A Generic Framework for Enforcing Security in Distributed Systems

vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
von

Dipl.-Inform. Richard Gay

geboren in Würselen, Deutschland

Referenten der Arbeit:	Prof. Dr. Heiko Mantel
	Prof. Dr. Christian W. Probst
Tag der Einreichung:	12.09.2017
Tag der mündlichen Prüfung:	27.10.2017
Hochschulkennziffer:	D 17

Darmstadt, 2017

Abstract

A large extent of today's computer programs is distributed. For instance, services for backups, file storage, and cooperative work are now typically managed by distributed programs. The last two decades also brought a variety of services establishing social networks, from exchanging short messages to sharing personal information to dating. In each of the services, distributed programs process and store sensitive information about their users or the corporations their users work for.

Secure processing of the sensitive information is essential for service providers. For instance, businesses are bound by law to take security measures against conflicts of interest. Beyond legal regulations, service providers are also pressed by users to satisfy their demands for security, such as the privacy of their profiles and messages in online social networks. In both instances, the prospect of security violations by a service provider constitutes a serious disadvantage and deters potential users from using the service.

The focus of this thesis is on enabling service providers to secure their distributed programs by means of run-time enforcement mechanisms. Run-time enforcement mechanisms enforce security in a given program by monitoring, at run-time, the behavior of the program and by intervening when security violations are about to occur. Enforcing security in a distributed program includes securing the behavior of the individual agents of the distributed program as well as securing the joint behavior of all the agents.

We present a framework for enforcing security in distributed programs. The framework combines tools and techniques for the specification, enforcement, and verification of security policies for distributed programs. For the specification of security policies, the framework provides the policy language CoDSPL. For generating run-time enforcement mechanisms from given security policies and applying these mechanisms to given distributed programs, the framework includes the tool CliSeAu. For the verification of generated enforcement mechanisms, the framework provides a formal model in the process algebra CSP. All three, the policy language, the tool, and the formal model allow for the distributed units of enforcement mechanisms to cooperate with each other. For supporting the specification of cooperating units, the framework provides two techniques as extensions of CoDSPL: a technique for specifying cooperation in a modular fashion and a technique for effectively cooperating in presence of race conditions. Finally, with the cross-lining technique of the framework, we devise a general approach for instrumenting distributed programs to apply an enforcement mechanism whose units can cooperate.

The particular novelty of the presented framework is that the cooperation to be performed can be specified by the security policies and can take place even when the agents of the distributed program do not interact. This distinguishing feature of the framework enables one to specify and enforce security policies that employ a form of cooperation that suits the application scenario: Cooperation can be used when one's security requirements cannot be enforced in a fully decentralized fashion; but the overhead of cooperation can be avoided when no cooperation is needed.

The case studies described in this thesis provide evidence that our framework is suited for enforcing custom security requirements in services based on third-party programs. In the case studies, we use the framework for developing two run-time enforcement mechanisms: one for enforcing a policy against conflicts of interest in a storage service and one for enforcing users' privacy policies in online social networks with respect to the sharing and re-sharing of messages. In both case studies, we experimentally verify the enforcement mechanisms to be effective and efficient, with an overhead in the range of milliseconds.

Zusammenfassung

Ein großer Anteil heutiger Computerprogramme wird durch verteilte Programme abgedeckt. So werden beispielsweise Dienste für Sicherungskopien, Dateiaustausch und computergestützte Zusammenarbeit üblicherweise durch verteilte Programme realisiert. Hinzu kamen insbesondere in den letzten zwei Jahrzehnten eine Vielzahl von Diensten für soziale Netzwerke zum Austausch von Kurznachrichten und persönlichen Informationen sowie in Form von Partnerbörsen. In jedem dieser Dienste speichern und verarbeiten verteilte Programme vertrauliche Informationen über die Benutzer der Programme oder auch deren Arbeitgeber.

Für Anbieter von Diensten („service provider") ist es essenziell, die Verarbeitung vertraulicher Informationen abzusichern. Beispielsweise werden Firmen gesetzlich dazu verpflichtet, Maßnahmen gegen finanzielle Interessenskonflikte zu ergreifen. Jenseits gesetzlicher Bestimmungen üben auch Nutzer Druck auf Dienstanbieter aus, Sicherheitsanforderungen wie beispielsweise den Schutz von Profilen und Nachrichten in sozialen Netzwerken umzusetzen. In beiden Fällen stellt bereits die Aussicht auf Sicherheitsverletzungen durch einen Dienstanbieter einen ernstzunehmenden Nachteil dar, der potenzielle Nutzer eines Dienstes abschreckt.

Der Fokus dieser Arbeit ist, Dienstanbietern zu ermöglichen, mit Hilfe von Laufzeit-Sicherheitsmechanismen („enforcement mechanisms") verteilte Programme abzusichern. Diese Mechanismen setzen Sicherheit in einem gegebenen Programm durch, indem sie das Programmverhalten zur Laufzeit beobachten und in den Programmablauf eingreifen wenn Sicherheitsverletzungen bevorstehen. In einem verteilten Programm umfasst dies nicht nur, das Verhalten der einzelnen verteilten Agenten des Programms abzusichern sondern auch das gemeinsame Verhalten aller dieser Agenten.

Diese Arbeit stellt ein Rahmenwerk für das Durchsetzen von Sicherheit in verteilten Programmen vor. Das Rahmenwerk integriert Werkzeuge und Techniken für die Spezifikation, das Durchsetzen und die Verifikation von Sicherheitsrichtlinien für verteilte Programme. Zur Spezifikation von Sicherheitsrichtlinien („security policies") stellt das Rahmenwerk eine Policysprache, CoDSPL bereit. Mit Hilfe des Werkzeugs CliSeAu können Sicherheitsmechanismen für gegebene Sicherheitsrichtlinien generiert und diese Mechanismen dann auf gegebene verteilte Programme angewandt werden. Für die Verifikation der so generierten Mechanismen bietet das Rahmenwerk ein formales Modell in der Prozessalgebra CSP an. Diese drei Bestandteile des Rahmenwerks bieten die Möglichkeit zur Kooperation zwischen den verteilten Einheiten des Mechanismus. Zur Unterstützung für die Spezifikation solcher Kooperation stellt das Rahmenwerk zwei Techniken in Form von Erweiterungen von CoDSPL bereit: eine Technik, mit der Kooperation in modularer Form spezifiziert werden kann sowie eine Technik für effektive Kooperation auch in Gegenwart von Race Conditions. Die Crosslining-Technik des Rahmenwerks beschreibt einen allgemeinen Ansatz zur Instrumentierung verteilter Programme mit Mechanismen deren verteilte Einheiten miteinander kooperieren können.

Neu am vorgestellten Rahmenwerk ist inbesondere, dass die gewünschte Form der Kooperation durch Sicherheitsrichtlinien spezifiziert werden kann und auch dann möglich ist, wenn die Agenten des verteilten Programms selbst nicht miteinander kommunizieren. Dadurch ermöglicht das Rahmenwerk sowohl die Spezifikation als auch das Durchsetzen von Sicherheitsrichtlinien mit einer auf das Anwendungsszenario zugeschnittenen Form von Kooperation. So kann Kooperation eingesetzt werden wenn die Sicherheitsanforderung nicht vollständig dezentral durchgesetzt werden kann. Jedoch kann Kooperation auch vermieden werden wenn sie im Anwendungsszenario nicht benötigt wird.

Die in dieser Arbeit vorgestellten Fallstudien bestätigen, dass sich das Rahmenwerk dafür eignet, spezifische Sicherheitsanforderungen in verteilten Diensten durchzusetzen. In den Fallstudien wird das Rahmenwerk verwendet um zwei Laufzeit-Sicherheitsmechanismen zu entwickeln: ein Mechanismus zum Durchsetzen einer Sicherheitsrichtlinie gegen Interessenskonflikte in einem verteilten Speicherdienst sowie ein Mechanismus zum Durchsetzen nutzerspezifischer Datenschutzrichtlinien beim Teilen von Nachrichten in sozialen Netzwerken. Experimentelle Evaluationen zu beiden Fallstudien bestätigen, dass die entwickelten Mechanismen sowohl effektiv als auch, mit einem Overhead im Bereich weniger Millisekunden, effizient sind.

Publications

Until the submission of this thesis, we have published contributions contained in this thesis at peer-reviewed conferences and workshops as follows:

- Partial descriptions of the cross-lining technique (Chapter 4) and of the syntax of the policy language CoDSPL (Chapter 3) have been published in [GHM14]. Detailed language semantics are published in this thesis for the first time. The architecture of CliSeAu (Sections 5.2 and 5.3) was also published in [GHM14].
- The application scenario of Chapter 6 and a high-level variant of the approach to enforcing security in the application scenario using CliSeAu have been submitted for publication [GHM+17a]. Moreover, parts of the description of related works in Section 9.8.1 share content with [GHM+17a].
- The case study of enforcing a Chinese Wall security requirement (Section 7.4) has been published in [GHM14]. For this thesis, the experimental evaluation was conducted again in a distributed setup and with the version of CliSeAu described in this thesis.
- The formal cooperation model for CliSeAu as well as the instance of this model for a concrete application scenario (Chapter 8) have been published in [GMS12].

The following of our contributions have been peer-reviewed and published but have not been included in this thesis.

- We have developed an exploit for interrupt-related covert channels and derived lower bounds for such channels through empirical evaluations and a model based on Shannon's information theory. The results were published in [GMS13; GMS15].
- We developed a technique for accelerating usage control by pre-computing decisions based on access correlations, and we developed an implementation of this technique based on CliSeAu [GHM+17b].

Acknowledgements

No person will make a great business who wants to do it all himself or get all the credit.

— Andrew Carnegie

First of all, I thank Heiko Mantel and Christian W. Probst for reviewing this thesis and for spending the time and energy this requires.

In particular, I want to thank Heiko also in his role as my Ph.D. supervisor. He gave me the opportunity to investigate the topic of this thesis and provided me with an environment that made this thesis possible. Neither the amount of time he dedicated for the supervision nor the breadth of his supervision is by any means self-evident. His dedication and the responsibilities he entrusted me with gave me the opportunity to grow in many ways. Finally, I thank Heiko for his patience with me—I suspect this has not always been easy.

I would like to thank Lujo Bauer, Úlfar Erlingsson, Cédric Fournet, and Joshua Guttman for intense discussions about my research that helped me sharpen my understanding. I am grateful to Arndt Poetzsch-Heffter, Alexander Pretschner, and Wolfgang Reif for giving me the opportunity to present and discuss my research results in their research groups and obtaining valuable feedback.

From the MAIS research group, I thank my former colleagues Markus Aderhold, Baskar Anguraj, Yuri Gil Dantas, Renate Drießler, Sarah Ereth, Sylvia Grewe, Tobias Hamann, Gudrun Harris, Jinwei Hu, Elisabeth Johannes, Abdullah Abdul Khadir, Ximeng Li, Steffen Lortz, Alexander Lux, Matthias Perner, Heide Rinnert, Jens Sauer, Johannes Schickel, David Schneider, Miriam Rifai-Schön, Dieter Schuster, Barbara Sprick, Artem Starostin, Henning Sudbrock, Markus Tasch, and Alexandra Weber for the countless insightful scientific as well as non-scientific discussions and for all the many fun times we had together. I am deeply indebted to Artem, my long-term office mate, for supporting me in many ways: sharing his experiences, his calm and thoughtful attitude, and lots of Russian proverbs. It has taken me a while to recognize how much this meant to me. Miriam deserves particularly warm gratitude for cheering me up in difficult times, for laughing with me, and for being a good friend. Special thanks also go to Markus, Matthias, and Sylvia for proof-reading parts of this thesis and providing valuable feedback.

I would like to thank Kevin Bouhsard, André Fischer, Christian Frieß, and Jan-Niklas Klocke, whom I (co-)supervised as apprentices and who allowed my to learn a lot about myself in the process. During my Ph.D. studies, I also had the pleasure of working with many students, in research as well as in teaching. Supervising and co-supervising Tobias Hamann, Sogol Mazaheri, Dominic Scheurer, Johannes Schickel, Daniel Specht, Markus Tasch, Moritz Tiedje, and Florian Wendel in their thesis projects was very inspiring and instructive for my own research. I also enjoyed the discussions with the students who participated in the "Dynamic Enforcement of Software Security" lab in 2013 and who provided valuable technical feedback about earlier versions of CliSeAu.

I would like to thank the DFG and LOEWE for their funding of parts of the research results presented in this thesis under the projects FM-SecEng (in the Computer Science Action Program, MA 3326/1-2) and CASED (www.cased.de). Project meetings provided opportunities for valuable exchange with other researcher, of which I particularly express my gratitude to Enrico Lovat, Andrei Popescu, Markus Rabe, and Tomasz Truderung. I also thank the developers of all the free software that I used in my research and in the production of this thesis, particularly including but not limited to: Awesome, Curl, Diaspora*, Gentoo, Java, Linux, R, Ruby, TeX and LaTeX, Vim, and Zathura.

Last but certainly not least I am forever grateful to my family and friends. My parents always gave me the opportunity and encouraged me to follow my interests; and yet they have been so incredibly forgiving that these interests often occupied too much of my time. I would like to thank my friends, most notably Manuel, with whom I found less time than I would have like to—and who nevertheless have always been there for me. Finally, I want to express my deepest gratitude to my love Sylvia, for giving me strength and security—the kind no mechanism can provide—and for infecting me again and again with her happiness.

Contents

Chapter 1 Introduction

With the ubiquitous availability of fast network access for businesses, private homes, and mobile devices, distributed software and services have taken over many traditional domains of non-distributed solutions. While services for file exchange over the Internet are widespread for a long time, even backups "in the Cloud" and office suites via online services as increasingly popular. At the same time, e-mail as the traditionally predominant technique for exchanging messages has been supplemented by a variety of online social networks for exchanging short messages and personal information as well as for dating. In each of these examples, client programs and services exchange, process, and store valuable and sensitive information about users or about the corporations their users work for.

The diversity of security requirements on distributed systems stretches beyond the domain of traditional access control for confidentiality and integrity. Businesses, for instance, are bound by law to take security measure, e.g., against conflicts of interest (in the USA, for instance, according to the Sarbanes-Oxley Act [Uni02]). The billions of today's users of social networks have privacy requirements on their profiles and messages [Gat07]. Incidents of security violations by service providers holding large amounts of sensitive data can harm not only the users of the system, including business customers, but through loss of reputation also the system provider itself. It is therefore crucial that the providers of online services employ adequate mechanisms for providing their users with the security they demand. In particular, these mechanisms should effectively ensure security despite the distributed nature of online services.

Already early work on access control points out networked computer systems as "likely to have a significant impact on computer security problems" [And72]. A variety of generic models [Sch00; LBW09], techniques [ES00b; CMJ+09], and tools [ES00b; BLW09; MJG+12] for enforcing security in non-distributed systems have been investigated. The introduction of usage control [PS04], which lifts access control to the subsequent usage of accessed resources, provoked the development of mechanisms for distributed enforcement [HPB+07; ZSS08; KP15].

The goal of this thesis is to improve the state of the art in security enforcement by lifting generic enforcement to distributed systems. Concretely, we focus on enabling generic means for cooperation within distributed enforcement mechanisms while preserving generic means of non-distributed enforcement mechanisms. To date, distributed enforcement mechanisms limit the security requirements that can effectively be enforced by constraining the possible forms of cooperation. For instance, mechanisms only allow cooperation to take place when agents communicate [MU00] and, particularly, only attached to the communication in the distributed system [SVA+04; OBM10; KP13], or build on general-purpose protocols for cooperation that constrain the security requirements that can be effectively enforced in a decentralized fashion [KP15].

In the remainder of this chapter we first introduce the state of the art in security enforcement with a particular focus on distributed systems (Section 1.1). Subsequently, Section 1.2 provides a more detailed account on the goals of this thesis. The chapter concludes with an overview of the contributions presented in this thesis and an outline of the organization of this thesis (Sections 1.3 and 1.4).

1.1. Enforcing Security

The security of a system can affect many and a whole range of stakeholders [SO05]. For instance, the users of an online shopping service require their transactions to be secured. Vendors of the shopping service require their listed products to be secured from manipulation. The provider of the shopping service requires that nobody can render the shopping service dysfunctional through the Internet.

The concrete security requirements of the individual stakeholders on a system can aim at several security goals [PP06, pp. 10–12]. For a secured transaction, for instance, a user's credit card details must not be accessible by other users. This requirement aims at *confidentiality*, requiring "that computer-related assets are accessed only by authorized parties" [PP06]. For another example, no user of the shopping service may modify the price of a product, unless she is the vendor of the product. This aims at *integrity*, requiring "that assets can be modified only by authorized parties or only in authorized ways" [PP06]. The requirement that it must not be possible that the shopping service's connection to its database is disabled through the Internet aims at *availability*, requiring "that assets are accessible to authorized parties at appropriate times" [PP06]. These three security goals – confidentiality, integrity, and availability, also called the *CIA triad* – constitute a common classification of security requirements in the literature [Bis03; PP06].

Stakeholders can take measures for their security requirements to be met. When a stakeholder has the system developed for herself and knows her security requirements at the time, she can demand the software architects and programmers to use techniques for producing a secure system in the first place (e.g, [Jür02; LBD02; Sea05]). For an existing system, a stakeholder can employ *static analyses* for security, i.e., analyses that examine a system's code without running it [CM04], to identify satisfaction or violation of security. (e.g., [VIS96; MS03; LL05; MS12]). When a stakeholder does not have any influence on the development of the system (e.g., by using off-the-shelf software) or does not know the security requirements at the time, she can use *run-time enforcement*, i.e., techniques that examine the program while it runs and intervene when necessary to make the program meet the security requirements (e.g., [ES00b; GBJ+08; BLW09; RBG+15]). Run-time enforcement involves a *security enforcement mechanism*, a “method, tool, or procedure for enforcing a security policy” [Bis03, p. 9]. In the literature, run-time enforcement is also known under the name *dynamic enforcement*. In the following, we abbreviate security enforcement mechanism by enforcement mechanism or simply mechanism.

1.1.1. Enforcing Security in Non-distributed Systems

For enforcing security in a non-distributed system, a stakeholder can choose from a variety of mechanisms proposed in the literature. Mechanisms have been developed for a wide spectrum of target systems (abbreviated *targets* in the following). One can distinguish two kinds of approaches for enforcing security on a target: mechanisms that control the target from another component of the system, such as the operating system, and mechanisms that are integrated into the code of the target. We introduce the former first in analogy with their historical appearance.

Early works in computer security advocate enforcement mechanisms for access control to protect resources stored in the system [War67; And72]. Such mechanisms typically have been and still are realized as part of the operating system (e.g., [Sal74; WCS⁺02]). For when the integrity of the operating system itself is to be secured, hardware-based mechanisms such as *CoPilot* [JFM⁺04] have been proposed. Some mechanisms residing within the operating system go beyond access control. When security requirements concern the access to resources as well as the subsequent use of these resources, *usage control* [PS04] can be employed. For usage control in the operating system OpenBSD, the mechanism by Harvan and Pretschner [HP09] can be integrated into the operating system. For the Android operating system, the Porscha [OBM10] mechanism is realized via a modified Android middleware to enforce security on transmitted messages. Other mechanisms address the particularities of Android’s confinement of applications, preventing privilege escalation through collusion attacks [GT14]. When the target is implemented in an interpreted programming language, the mechanism can also be placed into the interpreter rather than the operating system. This approach is pursued, e.g., by enforcement mechanisms for JavaScript targets that are integrated into the browser’s JavaScript interpreter [RHN⁺13; RBG⁺15] as well as by enforcement mechanisms for Java targets that are integrated into the JVM [NSC⁺08]. These mechanisms have in common that they are placed external to their targets’ code and control their targets through the targets’ interfaces to other system components.

The second approach pursued for enforcing security is to *encapsulate* the target by applying an enforcement mechanism directly to the code of the target [WLA⁺93]. Such encapsulation is realized by *instrumenting* the code of the target, i.e., by modifying the code to include the mechanism and to activate this mechanism when necessary. A variety of different techniques for instrumenting the code of the target have been proposed for Java, a language we focus on in this thesis. For instance, PoET [Erl04] and SASI [ES00b] perform the instrumentation before the target is started. JavaMOP [MJG⁺12] also performs the instrumentation before the start of the target but uses aspect-oriented programming for the instrumentation. In contrast, Polymer [BLW09] instruments the code after the target is started but before the code is loaded by the Java virtual machine. Beyond Java, instrumentation-based mechanisms have also been proposed, for instance, for x86 bytecode in the mechanisms Naccio [ET99] and a second variant of SASI, for Android's Dalvik bytecode [BGH$^+$14], and for Ruby [PBS14].

1.1.2. Enforceable Security Requirements

Several of the mechanisms named in Section 1.1.1 provide their users a policy language for tailoring the mechanism to their security requirements. For instance, PoET uses PSLang [Erl04] for its policy language, Polymer uses a Java-based policy language, and JavaMOP uses a combination of Java-based specifications with temporal logics or, alternatively, finite state automata specifications. The mechanisms support a diverse range of targets and, through their policy languages also a wide range of security requirements they can enforce. For the remainder of this thesis, we refer to enforcement mechanisms with these two characteristics as *generic*. Beyond PoET, Polymer, and JavaMOP, further mechanisms and likewise models of mechanisms aim at being generic [ET99; Sch00; ES00b; LBW09]. Examples of mechanisms that are not generic are the specialized mechanisms for access control mentioned in Section 1.1.1 [Sal74; WCS$^+$02].

Which security requirements a generic enforcement mechanism can enforce has been studied based on several formal models for capturing security requirements, with varied notions of what it means that a security requirement is enforced, and for several mechanisms. Some security requirements can be captured by predicates on individual sequences of events that model those executions of a target that satisfy the security requirement (e.g., [Sch00; LBW05; LBW09]). This class of security requirements includes safety and liveness [AS85; AS87]. Other security requirements demand that not individual executions are classified but that relationships between possible executions can be expressed. Such security requirements can be captured by predicates on *sets of* sequences of events [Man03; CS10] and include requirements for secure information flow or constraints on mean response times.

As criteria for what it means that a security requirement is enforced, Ligatti, Bauer, and Walker [LBW09] established the diametric soundness and transparency. *Soundness* means that with the mechanism applied to the target, the security requirement is never violated. *Transparency* means that when the target complies with the security requirement, the mechanism does not change the behavior of the target. Following Ligatti et al., we say that a mechanism *effectively enforces* a security requirement on a target if it soundly and transparently enforces it. Other criteria for enforcement mechanisms have been proposed. Traditionally, the so-called "principles for reference validation mechanisms" [And72] – subsequently also called "reference monitor assumptions" [Rus92] – demand a mechanism to be tamper-proof, to always be invoked (when an access takes place), and to be subject to analysis (i.e., to be *verifiable*) and testing. These criteria relate to effectiveness in that always being invoked is a prerequisite for effectiveness, being tamper-proof can be viewed as a variation of effectiveness under a strong attacker, and verifiability provides assurance of effectiveness. A complementary criterion is the *efficiency* with which a mechanism enforces a particular security requirement or the members of a range of security requirements, i.e., the amount of absolute or relative performance overhead of the mechanism at the run-time of the target in enforcing security. Newly proposed implementations of enforcement mechanisms often include an evaluation of the efficiency (e.g., [HP09; HMR12; KP13; DLJ15]). In the remainder of this thesis, we use the term "*enforceable*" without further qualification for security requirements that can be enforced effectively by a mechanism on a collection of targets.

Schneider [Sch00] has shown that the class of security properties that are soundly enforceable with *security automata*, a variant of non-deterministic automata that captures the termination of a target the possible countermeasure against security violations, falls into the class of all safety properties. Basin et al. [BJK+13] refine the notion and analysis of soundly enforceable properties by explicitly distinguishing observable and controllable events. Ligatti, Bauer, and Walker [LBW09] show that the class of security properties that are effectively enforceable with edit automata, automata that support countermeasures like suppressing and replacing of actions in addition to termination, subsumes also non-safety properties. Hamlen, Morrisett, and Schneider [HMS06b] characterize the class of properties enforceable by program rewriting. The program rewriting may analyse the target such that the class of enforceable properties essentially subsumes the class of decidable, i.e., statically analyzable, properties.[1]

1.1.3. Enforcing Security in Distributed Systems

In a distributed system, the target can itself possess a distributed architecture consisting of components that are distributed over several computers of the distributed system [TS14, p. 3]. In the following, we refer to such programs, which consist of components that are distributed over several computers, as *distributed programs*, and we refer to the individual autonomous, non-distributed components of a distributed program as *agents* [Fer99, p. 4]. For clarification, we refer to a target that is a distributed program as a *distributed target*. The security requirements for a distributed target might extend to more than individual requirements on the distributed components.

Firewalls [PP06, pp. 474–484], i.e., mechanisms for protecting targets by monitoring and manipulating network traffic within and at the boundaries of the distributed target, are one approach for enforcing security in distributed targets. Where the security requirement on a distributed target can be formulated in terms of network messages, a firewall can serve as a security enforcement mechanism. This would require, firstly, that the firewall can be placed at a node in the network through which the security-relevant network messages are transmitted. Secondly, it would require that the firewall is able to extract the security-relevant information from intercepted network messages and that this ability is not obstructed, e.g., by encryption.

Another approach for enforcing security in a distributed target is to apply instances of the non-distributed mechanisms discussed in Section 1.1.1 to the individual agents of the distributed target. This approach treats each agent, which by definition is non-distributed, as an individual non-distributed program. The individual mechanisms could observe the agents' internal behavior as well as the agents' interaction over the network, and thereby they are able to observe more than a firewall. In contrast to a firewall, such individual mechanisms can, however only observe the actions of individual agents. Consequently,

[1]The exception are unsatisfiable properties such as the property that forbids all executions, which can only be enforced through static analysis [HMS06b].

the approach can be pursued whenever the security requirement can be represented as a collection of security requirements on the individual agents of the distributed target.

Security requirements on distributed programs cannot be represented as a collection of security requirements on individual agents, in general, as observed by Martinelli and Matteucci [MM08] at the example of the Chinese Wall Security Policy [BN89]. Consider the following example scenario.

Example 1.1. A *storage service* provides a data storage infrastructure with servers in several data centers. Authenticated users can deposit files at the service and obtain files from the service on behalf of companies. The security requirement for the storage service is that, for avoiding conflicts of interest, no single user of the service may access files belonging to competing companies. $\diamondsuit$

The application scenario of Example 1.1 involves a distributed target whose agents are the servers distributed over the data centers as well as a security requirement against conflicts of interest. According to the security requirement, whether or not a user may access a file at one of the servers depends not only on the user's anterior accesses at the same server but also her accesses at other servers. Hence, the security requirement cannot be reduced to a collection of security requirements on the individual servers (i.e., the agents of the service). Isolated enforcement mechanisms, such as those aiming for non-distributed targets, could be applied to the servers in the scenario but could not enforce security in the example without at some point being overly conservative or letting security violations happen.

Mechanisms specifically proposed for enforcing security requirements in a distributed target often provide means for coordinating the enforcement that is performed at the individual agents of the distributed target. We refer to such mechanisms, i.e., enforcement mechanisms consisting of multiple distributed components whose components *cooperate*, as *distributed enforcement mechanisms*. To avoid confusion between the distributed components of a distributed target and the distributed components of a distributed enforcement mechanism, we refer to the latter as *units*. Several mechanisms for distributed targets attach the cooperation between units to the communication between the agents of the target. The units of Porscha and of the enforcement mechanism by Kelbert and Pretschner [KP13] exchange information about policies affecting the data that is exchanged between agents. The units of DiAna [SVA+04] cooperate by exchanging information about their state piggy-backed on the communication of the agents. The units of Moses [MU00] can cooperate by modifying messages that are exchanged by the agents of the target.

More recently, other forms of cooperation have been proposed. For instance, the mechanism by Kelbert and Pretschner [KP15] utilizes a distributed database for the cooperation between units. In contrast to the mechanisms mentioned in the previous paragraph, the cooperation performed by this mechanism can take place even at times when the agents of the target do not communicate. In the mechanism proposed by Decat, Lagaisse, and Joosen [DLJ15], the units also cooperate independently of communication between agents. The cooperation performed by this mechanism is used for exchanging information about authorizations for resources.

When security is defined by authorizations of individual users of the distributed target, a mechanism could require the users to prove that they are authorized (e.g., [LAB+92]). In

Example 1.1, a user could request from the mechanism at each individual server a certificate that no accesses to files in conflict with a particular company have been accessed. The user could then present such certificates to the mechanism at the one server at which she wants to access a file.

For non-distributed targets, generic enforcement mechanisms have been proposed and studied extensively [ES00b; Erl04; BLW09; MJG+12]. While such mechanisms exist also for distributed targets, the extent to which they are generic is rather limited compared to what is possible for a distributed enforcement mechanism: Some mechanisms perform cooperation only when the agents of the target communicate [MU00; SVA+04; OBM10; KP13]. A security requirement such as the one in Example 1.1, in which the security-relevant actions are not communication among the agents, can only be approximated by such mechanisms. A notable exception is the mechanism by Kelbert and Pretschner [KP15], which builds on the general-purpose distributed database Cassandra [LM10]. The mechanism uses the distributed database for cooperation by means of information exchange. As pointed out by Kelbert [Kel16, p. 90], establishing consistency in the distributed database incurs the penalty of synchronizing all nodes of the distributed database and thereby centralizing the database. In consequence, for effectively enforcing security in general, the mechanism can be used only in a centralized fashion. That is, the limitations constrain the security requirements that are effectively enforceable by the mechanism.

1.2. Goals of this Thesis

The goal of this thesis project is to provide techniques and tools for enforcing security in distributed programs. The security of a distributed program subsumes secure behavior of the individual agents of the distributed program as well as secure interplay between the agents and secure behavior of all the agents as a whole. In a distributed program neither information about the global state nor about the global ordering of events are known a-priori [Lam78; CL85] at the individual agents of the distributed program. A key challenge, thus, is establishing sufficient awareness in a distributed enforcement mechanism about global state and events for enforcing security without imposing unnecessary bottlenecks through synchronization.

Concretely, this thesis focuses on the providers of distributed systems, henceforth called *service providers*, as the target stakeholders and on malicious users of the distributed systems as the adversaries. These service providers apply security enforcement mechanisms for securing their distributed systems. Supporting other stakeholders, such as benign developers, as well as taking measures against other adversaries, such as malicious developers, are outside the scope of this thesis but is briefly discussed in Section 10.2.

A service provider desires a few elementary requirements to be satisfied by a security enforcement mechanism. Firstly, the service provider requires a sufficiently generic enforcement mechanism for her to enforce her particular security requirements. These security requirements can be originally hers or also originally the requirements of the system's users, henceforth simply called *users*. Secondly, the service provider demands verifiability of the mechanism. Verifiability shall assure the service provider of a low risk of security violations, because security violations can cause a loss of reputation of her

name	requirement	validation
(Req-1)	The enforcement mechanism should be generic	§ 3.6 (p. 47), § 5.6.1 (p. 76), § 5.7 (p. 80), § 6.3 (p. 86)
(Req-2)	The enforcement mechanism should be verifiable.	§ 3.6 (p. 47), § 5.6.3 (p. 79), § 5.7 (p. 80), § 5.7 (p. 80), § 6.3 (p. 86), § 8.4 (p. 148)
(Req-3)	The security requirements should be enforced effectively.	§ 5.5.2 (p. 74), § 5.6.2 (p. 77), § 5.7 (p. 80), § 6.4.4 (p. 103), § 7.4.4 (p. 122), § 8.5.3 (p. 153)
(Req-4)	The security requirements should be enforced efficiently.	§ 5.6.2 (p. 77), § 5.7 (p. 80), § 5.7 (p. 80), § 6.4.4 (p. 104), § 7.4.4 (p. 125)

Table 1.1.: Requirements of service providers and users

provided service. Thirdly, security requirements should be enforced effectively such that both security and functionality needs of the service provider are met. Fourthly, security requirements should be enforced efficiently, because bad service performance dissatisfies users of the service.

The malicious users we consider are characterized by the following capabilities and incapabilities. Malicious users are able to provide inputs to the target and to observe outputs of the target, and they are able to observe and modify network communication between agents of the target and between units of an enforcement mechanism. Malicious users are unable to influence the target or an enforcement mechanism by other means than inputs, outputs, and network communication. Particularly, they are unable to directly modify the target or the mechanism on the provider's computers.

We summarize the four requirements of service providers in Table 1.1. We validate the contributions against these requirements throughout the thesis, particularly at the places indicated in the last column of the table.

1.3. Overview of Contributions

This thesis presents a generic framework for enforcing security in distributed systems. The framework enables the specification of security policies for distributed programs, the enforcement of specified security policies, and the verification of effective enforcement. The framework is validated in two case studies. The framework consists of techniques, languages, tools, models, and case studies. Figure 1.1 displays the individual parts of the framework.

For the specification of security policies, the framework provides the policy language CoDSPL. CoDSPL supports the specification of policies for a wide range of security requirements in distributed programs. Through a technique for identifying and separating concerns, the framework enables the modularization of cooperative security policies. The modularity of resulting policies aims at a reduced complexity of policy specifications.

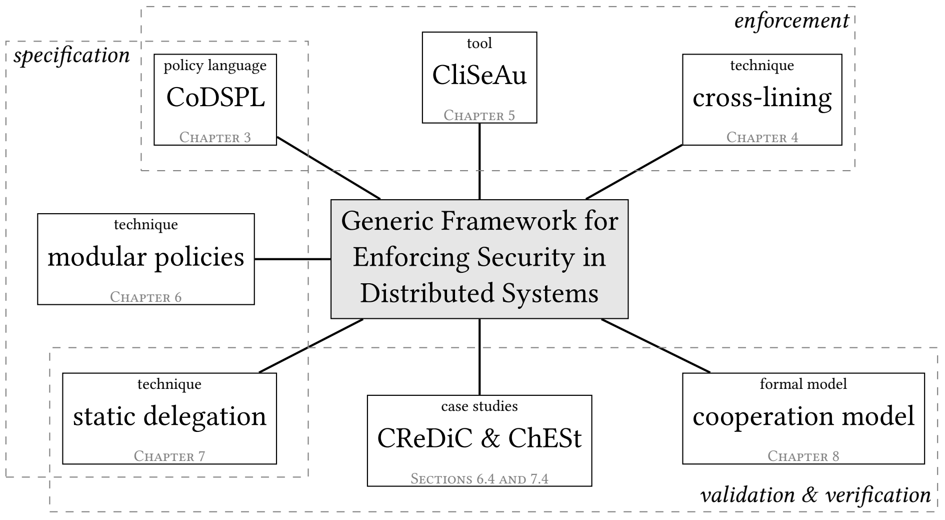


Figure 1.1.: Main contributions of this thesis

With static delegation, the framework provides a specialized cooperation technique for effectively enforcing a class of security requirements under race conditions. Both techniques, for modular policies and for static delegation, include extensions of CoDSPL through which the techniques can be applied in CoDSPL policies.

For the enforcement of security requirements, the framework provides CliSeAu, a tool for enforcing security in distributed systems. CliSeAu consists of a parametric implementation of a distributed enforcement mechanism that can be instantiated for enforcing given CoDSPL policies as well as a tool that instantiates and applies the parametric mechanism for enforcing a given CoDSPL policy in a given distributed target implemented in Java or in Ruby. As a generalization of the technique underlying CliSeAu for the design and application of distributed enforcement mechanisms, the framework comprises the cross-lining technique. The cross-lining technique enables cooperation within a distributed enforcement mechanism even when some agents of the distributed target do not communicate or are even idle.

For validating the virtues of the framework, the thesis provides two non-trivial case studies. The first case study presents CReDiC, a mechanism for enforcing users' privacy policies in a decentralized online social network. The second case study presents ChESt, a mechanism for enforcing a Chinese Wall Security Policy in a distributed storage service. Moreover, for enabling the modeling and verification of particular security policies, the framework provides a formal model of a distributed enforcement mechanism in CSP. The model captures the parametric modular architecture of CliSeAu's units.

Overall, the framework presented in this thesis covers specification, enforcement, validation and verification for security in distributed systems, addressing the four requirements of the addressed stakeholder, the service provider. The presented enforcement mechanism and policy language are generic (Req-1). Verifiability (Req-2) is addressed through modularity in the design of CliSeAu and in the design of CoDSPL policies as well as the formal cooperation model, which enables formal verification of sound enforcement. Effectiveness (Req-3) and efficiency (Req-4) are evaluated twofold: firstly through static analysis of the code of CliSeAu against common implementation flaws and inefficiencies; secondly through validation testing [Som16, p. 227] in the two conducted case studies.

1.4. Organization of the Thesis

Notions underlying the contributions of this thesis along with notational conventions concerning text as well as formal parts of this thesis can be found in Chapter 2. The contributions of this thesis are contained in Chapters 3 to 8. For an overview on where particular contributions can be found, the reader is referred to Figure 1.1. Related works to the technical contributions described in this thesis are discussed in Chapter 9. The body of this thesis concludes in Chapter 10 with a summary and with an outlook on conceptual and technical challenges that are related to the topic of this thesis and yet awaiting a solution.

Chapter 2

Notions and Notation

In this chapter, we introduce basic notions and notations that are relevant for several of the following chapters. Concretely, we first introduce mathematical notation and concepts, security properties as models of security requirements, and the BNF notation (Sections 2.1 to 2.3) used for unambiguous formal specifications in this thesis. Subsequently, we introduce notions from object-oriented programming, with a focus on the Java language, and from aspect-oriented programming, as well as the software design patterns used in this thesis (Sections 2.4 to 2.6). Finally, we introduce to concepts from the area of run-time enforcement, namely inlining and service automata (Sections 2.7 and 2.8).

2.1. Basic Mathematical Concepts and Notation

In this section, we introduce basic mathematical notions and notation that are used in this thesis, covering first-order logics, sets, relations, functions, sequences, and languages.

First-order logic We use formulas in first-order logic [EFT94] in formal modeling as well as in formal reasoning. In the following, we give a primer on syntax and semantics of first-order logic as we use it in this thesis.

A *formula* in first-order logic is a string of symbols based on a signature, a set of variable symbols, and a set of terms. A *signature* is a set in which each element represents either a constant symbol, or an n-ary function symbol, or an n-ary relation symbol ($n \geq 1$). Given a signature $\mathcal{S}$ and a set V of variable symbols, the set $\mathcal{T}(\mathcal{S}, V)$ of *terms* and the set $\mathcal{F}(\mathcal{S}, V)$ of formulas are inductively defined by the left column in Table 2.1. We call $\neg\varphi$ the *negation* of φ, $\varphi \wedge \psi$ the *conjunction* of φ and ψ, and $\varphi \vee \psi$ the *disjunction* of φ and ψ.

The semantics of first-order logic is given by the satisfaction relation, which is defined based on the notions of signatures from above as well as the notions of structures and interpretations introduced below. Given a signature $\mathcal{S}$, an $\mathcal{S}$*-structure* is a tuple $\mathfrak{A} = (D, \mathfrak{a})$, where D is a non-empty set, called the *domain* of $\mathfrak{A}$, and where $\mathfrak{a}$ is a map defined on $\mathcal{S}$ that maps every constant symbol in $\mathcal{S}$ to an element in D, maps every n-ary function symbol

syntax	**semantics**
if $v \in V$, then $v \in \mathcal{T}(\mathcal{S}, V)$;	if $v \in V$, then $v^{\mathfrak{I}} = \beta(v)$;
if $cs \in \mathcal{S}$ is a constant symbol, then $cs \in \mathcal{T}(\mathcal{S}, V)$;	if $cs \in \mathcal{S}$ is a constant symbol, then $cs^{\mathfrak{I}} = \mathfrak{a}(cs)$;
if $fs \in \mathcal{S}$ is an n-ary function symbol and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{S}, V)$, then $fs(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{S}, V)$.	if $fs \in \mathcal{S}$ is an n-ary function symbol and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{S}, V)$, then $fs(t_1, \ldots, t_n)^{\mathfrak{I}} = \mathfrak{a}(fs)(t_1^{\mathfrak{I}}, \ldots, t_n^{\mathfrak{I}})$.
if $t_1, t_2 \in \mathcal{T}(\mathcal{S}, V)$, then $t_1 = t_2 \in \mathcal{F}(\mathcal{S}, V)$;	if $t_1, t_2 \in \mathcal{T}(\mathcal{S}, V)$ and $t_1^{\mathfrak{I}} = t_2^{\mathfrak{I}}$, then $\mathfrak{I} \models t_1 = t_2$;
if $rs \in \mathcal{S}$ is an n-ary relation symbol and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{S}, V)$, then $rs(t_1, \ldots, t_n) \in \mathcal{F}(\mathcal{S}, V)$;	if $rs \in \mathcal{S}$ is an n-ary relation symbol, $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{S}, V)$, and $(t_1^{\mathfrak{I}}, \ldots, t_n^{\mathfrak{I}}) \in \mathfrak{a}(rs)$, then $\mathfrak{I} \models rs(t_1, \ldots, t_n)$;
if $\varphi \in \mathcal{F}(\mathcal{S}, V)$, then $\neg\varphi \in \mathcal{F}(\mathcal{S}, V)$;	if $\varphi \in \mathcal{F}(\mathcal{S}, V)$ and not $\mathfrak{I} \models \varphi$, then $\mathfrak{I} \models \neg\varphi$;
if $\varphi, \psi \in \mathcal{F}(\mathcal{S}, V)$, then $(\varphi \land \psi) \in \mathcal{F}(\mathcal{S}, V)$;	if $\varphi, \psi \in \mathcal{F}(\mathcal{S}, V)$, $\mathfrak{I} \models \varphi$, and $\mathfrak{I} \models \psi$, then $\mathfrak{I} \models (\varphi \land \psi)$;
if $\varphi, \psi \in \mathcal{F}(\mathcal{S}, V)$, then $(\varphi \lor \psi) \in \mathcal{F}(\mathcal{S}, V)$;	if $\varphi, \psi \in \mathcal{F}(\mathcal{S}, V)$ and $\mathfrak{I} \models \varphi$ or $\mathfrak{I} \models \psi$, then $\mathfrak{I} \models (\varphi \lor \psi)$;
if $\varphi \in \mathcal{F}(\mathcal{S}, V)$ and $v \in V$, then $\forall v\colon \varphi \in \mathcal{F}(\mathcal{S}, V)$;	if $\varphi \in \mathcal{F}(\mathcal{S}, V)$, $v \in V$, and, for each $x \in D$, $\mathfrak{I}[v \mapsto x] \models \varphi$, then $\mathfrak{I} \models \forall v\colon \varphi$;
if $\varphi \in \mathcal{F}(\mathcal{S}, V)$ and $v \in V$, then $\exists v\colon \varphi \in \mathcal{F}(\mathcal{S}, V)$.	if $\varphi \in \mathcal{F}(\mathcal{S}, V)$, $v \in V$, and there is $x \in D$ such that $\mathfrak{I}[v \mapsto x] \models \varphi$, then $\mathfrak{I} \models \exists v\colon \varphi$.

Table 2.1.: Inductive definitions of $\mathcal{T}$, $\mathcal{F}$, $\cdot^{\mathfrak{I}}$, and $\models$

in $\mathcal{S}$ to an n-ary function on D, and maps every n-ary relation symbol in $\mathcal{S}$ to an n-ary relation on D. An $\mathcal{S}$-*interpretation* is a tuple $\mathfrak{I} = (\mathfrak{A}, \beta)$, where $\mathfrak{A}$ is an $\mathcal{S}$-structure and $\beta : V \to D$ is a map from variable symbols to elements of the domain, called *assignment*. Let $\mathfrak{A} = (D, \mathfrak{a})$ be an $\mathcal{S}$-structure and V be a set of variable symbols. The *valuation* of terms, $\cdot^{\mathfrak{I}} : \mathcal{T}(\mathcal{S}, V) \to D$, and the *satisfaction relation*, $\models$, on $\mathcal{S}$-interpretations and formulas in $\mathcal{F}(\mathcal{S}, V)$ are inductively defined by the right column in Table 2.1. In the inductive definition, $\mathfrak{I}[v \mapsto x]$ denotes the same interpretation as $\mathfrak{I}$ except that it maps the variable symbol v to the value x.

We write $\varphi \Longrightarrow \psi$ to denote that every $\mathcal{S}$-interpretation $\mathfrak{I}$ satisfying $\mathfrak{I} \models \varphi$ also satisfies $\mathfrak{I} \models \psi$. We write $\varphi \Longleftrightarrow \psi$ to abbreviate $\varphi \Longrightarrow \psi$ and $\psi \Longrightarrow \varphi$.

We use the following abbreviating notations for formulas. We omit outermost parentheses around formulas and omit parentheses also around conjunctions in conjunctions and disjunctions in disjunctions. Moreover, we omit parentheses with the convention that relation symbols bind stronger than negation, negation binds stronger than conjunction, and conjunction binds stronger than disjunction. For a binary relation symbol rs, we sometimes write $t_1 \mathrel{rs} t_2$ instead of $rs(t_1, t_2)$. We sometimes write $\varphi \rightarrow \psi$ instead of $\neg\varphi \lor \psi$ to denote *logical implication*, and write $\varphi \leftrightarrow \psi$ instead of $(\varphi \rightarrow \psi) \land (\psi \rightarrow \varphi)$. Concerning quantifiers, sometimes we write $\forall v, w\colon \varphi$ instead of $\forall v\colon \forall w\colon \varphi$, write $\forall v \in A\colon \varphi$ instead of $\forall v\colon (v \in A \rightarrow \varphi)$, and write $\exists v \in A\colon \varphi$ instead of $\exists v\colon (v \in A \land \varphi)$.

In this thesis, we often omit explicit signatures, structures, and interpretations. We then implicitly refer to the signature that contains one constant symbol for every used constant, one function symbol for every function, and one relation symbol for every relation defined in the thesis. When no set of variable symbols is given, we implicitly refer to a set of symbols for meta-variables that is clear from the context. When no structure is given, we implicitly refer to a structure $\mathfrak{A} = (D, \mathfrak{a})$, where D is a universe comprising simple objects (e.g., numbers) as well as complex objects (e.g., sequences of simple objects) that become clear from the context and where $\mathfrak{a}$ maps each function and relation symbol to the corresponding homonymous function and, respectively, relation. For a formula φ, we say that φ holds (or: is satisfied), if and only if all free variables in φ are bound in the context and $(\mathfrak{A}, \beta) \vDash \varphi$ holds for an assignment β that assigns to each free variable in φ the respective value specified in the context.

Given a *sentence* φ, i.e., a formula with no free variables, we denote by δ_φ the constant 1 if φ holds true and the constant 0 otherwise.

In the following, we use the symbols φ and ψ as meta-variables ranging over formulas.

Sets We denote the empty set by $\emptyset$. We denote *extensional set specifications*, i.e., explicit enumerations of the elements contained in a set, by terms of the $\{x, y, z\}$. We also use *intensional set specifications* in the form $\{t(x_1, \dots, x_n) \in A \mid \varphi\}$, where the set then contains exactly those instances of the terms $t(x_1, \dots, x_n)$ that are in set A and satisfy the first-order logic formula φ in which x_1 to x_n can occur as free variables. For an element x and a set A, we write $x \in A$ to denote that x is contained in A. We write $A \subseteq B$ to denote that set A is a *subset* of set B or equal to B, i.e., that the formula $\forall x : (x \in A \rightarrow x \in B)$ is satisfied.

We denote the *union* of two sets A and B, i.e., $\{x \mid x \in A \vee x \in B\}$, by $A \cup B$, the *intersection* of two sets A and B, i.e., $\{x \mid x \in A \wedge x \in B\}$, by $A \cap B$, and the *difference* of two sets A and B, i.e., $\{x \in A \mid \neg(x \in B)\}$ by $A \setminus B$. For a finite set I, called *index set*, and sets A_i for all $i \in I$ we denote by $\bigcup_{i \in I} A_i$ the union of the sets A_i, i.e., the set $\{x \mid \exists i \in I : x \in A_i\}$. For a finite set A, we denote by $|A|$ the *cardinality* of A, i.e., the number of elements in A. We denote the *powerset* of a set A, i.e., the set $\{A' \mid A' \subseteq A\}$, by $\mathcal{P}(A)$.

Given an index set I, we denote by $(x_i)_{i \in I}$ the collection of values x_i for all $i \in I$ and call $(x_i)_{i \in I}$ a *family* (of values). A *partition* of a set A is a family $(B_i)_{i \in I}$ of sets, such that $B_i \neq \emptyset$ for each $i \in I$, $B_i \cap B_j = \emptyset$ for $i \neq j$, and $\bigcup_{i \in I} B_i = A$. Given two partitions $\mathcal{A} = (A_i)_{i \in I}$ and $\mathcal{B} = (B_j)_{j \in J}$, of a set A, we say that $\mathcal{A}$ is a *refinement* of $\mathcal{B}$ if for each $i \in I$ there exists a $j \in J$ such that $A_i \subseteq B_j$ holds.

We use $\mathbb{N}$ to denote the set of *natural numbers*, i.e., $\{1, 2, \dots\}$ (excluding 0), and $\mathbb{R}$ to denote the set of *real numbers*. We use the following notation for *intervals*, i.e., particular ranges of real numbers: We denote the *closed interval* $\{z \in \mathbb{R} \mid x \leq z \wedge z \leq y\}$ by $[x, y]$ and we denote the *left-closed, right-open interval* $\{z \in \mathbb{R} \mid x \leq z \wedge z < y\}$ by $[x, y)$.

Generally, we use symbols A, A', B, and so forth for meta-variables ranging over sets. Specifically for *index sets*, we use meta-variables I and J. For elements of sets we use the meta-variables x, y, and z as well as their indexed and primed variants. For elements of index sets, we use the meta-variables i and j. For meta-variables denoting partitions, we use calligraphic letters – typically the calligraphic counterpart of the partitioned set, i.e., $\mathcal{A}$ for a partition of set A.

Relations We write $(x_1, \ldots, x_n)$ for the *tuple* of elements x_1 to x_n. We denote by $A_1 \times \ldots \times A_n = \{(x_1, \ldots, x_n) \mid \forall i \in \{1, \ldots, n\} \colon x_i \in A_i\}$ the *cross product* of sets A_1 to A_n and call a set $R \subseteq A_1 \times \ldots \times A_n$ an (n-ary) *relation*.

A *binary relation* $R \subseteq A \times A$ on A is a *partial order* if it is *reflexive* (i.e., $(x, x) \in R$ for each $x \in A$), *antisymmetric* (i.e., $(x, y), (y, x) \in R$ implies $x = y$), and *transitive* (i.e., $(x, y), (y, z) \in R$ implies $(x, z) \in R$). The binary relation R is a *total order* if it is antisymmetric, transitive, and *total* (i.e., one of $(x, y) \in R$ and $(y, x) \in R$ holds). Moreover, the binary relation R is *symmetric* if $(x, y) \in R$ implies $(y, x) \in R$ and is *irreflexive* if $(x, x) \notin R$ for all $x \in A$.

We make use of maximal as well as minimal values of finite, totally ordered sets. Concretely, we use $\max\{t(x_1, \ldots, x_n) \in A \mid \varphi\}$ to denote the maximal element $t(x_1, \ldots, x_n) \in A$ satisfying formula φ and use $\min\{x_1, \ldots, x_n\}$ to denote the minimal element x_i in the given set. We use both notations only where the total order of the elements is unambiguous from the context.

In the following, we use R as meta-variable ranging over relations. For an n-ary relation R, we sometimes abbreviate $(x_1, \ldots, x_n) \in R$, i.e., the membership of the tuple $(x_1, \ldots, x_n)$ in R, by $R(x_1, \ldots, x_n)$. Moreover, for a binary relation R, we sometimes abbreviate $(x_1, x_2) \in R$ by $x_1 R x_2$.

Functions We formally treat *functions* as special kinds of relations. Concretely, we denote the set of *partial functions* from *domain* A to *co-domain* B by $A \rightharpoonup B = \{R \subseteq A \times B \mid ((x, y) \in R \wedge (x, y') \in R) \Rightarrow y = y'\}$. We denote the set of *total functions* from a domain A to co-domain B by $A \rightarrow B = \{R \in A \rightharpoonup B \mid \forall x \in A \colon \exists y \in B \colon (x, y) \in R\}$. Given a partial function $f : A \rightharpoonup B$, we denote by $\mathit{dom}(f)$ the set $A' \subseteq A$ of values for which f is defined (called the *domain of definition*). A total function $f : A \rightarrow B$ is *injective*, if for all $x, x' \in A$, $f(x) = f(x')$ implies $x = x'$.

Given sets A, A', and B and two partial functions $f : A \rightharpoonup B$ and $g : A' \rightharpoonup B$, we denote by $f \oplus g : (A \cup A') \rightharpoonup B$ the partial function defined by

$$(f \oplus g)(x) = \begin{cases} g(x) & \text{if } x \in \mathit{dom}(g), \\ f(x) & \text{if } x \in \mathit{dom}(f) \setminus \mathit{dom}(g), \end{cases}$$

and undefined otherwise. We call $f \oplus g$ the *overriding* of function f by function g.

In the following, we use symbols f, f', g, and so forth for meta-variables ranging over functions.

Sequences A *sequence* is an ordered collection of elements. We use $\langle\,\rangle$ to denote the empty sequence and $\langle x_1, \ldots, x_n \rangle$ to denote the sequence consisting of the elements x_1 to x_n in this order. For a given set A, we denote by A^* the set of all finite sequences over A. By A^+, we denote the set of all non-empty finite sequences over A, i.e., $A^* \setminus \{\langle\,\rangle\}$. Given two sequences $t_1 = \langle x_1, \ldots, x_n \rangle$ and $t_2 = \langle y_1, \ldots, y_m \rangle$, we denote by $t_1.t_2$ the concatenation of t_1 and t_2, i.e., $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$. We denote the *projection* of a sequence t to a set A, i.e., the sequence resulting from pruning from t all elements not contained in A, by

$t \upharpoonright A$. Formally, $t \upharpoonright A$ is defined inductively by

$$\langle\,\rangle \upharpoonright A = \langle\,\rangle,$$
$$\langle x\rangle.t \upharpoonright A = \begin{cases} \langle x\rangle.(t \upharpoonright A) & \text{if } x \in A, \\ t \upharpoonright A & \text{if } x \notin A. \end{cases}$$

Given a sequence $t = \langle x_1, \dots, x_n\rangle$, we denote the length n of t by $|t|$. We denote by $x \lhd t$ that an element x occurs in a sequence t, i.e., that $\exists t_1, t_2 : \ t = t_1.\langle x\rangle.t_2$ is satisfied. That an element x does not occur in a sequence t, we denote by $x \not\lhd t$. We call a sequence t a *prefix* of sequence t', denoted by $t \preceq t'$, if there is a sequence t'' such that $t.t'' = t'$. We call a set $B \subseteq A^*$ of sequences over A *prefix-closed*, if $\forall t, t' : ((t \in B \wedge t' \preceq t) \rightarrow t' \in B)$ is satisfied.

In the following, we use t as well as its indexed and primed forms as meta-variable ranging over sequences.

Words and languages We use Σ to denote the universe of symbols we implicitly assume all languages introduced in this thesis to be constructed upon. This universe contains all alphanumeric symbols as well as punctuation symbols, whitespace symbols, control symbols (such as line breaks), and mathematical symbols. We use the symbol $\mathcal{W} = \Sigma^*$ to refer to the set of all words over Σ. For a set $A \subseteq \Sigma$, we denote by $\mathcal{W}_A = (\Sigma \setminus A)^*$ the set of all words in which symbols from A do *not* occur. By the symbol $\varepsilon \in \mathcal{W}$, we denote the *empty word*, and by $w\ u$, we denote the *concatenation* of the words w and u.

In the following, we use w and u as meta-variables ranging over words.

2.2. Security Properties

We capture security requirements on programs formally by security properties. By a security property, we unambiguously distinguish "secure" from "insecure" executions of programs. We capture this as follows.

Definition 2.1. An *event* is a term that captures an atomic action of an agent [Man03]. ◊

Definition 2.2. Given a program *Prog* and a set E of events in which the program can in principle engage, an *execution* of *Prog* is a (finite) sequence $t \in E^*$ of events and captures the individual atomic actions performed by the program in the order in which they are performed. ◊

For example, the action "open the file `/etc/passwd`" can be captured by the event *open*(`/etc/passwd`). When a program first reads the content of file `/etc/passwd` into a buffer and subsequently sends its content over the network to some recipient, can be modeled by the execution $\langle \mathit{read}(\texttt{/etc/passwd}, \mathit{buf}), \mathit{send}(\mathit{buf})\rangle$.

Definition 2.3. Given a set E of events a *security property* is a set $P \subseteq E^*$ of sequences of events. ◊

Given a security property P, we call each sequence $t \in P$ *security-compliant* and call each sequence $t' \notin P$ a *security violation*.

Security properties modeled as sets of sequences of events are sufficiently expressive to capture a variety of security requirements. For example, discretionary or mandatory access control requirements [Bis03, pp. 103–104] can be captured by the set of sequences in which each of the events either captures a non-access action or captures an allowed access action. For another example, some usage control requirements [PS04], such as the requirement that certain sensitive documents may be printed at most 5 times, can be captured as well. In this latter example, the security property would consist of all sequences in which for each sensitive document there are at most 5 events capturing a printing action.

Outside the expressiveness of security properties as defined in Definition 2.3 are requirements that cannot be expressed as properties of individual executions. Such security properties are also outside the scope of this thesis. Examples of such properties include formalizations of non-interference [GM82], capturing, e.g., the lack of dependencies between confidential inputs and public outputs. Such lack of dependence can be captured, e.g., by requiring that two executions that only differ in confidential inputs must not differ in the public outputs. That is, such properties do not classify individual executions but collections of executions. Another example of properties outside the scope of this thesis are properties that capture availability by constraining average response times of programs (e.g., [CS10]), which also involves classifying collections of executions rather than individual executions.

Our definition of security property is adopted from Ligatti, Bauer, and Walker [LBW05]. Other formal models of security properties also involve sets of sequences. For instance, sets of both finite and infinite sequences of events were proposed for capturing additionally properties of nonterminating programs [Sch00; LBW09; BJK+13]. Sets of sequences of *sets of* events were proposed for capturing simultaneously occurring actions [Lov15; Kel16]. For when the evolution of a program's state is in the focus of the security requirements, models of security properties as sets of infinite sequences of program *states* were proposed [AS85; AS87]. Beyond the scope of security, properties have also been defined to classify finite sequences of events into categories such as "match", "fail", and "don't know" [MJG+12]. The term security property has also been used for sets of sets of sequences of events [Man03], which in other works are called security policies [Sch00] or *hyperproperties* [CS10]. In the remainder of this thesis, we use the term security properties as introduced in Definition 2.3 on page 15 and use the term hyperproperties for sets of sets of sequences.

Safety and liveness Two particular classes of security properties relevant for this thesis are safety properties and liveness properties. Intuitively, a safety property specifies that "something bad" must never happen and, consequently, any execution containing a "bad" event violates the property [AS85]. Conversely, a liveness property specifies that "something good" must eventually happen during an execution. Safety and, particularly, liveness properties are commonly characterized based on infinite sequences [AS85]. In this thesis, we formally define the two classes of security properties as follows, where we adopt the formalizations of Ligatti, Bauer, and Walker [LBW05; LBW09], which build on finite rather than infinite sequences.

Definition 2.4. A security property $P \subseteq E^*$ is a *safety property* over set E of events if and only if $\forall t \in E^* : (\neg P(t) \rightarrow \forall t' \in E^* : (t \preceq t' \rightarrow \neg P(t')))$ holds [LBW05]. ◊

Definition 2.5. A security property $P \subseteq E^*$ is a *liveness property* over set E of events if and only if $\forall t \in E^* : \exists t' \in E^* : (t \preceq t' \rightarrow P(t'))$ holds [LBW09]. ◊

Security properties in distributed programs The notions of safety and liveness properties distinguish security properties by the kind of security required. In the context of distributed programs, we further introduce an architectural aspect of security properties. Concretely, some security properties for distributed programs can be represented as a collection of security properties on the individual agents of distributed programs in the following sense: Security violations can soundly and completely be detected by independently analyzing the behavior of the individual agents. We capture such security properties as follows.

Definition 2.6. Let *DP* be a distributed program whose individual agents are identified by the elements of the finite nonempty set I and let $(E_i)_{i\in I}$ be the family of sets of events of the individual agents of *DP*. Let P be a security property over E, where $E = \bigcup_{i\in I} E_i$. We call P *localizable* with respect to $(E_i)_{i\in I}$ if and only if there is a family $(P_i)_{i\in I}$ of security properties over $(E_i)_{i\in I}$ such that

$$\forall t \in E^* : (t \in P \leftrightarrow \forall i \in I : \; t \upharpoonright E_i \in P_i) \qquad \text{(LOC)}$$

holds. Conversely, we call P *non-localizable* with respect to $(E_i)_{i\in I}$ if P is not localizable with respect to $(E_i)_{i\in I}$. ◊

The family $(P_i)_{i\in I}$ in the definition captures the collection of security properties by which the individual agents of the distributed program can "locally" detect a security violation. The sentence (LOC) faithfully captures that such detection is possible soundly and completely. For each security violation $t \notin P$, there must be an agent $i \in I$ such that $t \upharpoonright E_i$, i.e., the local view of agent i on execution t, is a security violation with respect to the local security property, P_i, of agent i. That is, security violations can be detected completely by independently analyzing the agents' local views. Conversely, for each detection $t \upharpoonright E_i \notin P_i$, also $t \notin P$ must hold. That is, security violations can soundly be detected by independently analyzing the agents' local views.

A non-localizable security property cannot be represented as a collection of security properties on the individual agents of a distributed program. For detecting some of the security violations, the individual agents' views in isolation do not suffice. Rather, the security property requires the agents of the distributed program to act in concert. Therefore, to characterize non-localizable security properties positively, we also refer to them as *concerted* security properties.

In Definition 2.6, the family of security properties, $(P_i)_{i\in I}$, is existentially quantified. For simplifying the analysis of whether a security property is localizable, the following theorem provides a construction of the family of security properties.

Theorem 2.1. *Let E be a set of events, P be a security property over E, and $(E_i)_{i\in I}$ be a family of sets of events with $\bigcup_{i\in I} E_i = E$. Then P is localizable with respect to $(E_i)_{i\in I}$ if and only if sentence* (LOC) *holds for the family $(P_i)_{i\in I}$ of security properties with $P_i = \{t \upharpoonright E_i \mid t \in P\}$.* ◊

Proof. Let E be a set of events, P be a security property over E, and $(E_i)_{i\in I}$ be a family of sets of events with $\bigcup_{i\in I} E_i = E$. Let $(\tilde{P}_i)_{i\in I}$ be a family of security properties defined by $\tilde{P}_i = \{t \upharpoonright E_i \mid t \in P\}$. We show both directions of the "if and only if". Firstly, if sentence (LOC) holds for $(P_i)_{i\in I} = (\tilde{P}_i)_{i\in I}$, then trivially P is localizable, as $(\tilde{P}_i)_{i\in I}$ stands as a witness for the existentially quantified $(P_i)_{i\in I}$ in Definition 2.6. Secondly and conversely, if P is localizable, then there exists a family $(P_i)_{i\in I}$ of security properties that satisfies the sentence (LOC). Let such $(P_i)_{i\in I}$ be fixed in the following. By splitting the bi-implication in (LOC) we get

1. $\forall t \in E^* : (t \in P \rightarrow \forall i \in I : t \upharpoonright E_i \in P_i) \Longrightarrow \forall t \in P : \forall i \in I : (t \upharpoonright E_i \in P_i)$

   $\Longrightarrow \forall i \in I : \{t \upharpoonright E_i \mid t \in P\} \subseteq P_i \Longrightarrow \forall i \in I : \tilde{P}_i \subseteq P_i$
2. $\forall t \in E^* : ((\forall i \in I : t \upharpoonright E_i \in P_i) \rightarrow t \in P)$

   $\Longrightarrow \forall t \in (E^* \setminus P) : \exists i \in I : (t \upharpoonright E_i \notin P_i)$

To show that (LOC) holds for $(\tilde{P}_i)_{i\in I}$, let $t \in E^*$ be arbitrary but fixed. If $t \in P$, then by definition of $\tilde{P}_i$, $t \upharpoonright E_i \in \tilde{P}_i$ holds for each $i \in I$. Conversely, if $t \notin P$, then by (2) we have $\exists i \in I : (t \upharpoonright E_i \notin P_i)$ and by (1) we particularly have $\exists i \in I : (t \upharpoonright E_i \notin \tilde{P}_i)$, as it was to be shown. □

We conclude by illustrating localizable and concerted security properties at the following two examples.

Example 2.1. For an example of a security property that is localizable, consider a distributed storage service with the requirement that user Alice must not log in at any of the distributed servers. The service consists of 5 servers with identifiers $I = \{1, 2, 3, 4, 5\}$. The event $login_i(Alice) \in E_i$ captures a successful login of Alice at server $i \in I$. We capture the security requirement by the security property $P = \{t \in E^* \mid t \upharpoonright E_{login} = \langle\,\rangle\}$, where $E = \bigcup_{i\in I} E_i$ and $E_{login} = \{login_i(Alice) \mid i \in I\}$.

The security property P is localizable with respect to $(E_i)_{i\in I}$, as the family $(P_i)_{i\in I}$ of security properties defined by $P_i = \{t \in {E_i}^* \mid t \upharpoonright \{login_i(Alice)\} = \langle\,\rangle\}$ satisfies the sentence (LOC). ◊

Example 2.2. For an example of a concerted security property, consider a distributed storage service consisting of 5 servers with identifiers $I = \{1, 2, 3, 4, 5\}$. The security requirement on the service is that user Bob must not access both the balance sheets *bsA* and *bsB* of Bank A and, respectively, Bank B during one execution of the service. The event $access_i(Bob, bs) \in E_i$ captures a successful access by Bob to the balance sheet $bs \in \{bsA, bsB\}$ at server $i \in I$. We capture the security requirement by the security property $P = \{t \in E^* \mid \neg\exists i, j \in I : (access_i(Bob, bsA) \lhd t \wedge access_j(Bob, bsB) \lhd t)\}$, where $E = \bigcup_{i\in I} E_i$.

The security property P is concerted with respect to $(E_i)_{i\in I}$. According to Theorem 2.1, P could only be concerted (i.e., non-localizable), if sentence (LOC) holds for $(P_i)_{i\in I}$ defined by $P_i = \{t \in E^* \mid \neg(access_i(Bob, bsA) \lhd t \wedge access_i(Bob, bsB) \lhd t)\}$. However, for the sequence $t = \langle access_1(Bob, bsA), access_2(Bob, bsB)\rangle$ we have $t \notin P$ but $\forall i \in I : (t \upharpoonright E_i \in P_i)$. Hence, sentence (LOC) is not satisfied and P is concerted with respect to $(E_i)_{i\in I}$. ◊

Remark 2.1. Literature does not provide generally accepted meanings of the terms "security requirement", "security policy", and "security property". In this thesis we refer to natural-language statements of requirements from the CIA triad as security requirements and refer to formalizations of security requirements as sets of sequences as security properties. By the term “*security policy*”, we refer to a machine-readable expression in a policy language understood by some enforcement mechanism (as, e.g., Ponder [DDL+01]). In particular, a security policy can be operational as well as declarative. Intuitive, the purpose of a security policy is to make the enforcement mechanism enforce particular security requirements. We do not use other meanings of the term “security policy” found in the literature, notably a strategy addressing a whole organization rather than an enforcement mechanism (e.g., Schneier [Sch04, pp. 308–309]) and a set of sets of executions (e.g., Schneider [Sch00]). As an exception to this nomenclature, we use the established term “Chinese Wall Security Policy” for the (natural-language) security requirement stated in the work by Brewer and Nash [BN89]. ◊

2.3. BNF

In this thesis, we use Wirth’s variant [Wir77] of the *Backus-Naur Form* [BBG+60] for specifying context-free grammars that define the syntax of a language. We abbreviate this variant by *BNF* in the remainder of the thesis. Following Wirth, we provide the syntax of BNF using BNF itself:

$$\begin{array}{ll}
\mathit{grammar} & = \{\, \mathit{production}\ \}. \\
\mathit{production} & = \mathit{identifier}\ \text{‘}{=}\text{’}\ \mathit{expression}\ \text{‘.’}. \\
\mathit{expression} & = \mathit{term}\ \{\ \text{‘}|\text{’}\ \mathit{term}\ \}. \\
\mathit{term} & = \mathit{factor}\ \{\, \mathit{factor}\ \}. \\
\mathit{factor} & = \mathit{identifier}\ \mid\ \mathit{literal}\ \mid\ \text{‘(’}\ \mathit{expression}\ \text{‘)’} \\
 & \quad \mid\ \text{‘[’}\ \mathit{expression}\ \text{‘]’}\ \mid\ \text{‘\{’}\mathit{expression}\ \text{‘\}’}. \\
\mathit{literal} & = \text{‘“’}\ \mathit{character}\ \{\ \mathit{character}\ \}\ \text{‘“’}.
\end{array}$$

In this BNF, *identifier* denotes a non-terminal symbol. A sequence of ASCII characters enclosed in single quotes denotes a terminal symbol. Two successive single quotes within a terminal symbol represent the single quote character. An expression enclosed in curly braces denotes repetition, zero or more times. An expression enclosed in square braces denotes optionality. Terms separated by a pipe symbol denote alternative choice. A sequence of factors without separation symbol denotes concatenation. Then a *grammar* is a sequence of zero or more production rules, of which each production rule associates a non-terminal with an expression. An expression is a collection of alternatives, of which each is a sequence of terms, i.e., non-terminals, terminals, grouped expressions, optional expressions, and repeated expressions. For a given grammar specified in BNF, we refer to the set of all words that can be produced by the production rules for a given non-terminal *identifier*, by $\mathcal{L}(\mathit{identifier})$.

2.4. Java

Java is a general-purpose object-oriented programming language. All Java code developed for this thesis runs conforms to the *Java 8 SE* language and API [GJS+14], i.e., the standard edition of Java version 8. The presentation chosen for this thesis, however, does not go into a level of code details at which the precise version of Java makes a difference. In the following, we briefly introduce the Java-related terms used in this thesis.

Core concepts of the Java language In this thesis, we use the term *class* as in Java for a code template from which *objects*, i.e., individual instances of the class, can be created. A class can define *methods*, i.e., procedures associated with every object of the class, *constructors*, i.e., procedures specifically for the construction of objects of the class, as well as *fields*, i.e., variables associated with every object of the class. A class can furthermore define *static methods*, i.e., procedures associated to the class itself, as well as *static fields*, i.e., variables associated to the class. A method can have a possibly empty list of *formal parameters* and can have a *return type*. When a method is *called*, a list of *actual arguments* are provided for the formal parameters. A method with a return type must, upon non-exceptional completion of its execution, provide a *return value* to the caller of the method. Like a method, a constructor also has a list of formal parameters but does not have a return type. A class can *inherit* from another class, making the former a *subclass* of the latter. A class can also declare *abstract methods*, i.e., methods for which only the name, the formal parameters and the return type are declared but the implementation must be provided by a subclass. A class containing abstract methods is called an *abstract class*. Java also provides the concept of *interfaces*, constructs that declare methods (like abstract methods of a class) but cannot have any fields. When a classes implements an interface, it provides code for all methods declared by the interface. Methods and fields of a class can be *private*, *protected*, or *public*, meaning that they are accessible only from code of the class, code of the class as well as of subclasses, or code of any class, respectively.

The Java language provides a few *primitive types*, for holding integers, floating-point values, Boolean values, and characters. Additionally, the Java language includes a large collection of classes and interfaces, called the *Java API*, which implement commonly used functionality for, e.g., data structures and network sockets. The Java API also provides means for *serialization* and *deserialization*, i.e., for turning in-memory objects into a storable piece of data and, respectively, turning back the latter into the former. Java code is organized into *Java packages*, sets of files sharing the same identifier provided to the **package** keyword in the source code and typically residing in the same directory.

Compilation, execution, and analysis of Java programs Java code is typically compiled to *Java bytecode*, in the *Java virtual machine language* (*JVML*) [LYB+14], for being interpreted by the *Java virtual machine* (*JVM*). Multiple compiled units of Java code are typically placed into a *Java Archive* (*JAR* file), which is a container that maintains the association between the file names of the compiled units and their content. Code documentation for Java is typically specified through code comments with a specific syntax (e.g., comments initiated by the "/**" sequence) from which the *Javadoc* tool of Java generates a documentation of packages, classes, methods, and fields.

For analyzing Java source code as well as Java bytecode with regard to various aspects, several tools exist. In this thesis, we particularly make use of FindBugs [AHM+08], PMD [PMD], and Checkstyle [Che], each being the result of several years of development. FindBugs is a tool for finding bugs in Java bytecode based on static analysis. The tool analyzes the Java bytecode with regard to several kinds of problematic code, including dereferencing of null pointers, wrong use of the Java API, and even inefficient code. PMD is an analysis tool for several programming languages, particularly including Java source code, that checks the code for overly complex code parts, dead code, and inefficient code. Checkstyle is a tool for checking Java source code against stylistic criteria such as indentation, naming of variables, etc., as well as code documentation criteria, particularly regarding complete and properly formatted Javadoc comments. Checkstyle includes an implementation of checks against the *Google Java Style Guide*, i.e., "Google's coding standards for source code in the Java programming language" [Gooa].

2.5. Aspect-oriented Programming

Aspect-oriented programming (brief: *AOP*) is a technique for realizing separation of concerns in computer programs [KLM+97; KHH+01]. Frequently named examples of functionality for which AOP is employed to achieve separation of concerns include logging, transactions, and, most relevant for this thesis, security.

In this thesis, we use concepts, techniques, and tools from aspect-oriented programming for instrumenting the code of a target with an enforcement mechanism. In particular, we utilize *AspectJ*, the de-facto standard AOP implementation for Java. We here briefly recapitulate the key notions from AOP that are relevant for this thesis. For further details, we refer to [KLM+97; KHH+01; WKD04]. A *join point* is an execution of a *program operation* (an instruction or atomic statement in the language of a program). A *pointcut* is a set of join points. We also use the term pointcut for referring to a specification of a set of join points. An *advice* is a method-like piece of code that is bound to a pointcut and whose purpose is to define additional program behavior for all join points specified by the pointcut. A particular kind of advice is the *around-advice*, which is an advice that is executed instead of the join point and that may or may not invoke the join point during its execution. An *aspect* is a set of advice together with a set of pointcuts and ordinary code. A *join point shadow* for a given join point is a point in the code of a program whose execution triggers the join point [MKD03; BH12]. Given a program and an aspect, the process of realizing that during the execution of the program, the behavior specified by each advice of the aspect is performed at each join point matching the pointcut of the advice, is called *weaving*.

2.6. Software Design Patterns

A *software design pattern* "names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design" [GHJ+95, p. 3]. We use software design patterns in the high-level and low-level architectures proposed in this thesis.

The *Abstract Factory pattern* [GHJ+95, pp. 87–95] is a *creational design pattern*, i.e., a pattern that "concern[s] the process of object creation" [GHJ+95, p. 10]. The pattern can be applied, for instance, when the creation and representation of a family of products should be transparent to the clients using the concrete products. Conceptually, the Abstract

Factory pattern involves five kinds of entities: an abstract factory, one or more concrete factories, abstract products, concrete products, and a client. An *abstract product* defines an interface for a particular kind of products. The *abstract factory* defines an interface for the creation of each abstract product from the family of products: For each kind of product of the family, this interface declares a *factory method*, i.e., a method for creating instances of this kind. Each *concrete product* implements a particular representation of an abstract product. A *concrete factory* inherits from the abstract factory and, as such, implements a particular way of creating particular concrete products from the family of products. The *client* uses a concrete factory and concrete products through the interfaces defined by the abstract factory and the abstract products. The pattern allows the representation of products to be transparent to clients through the distinction between abstract products and concrete products, and allows the creation of products to be transparent through the distinction between the abstract factory and concrete factories.

The *Strategy pattern* [GHJ+95, pp. 315–323] is a *behavioral design pattern*, i.e., a pattern that "characterize[s] the ways in which classes or objects interact and distribute responsibility" [GHJ+95, p. 10]. The pattern can be applied, for instance, in an architecture in which several different but yet related behaviors exist and a client shall be configurable to use either of them. Moreover, the pattern can be applied when a client shall not know about internal data used by an algorithm that implements some behavior. Conceptually, the Strategy pattern involves three kinds of entities: an abstract strategy, one or more concrete strategies, and a context. The *abstract strategy* defines an interface through which the behaviors can be invoked. In the Java language, an abstract strategy can be implemented as an interface. Each *concrete strategy* is an instance of the abstract strategy and implements some concrete behavior. The *context* is the point in the architecture in which a concrete strategy is used through the interface defined by the abstract strategy. In particular, the context does not use specifics of the concrete strategy that exceed the interface defined by the abstract strategy.

The *Constructor Injection pattern* [Fow04] is a low-level architectural pattern for separating the concerns of constructing and, respectively, using of an object of a particular type. According to the pattern, the object that uses another object obtains the latter as an argument of its constructor. That is, the construction of the latter object is not performed by the former object. By avoiding that the former object creates the latter object itself, the concrete type of the latter object can vary independently of the former object. Note that the Constructor Injection pattern can be used for realizing the Strategy pattern, namely by passing a concrete strategy through a constructor.

2.7. Inlining

Instrumentation is one possible technique for encapsulating a given program by applying an enforcement mechanism. Several proposed security enforcement mechanisms for non-distributed programs have been proposed to be applied to a given program through instrumentation of the program [ET99; ES00b; Erl04; BLW09; ADG09; MJG+12; ZTE13; BGH+14; PBS14]. Generally, the goal of such instrumentation techniques is that the enforcement mechanism is invoked at run-time by the program itself whenever the program

```
1   File file = new File(fileName);
2   if (file.exists()) {
3     sendFile(file);
4   } else {
5     sendErrMsg(fileName);
6   }
```

Listing 2.1.: Example code subject to inlining

```
1   File file = new File(fileName);
2   if (file.exists()) {
3a    if (!EM.check("send", file, this.user)) {
3b      System.exit(403);
3c    } else {
3d      EM.record("send", file, this.user);
3       sendFile(file);
3e    }
4   } else {
5     sendErrMsg(fileName);
6   }
```

Listing 2.2.: Resulting code after inlining

is about to perform a security-relevant action.

Technically, the individual instrumentation techniques differ from each other in that the mechanisms target different programming languages or use particular tools and techniques for modifying the relevant portions of the target's code. The techniques have in common that they place code of the enforcement mechanism figuratively around those lines of the existing code of the target whose execution might be relevant for security. When the resulting code is executed, the code of the mechanism is carried out sequentially interleaved with the original code of the target. Following Erlingsson and Schneider [ES00a], we refer to such program instrumentation techniques for enforcing security as *inlining*. Accounting for the technical differences between the individual inlining techniques, we use the term here in a way that aims to account for the variations found in the literature. Note that the inlining technique does not involve modifications of the environments in which the agent is supposed to be executed. We illustrate inlining at the following example.

Example 2.3. Let the Java-like code fragment in Listing 2.1 be part of the code of a program that provides access to files as a service. The code checks whether a file, whose name is stored in variable fileName, exists or not (Lines 1 and 2). If the file exists, the code sends the file to its destination (Line 3). Otherwise, the code sends out an error message (Line 5). The code is part of a program in which conflicts of interest shall be prevented (in analogy to Example 1.1 on page 6). In this context, calls to the sendFile method are security-relevant actions: They could violate the security requirement by sending a file to a user who previously already accessed a conflicting file.

The result of inlining an enforcement mechanism for the security requirement could

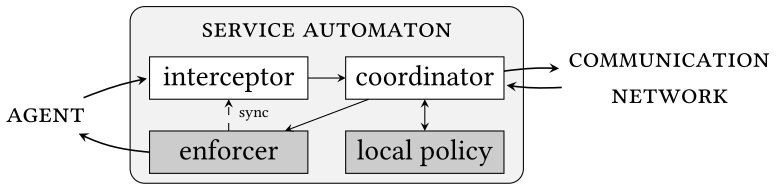


Figure 2.1.: Architecture, interfaces, and parameters of a service automaton [GMS12]

be as shown in Listing 2.2 on page 23, in which the shaded lines are added compared to Listing 2.1 on page 23. The added code first performs a check whether executing sendFile should be permitted (Line 3a). If the check fails, the code terminates the program before sendFile can be performed (Line 3b). Otherwise, the code records the upcoming occurrence of sendFile (Line 3d) before it calls sendFile. ◊

2.8. Service Automata

Service automata [GMS12] are a concept of a generic enforcement mechanism for security in distributed systems such as service-oriented architectures (hence the name service automata). The goal of the concept is to enable decentralized enforcement in a coordinated fashion. Coordination aims for the sound enforcement of security aspects that cannot be achieved locally in a distributed target. Decentralization aims to avoid bottlenecks and a reduction of communication overhead.

An individual *service automaton* constitutes a unit of a distributed enforcement mechanism. A particularity of a service automaton is that it can communicate with other service automata. This enables service automata to coordinate their enforcement of a security property on a distributed program.

A service automaton has a modular architecture comprising four components. Figure 2.1 shows these components. Each component takes over a particular task during the enforcement. At run-time, the *interceptor* continuously *intercepts* the next security-relevant action of the agent. That is, it anticipates the action before the action is performed and temporarily blocks the execution of the agent. The *local policy makes local decisions* about intercepted actions, i.e., determines whether the actions comply with the security property and determines suitable countermeasures, when possible. The *coordinator* uses the local policy and a network of other service automata to decide whether an intercepted action may be performed by the agent or, if not, to decide which countermeasure to impose on the agent. The *enforcer* implements these *decisions* and, if mandated by a decision, unblocks the execution of the target.

When the coordinator cannot make a decision about an intercepted action using the local policy, it can involve another service automaton in the decision-making. The coordinator then *delegates* the decision-making, i.e., transfers the responsibility to make a decision, for the intercepted action to the other service automaton by sending a *delegation request*. The coordinator can obtain the result of the delegation by receiving a *delegation response* from another service automaton. The coordinator can also assume the converse role, receiving delegation requests from other service automata and replying with a delegation response or further delegating the decision-making.

The generic aspect of the Service Automata concept is achieved by parametric components of the individual service automata: the local policy and the enforcer (the boxes with the dark gray background in Figure 2.1). These components can be *instantiated* by a concrete local policy and a concrete enforcer for enforcing a particular security property on a particular target.

Chapter

3

A Language for Specifying Modular Cooperative Policies

3.1. Introduction

We present *CoDSPL* (pronounced "code-spell" and abbreviating "Cooperative Distributed Security Policy Language"), a language for specifying policies for cooperative distributed enforcement mechanisms. The policy language allows policies to specify which units apply to which agents of a distributed program, which actions of the agents are security-relevant, and how countermeasures are decided, possibly by cooperation among units. In particular, CoDSPL allows the policies to specify for each unit of a distributed enforcement mechanism when and how this unit initiates cooperation and how it responds to other units' requests for cooperation. CoDSPL supports targets implemented in Java or in Ruby.

Specifying the cooperation in a security policy allows the designer of a policy to tailor cooperation to, e.g., the distributed program, the architecture of the underlying distributed system, and the security requirements. Beyond expressiveness regarding cooperation, CoDSPL is also expressive with regard to aspects that apply to non-distributed settings. Namely, CoDSPL allows wide ranges for the specification of which actions shall be intercepted, how decisions about intercepted actions are made, and what countermeasures shall be taken against security-violating actions.

Structure The remainder of this chapter is structured as follows. Section 3.2 introduces the reference architecture of enforcement mechanisms assumed by the policy language. In Section 3.3, we present the high-level structure of the policy language. Section 3.4 presents the syntax of our policy language, structured into the language's three sub-languages. Section 3.5 provides the semantics of CoDSPL, i.e., how the individual units specified by a CoDSPL policy behave at run-time. We summarize the presented contributions in Section 3.6.

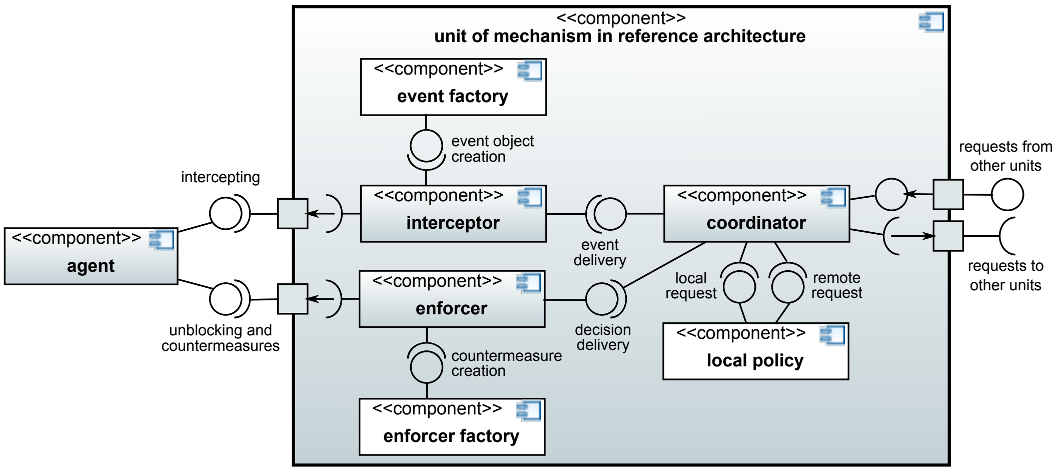


Figure 3.1.: Reference architecture of units for CoDSPL (UML component diagram)

3.2. Reference Architecture

The reference architecture for CoDSPL specifies basic architectural assumptions about enforcement mechanisms that shall enforce CoDSPL policies. Since CoDSPL targets enforcement mechanisms for distributed programs, the reference architecture consists of two architectural aspects. The reference architecture specifies the distribution of the mechanism in relationship to the agents of the distributed target. The reference architecture specifies the architecture of the individual mechanism's units, particularly with regard to the aspects that specified by CoDSPL policies.

The enforcement mechanism of the reference architecture is itself distributed. It consists of one unit for each agent of the distributed target that shall be subject to the enforcement. Moreover, each unit of the mechanism can be configured individually by the CoDSPL policy. That is, firstly, agents of the target that are irrelevant for security need not be subject to the enforcement mechanism and, hence, need not be subject to a CoDSPL policy. Secondly, for those agents that are subject to the mechanism, the mechanism allows the CoDSPL policy to tailor the units individually to these agents.

Figure 3.1 depicts the architecture of units in the reference architecture, modeled as a UML [BME$^+$07] component diagram. The figure displays the fixed and parametric components of the unit, the interfaces between the components, and the interfaces to the environment.

Components The reference architecture of units refines the architecture of the service automata concept (depicted in Figure 2.1 on page 24). It retains the interceptor, coordinator, local policy, and enforcer components and introduces two novel components: the *event factory* and the *enforcer factory*. The event factory abstracts from intercepted operations of an agent to objects capturing these operations at a level suitable for the local policy. In the following, we refer to these objects as *event objects*. The enforcer factory concretizes decisions at the level of the local policy to concrete countermeasures that can be performed on an agent. In the following, we refer to such decisions as *decision objects* and to the resulting concrete countermeasures as *countermeasure objects*.

A subset of the components in the reference architecture are parametric while the remaining components are fixed. In Figure 3.1, parametric components (event factory, local policy, and enforcer factory) are displayed with a bright background, and fixed components (interceptor, coordinator, and enforcer) are displayed with a gray background. That a component is parametric means that its functionality is fully or in part determined by the CoDSPL policy. Conversely, the functionality of a fixed component is not determined by the policy.

Note that the division between fixed and parametric components in the reference architecture deviates from the one of the service automata concept. While the local policy is parametric and the interceptor is fixed in both architectures, the enforcer component is parametric only in the service automata concept. In the reference architecture, the enforcer is reduced to the fixed task of implementing the countermeasure objects obtained from the parametric enforcer factory.

Interfaces The parametric components, i.e., those components whose functionality is determined by a CoDSPL policy, offer several interfaces to the fixed components. The local policy component provides a *local request* interface, for processing event objects, and a *remote request* interface, for processing delegation requests and delegation responses. The event factory provides an interface for the creation of event objects. The enforcer factory provides an interface for the creation of countermeasures from decision objects.

The interfaces between the fixed components are indirectly relevant for CoDSPL policies. Concretely, the interfaces between the interceptor, coordinator, and enforcer components establish that event objects and decision objects can be delivered to and, respectively, from the local policy. The external interfaces to the agent establish that security-relevant actions of the agent are intercepted and that countermeasures are applied. The interface to other units establishes that requests can be exchanged with other units.

Separation of concerns The reference architecture of units achieves separation of concerns in two regards. Firstly, the architecture separates operations of an agent from event objects as they are used by the local policy. Secondly, the architecture separates decision objects of the local policy from the countermeasures that are applied to the agent. The following example illustrates both.

Example 3.1. Consider Example 2.3 on page 23, in which an agent provides access to files as a service and the enforcement mechanism prevents conflicts of interest by checking the accesses of individual users. In the example, calls to the `sendFile` method are security-relevant operations, as they might allow a user to access conflicting files.

Of the method call, both the actual argument (`file`) and object (**`this`**) on which the method is called can be used to determine whether an access is complies with the security requirement or not. For the local policy of an enforcement mechanism, not the full `file` and **`this`** objects but rather only the names of the file and of the user stored in these objects suffice for checking compliance in a local policy. By extracting only the names of the file and the user, the event factory can separate agent-specific data structures from event objects.

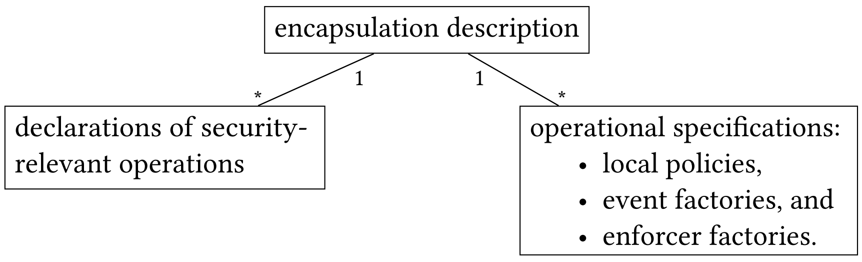

encapsulation description
1
1
*
*
declarations of security-relevant operations
operational specifications:
• local policies,
• event factories, and
• enforcer factories.

Figure 3.2.: Entities of a CoDSPL policy and their relationships

As a countermeasure against security-violating accesses, a call to method System.exit can be used. For the local policy of an enforcement mechanism, however, not the precise method call but rather the mere decision to call the method or not suffices. By generating the precise method call or its absence as a countermeasure object, the enforcer factory can separate the agent-specific countermeasure from decision objects of the local policy. ◊

3.3. Structure of CoDSPL policies

CoDSPL consists of three sub-languages:

1. the *encapsulation description*, a specification of which units to generate and of basic properties of these units and the agents they encapsulate;
2. the units' declarations of security-relevant operations; and
3. *operational specifications* of parametric unit components, i.e., of local policies, event factories, and enforcer factories.

The language uses three sub-languages for these three parts to cater for the different nature of the respective specified entities. Figure 3.2 shows these entities and their relationships.

A CoDSPL policy consists of a single encapsulation description, multiple declarations of security-relevant operations (one for each unit to generate), and multiple operational specifications (one local policy, one event factory, and one enforcer factory for each unit). The encapsulation description references the locations of the remaining constituents of a CoDSPL policy.

3.4. Syntax of CoDSPL

We introduce the three sub-languages of CoDSPL separately in Sections 3.4.1 to 3.4.3. In the presentation of the syntax, we abstract away some technical details to keep the presentation focused on the essential aspects of the language. Concretely, we omit the treatment of line breaks, whitespace, and escape characters.

3.4.1. Syntax of Declarations of Security-Relevant Operations

CoDSPL's syntax for declaring security-relevant operations borrows from aspect-oriented programming, where expressive languages for specifying program operations are used for identifying join points at which the functionality of aspects shall be added. For a Java

```
PointcutSpec = { ( PointcutDecl | Import ) ‘;’ }.
Import       = ‘import’ JavaFQN.
PointcutDecl = ‘pointcut’ Id Formals ‘:’ PointcutExpr.
Formals      = ‘(’ [ ParamList ] [ ( ‘>’ JavaFQN ) ] ‘)’.
ParamList    = JavaFQN Id { ‘,’ JavaFQN Id }.
PointcutExpr = PointcutPrim
               | ‘!’ PointcutExpr | ‘(’ PointcutExpr ‘)’
               | PointcutExpr ‘&&’ PointcutExpr
               | PointcutExpr ‘||’ PointcutExpr.
PointcutPrim = Id ‘(’ JavaExpression ‘)’.
JavaFQN      = Id { ‘.’{ ‘.’ } Id }.
Id           = [‘a’-‘z’,‘A’-‘Z’] { [‘a’-‘z’,‘A’-‘Z’,‘0’-‘9’] }.
```

Figure 3.3.: Syntax for declaring security-relevant operations in BNF

agent, the CoDSPL syntax is derived from the language of AspectJ for specifying pointcuts [Tea05]. For a Ruby agent, the syntax is derived from the language of *Aquarium* [Aqu], a tool for aspect-oriented programming with Ruby. Both languages already provides rich means for selecting method calls and binding values of actual arguments passed to these methods. In the following, we present CoDSPL's syntax for Java agents.

Figure 3.3 shows the syntax of CoDSPL's sub-language for declaring security-relevant operations of agents. The syntax is defined by the *PointcutSpec* non-terminal. The words of the language are semicolon-separated sequences of *pointcut declarations*, captured by the *PointcutDecl* non-terminal, and *import declarations*, captured by the *Import* non-terminal. Import declarations, as in regular Java code, serve the purpose of making name-spaces available for use in the pointcut declarations. In the following, we focus on the pointcut declarations, which can be used to specify a set of security-relevant program operations.

A pointcut declaration is indicated by the ‘`pointcut`’ keyword and consists of a pointcut identifier, a list of formal parameters, a pointcut expression, and a list of pointcut modifiers. The *pointcut identifier* is a unique name of the pointcut. The formal parameters list names and types of parameters of the pointcut as well as the return type of the pointcut. The purpose of the parameter names is to make actual parameters of a program operation captured by a pointcut accessible. The *pointcut expression* is either a pointcut primitive or a possibly grouped, logical combination of pointcut expressions (including negation ‘!’, conjunction ‘&&’, and disjunction ‘||’). The *pointcut primitive* is a combination of an identifier and an expression. The identifier specifies the type of pointcut and the expression specifies the program operations matched by the pointcut primitive.

The syntax used by CoDSPL deviates from AspectJ's pointcut syntax as follows. Firstly, CoDSPL's syntax adds the possibility to specify a return type of methods, in the form of the *JavaFQN* in the *Formals* non-terminal. This return type is added for capturing what type of return values may be substituted by a countermeasure. Secondly, CoDSPL's syntax over-approximates the pointcut primitives in that, firstly, an arbitrary identifier rather than one of AspectJ's 23 keywords are permitted and, secondly, that the subsequent expression is required only to be well-bracketed (when strings enclosed in double quotes

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.charset.Charset;

pointcut fileRead(Path path):
  (call(* Files.readAllBytes(Path)) && args(path))
  (call(* Files.readAllLines(Path, Charset)) && args(path));
```

Listing 3.1.: Example declaration of security-relevant file-reading operations

```
pointcut printSens(Document doc):
  (call(void Document.printSensitive()) && target(doc));
```

Listing 3.2.: Example declaration of security-relevant printing operations

are factored out) rather than an expression from a language determined by the respective keyword. This simplifies the syntax and allows an implementation of CoDSPL to pass pointcut primitives directly to the AspectJ backend. Finally, CoDSPL's syntax leaves out so-called pointcut modifiers, which declare the visibility of pointcuts in AspectJ and, as such, are not relevant for merely declaring security-relevant operations.

CoDSPL's sub-language for declaring security-relevant operations aims at being used for capturing method calls. For this purpose, each pointcut primitive must have '`call`', '`execute`', '`target`', '`this`', or '`args`' is its identifier. The first two identifiers declare a method call or, respectively execution as the operation to be security-relevant. The latter bind the object on which the method is called, the current **this**-object, and the actual arguments of the method call to the formal parameters of the security-relevant operation.

Example 3.2. Consider a security requirement that specifies that a program shall only read from files located in the home-directory of the user on whose behalf the program is run. Consider an agent whose code reads the content of files via the two Java API methods Files.readAllBytes and Files.readAllLines. For specifying that calls to these two methods are security-relevant, we use the declaration in Listing 3.1. The fileRead pointcut matches all operations that call (specified by '`call`') a method Files.readAllBytes or a method Files.readAllLines: the first with a single formal parameters of type Path, the second with formal parameters of types Path and Charset, and both with neglected return type (specified by '$*$'). The '`import`' declarations allow the short form of types Files, Path, and Charset to be used in the pointcut declaration. The '`args`' pointcut primitive binds the first argument of the called methods to the parameter path of the pointcut. The return values and the second argument to Files.readAllLines are not bound to parameters of the pointcut. ◊

Example 3.3. Consider a security requirement demanding that sensitive documents may be printed at most 5 times, and consider a program in which the printing of a sensitive document is performed by the custom method printSensitive of class Document declared as "**void** printSensitive();", i.e., without formal parameters and return type. For specifying that calls to the method are security-relevant, we use the declaration in Listing 3.2. The printSens pointcut matches all operations that call the method printSensitive of class Document. The pointcut has a formal parameter doc of class Document, which is bound to the object on

```
print_sens = Aquarium::Aspects::Pointcut.new :calls_to => :print_sensitive,
  :for_types => [Document]
```

Listing 3.3.: Example declaration of security-relevant Ruby printing operations

which the method printSensitive is called (specified by the term target(doc)). ◊

Like the syntax of declarations of security-relevant operations for Java agents is derived from the AspectJ language, the syntax for Ruby agents is derived from the language of the Aquarium [Aqu], an AOP tool for Ruby. Conceptually, the relationships between CoDSPL and AspectJ and, respectively, Aquarium are almost the same: CoDSPL uses the syntax of pointcut declarations but does not include syntax for specifying advice and aspects. While CoDSPL's syntax for Java makes small modifications to AspectJ's pointcut syntax (recall: adding a return type and leaving out pointcut modifiers), CoDSPL's syntax for Ruby does not make such modifications. We therefore refrain from introducing the syntax here and instead refer to the resources on Aquarium and illustrate the language by the following example as the Ruby counterpart to Example 3.3.

Example 3.4. Consider the security requirement and program of Example 3.3, except that the program is implemented in Ruby. For specifying that calls to the print_sensitive method[1] are security-relevant, we use the declaration in Listing 3.3. The declaration defines the print_sens pointcut to match calls to the printSensitive method of the Document type in a self-explanatory fashion. A difference to the Java counterpart in Listing 3.2 is that the Document object on which the method is invoked is not explicitly bound to a parameter. This difference takes into account that Aquarium automatically makes all actual arguments and the target object available to advice. ◊

Overall, CoDSPL's sub-language supports the declaration of a set of method calls as security-relevant operations. The operations can be named and have a list of formal parameters such that actual arguments to method calls can be captured at run-time. Methods can be matched by their name, the class they belong to, and their formal parameters list, and multiple methods can be combined to a single pointcut declaration. This provides a rich means for identifying and grouping security-relevant operations. The restriction to the granularity of method calls limits which operations can be declared security-relevant. For instance, accesses to variables cannot be declared security-relevant. However, in Java calls to methods must be made whenever system resources (e.g., files or network sockets) are accessed. Moreover, method calls are used for application-specific abstractions such as exemplified by the methods sendFile and printSensitive in Examples 2.3, 3.3 and 3.4. Hence, for agents by benign developers, we expect the granularity of method calls to suffice for capturing security-relevant operations in a wide range of scenarios.

3.4.2. Syntax of Operational Specifications

Operational specifications in CoDSPL policies serve the specification of the parametric components of the units, i.e., the local policies, event factories, and enforcer factories in the

[1]In Ruby-related code, we follow the Ruby naming convention for variables, fields, and methods using underscores rather than Java's camel case.

```
interface Event extends Serializable {}
interface Decision extends Serializable,LocalPolicyResponse {}
interface DelegationReqResp extends Serializable {}

class DelegationLocPolReturn implements LocalPolicyResponse {
  private String destinationId;
  private DelegationReqResp delReqResp; }
interface LocalPolicyResponse {}

interface EventFactory {
  /* one method per pointcut of the EC, of the form
      public Event <pointcutId>(<formals>); */}
interface EnforcerFactory {
  Countermeasure fromDecision(Decision dec); }
interface Countermeasure {
  void before();
  void after();
  boolean suppress(); }
abstract class LocalPolicy { /* fields, getters, setters, and constructor omitted */
  abstract LocalPolicyResponse localRequest(Event ev);
  abstract LocalPolicyResponse remoteRequest(DelegationReqResp dr); }
```

Listing 3.4.: Java base classes and interfaces for operational specifications

architecture proposed in Section 3.2 on page 28. The syntax for operational specifications in CoDSPL aims at allowing a wide range of algorithmic specifications.

Specifications of local policies are always in Java whereas the event factory and enforcer factory are specified in the language of the respective agent, i.e., Java or Ruby. We first present CoDSPL's syntax for Java agents. Operational specifications for Java agents assume the form of JAR files, i.e., compressed files containing compiled Java class files. CoDSPL places no further constraints about the content of the JAR files, including the way the Java code that constitutes the operational specifications is structured. However, operational specifications are required to provide implementations of classes that implement the interfaces EventFactory and EnforcerFactory or extend the abstract class LocalPolicy, respectively. Listing 3.4 shows how these interfaces and abstract class are defined in CoDSPL.

Local policies A CoDSPL policy must specify a subclass of the abstract LocalPolicy class, thereby providing implementations of the methods localRequest and remoteRequest. The former is expected to take an event object, of type Event, and return an object of type LocalPolicyResponse. The latter is expected to take a delegation request or delegation response, of type DelegationReqResp, and return a LocalPolicyResponse object. An object of type LocalPolicyResponse is either a decision object (if implementing the Decision interface) or a unit identifier and a delegation request or delegation response (encapsulated in a DelegationLocPolReturn object). By extending the LocalPolicy class and, thus, implementing the methods localRequest and remoteRequest, the operational specification of a local policy provides the two interfaces to the coordinator of a unit (see Figure 3.1 on page 28).

Example 3.5. Consider the security requirement and program of Example 3.3, i.e., that

```
class PrintEvent implements Event {
  public String docId; }
class PrintDecision implements Decision,DelegationReqResp {
  public boolean permit; }
class PrintDelegation implements DelegationReqResp {
  public PrintEvent event;
  public String sourceUnit; }

class CentralPrintPolicy extends LocalPolicy {
  private Map<String, int> counters;
  private PrintDecision decide(PrintEvent event) {
    int current = counters.getOrDefault(event.docId, 0);
    if (current < 5) counters.put(event.docId, current+1);
    return new PrintDecision(current < 5); }
  public LocalPolicyResponse localRequest(Event event) {
    return decide((PrintEvent)event); }
  public LocalPolicyResponse remoteRequest(DelegationReqResp delReqResp) {
    PrintDelegation del = (PrintDelegation)delReqResp;
    return new DelegationLocPolReturn(del.sourceUnit, decide(del.event)); }
}
class ClientPrintPolicy extends LocalPolicy {
  public LocalPolicyResponse localRequest(Event event) {
    return new DelegationLocPolReturn("central",
        new PrintDelegation((PrintEvent)event, getIdentifier())); }
  public LocalPolicyResponse remoteRequest(DelegationReqResp delReqResp) {
    return (PrintDecision)delReqResp; }
}
```

Listing 3.5.: Example local policy for limiting printing of documents

sensitive documents may be printed at most 5 times by the target. We specify centralized local policy components, with one central unit that the makes decisions and all other units delegating decision-making to the central unit. Listing 3.5 shows the definition of two local policy classes for limiting the printing of each individual sensitive document to at most 5 times, and shows the underlying definitions of event and decision objects. For readability, the listing does not include **import** statements and trivial constructors.

The PrintEvent class comprises the docId field for storing unique document identifiers in event objects. The PrintDecision class for decision objects comprises the Boolean permit field for storing whether printing of the respective document shall be granted (value **true**) or not (value **false**). The PrintDelegation class for delegation requests comprises fields for storing the delegated event (event) and for the identifier of the unit at which the event originated (sourceUnit).

The class CentralPrintPolicy is the local policy of the central unit. It defines a field counters as a map from document identifiers to print counts for storing how often individual documents were printed. It further defines a method decide that takes an event object and uses the counters field for checking and updating how often the document referenced by the event object (event.docId) has been permitted to be printed. The method returns the decision to permit the printing if and only if the document has been printed 5 times or

```
class PrintEventFactory implements EventFactory {
  public static Event printSens(Document doc) {
    return new PrintEvent(doc.getID());
}}
```

Listing 3.6.: Example event factory for limiting printing of documents

less so far. The localRequest method of the central unit's local policy simply returns the decision provided by the decide method. The remoteRequest method takes a delegation request from another unit and returns the decision provided by the decide method to the source of the delegation request.

The local policy class ClientPrintPolicy implements the local policy of a non-central unit. Its localRequest method delegates all events to the unit with identifier "central", and its remoteRequest method can directly return a received delegation object, because for a non-central unit this is always a delegation response. ◊

The example particularly illustrates two aspects about local policies of CoDSPL. Firstly, the example shows that local policies can employ delegation without using network sockets. Rather, local policies can delegate by returning objects of the DelegationLocPolReturn type, which specify the destination unit. This reduces the complexity of local policy implementations by network connection management and the serialization and deserialization of transferred data structures. Secondly, the example indicates how local policy implementations can make use of the expressiveness of Java: Methods like decide in class CentralPrintPolicy can use data structures of the Java API such as maps and Java control structures like branching, loops, and recursion.

Event factory and enforcer factory CoDSPL imposes two constraints on operational specifications for event factories and enforcer factories. Firstly, an event factory must implement the EventFactory interface and must provide one method for each pointcut declaration in the declaration of security-relevant operations for the same unit. This method must share the name and the formal parameter list of the respective pointcut declaration. Note that these methods together provide the interface for event object creation to the interceptor component of the unit's architecture (see Figure 3.1 on page 28). Secondly, an enforcer factory must implement the EnforcerFactory interface and, by definition of the latter, must implement a fromDecision method. This method takes a decision object and returns a Countermeasure object, which must implement the before, after, and suppress methods for specifying what code shall be inserted before and, respectively, after performing the intercepted operation and for specifying whether the operation shall be suppressed. As such, the fromDecision method providing the interface for countermeasure creation to the enforcer component of the unit's architecture (see Figure 3.1 on page 28).

Example 3.6. We resume Example 3.5 by providing an event factory for the scenario. Listing 3.6 shows the implementation of the event factory by the PrintEventFactory class. The class implements the EventFactory interface and defines one method, printSens. The definition of this method matches the security-relevant operation declared in Example 3.3 on page 32, whose name and formal parameter list it shares. The method simply returns

```
class PrintEnforcer implements Countermeasure {
  public boolean permit;
  public void before() {
    if (!permit) {
      JOptionPane.showMessageDialog(null,
        "Printing prohibited (limit reached).", JOptionPane.ERROR_MESSAGE);
  }}
  public void after() { /* nothing */}
  public boolean suppress() { return !permit; }
}
class PrintEnforcerFactory implements EnforcerFactory {
  public Countermeasure fromDecision(Decision dec) {
    return new PrintEnforcer(dec.permit); }
}
```

Listing 3.7.: Example enforcer factory for limiting printing of documents

an event object of type PrintEvent whose stored document identifier is initialized to the identifier of the document object in the actual argument doc. ◊

Example 3.7. We continue Example 3.5 by providing an enforcer factory. Listing 3.7 shows the implementation of the enforcer factory by the PrintEnforcerFactory class together with an implementation of a class for countermeasure objects by PrintEnforcer. The PrintEnforcerFactory class implements the EnforcerFactory interface by defining the method fromDecision. Independently of the concrete Decision object passed to this method, the method returns an object of type PrintEnforcer. In the construction of the object, the method uses the Boolean value permit stored in the decision object. The PrintEnforcer class implements the three methods of the Countermeasure interface. Through its implementation of the suppress method, objects of the class suppress an intercepted event if and only if the Boolean field permit stores value **false**. Before an event is suppressed by the enforcer object, the implementation of method before ensures that a message dialog signals the security violation to the user. ◊

For agents implemented in Ruby, the event factory and enforcer factory are specified by Ruby code; the local policy component for Ruby agents is specified in Java. Event factories and enforcer factories for Ruby agents, however, share the same low-level architecture as for Java agents. Besides the different programming language, two technical differences to the Java code exist. Firstly, a method in an event factory class that matches the name of a security-relevant operation has a formal parameter list that begins with the object on which the intercepted method was invoked and that is followed by the list of the intercepted method's arguments. Secondly, when the definitions of event classes used by the local policy reside inside a Java package, then the event factory in Ruby must provide a method java_package that returns the name of this package. This allows a connection between the Ruby event objects and the Java event objects to be established unambiguously. We show how a concrete event factory and enforcer factory can be specified in an example.

Example 3.8. Again we resume Example 3.5 by providing an event factory and an enforcer factory for the scenario, but here for a Ruby agent. Listing 3.8 on the following page

```
class PrintEvent
  attr_accessor :doc_id
end
class PrintDecision
  attr_accessor :permit
end
class PrintEnforcer
  attr_accessor :permit
  def before
    if !@permit puts "Printing prohibited (limit reached)." end
  end
  def after end
  def suppress return !@permit end
end
class PrintEventFactory
  def self.print_sens(doc) return PrintEvent.new(doc.get_id) end
end
class PrintEnforcerFactory
  def self.from_decision(dec) return PrintEnforcer.new(dec.permit) end
end
```

Listing 3.8.: Example event factory and enforcer factory for Ruby agents

shows the implementation of both components as well as of underlying classes. As in previous listings, we omit constructors of classes in the listing. The defined classes directly correspond to the Java classes of the same name in Listings 3.5 to 3.7. We therefore abstain from a description here. The only difference in the code is that the PrintEnforcer prints an error message on the console rather than displaying a dialog in the Java code. ◊

CoDSPL's syntax for operational specifications of local policies, event factories, and enforcer factories inherits the expressiveness of the general purpose languages Java and Ruby. This allows the specifications of these components to express a variety of possible abstractions for events and decisions, to express complex algorithms for decision-making and recording state, and to express when to cooperate and with which other units.

3.4.3. Syntax of Encapsulation Descriptions

Encapsulation descriptions are specified in a simple syntax. They capture which units exist and basic properties of these units. They do not contain the technically more complex entities of CoDSPL, namely the security-relevant operations as well as the parametric unit components. To these more complex entities, encapsulation descriptions contain pointers.

Syntactically, the language for encapsulation descriptions is a sub-language of the language used by Java for representing Properties objects [Ora]. The simplified syntax for encapsulation descriptions is captured by the BNF in Figure 3.4 on the next page. This syntax abstracts from details such as character escaping, line breaks, and comments. For the technical details, the reader is referred to the Java documentation [Ora].

The syntax for encapsulation descriptions is defined by the *policy* non-terminal. The words of the language are nonempty sequences of *keyvalue* elements. Each *keyvalue*

```
policy    = keyvalue | policy '↵' keyvalue.
keyvalue  = 'cfg.units' '=' IdList
            | 'cfg.destdir' '=' FileName
            | 'cfg.crypto' '=' ('true' | 'false')
            | Id '.policy.'Id '=' value
            | Id '.type' '=' ('Java' | 'Ruby')
            | Id '.'addrkey '=' DomainOrIP
            | Id '.'portkey '=' PortNumber
            | Id '.'classkey '=' JavaFQN
            | Id '.'filekey '=' FileName
            | Id '.'cpkey '=' FileName { ':' FileName }
            | Id '.'enckey '=' ('serialization' | 'JSON')
            | Id '.loglevel' '=' ('WARNING' | 'INFO').
addrkey   = 'ext-host' | 'cor-host' | 'enf-host'.
portkey   = 'ext-port' | 'cor-port' | 'enf-port'.
classkey  = 'event-factory' | 'enforcer-factory' | 'policy'.
filekey   = 'target' | 'target-dir' | 'inline-dir' | 'pointcuts' |
            'target-javavm'.
enckey    = 'cooperation-encoding' | 'local-encoding'.
cpkey     = 'inline-classpath' | 'policy-classpath'.
```

Figure 3.4.: Syntax of encapsulation descriptions in BNF

element is of the general form *key* '='*value*, but only specific *key* instances are allowed in the language. We discuss the allowed *key* and *value* combinations as well as the purpose of the *keys* in the following.

The keys in the language can be distinguished into three classes. The first class contains the keys with prefix '`cfg.`'. Those keys apply not to a particular unit but to the whole distributed enforcement mechanism. The second class contains the keys with prefix *Id* and suffix *Id*. These keys specify a named value for a local policy component, where the suffix *Id* identifies the name and the prefix *Id* identifies the unit for whose local policy the named value is specified. The third class contains all the remaining keys, which all have a prefix *Id*. These keys specify settings for the unit identified by the *Id*. In the following, we refer to such identifiers of units as *unit identifiers* or just identifiers. Instead of a unit identifier, the word '`unit`' can be used in the language for specifying that the value shall apply to each unit.

Intuitions for all the keys in the language for encapsulation descriptions are provided in Table 3.1 on the following page. In the table, those keys that only apply to agents that are implemented in Java are marked with a "J" subscript, and those keys that only apply to Ruby agents are marked with an "R" subscript.

Example 3.9. We resume and complete Examples 3.3 to 3.8 by an encapsulation description. The distributed program that offers printing of sensitive documents now consists of three agents: two Java agents and one Ruby agent. Listing 3.9 on page 41 shows the encapsulation description for the scenario. The first line of the encapsulation description

‘`cfg.units`’	This key must be set to a comma-separated list of unit identifiers (from *IdList*). This list specifies unique names for the units specified by the policy.
‘`cfg.destdir`’	This key specifies the name of a directory into which the result of applying the policy to a given target shall be stored.
‘`cfg.crypto`’	This key specifies whether encryption of the communication between units shall be used or not.
Id ‘.’ ...	For the unit identified by *Id*, this key specifies...
‘`policy.`’*Id*	...a named value to be passed to the unit’s local policy.
‘`type`’	...the type of the unit’s agent, which is the programming language that the target is implemented in.
‘`ext−host`’, ‘`ext−port`’, ‘`cor−host`’, ‘`cor−port`’, ‘`enf−host`’, ‘`enf−port`’	...the host and port through which (a) the unit can be reached by other units, (b) the coordinator can be reached within the unit, and, respectively, (c) the enforcer can be reached within the unit.
‘`policy`’, ‘`event−factory`’, ‘`enforcer−factory`’	...the fully-qualified class names of the local policy, the event factory, and the enforcer factory, respectively.
‘`target`’$_{\mathcal{J}}$	...the path to the JAR file that holds the Java bytecode of the target agent’s implementation.
‘`target−dir`’$_{R}$	...the path to the directory in which the code of the agent resides.
‘`target−javavm`’$_{\mathcal{J}}$	...the path to the Java virtual machine executable on the computer on which the agent executes.
‘`pointcuts`’	...the path to the file containing the declaration of security-relevant operations of the agent.
‘`cooperation−encoding`’, ‘`local−encoding`’	...the data encoding to be used at run-time in the communication between units and, respectively, within units.
‘`policy−classpath`’	...the colon-separated list of paths to all JAR files in which the implementation of the local policy of the unit together with its dependencies resides.
‘`inline−classpath`’$_{\mathcal{J}}$	...the colon-separated list of paths to all JAR files in which the implementation of the event factory and enforcer factory together with their dependencies reside.
‘`inline−dir`’$_{R}$	...path to the directory that contains the implementation of the event factory and enforcer factory and their dependencies.
‘`loglevel`’	...the log level to use when encapsulating the agent.

Table 3.1.: Intuitions of the keys in encapsulation descriptions

```
cfg.units = central, java—cli, ruby—cli
cfg.destdir = ./instrumented/
cfg.crypto = true

central.type = Java
central.target = PrintService.jar
central.ext—host = central.example.com
central.policy = CentralPrintPolicy
central.pointcuts = JavaPrint.pc
central.inline—classpath = JavaFactories.jar

java—cli.type = Java
java—cli.target = PrintService.jar
java—cli.ext—host = java—cli.example.com
java—cli.pointcuts = JavaPrint.pc
java—cli.inline—classpath = JavaFactories.jar

ruby—cli.type = Ruby
ruby—cli.target—dir = ./PrintService/
ruby—cli.ext—host = ruby—cli.example.com
ruby—cli.pointcuts = RubyPrint.pc
ruby—cli.inline—dir = RubyFactories/

unit.policy = ClientPrintPolicy
unit.policy—classpath = PrintPolicy.jar
unit.event—factory = PrintEventFactory
unit.enforcer—factory = PrintEnforcerFactory
unit.local—encoding = JSON
unit.cooperation—encoding = serialization
unit.loglevel = WARNING
unit.target—javavm = /usr/bin/java
```

Listing 3.9.: Example encapsulation description for limiting printing of documents

declares three units, named “central”, “java-cli”, and “ruby-cli”. The remaining two lines of the first block specify that the result of applying the policy shall be written to the sub-directory `instrumented` and that encryption for the communication between units shall be used. The subsequent four blocks of lines in the encapsulation description specify properties of the central unit, the two client units, and properties applying to all three units unless specified otherwise.

The units named “central” and “java-cli” encapsulate Java agents, whose implementation is in JAR file `PrintService.jar`. The unit named “ruby-cli” encapsulates a Ruby agent, whose implementation is stored in directory `./PrintService/` and its sub-directories. The specified values for the units reference several files and directories that contain the code or, respectively, compiled code of listings shown before: `JavaPrint.pc` (Example 3.3), `RubyPrint.pc` (Example 3.4), `PrintPolicy.jar` (Example 3.5), `JavaFactories.jar` (Examples 3.6 and 3.7), `RubyFactories/` (Example 3.8). The ‘`unit.policy`’ key specifies the default local policy class of units to be ClientPrintPolicy, and the ‘`central.policy`’ key overrides this default for the “central” unit by the CentralPrintPolicy. Furthermore, the encapsulation description specifies network addresses in the domain “example.com” for the computers at which the agents and the units are run. Port numbers are not specified such that default ports can be used when interpreting the encapsulation description. For the communication between the units, serialization is specified as the encoding via key ‘`unit.cooperation−encoding`’, whereas the communication within units is specified to use JSON as an encoding that is understood by the Ruby components and the Java components. Finally, the encapsulation description specifies the log level of all units show warnings and more severe problems and the Java virtual machine to be located in file `/usr/bin/java` on the agents’ computers. ◊

Overall, an encapsulation description specifies which units constitute the specified distributed enforcement mechanism and to which agents the individual units apply. Moreover, it specifies basic properties such as hosts and ports of the individual units. Finally, it specifies pointers to separate files containing specifications in the other two sub-languages of CoDSPL: declarations of security-relevant operations (via ‘`pointcuts`’ keys) and operational specifications of the units’ parametric components (via the keys specified by the *cpkey* non-terminal). In recognition that encapsulation descriptions are the central entities of CoDSPL and contain references to files containing the remaining entities, we define CoDSPL policies as follows.

Definition 3.1. A *CoDSPL policy* is a tuple (ed, files) where $ed \in \mathcal{L}(\mathit{policy})$ is an encapsulation description, $\mathit{files} : \mathcal{L}(\mathit{FileName}) \rightharpoonup \mathit{FileContent}$ is a partial function mapping names of files and directories to their content, and *FileContent* models the set of all possible file and directory contents, i.e., all possible finite sequences of byte values. We call *files* the *file map* of the CoDSPL policy. ◊

The first element of a CoDSPL policy tuple is the encapsulation description. The second element – the file map – captures that further named files can belong to a CoDSPL policy such that the declarations of security-relevant operations and the operational specifications of parametric unit components can be contained in separate files. These separate files must obey the syntax of the corresponding sub-languages of CoDSPL. We introduce the syntax of these sub-languages in the following sections. In the remainder of this thesis, we use the symbol *pol* to range over CoDSPL policies.

3.5. Semantics of CoDSPL

We define the semantics of CoDSPL in a semi-formal fashion. In the description of the semantics, we first provide a semantics for the key-value syntax of the encapsulation descriptions in CoDSPL policies. Subsequently, we cover how a unit is initialized based on a given operational specification and address how operational specifications determine a unit's processing of an individual intercepted operation and an individual delegation request. Finally, we specify how a sequence of operations and delegation requests is processed. The exposition is based on established semantics of incorporated sub-languages of Java and AspectJ where possible. We introduce the semantics of these sub-languages here only to the extent necessary for a self-contained presentation and refer to the literature on Java [GJS+14; LYB+14] and on AspectJ [KHH+01; WKD04] for a detailed exposition.

Key-value semantics of encapsulation descriptions We model the semantics of encapsulation descriptions such that it captures three main aspects. The first aspect is that encapsulation descriptions specify global as well as unit-local properties and, in particular, specify which units exist. The second aspect is that encapsulation descriptions shall use the same semantics as when they are parsed by Java Properties objects, which means that, for global as well as for local properties, if multiple values are specified for a property, the last specification overrides the preceding ones. The third aspect is that properties specified for a particular unit override properties specified for all units (via the '`unit`' prefix of keys instead of a unit identifier).

Definition 3.2. The semantics of an encapsulation description $ed \in \mathcal{L}(policy)$, is the triple $[\![ed]\!] = (glob, Ids, loc)$ where

$$\begin{aligned}
&glob = kvmap(ed, \text{‘}\texttt{cfg}\text{’}),\\
&Ids = \begin{cases} idsl(glob(\text{‘}\texttt{units}\text{’})) & \text{if ‘}\texttt{units}\text{’} \in dom(glob),\\ \emptyset & \text{otherwise}\end{cases}\\
&loc : \ Ids \to \mathcal{W}_{\{\text{‘}=\text{’}\}} \rightharpoonup \mathcal{W}, \quad \text{with } loc(id) = kvmap(ed, \text{‘}\texttt{unit}\text{’}) \oplus kvmap(ed, id), \qquad (3.1)
\end{aligned}$$

and

- $kvmap : \ \mathcal{L}(policy) \to \mathcal{W}_{\{\text{‘}.\text{’},\text{‘}=\text{’}\}} \to \mathcal{W}_{\{\text{‘}=\text{’}\}} \rightharpoonup \mathcal{W}$ such that

$$kvmap(kv, w) = \{(k, v) \in \mathcal{W}_{\{\text{‘}=\text{’}\}} \times \mathcal{W} \mid kv = (w \ \text{‘}.\text{’} \ k \ \text{‘}=\text{’} \ v)\}$$
$$kvmap(ed' \ \text{‘}\hookleftarrow\text{’} \ kv, w) = kvmap(ed', w) \oplus kvmap(kv, w) \qquad (3.2)$$

  for each $ed' \in \mathcal{L}(policy)$, $kv \in \mathcal{L}(keyvalue)$, and $w \in \mathcal{W}_{\{\text{‘}.\text{’},\text{‘}=\text{’}\}}$;

- $idsl : \ \mathcal{L}(IdList) \to \mathcal{P}(\mathcal{L}(Id))$ such that

$$idsl(id) = \{id\}$$
$$idsl(idl \ \ id) = idsl(idl) \cup idsl(id)$$

for each $id \in \mathcal{L}(\mathit{Id})$ and $idl \in \mathcal{L}(\mathit{IdList})$;

We call the first component of the semantics the *global configuration*, the second component the *identifier set*, and the third component the *local configuration*. ♢

The global configuration models all parts of the encapsulation description that apply to the whole distributed enforcement mechanism rather than to individual units. The identifier set models the unit identifier specified by the encapsulation description, agnostic of the ordering in which the names are provided. The local configuration models all parts of the encapsulation description that apply to individual units. Inside the definition, $\mathit{kvmap}(ed, w)$ models the mapping between suffixes of keys whose prefix is w and values in encapsulation description ed. Values $\mathit{idsl}(idl)$ model the set of all unit identifiers captured by the word $idl \in \mathcal{L}(\mathit{IdList})$.

The three components of the semantics – global configuration, identifier set, and local configuration – faithfully capture the three kinds of properties that can be part of an encapsulation description. Global properties of an encapsulation description are contained in keys that are prefixed with '`cfg.`', which is captured by the second parameter in $\mathit{kvmap}(ed, \text{'}\texttt{cfg}\text{'})$. Which units exist is specified via the '`cfg.units`' key of an encapsulation description and is faithfully captured by $\mathit{glob}(\text{'}\texttt{units}\text{'})$. Unit-local properties are specified via keys that are prefixed with the respective unit identifier or via keys prefixed by '`unit.`', where the former override the latter. This overriding is faithfully captured by Equation (3.1) on page 43. That the last specification of a value for a key overrides all previous specifications for the key is faithfully captured by Equation (3.2) on page 43 in the definition of *kvmap*, which is used for global as well as unit-local properties. That is, overall our semantics, as defined in Definition 3.2 on page 43 faithfully captures the three main aspects we aimed for.

Based on the semantics of encapsulation descriptions, we now have the ingredients to define when a CoDSPL policy is well-formed. Intuitively, a CoDSPL policy $pol = (ed, \mathit{files})$ is well-formed if values are defined for all keys and all references in the encapsulation description ed to other files are referring to files whose content is from the right language. Concretely, referenced pointcut files must have content from the pointcut language, referenced JAR files and executable files must obey the JAR file format. We formally capture this as follows.

Definition 3.3. A CoDSPL policy (ed, files) is *well-formed* if, for $(\mathit{glob}, \mathit{Ids}, \mathit{loc}) = [\![ed]\!]$ and for each $id \in \mathit{Ids}$, the following conditions hold:

- for each key $w \in \{\text{'}\texttt{units}\text{'}, \text{'}\texttt{destdir}\text{'}, \text{'}\texttt{crypto}\text{'}\}$, it holds that $w \in \mathit{dom}(\mathit{glob})$;
- for each key $w \in \mathcal{L}(\mathit{addrkey}) \cup \mathcal{L}(\mathit{portkey}) \cup \mathcal{L}(\mathit{classkey}) \cup \mathcal{L}(\mathit{filekey}) \cup \mathcal{L}(\mathit{enckey}) \cup \mathcal{L}(\mathit{cpkey})$, it holds that $(id, w) \in \mathit{dom}(\mathit{loc})$; in the following, particularly let $\mathit{type} = \mathit{loc}(id, \text{'}\texttt{type}\text{'})$;
- $\mathit{loc}(id, \text{'}\texttt{pointcuts}\text{'}) \in \mathit{dom}(\mathit{files})$ and $\mathit{files}(\mathit{loc}(id, \text{'}\texttt{pointcuts}\text{'})) \in \mathcal{L}(\mathit{PointcutSpec})$, where no two contained pointcuts may share a join point shadow;
- if $\mathit{type} = \text{'}\texttt{Java}\text{'}$, then $\mathit{loc}(id, \text{'}\texttt{target}\text{'}) \in \mathit{dom}(\mathit{files})$ and $\mathit{files}(\mathit{loc}(id, \text{'}\texttt{target}\text{'}))$ is in the JAR file format; if $\mathit{type} = \text{'}\texttt{Ruby}\text{'}$, then $\mathit{loc}(id, \text{'}\texttt{target-dir}\text{'}) \in \mathit{dom}(\mathit{files})$ and $\mathit{files}(\mathit{loc}(id, \text{'}\texttt{target-dir}\text{'}))$ is a directory;
- for each $w \in \mathcal{L}(\mathit{cpkey})$, if $(id, w) \in \mathit{dom}(\mathit{loc})$ holds, then for each cp contained in the colon-separated list $\mathit{loc}(id, w) \in \mathit{dom}(\mathit{files})$, $\mathit{files}(\mathit{loc}(id, \text{'}\texttt{target}\text{'}))$ is in the JAR

file format.

In the following, we denote by $\mathit{POL} \subseteq \mathcal{L}(\mathit{policy}) \times (\mathcal{L}(\mathit{FileName}) \rightharpoonup \mathit{FileContent})$ the set of all well-formed CoDSPL policies. $\Diamond$

The definition faithfully captures the intention of defining all keys (global as well as local keys) and of constraining the content of referenced files. Firstly, the definition includes a constraint for each specified unit and for each key whose value is from $\mathcal{L}(\mathit{FileName})$ or is composed of words from $\mathcal{L}(\mathit{FileName})$, as defined in Figure 3.4 on page 39. Secondly, this constraint specifies the referenced files to be contained in the CoDSPL policy's file map and to be of the respective expected format.

For the remainder of this section, let $(\mathit{ed}, \mathit{files}) \in \mathit{POL}$ be a well-formed CoDSPL policy, $(\mathit{glob}, \mathit{Ids}, \mathit{loc}) = [\![\mathit{ed}]\!]$ be the policy's encapsulation description, and $\mathit{id} \in \mathit{Ids}$ be an arbitrary but fixed unit identifier. Moreover, let $\mathit{LP} = \mathit{loc}(\mathit{id}, \text{‘}\texttt{policy}\text{’})$ be the name of the local policy class, $\mathit{EvF} = \mathit{loc}(\mathit{id}, \text{‘}\texttt{event-factory}\text{’})$ be the name of the event factory class, $\mathit{EnF} = \mathit{loc}(\mathit{id}, \text{‘}\texttt{enforcer-factory}\text{’})$ be the name of the enforcer factory class.

Initialization of units When a unit specified by a CoDSPL policy starts, it creates three objects of classes whose name is provided by the encapsulation description *ed*. An object of type *LP* is created through the constructor that accepts a single String argument, to which the unit identifier *id* is passed. In the following, we denote the local policy object by *lp*. An event factory object of type *EvF* and an enforcer factory object of type *EnF* are created without any arguments to their constructors.

Individual intercepted operation When an agent performs an operation that, according to AspectJ's semantics of pointcuts, matches a pointcut declaration from $\mathit{loc}(\mathit{id}, \text{‘}\texttt{pointcuts}\text{’})$, where *id* is the identifier of the agent's unit, then this operation is intercepted. That is, the agent does not perform this operation or continue with other operations from the agent's implementation, but rather the unit for the agent continues as follows.

Let the intercepted operation be the method call captured by $o.m(a_1, \dots, a_n)$, where *m* is the method to which the call is intercepted, *o* is the object on which the method is called, n is the number of formal parameters of *m*, and $a_1, \dots, a_n$ are the actual arguments in the method call. The event factory object is used by the unit for obtaining an event object of class Event from the intercepted operation. For this, the unit obtains the event object *eo* of type Event from the value returned from calling the method $\mathit{pcname}(a'_1, \dots, a'_k)$ on the event factory object, where *pcname* is the (unique) name of the pointcut matching the intercepted operation, $p_1, \dots, p_k$ are the names of the formal parameters of the pointcut with name *pcname*, and $a'_1, \dots, a'_k \in \{o, a_1, \dots, a_n\}$ are the following values:

- $a'_i = o$ if p_i is the argument of a ‘`target`’ pointcut primitive for *pcname* and
- $a'_i = a_j$ if p_i occurs in the j-th position of the formal parameters list of an ‘`args`’ pointcut primitive for *pcname*.

In the transmission from the interceptor component to the coordinator component, the event object is encoded for transmission and subsequently decoded. While this produces a new event object, it's content remains the same in the process. In the following, we therefore refer to the transmitted event object also as *eo*.

The unit's coordinator then invokes the local policy object *lp* for deciding about the event object by invoking the method lp.localRequest(eo). The object returned by the local policy is of type LocalPolicyResponse, i.e., of type Decision or of type DelegationLocPolReturn (see Listing 3.4 on page 34). In the latter case, delegation is requested by the local policy. The unit's coordinator extracts the identifier id' of the destination unit and the DelegationReqResp object, i.e., the delegation request or delegation response that the coordinator subsequently sends to the unit with identifier id'. For the address of the unit, the coordinator uses $loc(id', \text{`}\texttt{ext-host}\text{'})$ and $loc(id', \text{`}\texttt{ext-port}\text{'})$.

In the former of the two cases, i.e., if the local policy invocation yields an object do of type Decision, this object is transmitted to the enforcer component via address $loc(id, \text{`}\texttt{enf-host}\text{'})$ and $loc(id, \text{`}\texttt{enf-port}\text{'})$. The enforcer component uses the enforcer factory object for obtaining a countermeasure object from the decision do by calling EnF.fromDecision(do). On the returned countermeasure object, the unit then successively calls the methods before, suppress, and after. If suppress returns **false**, then the originally intercepted method call $o.m(a_1, \ldots, a_n)$ is performed before method after is called. The methods of the countermeasure object are called by the enforcer component, i.e., in the context of the agent which can, thus, be influenced by the countermeasure object.

In the above, we deliberately neglected the memory (stack and heap) of the unit. Rather, we implicitly consider the unit's memory at the point in time that the agent's operation is intercepted. This accounts for the fact that the components communicate by exchanging messages rather than using shared memory.

Individual delegation request and delegation response Let dr be the decoded delegation request object or delegation response object received by the coordinator of the unit. The unit's coordinator then invokes the local policy object lp for deciding about the delegation request or, respectively, delegation response by calling the method lp.remoteRequest(dr). The object returned by the local policy is of type LocalPolicyResponse and is further processed as is the result of the local policy invocation in the case of an individual program operation.

Sequences of intercepted operations and delegation For a given sequence of event objects, delegation requests, and delegation responses, the local policy object lp is invoked on a memory that is essentially the same as after the previous invocation. For the first invocation of lp, the memory is essentially the same as after lp's construction. More concretely, by "essentially the same", we refer to a memory in which

1. lp is the same object (including, transitively, all objects that lp references) and
2. the event object, delegation request, or delegation response on which the local policy object lp is invoked is added to the memory.

That is, the coordinator does not alter the part of the memory that is relevant to lp. Analogous to the local policy, also the memory for the event factory and the enforcer factory is essentially the same between two invocations. That is, the relevant part of the memory for the event factory and the enforcer factory is not changed between two invocations by the fixed components of the unit or by the agent.

Our semantics of CoDSPL, as defined in this section, formally specifies the semantics of encapsulation descriptions as this semantics deviates from typical key-value semantics and, moreover, constitutes the basis for the remaining semantics. We provide a semi-formal description of the declarations of security-relevant operations and of operational specifications as we expect this level of detail to suffice for clarifying the understanding. A full formal semantics would benefit, e.g., the formal verification of properties of individual CoDSPL policies or of the class of all CoDSPL policies. For such a model, existing formal semantics of Java (e.g., Jinja [KN06]) and AspectJ (e.g., [WKD04]) could be used. However, this is not our focus here. For the verification of sound enforcement, we propose to use a more light-weight, process-algebraic model in Chapter 8.

3.6. Summary

We presented the policy language CoDSPL for specifying cooperative distributed enforcement mechanisms. The language allows policies to specify which units encapsulate which agents of a distributed program, which actions of the agents are security-relevant, and how countermeasures are decided. In particular, CoDSPL allows the policies to specify when and how each unit of a distributed enforcement mechanism initiates cooperation and how it responds to other units' requests for cooperation.

The syntax of CoDSPL enables modular policy specifications, firstly through the modularization introduced through the three sub-languages and secondly through standard modularization techniques from object-oriented programming that can be applied to the operational specifications. This modularity simplifies both the specification of CoDSPL policies as well as the subsequent verifiability of the policies (thereby supporting Requirement (Req-2)). Modularity has been proposed for policy languages of security enforcement mechanisms before [BLW09], but to the best of our knowledge, CoDSPL is the first modular policy language that supports distributed targets and the first to incorporate cooperation.

CoDSPL is an expressive policy language. It obtains its expressiveness mainly from its integrated sub-languages: the general purpose programming languages Java and Ruby as well as the pointcut specification languages of the general-purpose AOP tools AspectJ and Aquarium. Due to its expressiveness, CoDSPL can be used for specifying policies for a wide range of security requirements (i.e., the policy language supports Requirement (Req-1)). For non-distributed programs implemented in Java, a similar combination has already been proposed for policy languages [MJG+12]. We are the first to apply this combination to Java and Ruby in one policy language and to apply it in a policy language for distributed programs.

CoDSPL particularly allows policies to specify which and when units shall cooperate. Cooperation can be specified in a centralized fashion (as, e.g., in Example 3.5) as well as in a decentralized fashion (for instance, by not always delegating to a central unit but delegating to event-specific units). Policies can also desist from cooperation (e.g., CentralPrintPolicy in Example 3.5 never delegates). That is, cooperation can be specified in a way that suits the security requirement and/or the architecture of the distributed target. We are not the first to propose policy-determined cooperation [MU00], but we are the first to support such parametric cooperation in a policy language in which both when cooperation shall take place as well as which units shall cooperate can be specified in an expressive language (concretely, via Java). Moreover, we are the first to support such cooperation in a policy language that supports events beyond transmission of network messages.

Leaving the specification of cooperation up to CoDSPL policies evades theoretical limitations that suggest that automatically deriving a suitable form of cooperation might be infeasible in general. For instance, the problem of synthesizing a distributed program for a given architecture and a specification is infeasible for particular forms of cooperation such as chains and rings [KV01] and is undecidable in general [PR90]. Transferred to run-time enforcement, the distributed program corresponds to the distributed enforcement mechanism and the specification corresponds to the security requirement.

The policy language presented in this chapter integrates into the context of this thesis as follows: Firstly, CoDSPL is the policy language used by our tool CliSeAu, which is described in Chapter 5. Secondly, CoDSPL constitutes the basis for the delegation approaches presented in Chapters 6 and 7, which focus on modularity of policy specifications and, respectively, on the effectiveness of policies under race conditions.

Chapter

4

A Technique for Encapsulating Distributed Programs

4.1. Introduction

We propose *cross-lining*, a technique for encapsulating distributed programs with distributed enforcement mechanisms. The technique is formulated at an architectural level, based on a very basic common architecture of enforcement mechanisms. Based on such modular units, the technique consists of two parts: encapsulating each agent such that the agent's security-relevant actions can be intercepted and countermeasures can be applied; and generating a component for each agent that is external to the agent's code and that can make decisions and cooperate with other units.

The cross-lining technique enables, as a result of the encapsulation, each unit of the distributed enforcement mechanism to intercept the security-relevant actions of an agent before they take place, to impose timely countermeasures on the agent, and to cooperate with other units of the mechanism. Timely countermeasures against an intercepted action especially include two kinds of measures of the unit: firstly, measures that take place directly before or after the intercepted action is performed; secondly, measures that alter or suppress the intercepted action altogether. The cooperation enabled by the cross-lining technique encompasses cooperation that may or may not follow the respective agents' communication. That is, two units are enabled to cooperate when a communication attempt between their agents was intercepted. A unit may also cooperate with another unit if the former unit's agent attempted to communicate with a different agent or did not attempt to communicate at all. Particularly, the technique enables a unit to cooperate with other units even when the unit's local agent is not engaging in security-relevant actions or is idle.

Structure The remainder of this chapter is structured as follows. In Section 4.2, we introduce the basic unit architecture underlying the cross-lining technique. Section 4.3 presents the cross-lining technique. In Section 4.4, we instantiate the technique for selected

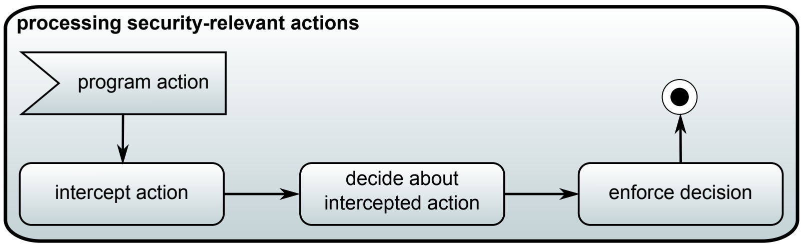

processing security-relevant actions
program action
intercept action
decide about intercepted action
enforce decision

Figure 4.1.: Processing of security-relevant actions by an enforcement mechanism (UML activity diagram)

example architectures of enforcement mechanisms, and in Section 4.5, we compare crosslining with inlining. We summarize the content of the chapter in Section 4.6.

4.2. Functionality of Enforcement Mechanisms

Enforcement mechanisms provide functionality for being able to enforce certain security requirements on certain kinds of targets. While the concrete functionality differs from mechanism to mechanism, we can identify a common baseline of functionality of enforcement mechanisms. Here, we focus on the run-time functionality of enforcement mechanisms, i.e., the functionality used by the mechanisms at run-time for processing security-relevant actions of a target.

Figure 4.1 shows the three activities that, in some form, are part of the functionality of enforcement mechanisms. For each potentially security-relevant action performed by the target at run-time, an enforcement mechanism intercepts the action. Intercepting an action involves at least detecting that the action is occurring or is about to occur. It possibly also involves temporarily preventing the action and subsequent actions from occurring. After intercepting an action, the mechanism decides about the action. Depending on the mechanism, the security requirement, and the action, the deciding may involve several subordinate activities: retrieving information (state), the actual decision-making, and recording the decision made (state-keeping). Possible decisions are, e.g., to allow an intercepted action, to disallow the action by performing some countermeasure, or to allow the action along with, e.g., logging the occurrence of the action or raising an alarm to a stakeholder. The enforcement mechanism finally enforces the respective decision and thereby completes the processing for the intercepted action.

The three activities can be found in many enforcement mechanisms (e.g., [ES00b; MU00; BLW09; PLB11; MJG⁺12; KP15]). Some mechanisms reflect the three activities in their component architecture. For instance, the XACML [OAS13] reference architecture [OAS13, Section 3.1] distinguishes the *policy enforcement point* (*PEP*), a component for intercepting access requests and for imposing obligations, and the *policy decision point* (*PDP*) and *policy information point* (*PIP*) for deciding about an access request based on information about the target's environment and about subjects relevant to the target. This reference architecture constitutes the basis for several enforcement mechanisms (e.g., [PLB11; KP15]). The architecture of service automata [GMS12] also reflects the three activities by two separate components for intercepting actions and enforcing decisions as well as two components

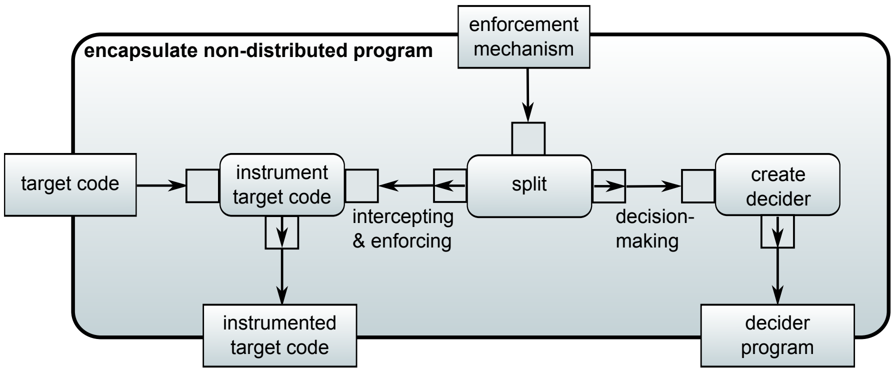


Figure 4.2.: Encapsulation of agents (UML activity diagram)

for decision-making. Some mechanisms also reflect the three activities in the structure of their policy languages, in which the individual activities are specified separately (e.g., [BLW09; MJG+12] and also CoDSPL in Chapter 3).

4.3. The Cross-lining Technique

The cross-lining technique employs aspects of the inlining technique, which weaves *all* the code of an enforcement mechanism into the code of a target, but only weaves the code implementing selected functionality of the mechanism. The technique places the remaining functionality of the mechanism into a separate program that is supposed to run in parallel to the target.

Encapsulating non-distributed programs Cross-lining uses the inlining technique for the functionalities of intercepting program actions and of enforcing decisions. It creates a separate program, called *decider program*, that contains and executes the functionality of decision-making. That is, the technique instruments the target by placing the code of the enforcement mechanism that implements the functionalities of intercepting program actions and of enforcing decisions into the code of the target. More concretely, this code of the enforcement mechanism is placed around the instructions (figuratively: the lines) of potentially security-relevant operations such that the code of the mechanism is invoked whenever a security-relevant operation is performed. For establishing that all three activities of the enforcement mechanism connect as displayed in Figure 4.1, cross-lining further places functionality of inter-process communication (*IPC*) into both the instrumented target as well as the separate program such that both interface with each other.

Figure 4.2 shows the encapsulation a single non-distributed program with a focus on the involved activities. The technique takes the code of a non-distributed target as well as the code of an enforcement mechanism as input. From these inputs, the technique produces as its two outputs firstly, the instrumented code of the target, and, secondly, the decider program, i.e., the program containing the decision-making functionality.

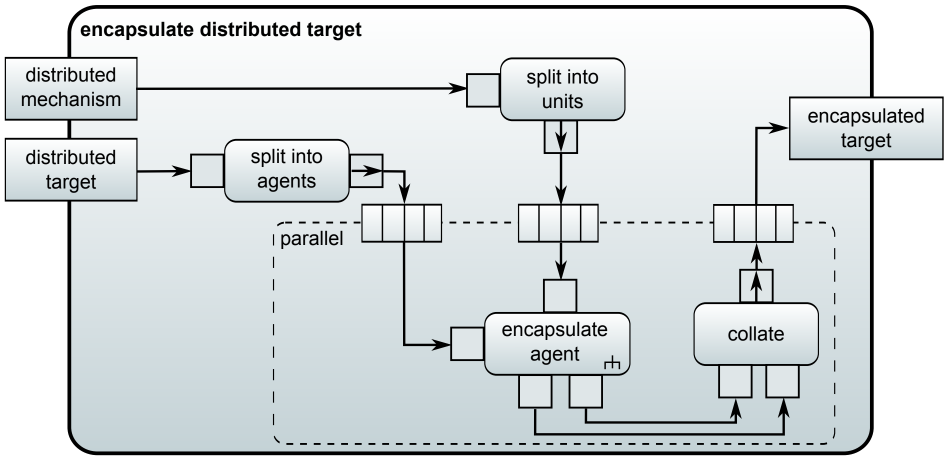


Figure 4.3.: Cross-lining for distributed targets (UML activity diagram)

Encapsulating distributed targets The cross-lining technique lifts to a distributed target and a distributed enforcement mechanism by applying the technique for each unit of the distributed enforcement mechanism and the corresponding agent of the distributed target that shall be encapsulated by this unit. In the context of cross-lining for distributed targets, we refer to the two outputs of the cross-lining for an individual agent as the *encapsulated agent*.

Figure 4.3 depicts the cross-lining technique for encapsulating a distributed target. The technique takes as input a distributed target whose individual agents can be derived from this input. Analogously, it takes as input a distributed enforcement mechanism whose units can be derived from this input and additionally obey the service automata architecture. As output, the technique produces an *encapsulated target*: a collection of encapsulated agents. The activity splits the distributed target into its agents and splits the distributed enforcement mechanism into its units. For each agent and the corresponding unit of the enforcement mechanism, the activity encapsulates the agent by performing the "encapsulate non-distributed program" activity, depicted in Figure 4.2 on page 51. As there are no dependencies between the agents or units with regard to the encapsulation, cross-lining can perform the encapsulation of the individual agents in parallel. The resulting instrumented agents' codes and decider programs are then collated to the encapsulated agents, which in turn form the encapsulated target.

The particular design of the cross-lining technique enables the activities of intercepting and enforcing to take place synchronously to the agents of a distributed target and to make use of application abstractions (e.g., methods defined in the code of an agent). Moreover, the technique enables the decision-making functionality to take place in parallel to the running agent. This allows the decision-making functionality to be responsive both to input from the intercepting functionality and to cooperation requests from remote units. The responsiveness is particularly not limited to time frames in which the local agent activates the decision-making by performing security-relevant operations. Rather, the decider program can cooperate with remote units even when the local agent is idle, blocked, or terminated.

4.4. Example Instances of Cross-lining

We illustrate the cross-lining technique for three high-level architectures of enforcement mechanisms.

Service automata The service automata concept comprises four components: the interceptor, the coordinator, the local policy, and the enforcer. Of these components, the interceptor intercepts security-relevant actions, the local policy decides about locally intercepted actions as well as about delegation requests from remote service automata, and the enforcer implements decisions made by the service automaton. Finally, the coordinator manages the delegation between service automata.

We can instantiate cross-lining for an enforcement mechanism that implements the service automata concept as follows. Firstly, the code of the interceptor and enforcer components would be inlined into the respective target, since they implement the functionality of intercepting actions and enforcing decisions and no decision-making functionality. Conversely, the code of the coordinator and local policy components would be placed into the decider program, since they both implement functionality of decision-making and neither functionality of intercepting nor of enforcing decisions.

The CoDSPL reference architecture The reference architecture of CoDSPL, as introduced in Section 3.2, refines the architecture of the service automata concept. As part of this refinement, the architecture comprises two components in addition to the components in the service automata concept: the event factory and the enforcer factory. The two additional components are responsible for abstracting from intercepted operations to entities suitable for the local policy and, respectively, for concretizing decisions of the local policy to concrete countermeasures.

We can instantiate cross-lining for an enforcement mechanism that implements the CoDSPL reference architecture as follows. The four components inherited from the service automata concept are treated the same as described for cross-lining of service automata, since the classification of these components with regard to the three functionalities of enforcement mechanisms remains the same. The two additional components can be classified as being part of the decision-making functionality as well as being part of the other two functionalities. Inlining the code implementing the two components would allow a reduction of the amount of data transferred between the components, because the abstracted event objects and decision objects can be expected to be smaller in size than the intercepted operation and, respectively, countermeasure as a whole. Conversely, placing the two components into the decider program would effectuate a smaller modification of the target.

XACML The XACML reference architecture [OAS13, Section 3.1] comprises four central components: the PEP, the PDP, the PIP, and the context handler. The PEP is the component that intercepts access requests and enforces authorization decisions [OAS13, p. 10]. The PDP evaluates the given policy and makes authorization decisions. The PIP provides information for the decision-making by the PDP. Finally, the *context handler* is a component that manages both the interaction between PDP and PIP as well as between PDP and

PEP. As part of the latter, the context handler also performs translations between the XML-based XACML language and the respective "native" format understood by the PEP [OAS13, p. 10].

We can instantiate cross-lining for an enforcement mechanism that exhibits the XACML reference architecture as follows. Firstly, the code of the PEP would be inlined into the respective target, since it implements the functionality of intercepting actions and enforcing decisions and no decision-making functionality. Conversely, the code of the PDP and PIP components would be placed into the decider program. The context handler's functionality of managing the interaction between PDP and PIP can be attributed to the decision-making functionality and its code would therefore also be placed into the decider program. The context handler's translation functionality can be attributed to either of decision-making or intercepting/enforcing and its code can therefore be placed in either of the two outputs of cross-lining.

4.5. Comparison with Other Techniques

We compare the cross-lining technique against other possible techniques for applying a security enforcement mechanism to a program. For the comparison, we choose two aspects: the granularity of operations with which the enforcement mechanism interfaces with the program, and the possibility of cooperation for a distributed enforcement mechanism.

Granularity of operations and countermeasures Cross-lining instruments a target by placing the enforcement mechanism's functionality of intercepting actions and enforcing decisions into the code of the target's agents. By building on program instrumentation, cross-lining allows the units of the enforcement mechanism to control an agent by intercepting actions and imposing countermeasures at programming-language granularity and at a level of application abstractions. This granularity and level of abstraction, in contrast to, e.g., the level of hardware or operating system granularity, benefits the enforcement of security requirements that are formulated in terms of higher level operations.

Example 4.1. Consider the scenario of Example 1.1 on page 6, in which the security requirement constraining downloads of users. Consider furthermore that the services that provides files for download is implemented in an object-oriented programming language and following an object-oriented design [BME+07]. The implementation of the service, thus, contains abstractions such as user objects and session objects, where a session object encapsulates (in the OOP sense [BME+07, pp. 50-54]) the connection between a user who is logged in and the network connection through which she is logged in. In the implementation, a download of a file is performed by a method on a session object. An enforcement mechanism based on instrumentation of the services can intercept the download by intercepting calls to the particular method of the session object.

For an enforcement mechanism at a lower-level interface to the service, such as the level of an operating system, the download of a file by a user assumes a different form. At that level, the download would consist of a sequence of operations such as the establishment of a network connection, several network transmissions in a protocol such as FTP corresponding to the successful login of the user, the opening of a file, and the piecewise transfer of content from the file to the user's network connection. All these operations would have to be intercepted at the operating system level for detecting when a particular user downloads a particular file. ◊

While Example 4.1 illustrates the benefit of high level of abstraction for intercepting actions, the argument also transfers to countermeasures. At an application level, countermeasures can invoke custom error routines or through suitable exceptions. Countermeasures directly available in the operating system are more crude, such as terminating a program or closing a network connection.

Besides cross-lining, the inlining technique [ES00b] also supports an application-level granularity. Enforcement mechanisms implemented in an interpreter or virtual machine (e.g., [RHN+13; RBG+15]) also support this granularity, but require the presence of an interpreter or VM and particularly one that supports enforcing security requirements. Cross-lining, like inlining, does not require an interpreter or VM at all and, thus, particularly does not require a customized interpreter or VM that supports enforcing security policies.

Cooperation within a distributed enforcement mechanism Cross-lining places an enforcement mechanism's functionality of decision-making into separate programs that are run in parallel to the target's agents. This allows the decision-making functionality at the individual units to cooperate when the information available at only one unit does not suffice. More concretely, the cooperation can be performed by the units asynchronously to the communication of the agents of the target. That is, cooperation between the units can take place event when the respective agents do not communicate.

Example 4.2. Consider again the scenario of Example 1.1, and consider a distributed enforcement mechanism whose decision-making functionality supports cooperation among units. The enforcement mechanism is applied to the distributed storage service by means of cross-lining. For preventing conflicts of interest, the units cooperate as follows. The unit at which a file access by a user is intercepted requests from all other units (respectively, their decider programs) where the user could have accessed a conflicting file whether the user indeed accessed such a file. This form of cooperation is asynchronous to the communication of the agents, as the agent at which the file access is intercepted would communicate with the other agents as part of the access. Due to cross-lining, all involved decider programs are responsive to these requests.[1] ◊

To some extent also the inlining technique can be applied to cooperating units of a distributed enforcement mechanism. Firstly, inlined units can perform cooperation that is synchronous to the communication of agents. For this form of cooperation, the unit of a sending agent can piggy-back information required by the receiving agent's unit on the regular message; the unit of the receiving agent then extracts the piggy-backed information. Variations of this form of cooperation have been proposed for several distributed enforcement mechanisms, though none based on the inlining technique. Rather, dedicated communication libraries [MU00; SVA+04] or middleware have been used [OBM10].

[1]In Chapter 7, we describe in detail an approach that requires less communication and is effective even when a malicious user attempts to concurrently access conflicting files.

Secondly, inlined units can perform asynchronous cooperation as follows. An operation that is performed very early during the start-up of an agent is declared as a security-relevant operation. When the respective unit intercepts this operation (for the first time), the unit spawns a new thread or process that runs the decision-making functionality. The resulting architecture of the units after spawning the thread or process then resembles the architecture of units in the cross-lining technique, but for achieving this, the approach requires reliable means for identifying the start-up operation. We are not aware of distributed enforcement mechanisms based on inlining that have been used for asynchronous cooperation.

4.6. Summary

We proposed the cross-lining technique for encapsulating distributed programs with distributed enforcement mechanisms. The technique encapsulates each individual agent of the distributed target by a unit of the distributed enforcement mechanism. For encapsulating individual agents, the technique splits the units of the enforcement mechanism by three basic functionalities that are common to enforcement mechanisms: intercepting potentially security-relevant operations, deciding about intercepted operations, and enforcing decisions. The code of the agent is instrumented to contain the first and the last of the three functionalities, while the second functionality is placed into the separate decider program.

By building on program instrumentation, cross-lining enables a unit to control an agent by intercepting actions and imposing countermeasures at programming-language granularity and at a level of application abstractions. This granularity and level of abstraction, in contrast to, e.g., the level of hardware or operating system granularity, benefits the enforcement of security requirements that are formulated in terms of higher level operations. Cross-lining inherits this feature from the inlining technique [ES00b], which in variations has been adopted by several enforcement mechanisms (e.g., [Erl04; BLW09; MJG+12]).

By placing the decision-making functionality of an enforcement mechanism into separate programs that are run in parallel to the target's agents, cross-lining enables the decision-making functionality of a unit to take place even when the agent is idle, terminated, or performing security-irrelevant actions. This property of cross-lined units enables them to cooperate with each other in enforcing security requirements in a distributed program. In particular, the property enables a cross-lined distributed enforcement mechanism to enforce, concerted security properties such as the one of Example 2.2. We are not the first to propose cooperating enforcement mechanisms [MU00; SVA+04; OBM10; KP15; DLJ15] but, to the best of our knowledge, cross-lining is the first proposal for a generally applicable technique that enables cooperating distributed enforcement mechanisms.

In the context of this thesis, the cross-lining technique is relevant for the tool CliSeAu, which implements the technique and is presented in the next chapter.

Chapter

5

A Generic Enforcement Mechanism For Distributed Programs

5.1. Introduction

Generic enforcement mechanisms can be employed for enforcing a wide range of security requirements and can be applied to a wide range of targets. When the target is a distributed program, i.e., consists of several distributed agents, a generic enforcement mechanism can consist of several units that enforce security at the agents of the target. The units of such a mechanism can cooperate with each other for enforcing security requirements that constrain the joint behavior of the target's agents, such as those requirements captured by concerted security properties.

We present *CliSeAu* (abbreviating "Cross-lining Service Automata"), a tool for enforcing security requirements in distributed programs. CliSeAu consists of two parts: two parametric unit implementations, called *generic enforcement capsule* (*generic EC* in short), and a tool implementing the cross-lining technique, called the *encapsulation tool*. Figure 5.1 on the next page illustrates the two parts of CliSeAu together with their connections to inputs and outputs of CliSeAu. The encapsulation tool takes a distributed target and a CoDSPL policy as input artifacts. From these inputs, the encapsulation tool uses the generic ECs for creating an encapsulated target as the output artifact. The two generic ECs of CliSeAu implement the reference architecture of CoDSPL policy, which in turn derives from the service automata concept, for Java agents and, respectively, for Ruby agents. The encapsulation tool uses the cross-lining technique for creating an encapsulated target as described in Chapter 4.

By supporting CoDSPL as policy language for the enforcement mechanisms generated by encapsulation tool, CliSeAu allows for enforcing a wide range of security requirements on a wide range of Java and Ruby programs. Particularly, CliSeAu supports that the units of generated enforcement mechanisms can make local decisions and can also cooperate by means of delegation. The design of both the generic EC and the encapsulation tool of CliSeAu follows the principles of object-oriented design [BME⁺07; GHJ⁺95]. The

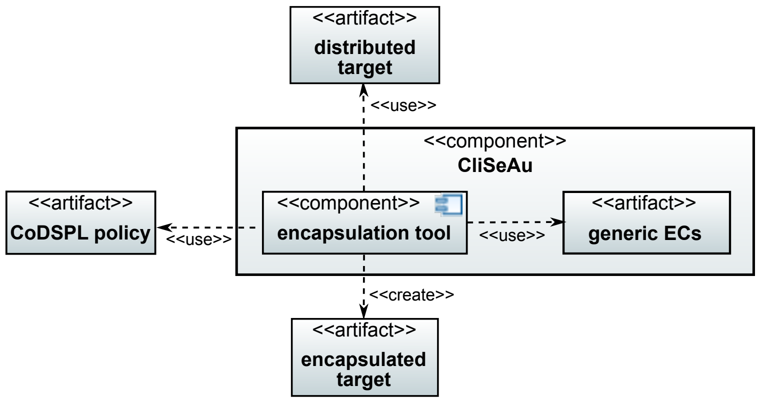


Figure 5.1.: High-level architecture and inputs/outputs of CliSeAu (UML component diagram)

implementation of CliSeAu passed three static analysis tools that identify common implementation flaws and inefficiencies, and the code is thoroughly documented. The design and implementation of CliSeAu's generic ECs enables cooperation for multiple intercepted actions to be performed concurrently to reduce latencies in the decision-making.

Structure The remainder of this chapter is structured as follows. Section 5.2 describes the high-level architecture of CliSeAu's encapsulation tool. In Section 5.3, the design of CliSeAu's generic ECs is presented with a focus on the how the generic ECs cooperate. How the cross-lining technique is instantiated by CliSeAu is modeled in Section 5.4. Section 5.5 describes selected aspects of CliSeAu's low-level architecture and implementation. An analysis of the design and implementation of CliSeAu with respect to its design, code quality, and documentation is provided in Section 5.6. Finally, Section 5.7 summarizes the contributions presented in this chapter.

5.2. Design of the Encapsulation Tool

CliSeAu's encapsulation tool generates a distributed enforcement mechanism from a given CoDSPL policy and applies the generated mechanism to a distributed target. For generating the mechanism, CliSeAu instantiates its generic ECs using the CoDSPL policy. For applying the mechanism, CliSeAu builds on the cross-lining technique introduced in Chapter 4. In this section, we introduce a white-box view on the high-level design of the encapsulation tool.

The architecture aims at separation of concerns for the individual activities and entities involved in encapsulating a distributed target by means of cross-lining. Concretely, the architecture identifies three concerns: the generation of the decider programs, the instrumentation of the agents, and the provisioning of policy elements.

Figure 5.2 shows the high-level component our model of the encapsulation tool based on the identified concerns. As input, the tool expects a CoDSPL policy as well as a distributed target and produces an encapsulated target as output. The encapsulation tool includes a

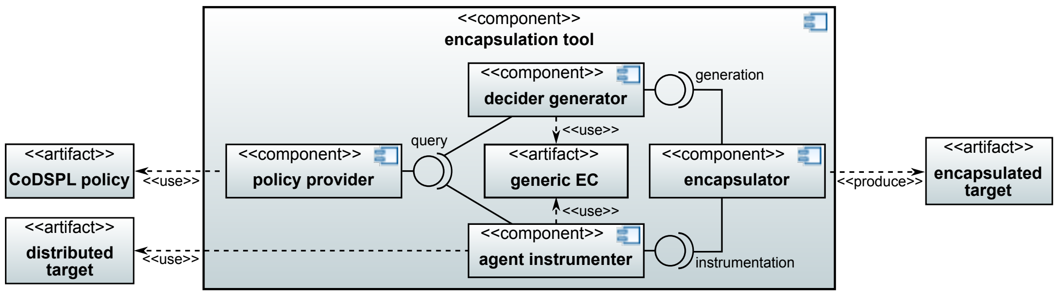


Figure 5.2.: Architecture of encapsulation tool (UML component diagram)

policy provider component, which offers an interface for querying individual parts of the given CoDSPL policy. Two components, the *decider generator* and the *agent instrumenter*, offer interfaces for generating decider programs and, respectively, instrumented code of agents. The two components use of the *generic EC implementation*, i.e., the second part of CliSeAu next to the encapsulation tool. The *encapsulator* is a component that uses the two components for producing the encapsulated target. The inputs and outputs of the encapsulation tool resemble those of the cross-lining activity displayed in Figure 4.3 on page 52. The difference is that the encapsulation tool takes a CoDSPL policy as input instead of a distributed enforcement mechanism, but internally generates the units of the distributed enforcement mechanism by instantiating the generic EC as specified by the CoDSPL policy.

The separation between the policy provider, decider generator, and agent instrumenter allows each of the components to vary with low impact on the other. For instance, how a decider program is generated by the decider generator component could vary independently from how the instrumentation is realized and vice versa. Moreover, the separate policy provider hides the specificities of the CoDSPL syntax, such as the key-value format of encapsulation descriptions and its key precedence rules, as defined in Definition 3.2 on page 43. This simplifies the realization of the other two components that use the policy provider and would furthermore allow for future extensions of CoDSPL, as already evidenced by Hamann [Ham16].

5.3. Design of Generic Enforcement Capsules

CliSeAu's generic ECs implement the reference architecture of CoDSPL presented in Section 3.2 on page 28. In this section, we present how the coordinator of CliSeAu's generic ECs realizes cooperation. The cooperation is realized such that it enables cooperation for multiple intercepted actions to be performed concurrently.

We capture the cooperation performed by the coordinator component together with a local policy through its involved activities. The overall activity, *cooperatively deciding*, takes an event object as input and produces a decision object as output. The activity itself may involve communication with other units, when helpful for determining an appropriate decision object, but is not required to. Whether communication takes place or not can be determined by the activity of cooperative deciding itself.

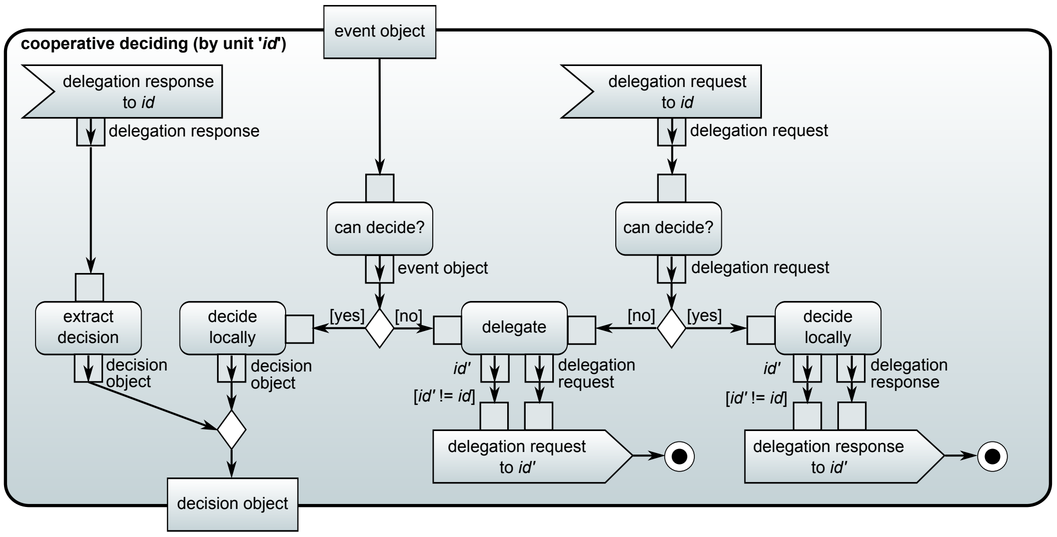


Figure 5.3.: Cooperative deciding by units (UML activity diagram)

The cooperative deciding can be triggered by an intercepted program operation and can also be triggered by the cooperative deciding activity of a remote unit. Overall, we identify five possible courses of action for the cooperative deciding that are enabled by the coordinator component. These courses of action are determined by the trigger as well as by when the local policy specifies that delegation shall take place. They correspond to the possible paths depicted in Figure 5.3, which we discuss in the following. In the exposition, we distinguishes the aspects 'when', 'where', and 'what' of delegation.

Case 1: locally making a decision for a locally intercepted event (path from event object to decision object).

This case occurs when an event object (top box) is given to the cooperative deciding and the result of the "can decide" activity is positive ('when'). Then the unit performs the "decide locally" activity and returns the result as a decision object (bottom box).

Case 2: delegating the decision-making for a locally intercepted event (path from event object to delegation request).

This case also occurs when an event object is given to the cooperative deciding and the result of the "can decide" activity is negative ('when'). In this case, the unit performs the "delegate" activity to determine the unit (*id′*) to which the delegation shall take place ('where') and the delegation request that shall be sent to that unit ('what').

Case 3: locally making a decision for a remotely intercepted event (path from delegation request to delegation response).

This case occurs when a delegation request is signaled to the unit and the result of the "can decide" activity is positive ('when'). Then the unit performs the "decide locally" activity to determine the unit to which the result shall be sent ('where') and the delegation response that shall be sent to that unit ('what').

Case 4: delegating the decision-making for a remotely intercepted event (path from delegation request to delegation request).

This case occurs when a delegation request is signaled to the unit and the result of the “can decide” activity is negative (‘when’). In this case, the unit performs the “delegate” activity to determine the unit to which the delegation shall take place (‘where’) and the delegation request that shall be sent to that unit (‘what’). This course of action can occur when a unit makes a partial decision.

Case 5: receiving a remotely made decision for a locally intercepted event (path from delegation response to decision object).

This case occurs when a delegation response is signaled to the unit. In this case, the unit performs the “extract decision” activity and returns the result as a decision object.

The cooperative deciding ends either by returning a decision object or by signaling a delegation request or delegation response to another unit. Which of the cases takes place is determined by a sequence of at most two activities that determine the aspects ‘when’, ‘where’, and ‘what’ of delegation. In case of delegation, the cooperative deciding ends after a signal is sent to another unit. This ensures that the cooperative deciding does not block until a decision for a delegated event is obtained. Non-blocking cooperation is also achieved when the decision-making for a received delegation request is further delegated (Case 4). This allows delegation responses to be returned directly to their destination without indirections over other units. It also allows the coordinator to be stateless with respect to received or sent delegation requests and responses.

5.4. Cross-lining

We specify how CliSeAu implements the cross-lining technique for a given CoDSPL policy in particular detail by providing a semi-formal model. In the model, we capture the implementation in a denotational form, as a function from CoDSPL policies to encapsulated targets, in the main definition of this section, Definition 5.4 on page 67. As prerequisite for this definition, we first provide a definition of template instantiation, how aspects are generated and woven into agent code, and how a decider program is generated.

Template instantiation For capturing complex strings that are constructed from multiple parameters, CliSeAu uses templates. These *templates* are strings (i.e., words) with placeholders specified in a particular syntax, which we describe next. Firstly, a template can contain placeholders enclosed in angle brackets that can be substituted by concrete parameters. Secondly, a template can contain conditional blocks (enclosed in guards of the form ‘$<$ `if(guard)` $>$’ and ‘$<$ `endif` $>$’) that are included in an instance of the template if and only if the parameter in the conditional is set to the word ‘`true`’. Placeholders inside included conditionals are again substituted by concrete parameters, but without further recursive substitution. The use of the placeholders and conditionals models the subset of the possibilities offered by the stringtemplate library for Java that is used by CliSeAu. We define templates and their instantiation as follows.

Definition 5.1. The *template instantiation* is the function $inst : \mathcal{W} \times (\mathcal{W} \rightharpoonup \mathcal{W}) \to \mathcal{W}$ for which $inst(w, sf) = w'$ holds if and only if w' is the result of substituting

1. every infix '$<$ `if(`' u '`)` $>$' v '$<$ `endif` $>$' of w that satisfies $u \in \mathcal{W}_{\{<,>\}}$ and '$<$ `endif` $>$' $\not\trianglelefteq v$ by v if $u \in dom(sf)$ and $sf(u) =$ '`true`' and by ε otherwise;
2. every infix '$<$' u '$>$' of w that satisfies $u \in \mathcal{W}_{\{<,>\}} \cap dom(sf)$ by $sf(u)$.

We call a function used as the second argument to the template instantiation a *substitution function.* $\Diamond$

The definition faithfully captures the intention of instantiating a template, both for parameters and for conditionals. In particular, the definition ensures that a parameter is substituted at most once and that the substitute is not subject to further substitution.

Agent instrumentation For instrumenting an agent, CliSeAu utilizes an AOP aspect that captures what methods of the agent's code are security-relevant and what functionality shall be performed when the agent is about to call one of such methods. In the following, we present the instrumentation of agents that are implemented in the Java programming language. We leave out the instrumentation for Ruby agents as it is largely analogous except that the respective templates are for the Aquarium tool rather than for AspectJ. The functionality performed by the aspect corresponds to the functionality of the interceptor and the enforcer components. We capture the generation of this aspect as follows.

Definition 5.2. The *aspect generation function genAspect* $: POL \times \mathcal{L}(Id) \rightharpoonup \mathcal{W}$ is defined as follows. For an arbitrary but fixed well-formed CoDSPL policy $pol = (ed, files) \in POL$, $(glob, Ids, loc) = [\![ed]\!]$, and a unit identifier $id \in \mathcal{L}(Id)$, the partial function *genAspect* is undefined for (pol, id) if $id \notin Ids$ holds and defined as follows otherwise.

$$genAspect(pol, id) = inst(aspt, sf)$$

where

- $pcs = loc(id,$ '`pointcuts`'$) \in \mathcal{L}(PointcutSpec)$,
- *aspt* is the content of Listing 5.2 on page 64,
- $sf : \mathcal{W} \rightharpoonup \mathcal{W}$ is defined by
  - $sf($'`Imports`'$) \in \mathcal{L}(\{Import\text{';'}\})$ is the adjoined import declarations occurring in *pcs*;
  - $sf($'`Pointcuts`'$) \in \mathcal{L}(\{PointcutDecl\text{';'}\})$ is the adjoined pointcut declarations occurring in *pcs* with two modifications: firstly, from the "formals" the return type is removed and, secondly, each pointcut expression *pe* is substituted by '(' *pe* '`)&&!within(CliSeAu)`';
  - $sf($'`LocalEncodingClass`'$) = x$, where $x =$ '`SerializationEncoding`' holds if $loc(id,$ '`local−encoding`'$) =$ '`serialization`' and $x =$ '`GsonEncoding`' holds if $loc(id,$ '`local−encoding`'$) =$ '`JSON`';
  - $sf($'`CorHost`'$) = loc(id,$ '`cor−host`'$)$;
  - $sf($'`CorPort`'$) = loc(id,$ '`cor−port`'$)$;
  - $sf($'`EnfHost`'$) = loc(id,$ '`enf−host`'$)$;
  - $sf($'`EnfPort`'$) = loc(id,$ '`enf−port`'$)$;
  - $sf($'`Advice`'$)$ is the result of concatenating $inst(advt, sf')$ for all pointcut declarations *pd* occurring *pcs*, for which $sf' : \mathcal{W} \rightharpoonup \mathcal{W}$ is defined by

  * sf'(‘`PointcutName`’) is the pointcut identifier of *pd*;
  * sf'(‘`ReturnType`’) is the return type of *pd*, if provided, or ‘`void`’ otherwise;
  * sf'(‘`TypedParameters`’) $\in \mathcal{L}(\mathit{ParamList})$ is the formal parameters, including their types, of *pd*;
  * sf'(‘`Parameters`’) is the comma-separated list of names of formal parameters of *pd*;
  * sf'(‘`EventFactory`’) $= \mathit{loc}(\mathit{id}, \text{‘}\texttt{event} - \texttt{factory}\text{’})$;
  * sf'(‘`EnforcerFactory`’) $= \mathit{loc}(\mathit{id}, \text{‘}\texttt{enforcer} - \texttt{factory}\text{’})$;
  * sf'(‘`hasReturnType`’) is ‘`false`’ if ‘`ReturnType`’ is mapped to ‘`void`’ and is ‘`true`’ otherwise;
  * sf'(‘`ObjReturnType`’) equals sf'(‘`ReturnType`’) is it is an object type and equals its *boxed type* (e.g., Integer for **int**) if it is a primitive type;
- *advt* is the content of Listing 5.1 on the next page. ◊

By the definition, *genAspect*(*pol*, *id*) faithfully captures an aspect for CoDSPL policy *pol* and unit identifier *id* that implements the functionality of the interceptor and enforcer components. The aspect contains all the specified pointcuts and corresponding advices and thereby causes the aspect to intercept all operations of the agent declared as security-relevant in the CoDSPL policy. The pointcuts of the specification of security-relevant program operations are all transferred to the aspect such that all specified operations are actually part of the aspect. The augmentation of pointcut expressions by ‘!`within(CliSeAu)`’ ensures that the pointcuts apply to the agent’s code only and do not accidentally apply to code of units.[1] The advices, generated from the template in Listing 5.1 on the following page,[2] combined into the aspect implement the interceptor and enforcer components: firstly, by invoking the event factory and enforcer factory for creating event objects and, respectively, countermeasure objects; secondly, by sending event objects to the coordinator and receiving decision objects from the coordinator; and thirdly, by invoking the countermeasure’s methods before, suppress, and after for inserting and suppressing actions as specified by the countermeasure object. The second aspect is implemented by the CoordinatorInterface class, whose network configuration is performed, based on the host and port specifications of *pol*, in the aspect’s constructor. The definition of the advice template is carefully crafted to treat return values of intercepted method calls adequately, for primitive types as well as objects and for the case of suppressed method calls.

We capture how AspectJ weaves an aspect into a given Java program by the function $\mathit{AspectJ} : \mathit{FileContent} \times \mathit{FileContent} \rightharpoonup \mathit{FileContent}$. This partial function takes an aspect and a JAR file and, if the inputs are valid for AspectJ, returns a JAR file. The returned JAR file corresponds to the JAR file provided to the function but with the provided aspect woven into the code. We refrain from further defining the working of AspectJ here and refer to Section 2.5 as well as to the literature for an informal overview over AspectJ [KHH+01] and for formalizations of AspectJ’s weaving of aspects (e.g., [BD06]).

[1]This is based on the assumption that the agent’s code does not define a ‘`CliSeAu`’ class.

[2]For the sake of brevity, handling of exceptional behavior is not displayed in the templates. The implementation of the templates in CliSeAu does handle exceptions.

```
<ReturnType> around(<TypedParameters>) : <PointcutName>(<Parameters>) {
  Event event = <EventFactory>.<PointcutName>(<Parameters>);
  Countermeasure cm = null;

  CoordinatorInterface.send(event);
  Decision decision = CoordinatorInterface.receive();
  cm = <EnforcerFactory>.fromDecision(decision);
  if (cm == null) {
    cm = <EnforcerFactory>.fallback(event);
  }
  cm.before();
  <if(hasReturnType)> <ReturnType> result;<endif>
  if (!cm.suppress()) {
    <if(hasReturnType)>result = <endif> proceed(<Parameters>);
  } else {
    <if(hasReturnType)>result =
        (<ObjReturnType>)cm.getReturnValue(<ObjReturnType>.class);<endif>
  }
  cm.after();

  <if(hasReturnType)>return result;<endif>
}
```

Listing 5.1.: Advice template used by the aspect generation function

```
import java.io.IOException;
import java.net.UnknownHostException;
import net.cliseau.runtime.javacor.Event;
import net.cliseau.runtime.javacor.Decision;
import net.cliseau.runtime.javatarget.Enforcer;
import net.cliseau.runtime.javatarget.CoordinatorInterface;
import net.cliseau.runtime.javacor.encoding.DataEncoding;
<Imports>

aspect CliSeAu {
  CliSeAu() throws IOException,UnknownHostException {
    DataEncoding encoding = new <LocalEncodingClass>();
    CoordinatorInterface.init(encoding,
      "<CorHost>", <CorPort>, "<EnfHost>", <EnfPort>);
  }
  <Pointcuts>
  <Advice>
}
```

Listing 5.2.: Aspect template used by the aspect generation function

Decider program generation The decider program is one of the two components produced in the encapsulation of an agent via cross-lining, next to the instrumented code of the agent (see Section 4.3 on page 51). The decider program contains the coordinator and the local policy of a unit and, as such, serves the purpose of cooperatively deciding about intercepted operations of the target. We model the production of the decider program by a function *genDecider*. In our model, we capture that CliSeAu refines the functionality of the decider program of the cross-lining technique: CliSeAu's decider program includes, besides the decision-making functionality, also the functionality to start the instrumented agent code after starting the decision-making functionality. The decider program is stored in the JAR file format. The JAR file contains a set of key-value files that configure the coordinator, the start-up of the agent, and the local policy based on the given CoDSPL policy.

Definition 5.3. The *decider generation function* $\mathit{genDecider} : \mathit{POL} \times \mathcal{L}(\mathit{Id}) \rightharpoonup \mathcal{W}$ is defined as follows. Let $\mathit{pol} = (\mathit{ed}, \mathit{files}) \in \mathit{POL}$ be an arbitrary but fixed well-formed CoDSPL policy. Let $(\mathit{glob}, \mathit{Ids}, \mathit{loc}) = [\![\mathit{ed}]\!]$ and let $\mathit{id} \in \mathcal{L}(\mathit{Id})$ be arbitrary but fixed. The partial function *genDecider* is undefined for $(\mathit{pol}, \mathit{id})$ if $\mathit{id} \notin \mathit{Ids}$ holds and defined as follows otherwise.

$$\mathit{genDecider}(\mathit{pol}, \mathit{id}) = \mathit{jar}(\mathit{cdFixed} \oplus \mathit{cdConfigs})$$

where

- $\mathit{cdConfigs} : \mathcal{L}(\mathit{FileName}) \rightharpoonup \mathit{FileContent}$ maps the three configuration file names shown in Table 5.2 on the following page to the *pol*-dependent key-value content displayed in the very same table (other file names are not in the domain of *cdConfigs*),
- $\mathit{cdFixed} : \mathcal{L}(\mathit{FileName}) \rightharpoonup \mathit{FileContent}$ models the fixed (i.e., *pol*-independent) implementation of the coordinator component and the implementation of the start-up of the decider program, as a map from file names to the content of the implementation's Java bytecode,
- $\mathit{jar} : (\mathcal{L}(\mathit{FileName}) \rightharpoonup \mathit{FileContent}) \to \mathit{FileContent}$ models the creation of a JAR file from a list of files and their content, as for instance performed by the jar tool of the Java distribution (we leave this function underspecified, as its precise definition is not of particular relevance for the model). $\Diamond$

The modeled generation of the decider program, by function *genDecider*, faithfully captures by its signature that a decider program can be generated for each unit declared by a given CoDSPL policy. The production of a JAR file is captured by the *jar* function, whose co-domain is the content of a single file and whose domain is a partial map of file names to their content. This function's signature captures faithfully that a JAR file is actually an archive in which the content of multiple files is stored in association with the names of these files. The fixed implementation of the coordinator and its start-up is faithfully captured by the underspecified map *cdFixed*, which reflects that the implementation consists of multiple Java classes and, hence, multiple files. Finally, the configuration files faithfully captured by the map *cdConfigs*, which transfers all those properties of the CoDSPL policy's encapsulation description that are relevant for the unit at run-time to key-value files for the decider program – including, e.g., network addresses and excluding, e.g., the specification of security-relevant program operations.

key	**value**	**quantification**
$\boxed{\textbf{file } \texttt{coordinator.cfg}}$		
‘`identifier`’	id	
‘`policyConfigRes`’	`policy.cfg`	
‘`units`’	$glob($‘`units`’$)$	
‘`local−host`’	$loc(id,$ ‘`cor−host`’$)$	
‘`local−port`’	$loc(id,$ ‘`cor−port`’$)$	
‘`enforcer−host`’	$loc(id,$ ‘`enf−host`’$)$	
‘`enforcer−port`’	$loc(id,$ ‘`enf−port`’$)$	
‘`remote−host.`’ id'	$loc(id',$ ‘`ext−host`’$)$	for all $id' \in Ids$
‘`remote−port.`’ id'	$loc(id',$ ‘`ext−port`’$)$	for all $id' \in Ids$
‘`LogLevel`’	$loc(id,$ ‘`loglevel`’$)$	
‘`localPolicyClass`’	$loc(id,$ ‘`policy`’$)$	
$\boxed{\textbf{file } \texttt{target.cfg}}$		
‘`type`’	$loc(id,$ ‘`type`’$)$	
‘`JARfile`’	$loc(id,$ ‘`target`’$)$	
‘`JavaVM`’	$loc(id,$ ‘`target−javavm`’$)$	
$\boxed{\textbf{file } \texttt{policy.cfg}}$		
w	$loc(id,$ ‘`policy.`’ $w)$	for all $w \in \mathcal{W}$ such that $(id,$ ‘`policy.`’ $w) \in dom(loc)$

Table 5.2.: Content of configuration files in the decider program of unit $id \in Ids$ for CoDSPL policy $(ed, files)$ and $(glob, Ids, loc) = [\![ed]\!]$

Main definition Building on the previous definitions, we model the cross-lining implemented by CliSeAu as a function from CoDSPL policies to encapsulated targets. We model the encapsulated agents of an encapsulated target by tuples from the set $EA = \mathit{FileContent} \times \mathit{FileContent}$ whose first component captures the instrumented agent code and whose second component captures the decider program.

Definition 5.4. The *cross-lining implementation* by CliSeAu is the function $[\![\cdot]\!] : \mathit{POL} \to \mathcal{P}(EA)$ defined by

$$[\![(ed, \mathit{files})]\!] = \{\mathit{crossline}((ed, \mathit{files}), id) \mid (\mathit{glob}, \mathit{Ids}, \mathit{loc}) = [\![ed]\!] \land id \in \mathit{Ids}\}$$

where $\mathit{crossline} : \mathit{POL} \times \mathcal{L}(Id) \rightharpoonup EA$ with

$$\begin{aligned}\mathit{crossline}(\mathit{pol}, id) = (&\mathit{AspectJ}(\mathit{genAspect}(\mathit{pol}, id), \mathit{files}(\mathit{loc}(id, \text{‘}\mathtt{target}\text{’}))),\\ &\mathit{genDecider}(\mathit{pol}, id))\end{aligned}$$

for each $\mathit{pol} = (ed, \mathit{files}) \in \mathit{POL}$ and $(\mathit{glob}, \mathit{Ids}, \mathit{loc}) = [\![ed]\!]$ and each $id \in \mathit{Ids}$. For $id \notin \mathit{Ids}$, $(\mathit{pol}, id) \notin \mathit{dom}(\mathit{crossline})$. ◊

The definition captures the cross-lining implemented by CliSeAu as a combination of the decider generator and the agent instrumenter, as depicted in Section 5.2.

The model of CliSeAu's cross-lining implementation makes explicit how CliSeAu processes CoDSPL policies and instantiates the cross-lining technique for creating an encapsulated target. In particular, the model highlights two properties of how CliSeAu implements cross-lining. Firstly, the model shows how CliSeAu internally uses AOP tools but hides the specification of AOP aspects from the user of CliSeAu. By keeping the amount aspect-oriented specification limited to the pointcuts specified in CoDSPL only, a concern raised by Kästner, Apel, and Batory [KAB07] about the readability and understandability of aspect specifications do not apply to CliSeAu. Secondly, the model clarifies the start-up of an encapsulated agent, which is not subject of the cross-lining technique itself. Concretely, CliSeAu places the functionality of starting the instrumented code of the agent into the implementation of the decider program. In consequence, the decider program can ensure that the instrumented agent is started only after the decision-making functionality is active, such that even early security-relevant operations of the agent can be processed by the unit.

5.5. Low-Level Architecture and Implementation

In this section, we discuss detailed aspects about the low-level architecture and the implementation of CliSeAu. Firstly, we point out how the design of CliSeAu's two main components is implemented using software design patterns (Section 5.5.1). Secondly, we present technical design decisions behind the implementation of cooperation (Section 5.5.2) and the use of AOP tools (Section 5.5.3).

5.5.1. Mapping between Design and Implementation

CliSeAu consists of two main parts, the generic enforcement capsules and the encapsulation tool. We present the mapping between design and implementation for each part separately.

Generic enforcement capsules The components of generic ECs, shown in Figure 3.1 on page 28, are implemented as Java classes, interfaces, and methods. The interfaces between the components of generic ECs are implemented through methods and network sockets. The parametric components are part of CoDSPL policies and are discussed earlier in this thesis, in Section 3.4.3 on page 38. In the following, we discuss the three fixed components: the interceptor, the enforcer, and the coordinator.

The coordinator component is implemented by CliSeAu's Coordinator class. This class implements the "event delivery" interface and the "requests from other units" interface (both in Figure 3.1 on page 28) through two threads that concurrently listen on network sockets for incoming event objects and, respectively, delegation requests or delegation responses. The processing of incoming messages is performed in the handleSocket method, which uses the local policy and immediately sends out a result based on the local policy's output, without waiting for a response to the sent message. This form of communication allows the coordinator implementation to be stateless and to handle delegation requests even while other delegations have not completed, as described in Section 5.3.

The interceptor and enforcer components are implemented in two parts. The first part is the CoordinatorInterface class, which both components use. This class encapsulates the components' code for communicating with the coordinator component. That is, the class provides functionality for using the "event delivery" interface of the coordinator as well as functionality based on network sockets for providing the "decision delivery" interface to the coordinator. The second part of the interceptor and enforcer component implementation is an AspectJ around-advice template. This advice template mainly consists of a block of Java code, of which the first part implements the main interceptor functionality and the second part implements the main enforcer functionality. The template contains placeholders for the names of the Java classes providing the concrete event factory and enforcer factory implementations. These placeholders are instantiated by the encapsulation tool. Being implemented as an around-advice, the interceptor and enforcer establish the interfaces to the agent as devised in the unit architecture: Firstly, as an around-advice, the components run sequentially to the agent and run instead of the respective security-relevant operations. This establishes the "blocking" and "unblocking" interfaces to the agent. Secondly, AspectJ provides around-advice with information about the security-relevant operations, such as the actual parameters of a method call, which establishes the "observation" interface. Finally, around-advice may or may not invoke the security-relevant operation and may run further code that modifies the state of the agent, which altogether establishes an "intervention" interface to the agent.

Encapsulation tool The components of CliSeAu's encapsulation tool, shown in Figure 5.2 on page 59, are implemented as Java classes and interfaces. The interfaces between components are realized with methods.

The policy provider component is implemented by the PropertiesConfig class. This class is part of a low-level architecture, which is shown in Figure 5.4. It consists of two main parts: The first part are abstract strategies for retrieving unit-specific information from a CoDSPL policy's encapsulation description (top part of the figure). The second part are abstract and concrete factories that create concrete strategies for the retrieval of the information (bottom part of the figure).

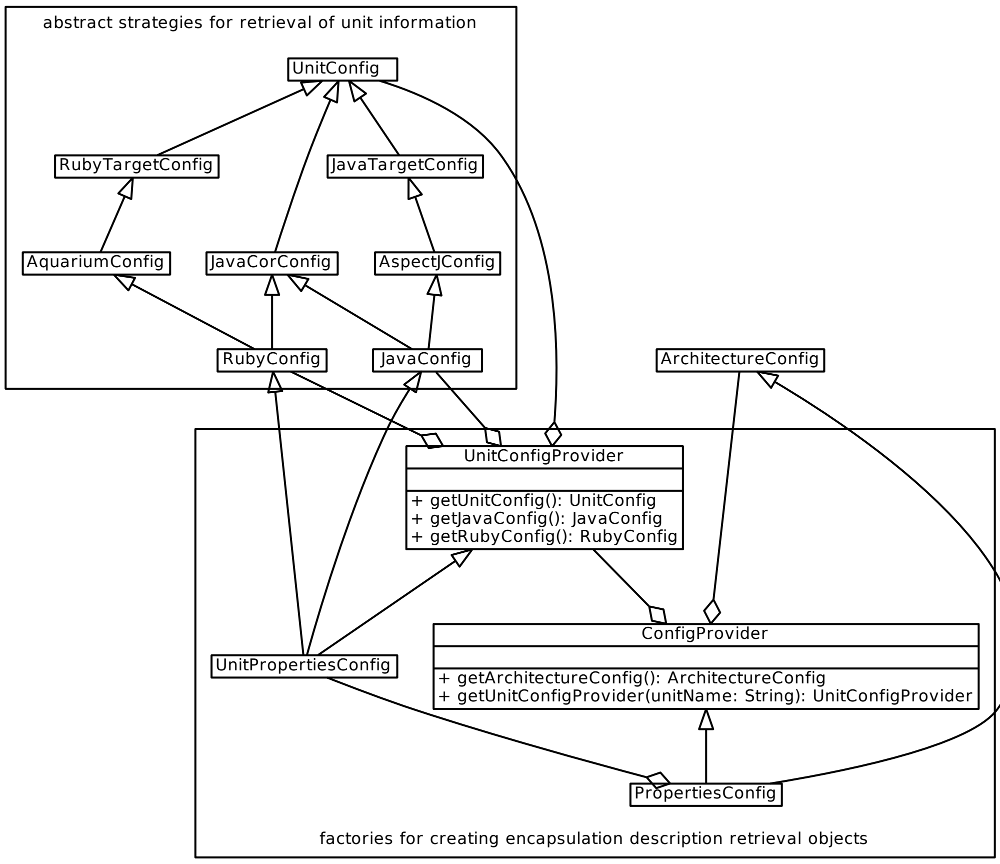

abstract strategies for retrieval of unit information
UnitConfig
RubyTargetConfig
JavaTargetConfig
AquariumConfig
JavaCorConfig
AspectJConfig
RubyConfig
JavaConfig
ArchitectureConfig
UnitConfigProvider
+ getUnitConfig(): UnitConfig
+ getJavaConfig(): JavaConfig
+ getRubyConfig(): RubyConfig
ConfigProvider
+ getArchitectureConfig(): ArchitectureConfig
+ getUnitConfigProvider(unitName: String): UnitConfigProvider
UnitPropertiesConfig
PropertiesConfig
factories for creating encapsulation description retrieval objects

Figure 5.4.: Low-level architecture for the policy provider component

The abstract strategies are realized as a hierarchy of Java interfaces. The root of this hierarchy, UnitConfig, provides methods for obtaining basic unit information from an encapsulation description. Towards the leaves, JavaConfig and RubyConfig, the interfaces provide further methods for obtaining more specific unit information. The intermediate interfaces provide methods for specific details like whether the coordinator is implemented in Java (JavaCorConfig) or what instrumentation tool is used (AspectJConfig and AquariumConfig). The interfaces are part of a Strategy pattern, in which the encapsulation tool provides one concrete strategy, implemented in UnitPropertiesConfig, that provides unit information from encapsulation descriptions given in the Java Properties format (as described in Section 3.4.3 on page 38. This separates concerns of obtaining unit information from encapsulation descriptions and using this information. The interfaces are used by clients according to the Constructor Injection pattern, i.e., the clients get a concrete strategy object passed via the clients' constructors. Concretely, clients only get the most narrow interface from the UnitConfig hierarchy, such that through the use of the pattern, the fine-granular hierarchy of interfaces realizes a need-to-know principle in the code.

The creation of encapsulation description retrieval objects is realized in an Abstract Factory pattern. The Java interfaces ConfigProvider and UnitConfigProvider constitute the pattern's abstract factories. They declare methods for producing objects that provide global and, respectively, unit-specific information of a CoDSPL encapsulation description. The Java classes PropertiesConfig and UnitPropertiesConfig constitute corresponding concrete factories for obtaining the information from encapsulation descriptions provided in Java's Properties format. These two concrete factories implement the interfaces for retrieval of unit information and, respectively, of ArchitectureConfig, themselves such that their factory methods simply return the concrete factory itself. This realization of the Abstract Factory pattern keeps the retrieval code bundled in two classes while still offering the discussed advantages of the fine-grained architecture.

The generic EC implementation component of the encapsulation tool consists of two kinds of artifacts: compiled Java classes and Java source code templates. The compiled classes capture the fixed components of the generic EC. These are in the format of Java bytecode and stored together in a JAR file, `CliSeAuRT.jar`. The Java source code templates contain placeholders for parameters whole values are obtained from the encapsulation description given to the encapsulation tool. For instantiating the templates, the encapsulation tool uses the stringtemplate library, and for turning the instantiated templates into executable code, the encapsulation tool uses the standard Java compiler, javac. An alternative design to the choice of source code templates would be to instead use compiled Java code and use parameters that expect Java class names through Java's reflection API. The approach implemented in the encapsulation tool has the advantage that parameter validation is anticipated from run-time of the encapsulated agent to the time of encapsulation.

The remaining components of the encapsulation tool's architecture – the decider generator, the agent instrumenter, and the encapsulator – are realized by the low-level architecture shown in Figure 5.5. The functionality of the decider generator is implemented by the UnitStartupCreatorJavaCor class. The agent instrumenter is implemented by the AspectWeaver class (for instrumenting Java agents) and the AquariumWeaver class (for instrumenting Ruby agents). The encapsulator is implemented by several Java classes, including the Encapsulate class, which contains the entry point of the encapsulation

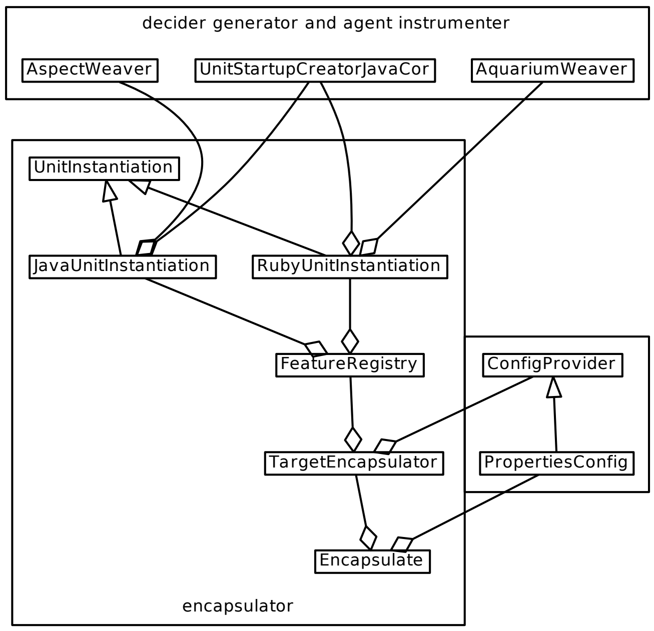


Figure 5.5.: Low-level architecture of decider generator, agent instrumenter, and encapsulator

tool (i.e., its main method). The Encapsulate class uses the PropertiesConfig class for obtaining the encapsulation description specified via the command line. Moreover, the Encapsulate class uses the TargetEncapsulator class for encapsulating the individual agents of the target specified in the encapsulation description. The TargetEncapsulator class, in turn, uses the FeatureRegistry class, which provides a factory method for producing UnitInstantiation objects (JavaUnitInstantiation for a Java agent and RubyUnitInstantiation for a Ruby agent). This architecture defines a clearly identified place for agents' language extensions (the FeatureRegistry) and achieves separation of concerns in two directions: Firstly, the TargetEncapsulator depends only on ConfigProvider rather than the storage-format specific PropertiesConfig. Secondly, the TargetEncapsulator depends only on UnitInstantiation rather than the language-specific implementations.

5.5.2. Cooperation

Coordinator implementation The coordinator component of CliSeAu's generic ECs is implemented as a Java class named Coordinator. The implementation enables the courses of action for cooperative deciding introduced in Section 5.3, implements the coordinator's share of the semantics of CoDSPL, and hides the technical details of network communication between components from the local policy component. Listing 5.3 on the following page shows the essential code of the implementation of the coordinator. The implementation handles a variety of possible exceptions in the network communication, but for the sake of brevity this exception handling is not included in the listing. The listing shows

```
void handleSocket(ServerSocket socket, DataEncoding sockEncoding) {
  while (true) {
    Socket connection = socket.accept();
    Object input = readObject(logger, connection, sockEncoding);
    LocalPolicyResponse resp;
    synchronized (localPolicy) {
      if (isLocal) {
        Event event = (Event) input;
        resp = localPolicy.localRequest(event);
      } else {
        DelegationReqResp delReqResp = (DelegationReqResp) input;
        resp = localPolicy.remoteRequest(delReqResp);
      }
    }
    if (resp instanceof Decision) {
      Socket out_connection = addressing.connectLocalEnforcer();
      localEncoding.write(out_connection.getOutputStream(), resp);
      out_connection.close();
    } else if (resp instanceof DelegationLocPolReturn) {
      DelegationLocPolReturn del = (DelegationLocPolReturn)resp;
      Socket out_connection = addressing.connectRemote(del.getDestinationId());
      cooperationEncoding.write(out_connection.getOutputStream(), del.getDelReqResp());
      out_connection.close();
    }
  }
}
```

Listing 5.3.: Essential code of the coordinator implementation

the handleSocket method, which is run in parallel by the Coordinator for a network socket listening to the interceptor component and, respectively, a network socket listening to input from other units. In Lines 3 to 4, the implementation reads the three possible inputs of the cooperative deciding activity. Subsequently, Lines 9 and 12 use the local policy for determining whether and where delegation shall be performed. Depending on the local policy's result, the coordinator then in Lines 17 and 22 sends a decision object or a delegation request or response to the enforcer or, respectively, a unit specified by the local policy. Overall, this implementation enables the five courses of action described in Section 5.3 as well as four additional cases (event to response, response to response, response to request, and request to decision) which the implementation allows as they are neither considered intuitively useful nor harmful. In particular, the implementation does not block after sending out a delegation request until a corresponding response is received. This allows the coordinator to continue cooperating, possibly with multiple other units, even when a locally intercepted operation is awaiting a delegation response. Moreover, the implementation realizes the interaction between the coordinator and the local policy specified by the semantics of CoDSPL (see Section 3.5 on page 43). Finally, the implementation takes over and hides the network communication from the local policy.

Network connection management CliSeAu's generic ECs establish network connections on demand and with the help of classes that form an Abstract Factory pattern. The generic ECs use these classes both for the network connections between the components within a unit as well as for the network connections between units. Establishing network connections on demand rather than establishing them once and then reusing them serves the purpose of simplicity: it renders checks for whether the remote end has the connection opened – for which there is no direct way in the Java API – unnecessary. The Abstract Factory pattern used by CliSeAu's generic ECs augments the Java API, which provides abstract factories for client-side as well as server-side network sockets but concrete factories only for SSL sockets. CliSeAu's generic ECs augment the Java API by concrete factories for plain, i.e., non-SSL TCP sockets, PlainSocketFactory and PlainServerSocketFactory. The generic ECs encapsulates the selection between the SSL and non-SSL factories based on the respective CoDSPL policy's '`cfg.crypto`' specification in the MetaSocketFactory class. In particular, the MetaSocketFactory produces concrete factories for server-side SSL network sockets that, upon creation, have client authentication enabled. Enabling client authentication improves on the Java default, which is to only enable server authentication. Note that, in the excerpt of the implementation of the coordinator component in Listing 5.3, the use of the network socket factories is encapsulated in the object stored in the addressing field of the coordinator object.

Encryption of cooperation CliSeAu supports encrypted, authenticated network communication for the cooperation among units. For each unit, the encapsulation tool generates a private key and a cryptographic certificate. Moreover, the encapsulation tool generates a single root certificate and uses the private key corresponding to the root certificate to sign the units' certificates. The encapsulation tool provides each generated unit with its own certificate and private key as well as with the root certificate. At run-time, each unit uses its certificate to authenticate itself against other units in the cooperation via SSL network sockets. Using the root certificate, each unit can validate the authenticity of another unit's certificate. By following this approach, CliSeAu provides encrypted and authenticated network communication while keeping the details about certificates and cryptographic keys transparent to the user of CliSeAu.

CliSeAu's encapsulation tool generates public/private key-pairs using Java's crypto API, particularly its KeyGenerator class, and generates cryptographic certificates using *BouncyCastle*, a "lightweight cryptography API for Java and C#" [Bou]. The public/private key-pairs are RSA keys of 2048 bit size. This key size follows recent recommendations by NIST [BD15, Table 2-1]. The generated certificates are X.509 certificates using the "SHA256withRSA" algorithm, which uses the padding scheme PKCS #1 v1.5 [Eas05, Section 2.3.2] and follows recent recommendations by NIST [BD15, Table 2-2].

The support of encrypted cooperation among units allows one to use CliSeAu for effectively enforcing security even in presence of attackers who have access to the network communication between units. More concretely, the encryption prevents attackers from forging delegation requests and delegation responses and thereby hindering and otherwise effective enforcement. The encryption also prevents attackers from learning potential secrets about the target by eavesdropping on the communication among units.

Encoding of information CoDSPL policies allows the specification of the encoding that units shall use for transmitting in-memory objects, namely event objects, delegation requests and responses, and decision objects over network connections. The language supports two encodings: the native serialization of the respective agent's programming language and the JSON format. In CliSeAu, this encoding is used on top of the encoding of the network sockets used (i.e., either TCP or SSL-over-TCP). CliSeAu supports Java's serialization of objects via ObjectOutputStream and ObjectInputStream, and supports the JSON format via Google's GSON library [Goob].

For realizing the selectable encoding, the low-level architecture of units includes a Java interface DataEncoding that provides methods for reading Java objects via a stream object and for writing Java objects via a stream object. This interface is implemented by the supported encodings, in the Java classes SerializationEncoding and GsonEncoding. CliSeAu's encapsulation tool, when generating units, instantiates the generic EC such that encoding objects of the right class are created. This design, on the one hand, hides details about the concrete encodings from the clients using the classes and, on the other hand, enables future extensions by further encodings without requiring changes to the architecture.

5.5.3. Instrumentation by AOP Tools

Instrumentation for Java agents using AspectJ The encapsulation tool implements the instrumentation of Java agents in its AspectWeaver class. At the implementation level, three aspects are noteworthy. Firstly, AspectWeaver invokes the command-line tool ajc of AspectJ, providing it the JAR file containing the respective agent's code as well as an AspectJ aspect that encodes the interceptor and enforcer components. This way the encapsulation tool hides details of AspectJ from users of CliSeAu. Secondly, AspectWeaver copies all the dependencies to a dedicated directory inside the directory into which the instrumented JAR file is placed. This allows a user of CliSeAu to simply copy the agent's directory to the machine from which it shall be run. Thirdly, AspectWeaver includes all the dependencies (i.e., JAR files) specified in the encapsulation description into the JAR file containing the instrumented agent, by modifying the classpath setting in the JAR's manifest file. This way, a user of CliSeAu need not specify paths to dependencies upon invocation of an agent. The dependencies are specified using relative paths to the subdirectory in which the dependencies are copied. This enables users of CliSeAu to copy instrumented agents even to a different directory than the one in which the instrumented agent is placed by the encapsulation tool.

Instrumentation for Ruby agents using Aquarium The encapsulation tool implements the instrumentation of Ruby agents in its AquariumWeaver class. At the implementation level, two aspects are noteworthy. Firstly, AquariumWeaver copies the whole code directory of the Ruby agent to a dedicated place. This allows modifications of the code without changing the original code and, thereby, allows the same code to be modified differently for different agents. Secondly, AquariumWeaver instruments the Ruby code of an agent by inserting two lines into the Ruby code, a require_relative and an include directive for loading and activating the respective aspect when the Ruby code is run. With the Aquarium framework, this suffices for activating the aspect at run-time at the specified join points.

5.6. Analytic Evaluation

We evaluated the design and implementation of CliSeAu in three directions: software design, code quality, and code documentation. Regarding software design, we investigate in Section 5.6.1 the object-oriented design and software design patterns used for realizing the generic ECs. Results from analyzing code quality and code documentation of CliSeAu are presented in Section 5.6.2 and Section 5.6.3, respectively.

5.6.1. Object-oriented Design in the Low-level Architecture

The architecture of CliSeAu follows principles of object-oriented design for enabling a generic architecture of units (the generic ECs) as well as for generally managing the complexity of the implementation. Through reducing conceptual complexity of individual components in the design, we particularly aim at better verifiability of CliSeAu through manual code review.

For the design of the generic ECs, software design patterns [GHJ+95] were used for combining the fixed components (interceptor, coordinator, and enforcer) with instances of the parametric components (event factory, enforcer factory, and local policy). Concretely, the Abstract Factory pattern is used for separating the (fixed) interceptor component from the concrete, policy-specific event objects. Analogously, the pattern is also used for separating the (fixed) enforcer component from the concrete, policy-specific decision objects and countermeasure objects. This allows the policy-specific objects to vary in their structure without requiring the fixed components to know this structure in advance. Finally, the Strategy pattern is used for separating the fixed coordinator component from the concrete, policy-specific algorithm for decision-making and delegation that is encapsulated by a local policy component. Again, this allows the algorithm for decision-making and delegation to vary independently from the coordinator component that uses the algorithm. For all three parametric components, the use of the design patterns enables a loose coupling to the fixed components by which the fixed components do not impose constraints on the parametric components. Thereby, the design preserves the expressiveness of CoDSPL and, hence, supports Requirement (Req-1).

In the design of the encapsulation tool, software design patterns are used primarily for a separation between the implementation of CoDSPL policies and the components that use parts of policies. Concretely, the Abstract Factory pattern as well as the Strategy pattern are used. The latter is used for obtaining particular elements of a CoDSPL policy independent of syntax-specific aspects (such as the key-value syntax of encapsulation descriptions). The former is used for separating the creation of the syntax-specific strategy objects. This design choice addresses Requirement (Req-2) by reducing the complexity in the implementation of components that use parts of policies and, secondarily, simplifies future enhancements to the policy language.

The design of CliSeAu's generic ECs is based on the service automata concept whose modular architecture and collaboration between its components constitute the key abstractions and mechanisms [BME+07, pp. 138–144] of the problem domain of a generic enforcement mechanism. An architectural documentation of CliSeAu is part of this thesis, based on UML component diagrams and activity diagrams for the high-level architectures and based on class diagrams for the low-level architecture [BME+07, pp. 320–322].

5.6.2. Code Quality

In the development of CliSeAu, we used the static code analysis tools FindBugs and PMD for improving the quality and thereby reliability of the implementation. We report on the results of analyzing the code of CliSeAu with the two analysis tools in the following.

We analyzed the code of CliSeAu with FindBugs version 3.0.1, the most recent version at the time of writing. For the hand-written code of CliSeAu, the analysis reported four potential problems, all of them in the code of CliSeAu's encapsulation tool and all of them unchecked type casts that FindBugs considers possible to fail. We could manually identify all four cases as false positives, as the program logic ensures that the type cast does not fail. CliSeAu's encapsulation tool also contains automatically generated code by the javacc parser generator for parsing specification of security-relevant operations of a CoDSPL policy. For this code, the analysis discovered 119 findings. None of the findings are classified by FindBugs as "troubling" and we assume that the input to javacc could not be improved for better parser code. Overall, the results of the analysis with FindBugs show that the code of CliSeAu's generic ECs did not trigger any findings. The findings in CliSeAu's encapsulation tool appear to be either false positives or, in the case of the parser code, unproblematic.

We analyzed the code of CliSeAu with PMD version 5.5.4, the most recent version at the time of writing. PMD includes a variety of so-called analysis rules for Java code which can be enabled/disabled and whose configuration can be customized. For our analysis, we used the set of all rules for Java code, from which we excluded the following: rule sets for Android, J2EE, and JavaBeans (they do not apply to CliSeAu); controversial rules (these rules are labeled controversial by PMD); a rule for the "Law of Demeter" (a controversial software design guide [Lie04] that is not labeled as such in PMD); rules limiting the length of variable names and comments (we prefer expressive names) and a rule for names of abstract classes (this rule is inconsistent with the naming used by the Java API).

The analysis of CliSeAu's code with PMD yielded 17 findings. All findings were carefully checked whether they actually constitute bad design or implementation and were manually confirmed to be spurious warnings by PMD. To account for the variety of different kinds of findings, we justify the individual kinds of findings in Table 5.4 on the following page. Each row of the table corresponds to a kind of finding. The black horizontal lines divide the rows into three groups of findings relating to efficiency, to code structure, and to logging. The columns of the table provide the concrete message of PMD for the finding, the number of times the finding occurred, the CliSeAu components in which the finding occurred, and our justification for why the respective finding is unproblematic.

During the development of CliSeAu, we used both analysis tools after the development of new features and code refactorings. The intermediate analyses during the development oftentimes produced several hundreds of findings that revealed incorrect, inefficient, as well as otherwise badly designed code. Some of the findings revealed mistakes in corner cases that might not have been revealed through test cases. Because of these experiences, we are confident that we applied the tools correctly and that the tools could indeed discover problematic code in CliSeAu.

Based on the results of both analysis tools, we are confident that the code of CliSeAu is of good quality. By making sure that the analysis tools do not reveal true findings of incorrect or inefficient code, we address Requirements (Req-3) and (Req-4). In particular, we expect that newly developed and previously untested CoDSPL policies do not trigger unexpected behavior in CliSeAu.

5.6.3. Code Style and Documentation

In the development of CliSeAu, we used a uniform code formatting according to a widely used coding style and specified a complete Javadoc documentation for CliSeAu's source code. We used two tools, Checkstyle and Java's javadoc, to find and remove violations of the code style and documentation. In the following, we report on the results of analyzing the code of CliSeAu with these two tools.

We conducted an analysis of the source code of CliSeAu with Checkstyle version 7.6, at the time of writing the most recent stable version, and analyzed against the Google Java Style Guide. The result of the analysis is that CliSeAu fulfills all checks performed by Checkstyle except for five findings: Four of the findings are lines longer than 100 characters, which we deem acceptable as these lines are code comments containing long hyperlinks that could not be broken into several lines. The fifth finding indicates a wrong level of indentation but appears to be a false positive that might be caused by an improperly supported language feature introduced in Java 7. That is, all source code of CliSeAu, including the encapsulation tool as well as the generic ECs uniformly follows the Google Java Style Guide.

The code of CliSeAu includes a complete Javadoc documentation of all classes and methods, including documentation of formal parameters, return values, and thrown

finding	#	components	justification
“Avoid instantiating new objects inside loops”	2	encapsulation tool	The individual objects are placed into a data structure for use outside the loop.
“A switch with less than three branches is inefficient”	1	encapsulation tool	The switch-statement was chosen to indicate possible future extensions by further branches (the statement is executed once per unit).
“StringBuffer constructor is initialized with size 16, but has at least 36 characters appended”	2	encapsulation tool	Spurious warning: The initialization is with size “4 * 50”, i.e., more than 36 characters.
“Possible God class”	1	encapsulation tool	The class contains many but mostly simple policy reading methods.
“The method [...] has a (Standard) Cyclomatic Complexity of 10”	2	encapsulation tool	Spurious warning: The method has 12 SLOC but many code paths in a switch-statement with 10 simple branches.
“The class [...] has a (Standard) Cyclomatic Complexity of 3 (Highest = 10)”	2	encapsulation tool	This complexity is solely due to the method causing the finding in the previous row.
“Consider simply returning the value vs storing it in local variable”	2	encapsulation tool, generic EC	Spurious warning: The local variables store values that cannot directly be returned.
“Overriding method merely calls super”	1	encapsulation tool	A code comment explains this deliberate choice.
“There is log block not surrounded by if”	2	generic EC	Spurious warning: A conditional using method isLoggable exists, but PMD falsely does not check for this method.
“The Logger variable declaration does not contain the static and final modifiers”	1	generic EC	The logger depends on the non-static unit identifier.
“Class contains more than one logger.”	1	generic EC	There is one logger for each of the two coordinator threads.

Table 5.4.: Findings by PMD upon analysing the CliSeAu implementation

CliSeAu component	files	SLOC	comment	license	empty	total
encapsulation tool	51	2003	1897	1024	1519	6443
generic ECs	25	852	945	500	604	2901
shared code	1	18	20	20	15	73
total	77	2873	2862	1544	2138	9417

Table 5.5.: Source code lines of the CliSeAu implementation

exceptions. The Javadoc documentation is confirmed to be complete by a run of the javadoc tool on the source code as well as by a run of the Checkstyle analysis tool. Both tools did not report missing documentation or inconsistencies between documentation and code, e.g., with regard to lists of formal parameters, return types, or exceptions.

For a complete picture of the code documentation, subsuming Javadoc as well as in-line comments, we counted the lines of source code (*SLOC*), lines of comment, license block lines, and empty lines. Table 5.5 summarizes the results. For the number of files as well as for each kind of lines, the table contains a column. For each component of CliSeAu, i.e., the encapsulation tool, the generic ECs, as well as the small amount of shared code between both parts, the table contains a row. The code lines were counted using the sloccount tool in its most recent version 2.26. Since each code file of CliSeAu begins with a license header, all lines in the first multi-line comment block were counted as license block lines. Empty lines were counted using the grep tool applied to a simple regular expression filter that also includes empty lines in multi-line comments. As comment lines, we counted all remaining lines. In the 51 files of the encapsulation tool, the 2003 SLOC are supplemented by 1897 lines of comments. The generic ECs are implemented in 25 files, consisting of 852 SLOC and 945 lines of comments. Altogether, the 2873 lines of CliSeAu's Java source code and 2862 lines of code documentation are very balanced.

The reported results of the tools confirm that the code follows an established code style and that the code of CliSeAu is quantitatively well-documented. This evidence was complemented by personal discussions with students who worked with CliSeAu and confirmed the code to be well-documented also qualitatively, most notably Hamann [Ham16], Schickel [Sch16], and Tiedje [Tie15]. Overall, we take these results as strong evidence that the implementation of CliSeAu, including the generic ECs as well as the encapsulation tool, is well-maintainable and understandable and, in this regard, also supports verifiability of the implementation (Requirement (Req-2)).

5.7. Summary

We introduced CliSeAu, a tool for enforcing security requirements in distributed programs. CliSeAu consists of generic enforcement capsules, parametric implementations of units for generic enforcement mechanisms, and the encapsulation tool, which instantiates and applies the generic enforcement capsules for a given CoDSPL policy and distributed program. For both parts of CliSeAu, the chapter presented the design as well as selected low-level architectural and implementation aspects. We analyzed the use of software design patterns in CliSeAu and applied static code analysis tools to check code quality, uniform code style, and documentation.

CliSeAu generates enforcement mechanisms for given CoDSPL policies by instantiating the generic enforcement capsules, which are parametric in the operational specifications of CoDSPL, and applying the cross-lining technique for instrumenting a given Java or Ruby target. The AOP tools used internally by CliSeAu allows CliSeAu to be applied to Java or Ruby programs that have not specifically designed for operating under an enforcement mechanism. CliSeAu inherits from CoDSPL that the units of generated enforcement mechanisms can make local decisions and can also cooperate by means of delegation. That is, CliSeAu allows for enforcing a wide range of security requirements on a wide range of Java and Ruby programs, as specified in Requirement (Req-1). CliSeAu is not the first generic enforcement mechanism for distributed programs [MU00; SVA+04; OBM10; KP15]. However, CliSeAu is the first that allows policies to specify cooperation to take place asynchronously to the communication of the target and is the first that can automatically be applied to a distributed Java or Ruby program.

The design of both the generic EC and the encapsulation tool of CliSeAu follows the principles of object-oriented design [BME+07; GHJ+95]. In particular, the design employs software design patterns for decoupling the fixed components of CliSeAu's generic enforcement capsules and the parametric components that are specified through CoDSPL policies. CliSeAu's design furthermore separates the concerns of CoDSPL syntax and the use of information contained in a policy by client components . The modular architectures of CliSeAu simplify the understanding and analysis of CliSeAu [BME+07, pp. 13–19] and thereby support verifiability (Requirement (Req-2)). We are not aware of other works applying principles of object-oriented design to generic enforcement mechanisms.

Our analysis of the CliSeAu implementation with the tools FindBugs, PMD, Checkstyle, and Javadoc with a manual inspection of the results revealed no problematic findings with regard to code quality, code style, and code documentation. Since FindBugs and PMD check code quality also with regard to incorrect and inefficient code, the result of our analysis support effectiveness and efficiency (Requirements (Req-3) and (Req-4)) for the fixed components of CliSeAu's generic ECs. Code style and documentation simplify the analysis of CliSeAu and thereby support verifiability (Requirement (Req-2)). We are not aware of other enforcement mechanisms whose implementation has been subjected to code analysis tools for ensuring code quality.

The design and implementation of the coordinator component of CliSeAu's generic ECs performs a non-blocking cooperation. That is, upon delegating the decision-making for an event, the coordinator does not block until it receives a delegation response but rather accepts delegation requests from other units in the meantime. Non-blocking cooperation enables a unit to concurrently contribute to the decision-making of multiple events. This avoids latencies and deadlocks through blocking and improves on the efficiency of enforcement mechanisms generated by CliSeAu (supporting Requirement (Req-4)).

In the context of this thesis, CliSeAu is used for encapsulating concrete distributed programs with concrete CoDSPL policies in the case studies of Chapters 6 and 7. Moreover, the formal model in Chapter 8 captures the high-level architecture of CliSeAu and the cooperation pursued by CliSeAu to enable formal verification of cooperative enforcement.

Chapter

6

Modular Delegation-Based Security Policies

6.1. Introduction

Cooperation among the units of a distributed enforcement mechanism enables the mechanism to coordinate the enforcement when units' locally available information does not suffice for enforcing security. This benefit of cooperation, however, comes at the expense of an increased complexity of the individual units: A unit has to simultaneously ensure a secure behavior of an agent and to interact with other units.

We provide an identification of concerns in cooperative enforcement of security by delegation. Based on this identification, we propose a modular design for a separation of concerns in CoDSPL and an implementation of this design as an extension library for CoDSPL. Technically, the extension library is realized as a Java package that can be used by CoDSPL policies without changes to the syntax and semantics of CoDSPL and without changes to CliSeAu. We demonstrate the applicability of the proposed design and extension library in a case study, in which we enforce compliance with users' privacy policies in the popular decentralized online social network Diaspora* [GSS+]. As part of the case study, we present a CoDSPL policy, called CReDiC, that enforces compliance with privacy policies based on the extension library and provide an empirical evaluation of CReDiC's effectiveness and efficiency.

Our modular design for CoDSPL can be adopted by CoDSPL policies such that fewer architectural design decisions remain to be made when a CoDSPL policy is specified. Moreover, by separating concerns, our modular design also facilitates mastering the potential complexity of a CoDSPL policy [BME+07, pp. 13–14]. CReDiC, on the one hand, provides an example for specifying a complex CoDSPL policy in a modular fashion and, on the other hand, provides a solution for controlled re-sharing in decentralized online social networks that allows the users of such social networks to expand the outreach of their messages in a controlled fashion. Our empirical evaluation shows that CReDiC is effective and introduces only minor run-time overhead into the encapsulated Diaspora* system.

Structure The remainder of this chapter is structured as follows. In Section 6.2, we identify the concerns of delegation-based local policies. The design and implementation of the modular extension library for local policies in CoDSPL based on the identified concerns is described in Section 6.3. Section 6.4 provides the case study of controlled sharing and re-sharing in decentralized online social networks. Section 6.5 summarizes the contributions presented in this chapter.

6.2. Concerns of Delegation-based Local Policies

Delegation enables a unit to cooperate with other units when the information available at the former unit does not suffice to make a decision locally. Delegating and contributing to the decision-making are not mutually exclusive: Before delegating, a unit might already use locally available information (i.e., local state) for performing parts of the decision-making. In this case, the unit delegates only the remaining parts of the decision-making. A delegate unit, in turn, can perform the same two steps – making a partial decision and further delegating the remainder. Delegation-based decision-making can, thus, involve a sequence of steps until a decision is made.

In the step-wise decision-making, each participating unit accomplishes a *subgoal* in the decision-making by either making a decision or by delegating a request that is simpler – in the sense of requiring less information or fewer computational resources – to accomplish than the decision-making prior to the delegation. We illustrate the step-wise decision-making with delegation based on the following example.

Example 6.1. System-6.1 is a distributed program consisting of several connected services through which users can exchange files. Each user of System-6.1 can connect to System-6.1 through only one of the services via her web browser. From her browser, the user can upload own files, can view files of other users, and can store copies of other users' files in her own directory at the service. The security requirement on System-6.1 is: A user may upload a file only if the upload does not exceed her storage quota at the service; A user may only make a copy of another user's file if the former user's storage quota are not exceeded by the copy and if the latter user declared the file to be approved for copying.

The enforcement mechanism that enforces the security requirement consists of one unit for each service of System-6.1. When a copy operation is intercepted by a unit, the unit at the respective service proceeds in the following steps: first, checking whether the copy would violate the user's storage quota; second, determining whether the file is stored at another service and at which one; third, delegating the decision-making for the intercepted operation to the unit encapsulating this other service; fourth, if the copy is permitted, the intercepting unit updates the user's amount of available storage space. When an upload is intercepted, only the first and the last step are performed. ◊

In the example, a unit of the enforcement mechanism can perform the decision-making for a copy operation alone if the two involved users (the source user for the file and the user who wants to copy the file) use the same service of the distributed program. The unit can also perform the decision-making alone if the storage quota of the copying user would be exceeded through the copy. Otherwise, the intercepting unit can first make a partial decision ("storage quota not exceeded") and then delegate the remainder to the unit for the service of the source user. A more elaborate variant of this example is used in the case study conducted in Section 6.4 of this chapter. The example illustrates how decision-making for a single operation can consist of multiple steps that may be performed by the same unit and may involve delegation.

We propose to separate the concerns in delegation-based local policies in order to reduce the complexity in the design and implementation of local policies.

Definition 6.1. We distinguish the following concerns of delegation-based local policies:

(a) *selection of decision-making subgoals*: Which subgoals exist and how is the next subgoal for the overall goal of making a decision selected for an intercepted operation, delegation request, or delegation response?
(b) *realization of decision-making subgoals*: How is the next subgoal accomplished, by means of using using available information and either making a decision or delegating?
(c) *state-keeping*: How is the state of an individual local policy updated upon accomplishing a subgoal?
(d) *routing*: How are delegation requests routed in a network of units to selected delegates, for instance to account for network topology? ◊

Separating the concerns of delegation-based local policies leads to local policies of reduced complexity. The separation between the selection and realization of decision-making subgoals reduces the complexity particularly in case of local policies with several subgoals and a non-trivial workflow for achieving the subgoals. In this case, the separation between selection and realization simplifies local policies by making the subgoals and their chronological orderings explicit and by reducing the conceptual complexity of the realization of separated, individual subgoals. Separating the decision-making concerns from the state-keeping simplifies subgoal realization by separating how state is used from how state is modified. Separating the selection of delegates from routing simplifies the delegation, as network topology can be neglected in the design of the latter.

Example 6.2. Consider the enforcement mechanism in Example 6.1. We can identify two subgoals in the description of the mechanism's decision-making for copy operations by users: firstly, excluding quota violations; secondly, ensuring that the file is approved for copying by the owner of the file. The subgoal selection is performed sequentially in the listed order of the subgoals. For realizing the first subgoal, the unit compares locally the size of the file against the available storage space of the user. For realizing the second subgoal, the unit delegates to the unit of the file's source service if this is a different unit; Otherwise or if the unit is already the unit of the source service, the unit checks locally whether copying the file is approved. State-keeping is performed only when a decision was made, is performed by the intercepting unit, and consists of reducing the amount of the user's available storage space. Routing is not specified, so delegates might directly be reachable in the network of units.

Note that depending on how the services are realized, an additional subgoal might improve the separation: Concretely, the source service might not immediately known for a file, for instance because the file has been propagated several times before. In this case, determining the source service can constitute an independent subgoal. ◊

Note that CoDSPL policies can specify local policy components that employ delegation by using their interface to the coordinator component (see Figure 3.1 on page 28). How a local policy determines when and how delegation shall take place and how it integrates with the state-keeping and local decision-making is not specified by CoDSPL. This keeps the policy language small and provides the individual CoDSPL policies with the flexibility to structure their local policies tailored for the respective application scenario.

6.3. Separation of Concerns in CoDSPL

In this section, we present the architecture and implementation of an extension library for modular local policies in CoDSPL. The extension library is implemented as a Java package that is part of CliSeAu but is not mandatory to be used by CoDSPL policies. The goal of the extension library is to enable the specification of local policies in a modular fashion that separates the concerns identified in Section 6.2.

The central concept of the extension library are micro-policies. A *micro-policy* is a component for realizing a particular subgoal in the decision-making for a security requirement as well as for performing the state-keeping corresponding to the respective subgoal. The subgoal selection, i.e., the selection of a micro-policy for a particular intercepted program operation or received delegation, is performed by another component of the extension library, the *micro-policy factory*. Finally, the extension library includes a *routing policy*, a component for determining next units on paths to a particular delegate unit.

Figure 6.1 on the next page shows the low-level architecture of the extension library. For micro-policies, the extension library provides the MicroPolicy interface. A local policy based on the extension library can provide multiple implementations of this interface, one for each logical subgoal to be accomplished by the local policy. For micro-policy factories, the extension library provides the MicroPolicyFactory interface. Both interfaces are defined as Java generics, i.e., parametric types: The interfaces are parametric in a type encapsulating local policy state (StateT) and a type providing an interface for reading state information (StateReaderT). For routing policies, the extension library provides the RoutingPolicy interface. The interfaces are combined in the ModularLocalPolicy class, an implementation of CoDSPL's abstract LocalPolicy class that is parametric in objects implementing these interfaces. Concrete instances of the interfaces are provided to the ModularLocalPolicy through its constructor, following the Constructor Injection pattern [Fow04].

The separation between subgoal selection and subgoal realization is implemented via the abstract factory design pattern: An implementation of the MicroPolicyFactory interface is an abstract factory whose methods, createFromEvent and createFromDelegation, select a subgoal in the form of producing a MicroPolicy object. The object implementing the MicroPolicy interface is expected to realize the subgoal, by providing a decision or a delegation, captured by a MicroPolicyResult object returned by the suggestPolicyResult method.

The extension library reifies the separation between decision-making and state-keeping by two means. Firstly, the extension library's design of micro-policies provides two methods – suggestPolicyResult and implementSuggestion – of which the first is intended to realize a subgoal of decision-making and of which the second is intended to perform the

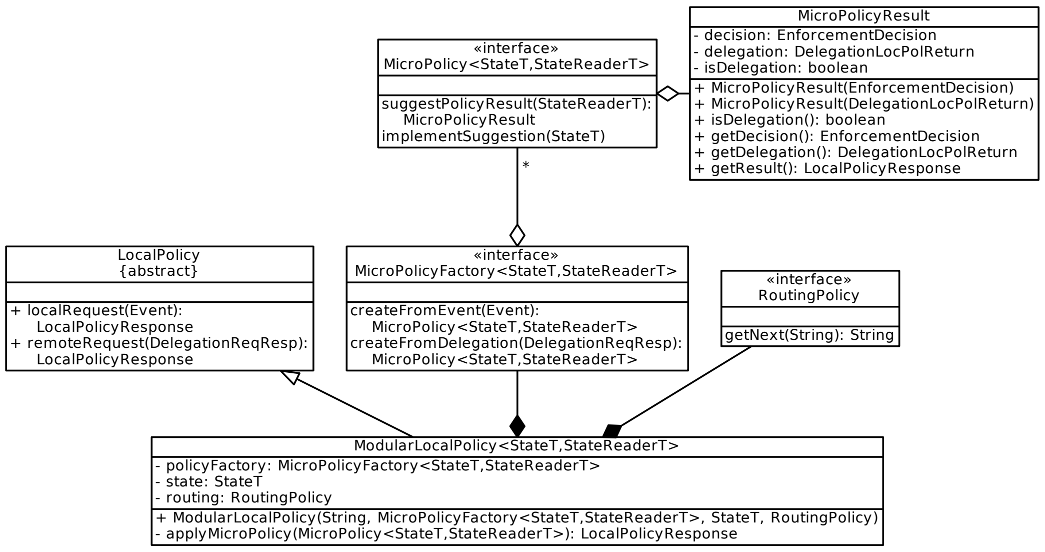

«interface»
MicroPolicy<StateT,StateReaderT>
suggestPolicyResult(StateReaderT): MicroPolicyResult
implementSuggestion(StateT)
MicroPolicyResult
- decision: EnforcementDecision
- delegation: DelegationLocPolReturn
- isDelegation: boolean
+ MicroPolicyResult(EnforcementDecision)
+ MicroPolicyResult(DelegationLocPolReturn)
+ isDelegation(): boolean
+ getDecision(): EnforcementDecision
+ getDelegation(): DelegationLocPolReturn
+ getResult(): LocalPolicyResponse
*
LocalPolicy {abstract}
+ localRequest(Event): LocalPolicyResponse
+ remoteRequest(DelegationReqResp): LocalPolicyResponse
«interface»
MicroPolicyFactory<StateT,StateReaderT>
createFromEvent(Event): MicroPolicy<StateT,StateReaderT>
createFromDelegation(DelegationReqResp): MicroPolicy<StateT,StateReaderT>
«interface»
RoutingPolicy
getNext(String): String
ModularLocalPolicy<StateT,StateReaderT>
- policyFactory: MicroPolicyFactory<StateT,StateReaderT>
- state: StateT
- routing: RoutingPolicy
+ ModularLocalPolicy(String, MicroPolicyFactory<StateT,StateReaderT>, StateT, RoutingPolicy)
- applyMicroPolicy(MicroPolicy<StateT,StateReaderT>): LocalPolicyResponse

Figure 6.1.: Low-level architecture of the extension library for separation of concerns in CoDSPL policies (UML class diagram)

state-keeping. In the design, we use the notion of “suggestions” in imitation of Polymer’s policies, in which decision-making and state-keeping is also separated. Secondly, to support the implementation of decision-making free of side-effects on the state, while still allowing the decision-making to make use of state, the extension library distinguishes the type parameters StateT (used in state-keeping) and StateReaderT (used in decision-making). This allows the specification of a read-only interface for the state through the StateReaderT parameter to prevent inadvertent state changes in the decision-making implementation.[1]

The extension library separates decision-making from routing by encapsulating both functionalities in separate interfaces. The implementation of ModularLocalPolicy then uses the provided routing policy after a MicroPolicy has returned a delegation object.

While the extension library provides an architecture that supports the implementation of modular delegation-based local policies, this architecture does not constrain the expressiveness of local policies. That is, the extension library is still generic, in the sense that for an arbitrary but fixed subclass C of LocalPolicy, a subclass C’ of ModularLocalPolicy can be constructed that performs the same computation as C. We show that this claim holds in Appendix B.1 on page 203 by construction of a concrete implementation of a class C’. Notably, while the functionality accomplished by the classes is the same, the constructed class performs more object creations at run-time and, thus, can be expected to perform slightly slower.

The presented extension library can be used in a CoDSPL policy by specifying the local policy component as a subclass of ModularLocalPolicy and, thus, by transitivity of subclassing, as a subclass of LocalPolicy. The constructor of the subclass implementation

[1]Through this technique, we compensate that Java does not allow the specification of method signatures that declare some arguments as being unmodified by the method body (such as achieved with the **const** keyword for formal method parameters in C++)

constructs objects that implement the MicroPolicyFactory interface and, respectively, the RoutingPolicy interface. Moreover, the constructor creates a state object (the type of the object must be the first type parameter of the ModularLocalPolicy subclass). Such a subclass can then be used as a local policy in a CoDSPL policy.

Overall, the presented extension library allows the specification of modular, delegation-based local policies in CoDSPL. When used by a local policy, this modularity can support the verifiability of the local policy. The architecture of the extension library uses software design patterns (abstract factory, constructor injection) for the modularity. The local policies specified using the extension library are as expressive as local policies in CoDSPL that do not use the extension library. The extension library, thus, preserves the expressiveness of CoDSPL and, hence, supports Requirement (Req-1).

6.4. Case Study

We demonstrate the use of delegation as a means for cooperatively enforcing security in distributed targets in a case study. In the exposition, we focus on the design and implementation of modular local policies, for which we build on the extension library proposed in this Chapter 6. Overall, we use the case study as evidence,

1. that delegation-based enforcement can be specified with CliSeAu in a modular fashion,
2. that the enforcement can be performed on a real-world target of decent code size, and
3. that the enforcement can be performed effectively and with moderate performance overhead.

The case study is structured as follows. We start by introducing the application scenario in Section 6.4.1. In Section 6.4.2 we develop a modular local policy for decentralized coordinated enforcement in the application scenario. We evaluate first the faithfulness of the policy analytically, in Section 6.4.3. Afterwards, in Section 6.4.4 we provide an empirical evaluation of the faithfulness and performance of the policy in an experimental setting.

6.4.1. Application Scenario

The application scenario we consider in this case study are decentralized online social networks, which we introduce first in the following. Afterwards, we introduce the security requirement in the scenario, which is fine-grained control by users over the propagation of their messages.

Decentralized Online Social Networks

Online social networks (*OSNs*) are web-based services that offer users the functionality to share text and multimedia messages with other users. A *decentralized online social network* (*DOSN*) [DBV⁺10; PFS14; YLL⁺09] is an OSN that is supported by multiple service providers. In a DOSN, users can, hence, choose a service provider whom they trust most to store their profiles.

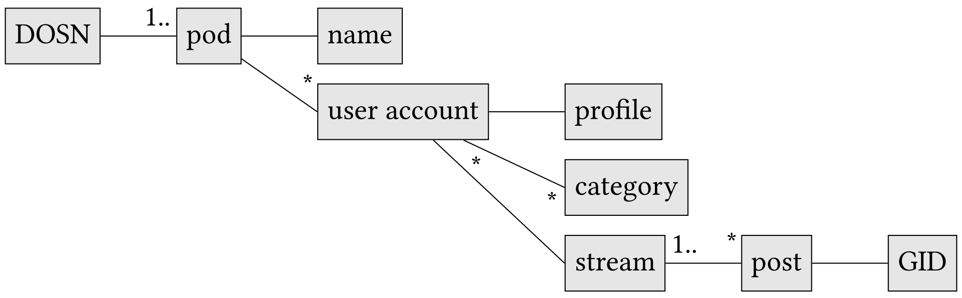


Figure 6.2.: Entity relationships in a DOSN

Target architecture The service that each service provider of a DOSN runs is a *pod*, a server program through which clients can connect via a web interface. That is, a DOSN consists of one or more pods. Each pod is identified by a unique name: the URL through which the pod is addressed in the network. In this exposition as well as in the following, we use the terminology of *Diaspora** [GSS+], at the time of writing this thesis the most popular DOSN concerning the number of users. The network topology is such that pods can communicate directly with each other (no routing needed).

Each pod of a DOSN stores a set of user accounts, one for each user who is registered with the pod. A *user account* consists of information about the user (the *user profile*), such as full name, birth date, picture, and biography, and it has a collection of categories as well as a collection of posts associated to it. A *category*[2] is associated with a set of user accounts, of which each user account can be with the same pod or with a different one. Examples for categories are “family”, “friends”, or “colleagues”. Through categories, DOSNs allow users to logically group connected users.

The set of posts of a user profile is called *stream* and includes posts of the user herself as well as posts of other users. A *post* is a message by a user, alongside with the information about which user is the post’s author and at what date and time the post was created. Each post has a unique global identifier (*GID*) in the DOSN and it is part of at least one stream, namely the stream of the post’s author. Figure 6.2 summarizes and visualizes the relevant entities and their relationships in DOSNs.

Stakeholders A DOSN involves two kinds of stakeholders: service providers, i.e., those who operate individual pods, and users, i.e., those who have an account at some pod of the DOSN and use the DOSN to socialize among each other.

Sharing and re-sharing A user of a DOSN can communicate to other users by sharing posts. *Sharing* is the operation by which a user enters a post into the DOSN. For sharing, the user composes the text of the post in the interface of the pod at which the user’s account resides and possibly attaches further objects, such as images to the text. The user then instructs the pod to share the post with other users. In the following, we refer to the user who creates a post as its *author*. DOSNs typically offer different modes of visibility

[2]Diaspora* uses the term “aspect” instead of “category”. We chose “category” to avoid confusion with “aspect” from AOP.

for posts: public visibility, visibility for selected categories, and visibility for selected users. Public visibility means that every other user can see the post on the author's stream and that every user who is associated with the author through one of the author's categories can see the post on her own stream as well. Visibility for selected categories or selected users mean that only the users from the selected categories or, respectively, only the selected users can see the post on the author's stream and on their own stream. This way, an author can selectively reach other users in the DOSN. When a user shares a post, the pod on which the user has her account distributes the post to the respective streams. In particular, when a user shares a post with users on other pods, then the pod communicates the post to the other pods.

Another form of communication in OSNs and DOSNs is re-sharing. *Re-sharing* is the operation by which a user can make a post that was shared with her visible for further users. Conceptually, the same modes of visibility as for sharing apply also to re-sharing. When a user re-shares a post, the post is displayed in other users' streams according to the same procedure as for sharing. The re-sharing user is then displayed in addition to the author in the respective streams.

Note that support for re-sharing of non-public posts is very restrictive in existing DOSN implementations. We discuss the limitations in more detail and provide a solution for controlled re-sharing in DOSNs in Section 6.4.

Controlled Re-sharing

Typical OSNs provide an author with means for sharing a post with the group of users she categorized as 'friends', 'family', 'colleagues', 'followers', or the like. As of today, DOSNs allow authors to share sensitive posts with selectable sets of users but forbid re-sharing of sensitive posts entirely.[3] A better support of controlled re-sharing in DOSNs would be beneficial.

Example 6.3. A researcher attends a conference to present a paper. Immediately when the paper got accepted, she informed her colleagues early that she will be attending the conference. During the conference talks, she writes her opinion about the presentations to her colleagues, and during the free time, she takes pictures of the landscape as well as of having fun with other researchers in bars. For privacy reasons, she wants to control spreading of this information: Pictures taken in bars should remain among her friends only and personal opinions about presentations should remain among her direct colleagues. The fact that she attends the conference should remain among colleagues and their colleagues. Her motivation for limiting spreading of her attendance could be to not provoke burglary [MSK11]. In contrast, she is less concerned about distributing pictures of the landscape. This information may be distributed further, but without becoming public. $\Diamond$

This example scenario illustrates the need for providing fine-grained control over sharing and re-sharing as well as over the distribution of re-shared posts. For brevity, we refer to the combination of these three forms of controlled information dissemination by the term

[3]Even the non-decentralized OSN Facebook supports controlled re-sharing only with users with whom the author of the post had already shared the post. The next less restrictive control over re-sharing in Facebook is already to allow arbitrary re-sharing of posts.

controlled re-sharing.

Privacy Policies

As the basis for controlled re-sharing of users' privacy in DOSNs, we first establish a model of users' privacy policies. The goal we pursue with the policies is threefold. Firstly, the policies shall allow fine-grained control over re-sharing. Secondly, the policies shall require low specification effort by users. Finally, the concepts used by the policies should be close to what existing DOSNs offer already. In the following, we first model privacy policies and subsequently define their semantics.

The set of users in a DOSN can vary over time. Moreover, also the categories of individual users can vary. For capturing snapshots of DOSNs and their pods, in the following we use $\mathcal{USER}$ to denote the universe of possible identifiers of users and $\mathcal{CAT}$ to denote the universe of all possible category names. The former universe particularly contains user names such as "*alice@example.com*". The latter universe particularly contains the words "*friend*", "*family*", and "*colleague*".

When a user shares or re-shares a post in a DOSN, the post's recipients might afterwards use the sensitive content of the post to the disadvantage of the post's author. Consequently, the user could use her trust in a particular recipient as a basis for deciding whether to share the sensitive post with this recipient. This instance of trust is well captured by the notion of *decision trust*, "the extent to which one party is willing to depend on something or somebody in a given situation with a feeling of relative security, even though negative consequences are possible" [JIB07]. Users' decision trust[4] constitutes a central element of the privacy policies we propose for constraining the propagation of sensitive posts in DOSNs.

Definition 6.2. A *privacy policy* of a user u is a triple $pp = (cat, rel, tv)$, where $cat \subseteq \mathcal{CAT}$ is a set, $rel \subseteq cat \times \mathcal{USER}$ is a binary relation, and $tv : cat \to [0,1]$ is a function. In the privacy policy, cat models the categories that u uses for categorizing other users, $rel(c, u')$ models that user u' is in the category c of user u, and $tv(c)$ models the trust of user u in her category c. We call rel the *relationships* of u and the values $tv(c)$ *trust values*. In the following, we denote the set of all possible privacy policies by PP and model several users' privacy policies by partial functions $pps : \mathcal{USER} \rightharpoonup PP$. We abbreviate $PPS = \mathcal{USER} \rightharpoonup PP$. ◊

Intuitively, the relation rel_u of a user u's privacy policy specifies which users may obtain a post that u (re-)shares with a particular category. The function tv_u specifies to which extent user u trusts other users in her categories to re-share her posts in her interests. Trust values are modeled as scalar values, where greater values mean greater trust. The maximal trust value 1 means that the respective user trusts the users in the category as much as herself wrt. the propagation of her posts. The minimal trust value 0 means that the user does not have any trust in the users in the category wrt. propagation of posts.

Example 6.4. A user Alice uses a privacy policy with two categories, "friend" and "colleague". She assigns user Bob to category "friend" and user Charlie to category

[4]In the following, we use "trust" and "decision trust" synonymously.

"colleague". Her trust in friends Alice specifies as 0.8, her trust in colleagues as 0.5. Bob has both Alice and Charlie in his only category "friend", in which his trust is 0.7 and Charlie considers only Bob a friend, also with trust 0.7. Consequently, Alice's privacy policy would be $(\mathit{cat}_a, \mathit{rel}_a, \mathit{tv}_a)$ with $\mathit{cat}_a = \{\mathit{friend}, \mathit{colleague}\}$, $\mathit{rel}_a = \{(\mathit{friend}, \mathit{bob}), (\mathit{colleague}, \mathit{charlie})\}$, $\mathit{tv}_a(\mathit{friend}) = 0.8$, and $\mathit{tv}_a(\mathit{colleague}) = 0.5$. Independently, Bob's chosen privacy policy is $(\mathit{cat}_b, \mathit{rel}_b, \mathit{tv}_b)$ with $\mathit{cat}_b = \{\mathit{friend}\}$, $\mathit{rel}_b = \{(\mathit{friend}, \mathit{alice}), (\mathit{friend}, \mathit{charlie})\}$, $\mathit{tv}_b(\mathit{friend}) = 0.7$. Finally, Charlie's privacy policy is $(\mathit{cat}_c, \mathit{rel}_c, \mathit{tv}_c)$ with $\mathit{cat}_c = \{\mathit{friend}\}$, $\mathit{rel}_c = \{(\mathit{friend}, \mathit{bob})\}$, $\mathit{tv}_c(\mathit{friend}) = 0.8$. $\Diamond$

The example shows how a user's privacy policy can look like and how individual users' privacy policies can differ in their sets of categories as well as in the trust they associate with their categories. We revisit this example after defining the semantics of privacy policies for controlled re-sharing.

When a user has a post in her stream that is not her own post, then she must have obtained it from another user. This other user must either have shared the post herself or have re-shared another user's post she had in her own stream. That is, the user obtained the post through a path of sharing and re-sharing operations of other users. Each of the share and re-share operations along the path was performed by a user and with a particular set of categories to which the next user in the path belongs. To capture how a post has been propagated from the post's author to a recipient, we introduce re-share paths, defined as follows.

Definition 6.3. A *re-share path* for a post to a user u is a list of users who actually or hypothetically shared and re-shared the post with particular categories of a post before it reached user u. We model a re-share path as a non-empty, alternating sequence of users and sets of categories $\pi \in (\mathcal{USER} \times \mathcal{P}(\mathcal{CAT}))^+$. A re-share path $\pi = (u_1, C_1, \ldots, u_n, C_n)$, for $n \geq 1$, models that user u_1 shared (as author) with all categories in C_1 to user u_2, who in turn re-shared with all categories in C_2, and so forth to user u_n, who in turn re-shared with all categories in C_n. We denote the *set of possible re-share paths* by $\mathit{PATH} = (\mathcal{USER} \times \mathcal{P}(\mathcal{CAT}))^+$. $\Diamond$

Such re-share paths faithfully capture the propagation of a post from the post's author to a recipient. This is because a re-share path captures all share and re-share operations that caused a post to be propagated to a recipient, in terms of which user (re-)shared and which categories the user (re-)shared with. Moreover, a re-share path captures all share and re-share operations and captures them in the order in which they occur in the propagation. Note that a re-share path does not contain the post itself or a recipient of the last (re-)share, as they are not required for capturing the propagation.

Example 6.5. The re-share path $(\mathit{alice@pod1}, \{\mathit{friend}, \mathit{colleague}\}, \mathit{bob@pod2}, \{\mathit{family}\})$ captures that a user *alice@pod1* shared some post with all users in category *friend* and all users in category *colleague* and that, subsequently, a user *bob@pod2* re-shared the post with all users in category *family*. $\Diamond$

Sharing and re-sharing in a DOSN propagates posts along the relationships between users. We capture this aspect as a property of re-share paths, particularly capturing also that the relationships are part of users' privacy policies and can even change over time.

Definition 6.4. The relation $PC \subseteq PPS \times PATH \times \mathcal{USER}$ is defined inductively by:

(a) $(pps, \langle(u, C)\rangle, u') \in PC$ holds if $pps \in PPS$ and $u \in dom(pps)$ hold and, with $(cat_u, rel_u, tv_u) = pps(u)$, furthermore $C \subseteq \mathcal{CAT}$, $u' \in \mathcal{USER}$, and $rel_u(c, u')$ hold for some $c \in C \cap cat_u$; and

(b) $(pps, \pi.\langle(u, C)\rangle, u') \in PC$ holds for $\pi \in PATH$ if both $(pps, \langle(u, C)\rangle, u') \in PC$ and $(pps, \pi, u) \in PC$ hold.

We call a re-share path π to a user u *connected* for a family *pps* of users' privacy policies if and only if $(pps, \pi, u) \in PC$ holds. $\lozenge$

The definition of faithfully captures that subsequent users in a connected re-share path are related according to the respective users' privacy policies. For ensuring this property, the definition primarily requires the predicate $rel_u(c, u')$ between subsequent users u and u' and some category c of the respective (re-)share operation, where rel_u are the relationships of u's privacy policy. Our model of re-share paths supports changing users' privacy policies in multiple ways: Connectedness can be evaluated with the privacy policies *pps* as they were at the time of the respective (re-)shares in a re-share path, even if users' privacy policies changed afterwards; Connectedness can also be evaluated with the users' most recent privacy policies *pps*, even if the privacy policies were different at the times of the respective (re-)shares in a re-share path. Our model excludes the rather pathological case of different privacy policies for one and the same user if that user occurs more than once in a re-share path.

In their privacy policies, users specify their decision trust in categories of users to which they are (directly) related. When a post is shared and re-shared, the recipients of the post might not be in a direct relationship to the post's author. That is, the privacy policy of an author might not assign trust values to all recipients of re-sharing. Moreover, even when a trust value is assigned to a user, this user might show different behavior with respect to re-sharing a post when she obtained the post directly than when she obtained the post indirectly: The fact that another user has already re-shared post might, on the one hand, make a user less cautious about further re-sharing the post than when she directly obtained the post. On the other hand, a user might have an incentive to be more cautious about re-sharing an already re-shared post because of the potential to displease not only the author of the post but also the previously re-sharing users. That is, "indirect trust" depends on the trust between users on a re-share path. To capture our notion of indirect trust based on users' privacy policies, we introduce path trust as the last remaining concept underlying our semantics of privacy policies.

Definition 6.5. The *path trust* is the decision trust that the author of a post has into an actual or hypothetical recipient of the post who obtained the post through a particular re-share path from the author. Formally, we capture path trust for a re-share path π to a user u under a family *pps* of users' privacy policies by $pt(pps, \pi, u)$, where $pt : PC \to [0, 1]$ is defined recursively over the length of the re-share path as follows:

$$\begin{aligned} pt(pps, \langle(u, C)\rangle, u') = {} & \max\{tv(c) \mid c \in cat \cap C \wedge rel(c, u')\}, \\ & \text{where } (cat, rel, tv) = pps(u) \\ pt(pps, \pi.\langle(u, C)\rangle, u') = {} & pt(pps, \pi, u) \cdot pt(pps, \langle(u, C)\rangle, u') \end{aligned}$$

where π ranges over re-share paths (i.e., π in particular is a non-empty sequence). $\diamondsuit$

By the definition, the path trust for a post is the product of all direct trust values specified by the users along a re-share path along which the post was obtained. That is, if a user obtained a post directly from the author u, the path trust equals u's trust value in the categories that she shared with. Path trust can be greater or lower than the direct trust and can, as such, capture the cases of more and of less cautious users with regard to re-sharing. The following example illustrates both cases.

Example 6.6. Consider the scenario and the privacy policies of Example 6.4 on page 89 and let *pps* be a family containing the three privacy policies of Alice, Bob, and Charlie. Then we have $pt(pps, \langle(alice, \{friend\}), (bob, \{friend\})\rangle, charlie) = 0.8 \cdot 0.7 > 0.5 = pt(pps, \langle(alice, \{colleague\})\rangle, charlie)$ for Alice's trust in Charlie (via Bob). Here, path trust is higher than direct trust due to the rather strongly trusted *friend* category, compared to the *colleague* category.

Conversely, $pt(pps, \langle(alice, \{colleague\}), (charlie, \{friend\})\rangle, bob) = 0.5 \cdot 0.8 < 0.8 = pt(pps, \langle(alice, \{friend\})\rangle, bob)$ holds for Alice's trust in Bob (via Charlie). Here, path trust is lower than direct trust due to the rather weakly trusted *colleague* category, compared to the *friend* category. $\diamondsuit$

Note that *pt* is well-defined even in the recursive case of the definition, because the relation *PC* is closed in the sense that $(pps, \pi.\langle(u, c)\rangle, u') \in PC$ implies $(pps, \pi, u) \in PC$ and $(pps, \langle(u, c)\rangle, u') \in PC$ by Definition 6.4.

To capture that the content of posts in a DOSN can be of varying sensitivity, we introduce sensitivity values. A *sensitivity value* is a value s from the half-open interval $[0, 1)$ that models the sensitivity of a post. Greater sensitivity values model greater sensitivity.

The goal behind our privacy policies is two-fold: Firstly, privacy policies shall constrain who, i.e., which users are allowed to obtain a post in their stream in consequence of a sharing or re-sharing. Secondly, privacy policies shall constrain who is allowed to re-share a post. We capture both aspects in the following definition of the semantics of privacy policies.

Definition 6.6. Let $sc \in [0, 1]$ be arbitrary. Re-sharing of a post p with sensitivity value $s \in [0, 1)$, which had been received via a re-share path π, by a user u with a set of categories C *complies* with the family $pps \in PPS$ of users' privacy policies if and only if each of the following conditions is satisfied:

(a) $C \subseteq cat_u$;
(b) $(pps, \pi.\langle(u, C)\rangle, u') \in PC$ holds for all users u' who receive p due to this sharing; and
(c) $pt(pps, \pi, u) \geq \frac{sc}{1-s}$ holds,

where $(cat, rel, tv) = pps(u)$. $\diamondsuit$

The semantics is parametric in *sensitivity coefficient* $sc \in [0, 1]$, for which higher values make the privacy policies more restrictive (Definition 6.6 (c)). For instance, the choice of sensitivity coefficient $sc = 0.35$ ensures that re-sharing is completely forbidden along re-share paths with low trust ($pt(pps, \pi, u) < 0.3$), is allowed for low-sensitivity posts (with a sensitivity value below 0.3) along re-share paths with medium trust ($0.5 \leq pt(pps, \pi, u) \leq 0.6$), and is completely forbidden for posts with a sensitivity value above 0.65. Sensitivity coefficients have been used before in the semantics of privacy policies for controlling the direct sharing of posts between users. For instance, Kumari et al. [KPP+11] propose sensitivity coefficients for different kinds of operations.

The definition faithfully captures the two aspects of re-sharing we aim to control: who may re-share and who may obtain a re-shared post. Regarding the latter aspect, the semantics requires that all recipients of a re-shared post must be related to the re-sharing user in some category with which she re-shared. This reflects the intuition behind categories in DOSNs. Regarding the former aspect – who may re-share –, the semantics requires that a re-sharing user must be in a connected re-share path together with the re-share path along which she obtained the post (Definition 6.6 (b)) and must be trusted enough by the author of the post along the re-share path (Definition 6.6 (c)). This reflects the intuition behind trust values of privacy policies.

The focus in this chapter is controlled re-sharing. For the sake of completeness, we introduce compliance of sharing with users' privacy policies as well. This shall demonstrate that our privacy policies are compliant with existing mechanisms of DOSNs to control the sharing of posts.

Definition 6.7. Sharing of a post p with sensitivity value $s \in [0, 1)$ by a user u with a set of categories C *complies* with the family $pps \in PPS$ of users' privacy policies if and only if each of the following conditions is satisfied:

(a) $C \subseteq cat_u$ and

(b) $(pps, \langle(u, C)\rangle, u') \in PC$ holds for all users u' who receive p due to this sharing,

where $(cat, rel, tv) = pps(u)$. ◊

The conditions for compliant sharing are enumerated to expose the similarity to their counterparts in Definition 6.6. The recipients of shared posts are constrained by Definition 6.7 (b) in the same way as for re-sharing in that they must be related with the sharing user in some category that was shared with. Who may share is not constrained: Sharing is always allowed as long as the sharing user chooses a valid set of categories. That is, overall the semantics of privacy policies for sharing captures what is typically implemented in OSNs.

Example 6.7. Consider again Example 6.3 on page 88 and suppose Alice, whose privacy policy is specified in Example 6.4 on page 89, is the researcher who travels to the conference. Let the sensitivity coefficient be $sc = 0.35$. When Alice shares her personal opinions about presentations in posts to category *colleague* or sends pictures of having fun in bars in posts to category *friend* with sensitivity value $s_1 = 0.7$, then her colleague Charlie obtains her shared opinions and her friend Bob obtains the pictures, but neither Bob nor Charlie can re-share the posts, because $\frac{sc}{1-s_1} \approx 1.17$ and no path trust can exceed this value to satisfy Definition 6.6 (c). When Alice shares her attendance of the conference with colleagues in posts with sensitivity value $s_2 = 0.3$, then Charlie may re-share these posts but a recipient of the posts re-shared by Charlie could not re-share further, because a trust value of Charlie in this recipient of at least $\frac{sc}{(1-s_2)\cdot tv_a(colleague)} = 2$ would be required. Finally, when Alice shares pictures from the landscape around the conference venue with friends in posts with low sensitivity value $s_3 = 0.2$, then Bob could re-share with his friend Charlie and Charlie could re-share further with his friends, because

$tv_a(\mathit{friend}) \cdot tv_b(\mathit{friend}) \cdot tv_c(\mathit{friend}) = 0.448 \geq \frac{sc}{1-s_3} = 0.4375.$ ◊

The example demonstrates that our privacy policies together with sensitivity values enable users of a DOSN to control re-sharing in a fine-grained fashion.

Conceptually, our privacy policies are policies for *relationship-based access control* (*ReBAC*) [Gat07; Fon11], i.e., access control in which the permissibility of an access is determined by the relationships between involved users. Our privacy policies combine qualitative relationships (captured by relations *rel*) as well as quantitative relationships (captured by functions *tv*). Both relationships are taken into account by the semantics of privacy policies in Definition 6.6. In the following, we use CliSeAu for enabling controlled re-sharing by enforcing the privacy policies introduced in this section.

6.4.2. CReDiC – a CoDSPL policy for Controlled Re-Sharing

In this section, we present *CReDiC* (abbreviating "Controlled Re-Sharing in Diaspora* with CliSeAu"), a local policy for enforcing controlled re-sharing in DOSNs and a CoDSPL policy based on this local policy for the Diaspora* DOSN. The local policy uses delegation and is designed in a modular fashion with separation of concerns, built with the extension library introduced in Section 6.3. In the following, we first introduce the basic design decisions behind the local policy and, second, present the modular architecture and implementation of the local policy for the application scenario.

Goals and design choices More concretely, the main goal of this section is enforcing controlled re-sharing with users' privacy policies in DOSNs. In particular, we aim at an enforcement mechanism that supports two common properties of DOSNs: firstly that, like OSNs, DOSNs allow their users to change their relationships to other users (i.e., the privacy policies in our model) over time and, secondly, that DOSNs store users' profiles (including the privacy policies) in a decentralized fashion at the DOSN's pods. As the target for the enforcement, we pick Diaspora* for its popularity among DOSNs. Diaspora* shares both of the aforementioned properties. The two properties provoke two main design choices. Firstly, that privacy policies in DOSNs can change over time raises a design space over which privacy policies to use in the enforcement. For instance, the enforcement could use the users' privacy policies as they were at the time they performed a (re-)share operation. Alternatively, users' most recent privacy policies, at the time of checking whether a re-share is compliant, could be used. Secondly, that privacy policies are stored in a decentralized fashion implies that they yet have to be combined for determining the permissibility of a re-share operation (as specified by the semantics of privacy policies in Definition 6.6). For instance, all relevant relationships and trust values could be gathered by the enforcement mechanism for determining the compliance of a re-share. Alternatively, the permissibility could be determined in a more decentralized fashion.

The local policy we design for enforcing security in the case study is based on the following design decisions. Concerning the recency of privacy policies, we aim at taking the most recent ones, such that changes of privacy policies over time can influence even posts that were shared and re-shared before the change of policy. However, we slightly relax the recency when privacy policies change during the decision-making: In this case, the enforcement utilizes the respective most recent privacy policy at some point in time during the decision-making. This way, the enforcement mechanism can avoid overly synchronizing and, thereby, slowing down the DOSN. Concerning the second design choice, how to cope with the decentralized privacy policies, we aim for a decentralized decision-making based on partial decisions. More concretely, the partial decisions essentially consist of intermediate path trust values and, thus, avoid gathering privacy policies at a central location.

At a more detailed level, the local policy is based on further design choices regarding the storage of relevant information and regarding the cooperation. Users' privacy policies as well as posts' sensitivity values are stored locally, i.e., at the unit at whose pod the post is created or, respectively, the users' profile resides. Re-share paths for posts are stored at a unit whenever the unit accepted a security-compliant re-share operation. Sensitivity values and trust values are never directly exchanged in the process of cooperation between units. Concerning the routing, units communicate with each other directly (as do the pods).

Note that in this case study, we focus on controlling who may re-share. The proposed mechanism does not control who receives posts due to (re-)sharing, given that this functionality is typically provided by DOSNs already.

By the choice of this design, we follow the work of Mazaheri [Maz12, Section 4.1]. Our main contribution here is the entirely re-designed enforcement mechanism, designed with a modular architecture of the delegation-based local policies and built on a more sophisticated model of privacy policies. We provide a delineation between her contributions and the contributions made by this thesis in the related works, Section 9.8.1 on page 165.

Policy design Each instance of our local policies is for one pod of a DOSN. At this pod, the local policy handles four kinds of operations occurring in the pod and produces two possible decisions. The operations, captured by event objects, are sharing of posts, changing of relationships between users, changing of trust values, and re-sharing of posts. The two decisions are to allow an operation and to disallow an operation. Note that we do not consider operations that change the sensitivity value of a post. That is, sensitivity values of posts are constant and determined by authors when posts are shared.

For being able to enforce users' privacy policies, the local policy maintains state. This state consists of the following parts:

- a map from user identifiers to privacy policies,
- a map from post identifiers to sensitivity values, and
- a map from post identifiers to re-share paths.

The privacy policies in the state are straightforward implementations of Definition 6.2 on page 89, i.e., consisting of a set of categories, the relationships, and the trust values for categories. The state is implemented by a `State` class. For each part of the state, including the parts of users' privacy policies, the `State` class provides methods for retrieving as well as for updating this part. State-keeping, through updating the `State` object of the local policy, is invoked upon occurrence of intercepted operations, as we discuss next for the micro-policies. This design separates the details of state-keeping (implemented in the `State` class) from the realization of decision-making subgoals (implemented in micro-policies).

Some kinds of operations handled by the local policy are always allowed by the local policy and solely handled for the purpose of maintaining the local policy's state. These kinds of operations are those that change users' privacy policies (adding/removing categories, adding/removing users from/to categories, and changing trust values of categories) and those that share posts. The state-keeping provoked by these operations is always performed by the local policy of the pod at which the operations are performed. That is, no delegation takes place for these operations. For each of the four operations, our local policy provides a single micro-policy. The respective micro-policy always produces a decision to allow the operation and implements this decision by invoking an update of the local policy's state.

The main operation for controlling re-sharing in DOSNs is the re-sharing itself. For handling this operation in our local policy in accordance with the semantics of users' privacy policies (see Definition 6.6 on page 92), our local policy provides several micro-policies. These micro-policies encapsulate subgoals that can be performed by a single unit without intermediate delegation. Concretely, the micro-policies for handling re-sharing operations are the following.

- The "path query" micro-policy resolves the re-share path of the post that is to be re-shared. This re-share path is required for computing the path trust.
- The "sensitivity query" micro-policy retrieves the sensitivity value of the post that is to be re-shared. This value is required for checking compliance.
- The "trust query" micro-policy performs a partial check for connectedness of the re-share path and a partial computation of the path trust. Both tasks are performed for those elements of the re-share path for which the respective users' privacy policies are available at the unit.
- The "decision delivery" micro-policy realizes that a delegation response is delivered to the unit at which the re-share shall be allowed or denied.

Each of the micro-policies includes a branching based on whether the locally available state provides the information for realizing the subgoal. If this is not the case, the micro-policy triggers delegation to a unit that is expected to provide the required information. The delegation objects used by the micro-policies for exchanging partial decisions store the event object for the re-share as well as the identifier of the unit that expects a decision for the event object, the re-share path, the identifiers of the units whose users' privacy policies remain to be involved for determining compliance, and a variable holding an intermediate result of the computation of $pt(pps, \pi, u) \cdot (1 - s)$, where pps are the users' most recent privacy policies, π is the re-share path of the post that is to be re-shared, u is the user who wants to re-share, and s is the sensitivity of the post. By exchanging last-mentioned intermediate result – rather than gathered users' privacy policies –, the local policy avoids gathering users' privacy policies at a central location that would otherwise be stored decentralized. For determining compliance of a re-share, the micro-policies use the sensitivity coefficient $sc = 0.35$, which we have already used earlier in our exposition, in checking $pt(pps, \pi, u) \cdot (1 - s) \geq sc$.

Figure 6.3 on the facing page shows all micro-policies used (shaded boxes), their triggers (white boxes with solid arrows), and their temporal ordering (uncontinuous arrows). The trigger for a micro-policy is either an operation intercepted by a unit or a delegation request received by a unit from another unit. The small black circle represents the end of the decision-making. Temporal ordering comes in three variants: always with delegation, meaning that the next micro-policy is invoked at another unit (dashed arrows), always

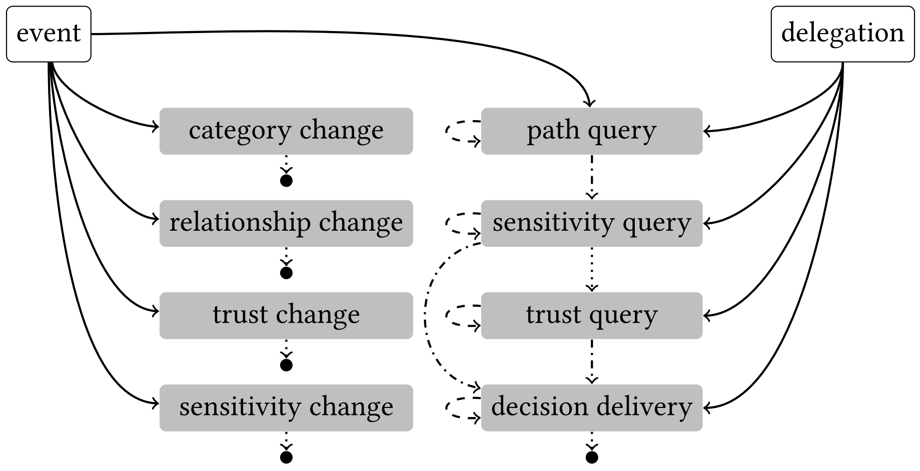


Figure 6.3.: Schematic visualization of the micro-policies in CReDiC

without delegation (dotted arrows), and sometimes with and sometimes without delegation (dash-dotted arrows).

The figure shows that for each of the four operations that are always allowed and only affect the local policy's state is handled by a separate micro-policy (left column of gray boxes). Each of these micro-policies, the decision-making ends after the micro-policy's invocation and no delegation takes place. Note that the ordering indicated by discontinuous arrows is a temporal ordering of micro-policies but, in the design and implementation of the micro-policies, it is not caused by one micro-policy directly invoking another micro-policy. Rather, the successor of a micro-policy is determined by the micro-policy factory, which selects the next micro-policy based on the (intermediate) result produced by the former micro-policy. That is, subgoal selection and subgoal realization are separated in the design of the local policy.

As we discussed, as part of the design decisions, our local policy aims at routing where units communicate with each other directly. In the design and implementation of the local policy, we realize this through a simple class, named DirectRoutingPolicy, which always provides the destination unit of a delegation itself as the next unit to route to. Through this separate class, we achieve a separation of the routing concern from the remaining concerns of the local policy.

Mapping Diaspora on our trust model* We map the trust model introduced in Section 6.4.1 to the particularities of Diaspora* (version 0.5.3.1) through three kinds of adaptations: small extensions to Diaspora*, realized as patches to the Diaspora* code; usage of reasonable defaults for some parameters in event objects implemented in the event factory of CReDiC; and altered use of existing Diaspora* elements. We list all concrete instances of adaptations below to shed light on the extent of the adaptations made.

The extensions made to Diaspora* compensate two particularities of Diaspora*: Firstly, Diaspora* prohibits re-sharing of sensitive posts (i.e., posts not classified as "public"). Secondly, the Diaspora* code lacks, for the enforcement, once a sufficiently fine method granularity, two accessible object attributes, and public rather than private accessibility for one method (required by the Aquarium tool). The code patch remediating these two particularities consists of 22 deleted lines and 20 inserted lines, excluding code comments.

Default values for some parameters of event objects are used in the following cases: sensitivity of posts, re-sharing with selected categories. In Diaspora*, posts cannot be assigned a sensitivity value. We simulate sensitivity values of posts by assigning the least trust value into categories with which a post is shared as the post's sensitivity value. In Diaspora*, re-sharing (as opposed to the initial sharing) cannot be further parameterized with a set of categories. Rather, Diaspora* delivers a re-shared post to all users who are related to the re-sharing user. In accordance with this behavior of Diaspora*, we therefore simulate the categories of a re-sharing operation by taking all categories of the re-sharing user's privacy policy at the time of re-sharing.

Altered use of existing Diaspora* elements applies to the names of categories. In Diaspora*, users cannot assign trust values to categories.[5] We compensate this by expecting users of Diaspora* to specify trust in categories as part of the categories' name in parentheses, e.g., "family (0.9)", "friend (0.8)", "colleague (0.5)". This allows a user to change the trust value in a category by renaming the category. In the event factory of CReDiC, we decompose these augmented category names such that this abuse of category names becomes transparent to the local policy.

Overall, the described mapping of Diaspora* to our trust model constitutes the basis for the design of the CoDSPL policy of CReDiC on top of our local policy. This CoDSPL policy enables controlled re-sharing of posts in Diaspora* based on users' most recent privacy policies. Through the particular design of the local policy based on delegation, controlled re-sharing is performed in a decentralized fashion.

6.4.3. Analytic Evaluation of CReDiC

We analyze CReDiC with regard to the two key notions of this chapter: delegation and modularity through separation of concerns.

Delegation Delegation is an essential concept used by the local policy for enforcing controlled re-sharing in the case study. However, as network communication is time-intensive compared to local computations, delegation is also a crucial factor for the performance of the enforcement mechanism. We, thus, qualify and quantify the use of delegation by the local policy in the following.

Delegation is performed by the local policy when its own state does not provide all information needed for checking compliance of a re-share operation. We analyze the delegation behavior of the local policy along the involved micro-policies for an abstract instance of an intercepted re-share of a post by a user u. The "path query" micro-policy performs delegation to the unit at the pod from which user u obtained the post, unless it is part of this unit itself. Analogously, the "sensitivity query" micro-policy performs delegation to the unit at the pod from which the post was originally shared, unless it is part of this unit itself. The "trust query" micro-policy delegates to a unit whose contribution to computing the path trust is outstanding, if such unit exists. In picking the delegate,

[5]Note that in Diaspora* the categories are called "aspects". We prefer to use the term categories to avoid confusion with AOP terminology.

this micro-policy ensures that, if the unit at which the re-share was intercepted is in the list, it is picked last. This avoids one delegation when the recipient unit of the decision participates in the path trust computation. Finally, the "decision delivery" micro-policy returns a decision object for the unit's enforcer if it is part of the unit that intercepted the re-share, and delegates to that unit otherwise. That is, in the process of decision-making for an intercepted re-share, each of the micro-policies might delegate once or not at all, and only "trust query" might delegate more than once.

The number of delegations performed by the local policy for controlling the re-sharing of a single post is determined by how many units store relevant information for determining compliance according to Definition 6.6. The following theorem provides the exact number of delegations as well as lower and upper bounds.

Theorem 6.1. *Let u be the user whose re-sharing operation is intercepted, π be the re-share path along which u received the post, r be u's pod, p be the pod of the user who (re-)shared the post with u, s be the pod on which the post was originally shared, and S be the set of all pods of users in π. Then*

(a) the exact number of delegations performed by the local policy is

$$\delta_{r \neq p} + \delta_{p \neq s} + (|S| - \delta_{s \in S}) + \delta_{r \notin S},$$

(b) the minimal number of delegations is 0, and

(c) the number of delegations has the tight upper bound $2 + |\pi|$. ◊

Proof. We show the individual claims of the theorem separately.

- Theorem 6.1 (a) is the immediate result of the number of delegations contributed by the individual micro-policies involved in the decision-making for a re-share operation. The "path query" micro-policy delegates $\delta_{r \neq p}$ times, "sensitivity query" delegates $\delta_{p \neq s}$ times, "trust query" delegates $|S| - 1$ times (because $s \in S$ is always the case) until all units involved in the re-share path are visited, and "decision delivery" delegates $\delta_{r \notin S}$ times.
- The minimal number of 0 delegations, as claimed by Theorem 6.1 (b), follows immediately from Theorem 6.1 (a) and the fact that the conditions $r = p$, $p = s$, $|S| = 1$, and $r \in S$ can be satisfied simultaneously.
- We argue separately that the formula of Theorem 6.1 (c) is an upper bound and that it is a tight bound. That the formula provides an upper bound follows immediately from Theorem 6.1 (a) and the fact that $|S| \leq |\pi|$, $\delta_\varphi \leq 1$, and $s \in S$, which yields

$$\delta_{r \neq p} + \delta_{p \neq s} + (|S| - \delta_{s \in S}) + \delta_{r \notin S} \leq 2 + (|\pi| - 1) + 1 = 2 + |\pi|.$$

 That the upper bound is tight follows from the fact that the conditions $r \neq p$, $p \neq s$, $|S| = |\pi|$, and $r \notin S$ can be satisfied simultaneously. □

Overall, we have seen that the local policy avoids delegation for controlling re-sharing of a post when the re-share path is confined to a single pod. The local policy moreover avoids a delegation that follows the re-share path but rather clusters the subgoals it performs by the pods in the re-share path, thereby avoiding delegation when pods occur multiple times in the re-share path.

Complexity of local policy design and implementation As we have discussed in Section 6.4.2, the design separates the concerns of subgoal selection, subgoal realization, state-keeping, and routing. The design of the local policy based on micro-policies consists of one micro-policy factory, one routing policy, seven micro-policies, and one state component. Despite the involved cooperation performed by the local policy, the code of the local policy consists of only 614 source lines of code (SLOC). The code of the micro-policy factory consists of 38 SLOC. The code size of the micro-policy implementations ranges from 12 to 41 SLOC, thus, still fitting on a single screen in full. These numbers provide evidence that the design of the local policy is very modular and in a way that is beneficial for the comprehensibility and, thus, maintainability of the code.

Notably, in contrast to the rather small code size of our local policy, Diaspora* itself is of decent code size: The version of Diaspora* we used in our application scenario consists of 31717 Ruby SLOC.

6.4.4. Empirical Evaluation of CReDiC

We empirically evaluate effectiveness and performance of CReDiC. Concretely, concerning the effectiveness, we evaluate selected test cases – policy-compliant as well as policy-violating –, showing the respective expectation (hypothesis) and the outcome of the evaluation. Concerning the performance, we present the overhead caused by our enforcement mechanism at run-time of the target, and we present the time taken to instrument the target using CliSeAu. The evaluation shall provide evidence that the proposed policy indeed enforces compliance with users' privacy policies in the application scenario and does so with reasonable performance.

For the empirical evaluation, we used three Intel Quad-Core (i5-4590 3.3GHz) machines with 32 GB RAM. The machines ran Ubuntu 14.04.2 LTS with a 3.13.0 kernel and OpenJDK 7. We ran three patched Diaspora* pods (see Paragraph 'Mapping Diaspora* on our trust model' on page 97) in production mode with Ruby 2.1.1 and a MySQL 5.5.54 database. Four user profiles were hosted by the three pods. For static content, we ran an Apache 2.4.7 web server. We measured page fetch times on an Intel Quad-Core (i7-6600U 2.6GHz) machine with 16 GB RAM, a 4.4.26 kernel, OpenJDK 8, and AspectJ 1.8.10, using curl 7.52.1 with `trace-time` option. All four machines were connected through a 1 Gbps wired network.

For the evaluation, we use the following concrete setup of users and their privacy policies of the Diaspora*-based DOSN. The setup refines the one from Example 6.4 on page 89 with regard to the distribution of users to pods and augments the example by another user. A visualization of the setup is provided by Figure 6.4 on the facing page. The three pods of the DOSN are named "pod0", "pod1", and "pod2", respectively. Four users are registered in the DOSN: "alice", "bob", "charlie", and "dave". Alice has her profile at pod0, Bob at pod1, and Charlie and Dave at pod2. The privacy policies of Alice, Bob, and Charlie are as in Example 6.4, except that Alice's relationships are expanded to have Dave as a friend and colleague. Dave's privacy policy has the same categories and trust values as Alice and has Alice as her only friend and colleague.

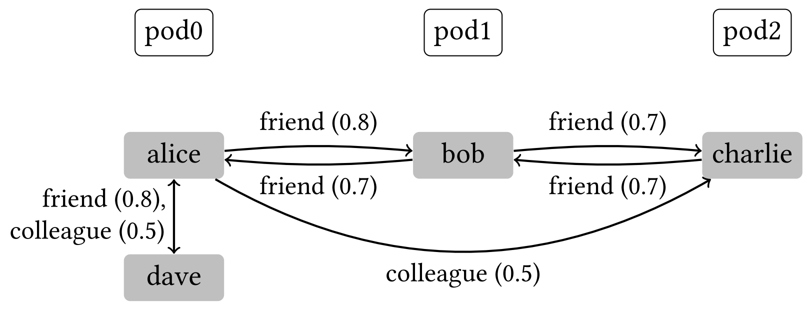


Figure 6.4.: Example scenario for the empirical evaluation

Effectiveness of the Enforcement

In this section, we provide empirical evidence about the effectiveness of our enforcement mechanism, i.e., to what extent the mechanism soundly and transparently enforces users' privacy policies in the application scenario. Our hypothesis is that our enforcement mechanism is effective in absence of race conditions between re-sharing operations and changes of privacy policies and can be unsound or intransparent otherwise. This is coherent with observations by Mazaheri [Maz12].

We select test cases from two categories: tests for operations that are intercepted by our enforcement mechanism but are always permitted (sharing and changing of relationships and trust values) and tests for operations that might violate security (re-sharing). In recognition of the higher complexity of the re-sharing compared to, e.g., the change of relationships, we provide more test cases for re-sharing: test cases for re-sharing and re-re-sharing, for one up to three units involved in the decision-making, and for different causes of dissatisfied compliance. Table 6.1 on the next page displays the list of test cases, along with their expected results and their empirically observed results.

The test cases are structured into several groups, separated by horizontal rules in the table. In the table, we use a short-hand notation in which an arrow $\xrightarrow[s]{c}$ abbreviates "shares a post of sensitivity value s with categories c", an arrow $\xrightarrow{c}$ abbreviates "re-shares" with categories c, and a boxed user name marks the user who's re-sharing is tested (all other operations are for establishing the prerequisites of the test case).

- Test cases 6.1–6.4 aim to ensure that sharing and changing of privacy policies is maintained functional by CReDiC.
- Test cases 6.5 and 6.6 address re-sharing involving a single unit (i.e., no delegation). At the same time, these test cases address the impact of posts' sensitivity values on the decisions made by CReDiC.
- Test cases 6.7 and 6.8 address re-sharing involving two units and test the impact of the author's trust value on the decisions made.
- Test cases 6.9 and 6.10 address re-sharing involving three units and test the impact on the connectedness of the re-share path on the decisions made.
- Test cases 6.11 and 6.12 tests effectiveness of CReDiC under simultaneous re-shares and changes of privacy policies.

Consider Test case 6.7 for a closer look. Alice shares a post with categories "friend"

#	test case	expectation	result
6.1	Alice shares a post with category "friend"	Alice's post appears in streams of Bob and Dave (same as without CReDiC)	✔
6.2	Alice changes trust value in category "colleague" to 0.3	Alice's former category "colleague (0.5)" is now "colleague (0.3)"	✔
6.3	Alice adds Charlie to category "friend"	Charlie is displayed in categories "friend" and "colleague" of Alice (same as without CReDiC)	✔
6.4	Alice removes Charlie from category "friend"	Alice is displayed to no longer share with Charlie (same as without CReDiC)	✔
6.5	Dave$\xrightarrow[0.5]{\text{fri,col}}\boxed{\text{Alice}}\xrightarrow{\text{fri,col}}$ Bob	re-sharing is allowed; the post appears in Bob's stream	✔
6.6	Dave$\xrightarrow[0.8]{\text{fri}}\boxed{\text{Alice}}\xrightarrow{\text{fri,col}}$ Bob	re-sharing by Dave is not permitted	✔
6.7	Alice$\xrightarrow[0.5]{\text{fri,col}}\boxed{\text{Bob}}\xrightarrow{\text{fri}}$ Charlie	re-sharing by Bob is allowed; the post appears in Charlie's stream	✔
6.8	like 6.7 but after sharing, Alice sets trust in "friend" to 0.66	re-sharing by Bob is not permitted	✔
6.9	after executing Test case 6.2, Alice$\xrightarrow[0.3]{\text{fri,col}}Bob\xrightarrow{\text{fri}}\boxed{\text{Charlie}}\xrightarrow{\text{fri}}$Bob	re-sharing by Charlie is allowed; the post appears in Bob's stream	✔
6.10	like 6.9 but after re-sharing, Bob removes Charlie as "friend"	re-sharing by Charlie is not permitted	✔
6.11	like 6.10, but Bob changes his privacy policy *while* Charlie's re-sharing is performed	re-sharing by Charlie is not permitted	✘
6.12	Alice$\xrightarrow[0.5]{\text{col}}\boxed{\text{Charlie}}\xrightarrow{\text{fri}}$Bob, while Alice changes her trust in "colleague" to 0.7	re-sharing by Charlie is allowed	✘

Table 6.1.: Test cases for the empirical evaluation of CReDiC's effectiveness

and “colleague” (abbreviated “fri” and “col”). Given that in CReDiC, we assign the use the least trust value for the sensitivity value of the post, this value is $s = \min\{0.8, 0.5\} = 0.5$. The re-share by Bob is expected to be allowed because the path trust in this case is $0.8 \geq \frac{0.35}{1-s} = 0.7$.

We conducted each of the test cases at least once for the version of CReDiC described in this thesis. Each of the test cases was conducted independently from the other test cases. That is, changes to users’ privacy policies were reset before another test case was conducted. The observed results are presented in the rightmost column of Table 6.1. A checkmark (✔) indicates that the test succeeded in all instances of the test; a cross (✘) indicates that the test failed at least once. That is, Test cases 6.1–6.10 succeeded while Test cases 6.11 and 6.12 failed.

Overall, the empirical evaluation of effectiveness based on the presented test cases provides evidence that CReDiC is indeed effective as long as race conditions between decision-making for re-share operations and changes of privacy policies are not present. By deliberately provoking race conditions, we could show that CReDiC can indeed produce unsound as well as intransparent decisions. We consider this ineffectiveness, which appears to be limited to occurrences of race conditions, acceptable for two reasons: Firstly, the cases of ineffectiveness are constrained to the time windows during which decisions for re-share operations are made. These time windows are short compared to, e.g., the times users need to react to posts, as also the results of our performance evaluation suggests. Secondly, in a case of intransparency, a user could succeed in re-sharing by repeating her attempt because CReDiC does not memorize failed attempts, and, in situations in which a case of unsoundness is actually harmful, the user whose racing change of privacy policy was not taken into account for the decision could have made a more sensible choice of privacy policy in the first place. Therefore, we find that CReDiC can be considered sufficiently effective in enforcing controlled re-sharing based on users’ privacy policies.

Run-time Performance

CReDiC is an enforcement mechanism that operates at run-time of its target, the DOSN Diaspora*. As such, it introduces a certain amount of overhead in order to check compliance with users’ privacy policies and, if necessary, prevent violations. In the following, we evaluate the amount of the introduced overhead by comparing the performance of CReDiC to the performance of Diaspora*. For the comparison, we borrow selected test cases from Table 6.1 on page 102.

For the performance evaluation, we selected experiments based on two test cases of operations that are always permitted by CReDiC, as well as three test cases of allowed re-share operations, with one to three involved units. These experiments allow for a comparison of CReDiC’s performance with Diaspora*’s performance.

We conducted the experiments by sending requests to the HTTP servers running the pods and measuring the duration until a response was returned. By including the operations performed by the HTTP server and all potential overhead incurred by Diaspora*’s background services, we aimed at being close to the overhead as it would be perceived by an actual user of Diaspora*. For the sake of conducting many experiments, we conducted the experiments from a command line rather than from a browser. Our results, thus, do

experiment	test case	CReDiC	Diaspora*	absolute overhead	relative overhead
share message	6.1	231.5ms	230.3ms	1.2ms	0.52%
change trust	6.2	36.0ms	31.1ms	4.9ms	15.76%
re-share/intra	6.5	294.6ms	290.7ms	3.9ms	1.34%
re-share/inter	6.7	294.3ms	281.8ms	12.5ms	4.44%

Table 6.2.: Run-time performance evaluation results for CReDiC

not include the page rendering times that a normal user would experience. While we expect the absolute overhead we determined to be faithful nevertheless, we expect the relative overheads to be upper bounds for a normal user's perceived overhead in our setup.

Table 6.2 shows our experimental results. For each of the performed individual experiments we conducted, the table contains a separate row. The first and the second column shows the name of the operation for which the overhead was measured as well as the number of the concrete test case from Table 6.1 on page 102 conducted for the experiment. The third column shows the durations of the respective operations in Diaspora* when CReDiC is enabled and the fourth column shows the durations of the operations in Diaspora* when CReDiC is disabled. The fifth and the sixth column display the absolute overhead (in ms) and the relative overhead (in %), respectively, as computed from the two preceding columns.

Sharing a post took 231.5ms with CReDiC enabled, compared to 230.3ms with CReDiC disabled, which corresponds to an overhead of 1.2ms (0.52%). For re-sharing, we evaluated two cases: intra-provider re-sharing and inter-provider re-sharing. For the two operations, the overhead of CReDiC ranges from 3.9ms to 12.5ms (1.34% to 4.44%). CReDiC's support for dynamically changing trust between users effects an overhead of 4.9ms (15.76%, due to the comparatively low baseline duration of 31.1ms) to operations that change a user's trust in another user. Each duration value given in Table 6.2 reflects the mean of the lower 90th% of 1000 measurements [Oak14, p. 28]. The results confirm that CReDiC efficiently enforces users' privacy policies.

Encapsulation Performance

In order to use CReDiC, CliSeAu must be used to apply CReDiC to Diaspora*. This step has to be performed only once before Diaspora* is started. In the following, we show for a varying number of pods, the time taken by CliSeAu to apply CReDiC.

In the evaluation, we used the same configuration for each pod and its unit except that each unit had a distinct unit identifier. The CoDSPL policies provided to CliSeAu in the individual experiments, thus, varied only in the lists of unit identifiers. For each number of pods encapsulated, we conducted 100 measurings.

The plots in Figure 6.5 on the next page show the results of the evaluation. The horizontal axes of both plots indicate the number of pods encapsulated in the respective experiment. The vertical axes indicate the time taken, in seconds. For each number of pods, the plot

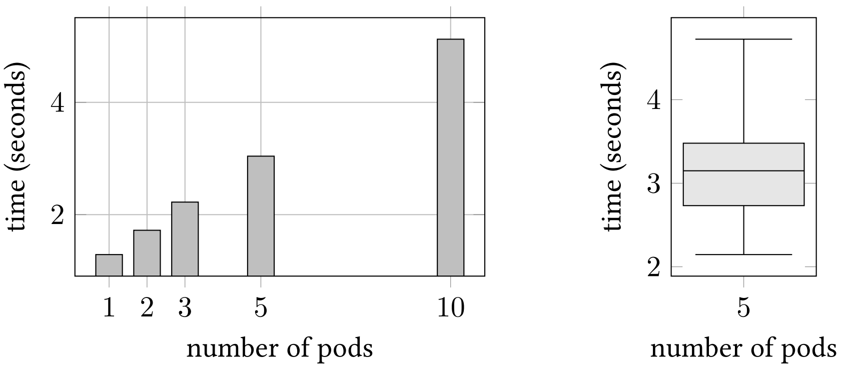


Figure 6.5.: Encapsulation performance evaluation results for CReDiC

on the left of the figure shows a bar depicting the mean of the lower 90th% of the 100 conducted measurements [Oak14, p. 28]. For 5 pods, the plot on the right of the figure contains one box plot, displaying five values: the first and third quartile of measured times (lower and upper extents of the gray boxes), the median of measured times (horizontal lines inside the gray boxes, labeled by the concrete value), and the minimum and maximum measured times (lower and upper extents of the whiskers extending from the boxes).

The left plot in Figure 6.5 shows the results of measuring the time taken by CliSeAu for applying CReDiC for varying numbers of pods. The time grows about linearly from 1.29s for 1 pod to 5.13s for 10 pods. The box plots of the figure shows that for encapsulating 5 agents, the time taken varied by 2.59s between the minimal and maximal time, and by 0.75s between the lower and upper quartile of measurings. While the overall variation in the time taken by the encapsulation was rather large, even the maximal time taken still remained below 1s per pod.

6.5. Summary

We presented four concerns, subgoal selection and realization as well as state-keeping and routing, in delegation-based security policies. In our extension library for modular CoDSPL policies separation of concerns is achieved through the concept of micro-policies, which capture individual subgoals in the decision-making. We applied this extension library in a case study of enforcing controlled re-sharing based on users' privacy policies in decentralized online social networks. The mechanism we developed for the case study, CReDiC, is a usage control mechanism (re-sharing is usage of posts) that enforces controlled re-sharing in a decentralized fashion. Through analytic and empirical evaluations, we provided upper bounds on the number of delegations used by CReDiC and confirmed that CReDiC is efficient and is effective in absence of race conditions during small time windows.

Our extension library provides a base class, ModularLocalPolicy, from which concrete local policies of a CoDSPL policy can inherit to implement our modular design. The base class demands the implementation of micro-policies, a state component, a routing policy, and a concrete factory for micro-policy objects. In consequence, when specifying a

CoDSPL policy, fewer architectural design decisions [Som16, Chapter 6] remain to be made when a CoDSPL policy is specified. As we demonstrate in our case study, our extension library allows for small micro-policy implementations (each at most 41 lines of Java code) and, thereby, facilitates mastering the potential complexity of a CoDSPL policy [BME+07, pp. 13–14]. While our modular design shares the split between decision-making and state-keeping with the policy language of the non-distributed enforcement mechanism Polymer [BLW09], we are the first to propose a modular design for security policies that specify cooperation.

CReDiC, our mechanism for controlled re-sharing in decentralized online social networks allows the users to specify privacy policies in which they categorize users and specify trust values for the individual categories. Such a categorization can be found in many popular online social networks. Our augmentation with trust values and sensitivity values for posts enables CReDiC to allow re-sharing of posts in a controlled fashion only along paths of users whose concatenated trust [HWS09] is sufficiently high with regard to the sensitivity of the post. As of today, decentralized as well as centralized online social networks such as Facebook forbid such controlled re-sharing. Controlled re-sharing allows users to re-share posts and thereby expand the outreach of the post's author and yet prevents re-sharing to insufficiently trusted users. Although CReDiC performs the enforcement in a decentralized fashion by means of cooperation, our empirical evaluation in test cases shows that CReDiC is effective and our performance evaluation showed a minor run-time overhead of 12.5ms (4.44%) on the re-sharing.

Chapter

7

Cooperative Enforcement with Authoritative Delegates

7.1. Introduction

In a distributed program, the agents of the program run concurrently. Under the regime of a security enforcement mechanism, race conditions arise when the ordering of two actions of two agents determines how the enforcement mechanism treats the two actions. For instance, the enforcement mechanism might permit only the first occurring action but prevent the second action from occurring. In the enforcement mechanism, such race conditions can provoke *time-of-check to time-of-use flaws* [PP06, p. 289] – concretely between the time span of making a decision ("check") and the time of enforcing the decision ("use"). A second action occurring within this time frame might render the decision for the first action false.

We propose a technique, called static delegation, for the cooperation between the units of a distributed enforcement mechanism. According to the technique, the decision-making for an event is always delegated to the so-called responsible unit for the event, except when the event was already intercepted by this unit. We identify a class of security properties, called partitionable order-insensitive properties. We show how static delegation can be applied to such properties such that responsible units become authoritative, i.e., enabled to make decisions for events that, when enforced, lead to effective enforcement. For the specification of policies that employ static delegation, we present an extension library for CoDSPL. In a case study of enforcing a Chinese Wall Security Policy [BN89] on a distributed storage service, we show how static delegation can be used can be used for effectively and efficiently enforcing security with CliSeAu.

When static delegation is used as we propose in this chapter, effective enforcement is possible even in presence of race conditions. That is, an adversary that attempts to exploit the timing of events cannot circumvent the enforcement mechanism. Static delegation is parametric in how responsible units are assigned to security-relevant events and particularly allows the responsibility to be distributed over all the units of the enforcement mechanism. Such a decentralized enforcement avoids bottlenecks in the cooperation among units. By predetermining how units cooperate, static delegation simplifies the specification of CoDSPL policies conceptually and in terms of policy size.

Structure The remainder of this chapter is structured as follows. In Section 7.2, we introduce the static delegation and define classes of security properties for which authoritative responsible units are enabled by static delegation. Section 7.3 introduces the extension library for static delegation in CoDSPL. In Section 7.4, we present a case study in which the static delegation is used for effectively enforcing a Chinese Wall Security Policy in a distributed storage service. The case study encompasses a description of the scenario, the CoDSPL policy, and an empirical evaluation of this CoDSPL policy when applied by CliSeAu. Section 7.5 discusses possible optimizations of static delegation. Finally, in Section 7.6, we summarize the content of the chapter.

7.2. Static Delegation

Delegation as a form of cooperation among the units of a distributed enforcement mechanism can be characterized by several attributes, such as *when* delegation takes place, to *which* unit is delegated, and *what* information constitutes a delegation request. Generally, these attributes can depend on the concrete cause (such as an event or a delegation request from another unit), static information of the unit (such as the unit's identity), and dynamic information of the unit (such as the past history of events).

We propose a technique for delegation that reduces the degrees of freedom but is still parametric. In the following, we first describe the technique, which we coin static delegation. We then make several observations about the knowledge that the units of an enforcement mechanism can have. Finally, we introduce the notion of authoritative delegates and present a connection between static delegation and effective enforcement.

Definition 7.1. We say that a unit of a distributed enforcement mechanism performs *static delegation* if each of the following conditions is satisfied:

(a) The unit determines solely from an intercepted event itself whether and to which unit it delegates the decision-making for the event.
(b) The unit transmits, as a delegation request for in intercepted event, solely the event itself as well as, optionally, static information (such as the unit's identifier) that is the same for all events.
(c) Upon receiving a delegation request for an event, whether and where the unit delegates are the same as if the unit intercepted the event itself. If the unit delegates, then it transmits the received delegation request without change. ◊

We call the delegation static because information gathered by the unit at run-time does not influence the delegation behavior. As an information source, we even include the source for an event, namely interception by the unit or delegation from another unit, and require that this does not impact the delegation.

Example 7.1. Consider a distributed enforcement mechanism in which there is one central unit. Each unit that is not the central unit delegates the decision-making for all intercepted events and for all received delegation requests to the central unit. For the delegation request, the unit transmits the intercepted event together with the unit' identifier or, respectively, the received delegation request. Then each of the non-central units performs static delegation. Notably, the condition of Definition 7.1 (a) is satisfied since the unit always delegates to the central unit and, thus, whether and to which unit it delegates particularly does not depend on more than the intercepted event itself. $\Diamond$

We model the parametric aspect of static delegation by the responsibility function of a unit as follows.

Definition 7.2. Let $resp : E \to Ids$ be a total function, where E is a set of events and Ids is a set of unit identifiers. We say that $resp$ is the *responsibility function* of a unit with identifier id if the unit, upon intercepting or being delegated an event $e \in E$,

(a) delegates the decision-making for e to the unit with identifier id' if $resp(e) = id'$ and $id' \neq id$; and

(b) does not delegate the decision-making for e if $resp(e) = id$.

We call the unit with identifier id' *responsible* for an event e if $resp(e) = id'$. $\Diamond$

The definition captures whether and where a unit delegates the decision-making for an event. Since the decision-making for events as well as delegation requests for events in static delegation must be delegated to the respective same units, the responsibility function captures both cases, i.e., (i.e., Definitions 7.1 (a) and 7.1 (c)). As the information transmitted as part of a delegation request for an event is fixed, we do not capture it in the model.

We next lift the notion of static delegation from individual units to enforcement mechanisms consisting of one or more units.

Definition 7.3. An enforcement mechanism performs static delegation if each of its units performs static delegation and all units share the same responsibility function. $\Diamond$

In the following, we make some observations for enforcement mechanisms that perform static delegation.

Observation 1: For each event occurring at run-time in the distributed target of an enforcement mechanism performing static delegation, delegation is either not performed or performed exactly once. This is because a delegate shares the same responsibility function as the delegating unit and, thus, could only delegate to itself.

Observation 2: With static delegation, run-time information does not influence whether, where and what a unit delegates. Thus, static delegation facilitates separation of delegation and state-keeping in implementations of local policies.

Observation 3: At any point in time, a unit of the mechanism could "know"[1] about all intercepted events for which the unit is responsible, as well as the decisions made for them and a partial order on the occurrences of these events.

[1]We use *knowledge* in the sense of the set of propositions that hold in all possible worlds that are consistent with the units local view, as used, e.g., by Fagin et al. [FMH+03].

Observation 4: With static delegation, each event is decided by a single unit (the responsible unit for the event) and based on a single snapshot of this unit.[2]

As with the static delegation technique, we primarily aim at effectiveness, we first focus on Observations 3 and 4. We come back to Observation 2 in Section 7.3 and we discuss possible optimizations with regard to Observation 1 in Section 7.5.

The responsibility function of an enforcement mechanism performing static delegation impacts the knowledge that the units can develop at run-time about the target's behavior. At run-time, a unit gets to know about all intercepted occurrences of those events that the unit is responsible for. Moreover, since the unit also makes the decisions for those events, it also gets to know which of the events actually occur when the decisions are enforced. The unit can use this knowledge for future decisions. The choice of the responsibility function thus, influences whether the unit at run-time has sufficient knowledge to make decisions that render effective enforcement for a given safety property. We capture this condition as follows.

Definition 7.4. We call a unit of an enforcement mechanism *authoritative* for an event of an encapsulated target under a safety property if, for each execution of the encapsulated target and at any point in time during the execution, the unit's local information at that point in time suffices to determine whether the occurrence of the event at this time would comply with the safety property. We call a unit authoritative for a *set* of events if it is authoritative for each event in the set. ◊

The unit's local information aggregates knowledge of static as well as of dynamic kind. Static knowledge can include the safety property and its own as well as the other units' delegation and decision-making procedure. Dynamic knowledge, on the other hand, can include the events that occurred previously at this unit and remote occurrences of events obtained from prior communication with remote units.

Before we come to authoritativity for distributed enforcement mechanisms, we first introduce a particular class of security properties.

Definition 7.5. We call a security property P over event set E *order-insensitive* if

$$\forall t_1, t_2 \in E^* : \big((\forall e \in E \colon |t_1 \restriction \{e\}| = |t_2 \restriction \{e\}|) \rightarrow (t_1 \in P \leftrightarrow t_2 \in P)\big)$$

holds. ◊

Intuitively, a security property is order-insensitive if two sequences of events in which each event occurs the same number of times (i.e., only the ordering of events in the sequences might differ) are classified either both as security-compliant or both as security violations. Among the class of order-insensitive security properties are some concrete properties collected by Fong [Fon04]. From these security properties, we discuss the one-out-of-k property [EAC99; Fon04] in Example 7.2 within this section and study the enforcement of the Chinese Wall property [BN89] in Section 7.4.

In the following, we discuss authoritativity for several constellations in which this property can be achieved.

[2]This property differs from the delegation used in the case study of Chapter 6, in which a re-share can involve several delegation requests and global state can change during the delegation.

Conjecture 7.1. *The one unit of a non-distributed enforcement mechanism is authoritative for the set of all events that are subject to the security property if and only if this security property is soundly and transparently enforceable.* ◊

This is because the unit gets to know about all the events as it must be responsible for them. Hence, whether the unit can make sound and transparent decisions for the events depends solely on whether such deciding is possible at all. Note that we use conjectures for the sake of independence from a concrete formalism for capturing targets and enforcement mechanisms here.

In this thesis, the focus is on distributed targets. As the first step in this direction, the following conjecture makes a transition to a centralized delegation.

Conjecture 7.2. *If an enforcement mechanism performs static delegation with a responsibility function that makes one unit responsible for all events, then this unit is authoritative for the set of all events if the security property is (soundly and transparently) enforceable and order-insensitive.* ◊

A responsible unit as described in Conjecture 7.2 may know all granted events but not necessarily the ordering in which these events occurred. This is because the decisions made by the authoritative unit might be realized at the other units in a different ordering than the decisions, e.g., due to latencies in the cooperation among units. For an order-insensitive security property, this knowledge is sufficient, however, to be authoritative.

As an example for Conjecture 7.2, we discuss the one-out-of-k authorization property [EAC99; Fon04]. Although in the literature this property is described in a non-distributed system setting, we can naturally lift it to a distributed setting as follows.

Example 7.2. System "office" consists of several desktop computers on which from time to time new applications are installed. Each application, for security reasons, is required to access resources only in accordance with some class of applications. For example, an application of class "browser" may access network connections, access temporary files, and access the display to show content [Fon04]. Among the k classes of applications, the class that applies to an application is determined dynamically by the accesses made. No application may exceed the access rights provided by its class. ◊

The security property is enforceable, as it is a safety property and all events subject to the property are controllable by an enforcement mechanism. The property is order-insensitive, as the order of an application's permissible accesses is irrelevant for the classification and for the determination of which accesses the application may perform at any point. That is, by designing the units with static delegation where all events of a new application are delegated to one and the same responsible unit, this unit is authoritative for all the application's access events.

In Conjecture 7.2, a single unit is responsible for all events. That is, the conjecture refers to a centralized cooperation by the enforcement mechanism. We next address decentralized cooperation.

A security property might be the conjunction of several independent security properties, which we use as starting point for the decentralization. For instance, the property might constrain the permissible events of each user or each agent of the target independently from the events of other users and, respectively, agents. Rather than users or agents, the security property could also constrain the permissible events of resources managed by the target, such as files. We capture such conjoined security properties as follows.

Definition 7.6. We call a security property P over event set E *partitionable* into a partition $(E_i)_{i \in I}$ of E if a family of security properties $(P_i)_{i \in I}$ with $P_i \in {E_i}^*$ exists such that $P = \{t \in E^* \mid \forall i \in I : ((t \upharpoonright E_i) \in P_i)\}$ holds. $\Diamond$

In the definition, independence between the individual security properties P_i is achieved by the constraint that they are sequences over mutually disjoint sets of events. That the overall property, P, is indeed the conjunction of the individual properties is adequately captured by the last constraint in the definition: A sequence of events complies with P if and only if its projection to each of the individual properties' domains complies with the respective property. The following theorem provides two essential properties of partitionable security properties.

Theorem 7.1. *Let E be a set of events and let P be a security property over E.*

(a) P is partitionable into $(E_i)_{i \in \{\top\}}$ where $E_\top = E$.

(b) Let $\mathcal{E} = (E_i)_{i \in I}$ and $\mathcal{E}' = (E'_j)_{j \in \mathcal{J}}$ be partitions of E such that $\mathcal{E}$ is a refinement of $\mathcal{E}'$ and P is partitionable into $\mathcal{E}$. Then P is also partitionable into $\mathcal{E}'$. $\Diamond$

The theorem shows that there is a "trivial" partition into which every security property is partitionable and that partitionability is closed under less fine-grained partitions of events. A formal proof of the theorem can be found in Appendix A.1 on page 189.

In static delegation, the responsibility function establishes a partition of events through its assignment of the event to the respective responsible units.

Definition 7.7. Let $resp : E \rightarrow Ids$ be a responsibility function over sets E and Ids of events and, respectively, unit identifiers. Let $I = \{id \in Ids \mid \exists e \in E : resp(e) = id\}$ be the set of identifiers of units that are responsible for at least one event. We call $(E_{id})_{id \in I}$, where $E_{id} = \{e \in E \mid resp(e) = id\}$, the *induced event partition* of *resp*. $\Diamond$

Note that the induced event partition is indeed a partition of the respective set of events E: all elements of the partition are non-empty, mutually disjoint, and together cover the set of all events. The induced event partition is determined by the responsibility function, i.e., by a design choice in how the mechanism cooperates. In general, the induced event partition is unrelated to the partitionability of a security property.

Conjecture 7.3. *Let an enforcement mechanism performing static delegation be given and let $resp : E \rightarrow Ids$ be the responsibility function of the units. Let P be an enforceable security property over E. If P is order-insensitive and partitionable into the induced event partition of resp, then at any point in time during the execution of a target the responsible unit for an event is authoritative for this event.* $\Diamond$

A unit as described in Conjecture 7.3 may know all events for which it is responsible. If the security property is partitionable knowing about these events is sufficient for making sound and transparent decisions. As before, the unit does not necessarily know the ordering in which these events occurred, because the decisions made by a unit might be realized at the other units in a different ordering than the decisions. For an order-insensitive security property, this knowledge is sufficient, however, to be authoritative. Note that

Conjecture 7.3 generalizes Conjecture 7.2: The case that a single unit is responsible for all events is precisely the case considered in the latter conjecture and partitionability in this case is always given, as shown in Theorem 7.1 (a). Finally, we provide an example demonstrating that when ordering is relevant for a security property then authoritativity cannot be achieved in general.

Example 7.3. System "2-store" consists of two agents, each offering storage space to users. Users can utilize storage at each of the agents. However, the total amount of storage available to all users on the system is limited by a fixed quota. Users can consume storage space (by uploading files to the system) and can release storage space (by deleting files). The security property for the system specifies that no user at any time may exceed her quota on the system. $\Diamond$

The security property in the example is partitionable into the individual users' sets of consume and release events. However, the security property is not order-insensitive, as the ordering among consume and release events matters: If the quota are 3, then the sequence $t = \langle \text{consume}(2), \text{consume}(2), \text{release}(1) \rangle$ of events by one user violates the property while $t' = \langle \text{consume}(2), \text{release}(1), \text{consume}(2) \rangle$ does not. Regarding authoritativity, we consider sequence t after its first event, consume(2), was granted. When a delegation request for the second event, consume(2) is received by the responsible unit for the event, this unit must make a decision. If the unit allows the event to occur, then this decision is unsound if the release(1) occurs only after the second event occurred. On the other hand, if the unit disallows the event to occur, then this decision is intransparent if, before this decision is *realized*, the event release(1) is intercepted and allowed. Hence, static delegation cannot establish authoritative units for all events in this example. Notably, the security property is not order-insensitive, but for deciding whether permitting an event under a given execution does not require knowing the ordering of events in the execution.

The connection between static delegation and authoritativity that we have established with Conjectures 7.1 to 7.3 provides sufficient conditions for effective enforcement. In particular, we have identified a class of security properties – order-insensitive partitionable properties – that allow for an effective decentralized enforcement based on the static delegation technique.

7.3. Static Delegation in CoDSPL

For specifying local policies that perform static delegation, we present an extension library for CoDSPL. The extension library for static delegation is implemented as a Java package that is part of CliSeAu but is not mandatory to be used by CoDSPL policies. The extension library is provides simple means for specifying local policies using static delegation.

The extension library for static delegation consists of three classes, which are shown with shaded background in Figure 7.1 on the following page. For employing the extension library in a CoDSPL policy, a subclass of the abstract class StaticDelegationPolicy, which itself is a subclass of LocalPolicy, must be used for the local policy component of the units specified by the CoDSPL policy. The abstract class provides a public constructor, which receives as argument the identifier of the unit. For the static delegation, it provides three protected abstract methods that a concrete local policy must implement:

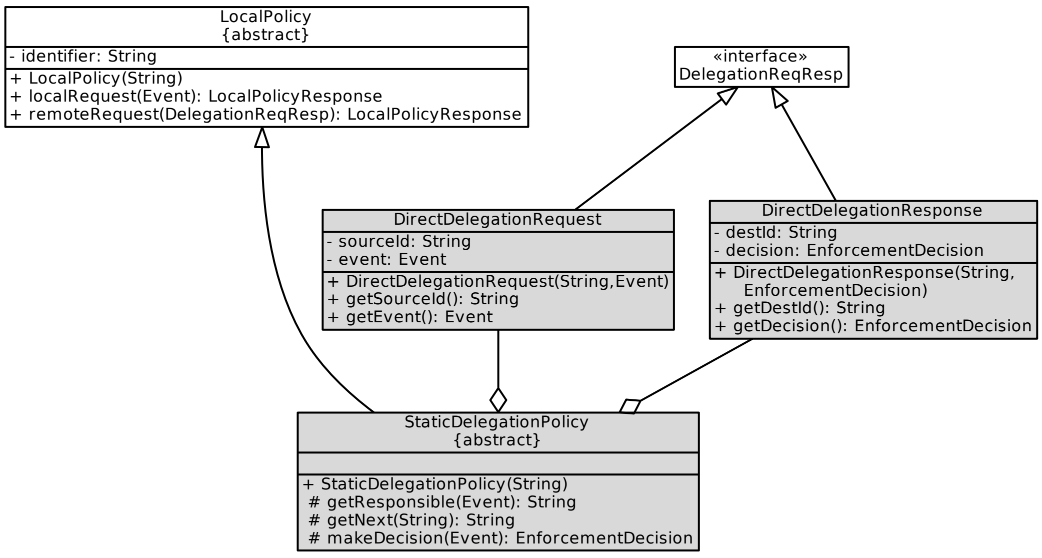

LocalPolicy
{abstract}
- identifier: String
+ LocalPolicy(String)
+ localRequest(Event): LocalPolicyResponse
+ remoteRequest(DelegationReqResp): LocalPolicyResponse
«interface»
DelegationReqResp
DirectDelegationRequest
- sourceId: String
- event: Event
+ DirectDelegationRequest(String,Event)
+ getSourceId(): String
+ getEvent(): Event
DirectDelegationResponse
- destId: String
- decision: EnforcementDecision
+ DirectDelegationResponse(String, EnforcementDecision)
+ getDestId(): String
+ getDecision(): EnforcementDecision
StaticDelegationPolicy
{abstract}
+ StaticDelegationPolicy(String)
# getResponsible(Event): String
# getNext(String): String
# makeDecision(Event): EnforcementDecision

Figure 7.1.: Low-level architecture of the extension library for static delegation in CoDSPL policies (UML class diagram)

- The method getResponsible takes an event and returns the identifier of the responsible unit. This method corresponds to the responsibility function we introduced in Section 7.2.
- The method makeDecision takes an event and returns a decision. This method may be invoked by the extension library with a locally intercepted event or with an event for which the decision-making was delegated by another unit.
- Finally, the method getNext takes a unit identifier and returns a unit identifier. This method allows one to implement routing separately from the determination of the responsible unit.

Through these three methods, our extension library facilitates the implementation of local policies that satisfy the conditions of static delegation of a unit (Definition 7.1 on page 108). Note that the extension library does not enforce that local policies satisfy the conditions, as a StaticDelegationPolicy subclass could, e.g., implement the determination of delegates based on more than just the respective event.

Internally the StaticDelegationPolicy class uses two classes, one for delegation requests and one for delegation responses. The former holds the event for which the decision-making is being delegated as well as the unit identifier of the event source. The class for delegation responses holds the decision made as well as the unit identifier to which the decision is to be delivered. Objects of these classes are able to encapsulate sufficient information for the StaticDelegationPolicy to perform the delegation in a way that a unit receiving a delegation request can return a decision to the unit that issued the delegation request. The use of these two classes, however, is transparent to local policies implemented using the extension library.

We illustrate the delegation specified by a local policy class that inherits from class StaticDelegationPolicy in the sequence diagram depicted in Figure 7.2 on the facing page.

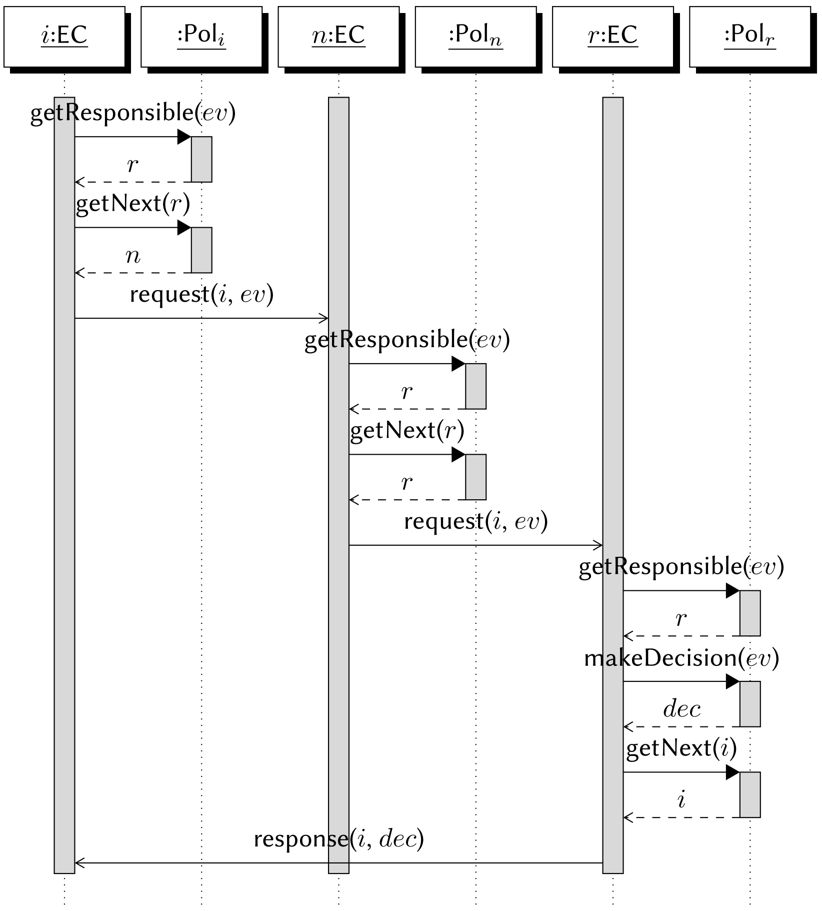


Figure 7.2.: Cooperation of units with static delegation (UML sequence diagram)

For the illustration we take the case that an event ev is intercepted by a unit that is not responsible for this event. The sequence diagram displays three units, named i, n, and r as well as their local policies of types Pol$_i$, Pol$_n$, and Pol$_r$, each with their own ***lifeline*** (the parallel vertical lines). In the sequence diagram, i is the unit at which an event ev is intercepted. By calling methods getResponsible and getNext, the unit uses Pol$_i$ to determine the responsible unit r for event evas well as the next unit n to unit r. The unit i then delegates to n, sending the event as well as its own identifier. At unit n, the same control flow is triggered by receiving the delegation request. At the responsible unit, the control flow starts with getResponsible as for the other units, but diverges when the unit obtains that it is responsible. Thereafter, the unit calls method makeDecision of Pol$_r$ to compute decision dec for ev and then calls method getNext to obtain the next unit to unit i, at which ev was intercepted. Finally, unit $r$sends decision dec and identifier i of the destination in a delegation response to unit i, at which the decision would afterwards be realized.

The sequence diagram particularly illustrates two properties of static delegation. Firstly, the decision-making for an event via method makeDecision is triggered only a single time during the process. That is, the decision-making is performed based on the one partial snapshot of the enforcement mechanism that the responsible unit has at the time of deciding. This is in contrast to, e.g., the decision-making of CReDiC (see Section 6.4), which can involve partial decisions of several units at various points in time and, thus, enable race conditions that result in unsound or intransparent decisions. Secondly, a decision made by a responsible unit is not further processed by the local policy of the recipient unit. That is, the responsible unit “knows” that its decisions are going to be realized and use this in its decision-making.[3]

Overall, the proposed extension library provides means for implementing local policies of CoDSPL policies that perform static delegation. The interface that such a local policy must implement is simple – three methods – and facilitates separation of concerns for delegation, decision-making, and routing.

7.4. Case Study

We demonstrate the application of the static delegation technique in a case study. In the case study, we show that static delegation can be used for effectively enforcing Chinese Wall Security Policies. For the specification of an enforcement mechanism for Chinese Walls, we utilize CoDSPL with the extension library for static delegation and provide evidence that the extension library enables modular local policy implementations. We empirically confirm that the enforcement can be performed effectively with moderate performance overhead.

The case study is structured as follows. We start by introducing the application scenario in Section 7.4.1. In Section 7.4.2 we develop a local policy for a decentralized cooperative enforcement in the application scenario based on the extension library introduced in Section 7.3. We evaluate first the faithfulness of the policy analytically, in Section 7.4.3. Afterwards, in Section 7.4.4 we provide an empirical evaluation of the faithfulness and performance of the policy in an experimental setting.

7.4.1. Application Scenario

The application scenario we investigate in this section is a distributed storage service in which conflicts of interest shall be prevented programmatically. We have briefly discussed this scenario already in Example 1.1 on page 6.

Target and stakeholders The target of our case study is a distributed storage service. It consists of several services, each of which constituting an agent of the target. The services store files and offer an interface to its users for downloading and uploading files. Figure 7.3 on the facing page shows the system’s architecture.

Overall, we identify three groups of stakeholders in the application scenario. The users of the storage service process the files as part of contracts with the proprietors. The *proprietors* are the legal owners of the files. They might interact with the storage service for placing the files. This task, however, could also be performed by the users as part of their contractual obligations. For an example, users could be financial auditors, files could

[3] We neglect circumstances such as message loss in the communication between units or an inadequate choice for the recipient’s enforcer component here.

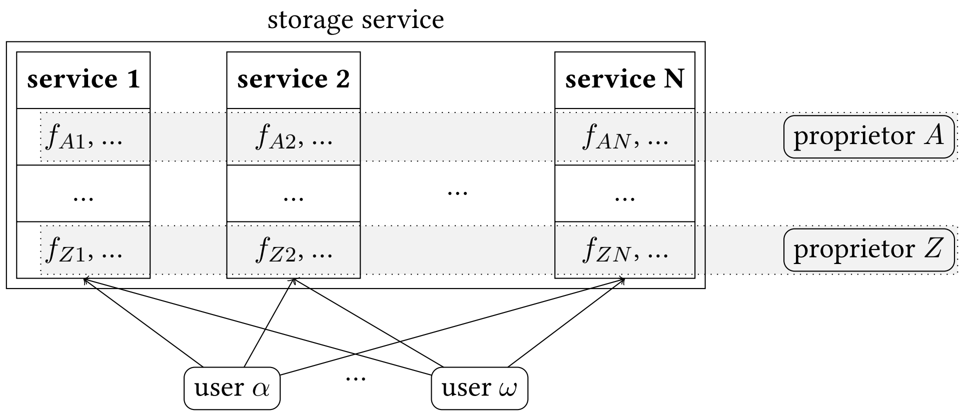


Figure 7.3.: Storage service architecture and stakeholders

be balance sheets, and proprietors could be large companies, such as banks. Next to users and proprietors, the third stakeholder is the provider of the storage service. Users and proprietors are depicted in the figure by rounded boxes. The shaded areas around the proprietors indicates the files that the respective proprietors own. Finally, as the arrows in the figure indicate, users can in general access files on any of the system's services.

Security requirement The security requirement whose enforcement we investigate in this case study is a requirement by the proprietors:

> No user may at any point in time have had access to two files that belong to competing proprietors.

The security requirement aims to prevent conflicts of interest among the users that might arise from competitors' business secrets. Note that the application scenario in practice typically involves further security requirements. For instance, proprietors would require that only authenticated and authorized users can access their files. We neglect this requirement in our case study, as commodity storage server protocols and implementations (FTP, HTTP, and cloud implementations on top of these protocols) already offer security features for them. Also outside the scope of our case study are requirements such as low downtime (availability) or low probability of data loss (integrity). Instead, our case study focuses on the prevention of conflicts of interest, a security requirement that is typically not fulfilled by commodity storage software.

The concrete requirement against conflicts of interest that we choose for the case study is the *Chinese Wall Security Policy* (brief: *CWSP*), which specifies:

> "Access [of a subject to an object] is only granted if the object requested:
> 1. is in the same company dataset as an object already accessed by that subject, i.e., within the Wall, or
> 2. belongs to an entirely different conflict of interest class." [BN89]

The requirement, thus, captures that a subject may access objects from the dataset of at most one company within a conflict of interest class. By the term *conflict of interest class* (brief: *COI class*) we refer to a set of companies that compete with each other. Transferred to our application scenario, users are the subjects, files are the objects, proprietors are the companies, and conflict of interest classes are sets of competing proprietors.

7.4.2. ChESt – CoDSPL policies for Chinese Walls

ChESt (abbreviating "Chinese Wall Enforcement for Storage Systems with CliSeAu") is a local policy for enforcing Chinese Wall Security Policies in distributed storage services and CoDSPL policies based on this local policy for two FTP servers, *AnomicFTPD* [Ano09] and *simple-ftpd* [Sim10]. The local policy is built with the extension library for static delegation proposed in Section 7.3 and utilizes a responsibility function that is suitable for authoritativity in the application scenario. In the following, we first introduce the event objects and states used by the local policy, afterwards describe the proposed local policy, and finally cover briefly how the local policy is completed to CoDSPL policies for AnomicFTPD and simple-ftpd.

For event objects we define a class, ConflictClassAccessEvent, that contains fields for four attributes: the name of the file being accessed, the proprietor to whom the file belongs, the COI class of this proprietor, and the user on whose behalf the access shall be performed. All fields are designed to hold String objects. In the model we pursue with this design of event objects, we rely on information for all four attributes to be available upon an intercepted access operation. While the name of the file is naturally available, the others deserve an explanation. The accessing user must be authenticated and authorized for accessing any files at all, so the user's identity is known and, depending on the implementation of the target, available as part of the session information maintained for the user's connection. The proprietor could be the owner of the file in the storage service's file system. Finally, the COI class might not be available in commodity storage services as such, but could, for instance, be preserved by the group of the file being accessed.

Listing 7.1 on the next page shows the code of ChESt's local policy class, which inherits from StaticDelegationPolicy. That is, the local policy of ChESt is built with the extension library for static delegation proposed in Section 7.3. The code omits exception handling, the constructor, and inline code comments. In the following, we discuss how the implementation realizes state, the responsibility function, routing, and decision-making.

The state of a local policy of a ChESt unit holds, at any point during run-time, an abstraction of the granted accesses of all users for which the unit is responsible. Concretely, the state stores for each user and each COI class the proprietor whose files the user has accessed. If a COI class has not been accessed by the user, then no entry for this pair is stored. Technically, the state is stored as a map from String objects to maps from String to String. This information is sufficient for recording all information relevant to the unit for making sound and transparent decisions for the Chinese Wall Security Policy. Further information, such as file names or ordering of accesses of the user are not required and, thus, abstracted away by the state for reducing the memory consumption of the unit.

In the design of the responsibility function for static delegation, we assign the responsible unit to an event based on the joint hash value of the user and the COI class of the event object. Concretely, the responsible unit is computed by taking this hash value modulo the total number of units in the enforcement mechanism and appended to the prefix "id-". The

```
public class ChineseWallPolicy extends StaticDelegationPolicy {
  private Map<String, HashMap<String, String>> accessedFiles;

  protected String getResponsible(Event event) {
    ConflictClassAccessEvent ChWev = (ConflictClassAccessEvent)event;
    return ”id−” + (Math.abs((ChWev.conflictClass.hashCode() ^
         ChWev.userName.hashCode()) % numberOfUnits));
  }
  protected String getNext(String destinationId) { return destinationId; }
  protected EnforcementDecision makeDecision(Event event) {
    ConflictClassAccessEvent ChWev = (ConflictClassAccessEvent)event;
    HashMap<String,String> accessed = accessedFiles.getOrDefault(ChWev.userName,
        new HashMap<String,String>(1));
    String accessedCompany = accessed.get(ChWev.conflictClass);
    if (accessedCompany != null && !ChWev.company.equals(accessedCompany)) {
      return BinaryDecision.REJECT;
    } else {
      accessed.put(ChWev.conflictClass, ChWev.company);
      accessedFiles.put(ChWev.userName, accessed);
      return BinaryDecision.PERMIT;
    }
}}
```

Listing 7.1.: The local policy of ChESt

encapsulation descriptions ChESt are defined such that they declare exactly those unit identifiers. The assignment of responsible units through hash values achieves a uniform distribution of responsibilities for avoiding bottlenecks. More importantly, the assignment enables authoritative units as we will analyze in Section 7.4.3.

The decision-making of the local policy uses the state to check whether the user who performs the intercepted access event already accessed files from the same COI class. If this is the case, the access is permitted if and only if the prior accesses to the COI class were to files of the same proprietor. Otherwise, i.e., if the user never accessed files from the COI class, the access is always permitted. When an access to a file is permitted, the local policy updates its state as follows. If the user already accessed a file from the same COI class before, then the state is not changed, because the user's access to files of the same proprietor in the COI class has already been recorded in the state. Otherwise, the state is updated to hold all information that was in the state before and additionally the information that the user accessed, in the COI class of the access, files from the proprietor of the access. That is, overall decision-making and state-keeping are implemented as for a non-distributed target.

7.4.3. Analytic Evaluation of ChESt

We analyze the design of ChESt with regard to authoritativity and analyze the implementation of ChESt with regard to their complexity.

Authoritativity We introduced the notion of authoritativity in Definition 7.4 to capture that a unit is able to determine whether an event would comply with or violate a security property. This ability, in turn, allows the unit to make sound and transparent decisions. In Conjecture 7.3, we provided a sufficient condition for authoritativity under decentralized enforcement based on static delegation. In the following, we utilize this condition to show that ChESt establishes authoritative units.

As a first step, we make an observation about properties of the Chinese Wall Security Policy.

Conjecture 7.4. *The Chinese Wall Security Policy, as introduced in this thesis based on Brewer and Nash [BN89], is order-insensitive and is partitionable into the sets of access events that share the same user and COI class.* ◊

Firstly, the CWSP is clearly order-insensitive, as it only constrains which access events may occur together in one execution of a target and which may not. Exchanging the ordering between access events in an execution, thus, does not change compliance with the CWSP. Secondly, the CWSP is also partitionable into the sets of events named in the conjecture. This is because an execution of a target complies with the CWSP if and only if the entirety of each user's accesses to each of the COI classes comply with the CWSP. Access events of distinct users or to distinct COI classes by the same user do not interfere with each other under the CWSP. The family of security properties demanded by Definition 7.6, thus, is constituted by the individual security properties that require for the respective user and COI class that all accesses of the user to this COI class are to files of the same proprietor.

The CWSP is not only partitionable as provided in Conjecture 7.4 but is also partitionable into the partition induced by the responsibility function in ChESt. Recall that by this responsibility function, a unit is responsible for the union of all access events of particular combinations of users and COI classes – namely those exhibiting a particular hash value. Consequentially, the partition provided in Conjecture 7.4 is a refinement of the induced event partition. Through Theorem 7.1 (b), we therefore obtain that the CWSP is indeed partitionable also into the partition induced by the responsibility function in ChESt.

The above arguments, together with Conjecture 7.3, allow the conclusion that for all access events the respective responsible unit is authoritative for the event. That is, in ChESt, sound and transparent decisions for all access events can be made by the respective responsible units.

Complexity of local policy design and implementation The local policy of ChESt, implemented in class ChineseWallPolicy, consists of 56 SLOC. Of these lines, 25 SLOC are for decision-making and state-keeping in method makeDecision, four SLOC are for the responsibility function and one SLOC implements for routing in methods getResponsible and getNext. The remaining lines distribute over package imports, the field definition for the state, and boiler-plate code. In addition, ChESt consists of 18 SLOC for the definition of the event objects 11 SLOC for the decision objects, 35 SLOC for the countermeasure objects, and 54 SLOC for the event factory and enforcer factory. In total, ChESt thus consists of 174 SLOC. These numbers indicate that policies for coordinated decentralized enforcement can be implemented in a modular fashion with low code complexity in CoDSPL based on the extension library for static delegation.

7.4.4. Empirical Evaluation of the Policy

We empirically evaluate effectiveness and performance of ChESt. We conduct the evaluation along the lines as the evaluation of Chapter 6. That is, concerning the effectiveness, we evaluate selected policy-compliant as well as policy-violating test cases. Concerning the performance, we evaluate the overhead caused by ChESt at run-time of the two targets as well as the time taken to instrument these targets using CliSeAu. The evaluation shall show that the Chinese Wall Security Policy is effectively enforced in the application scenario and with reasonable performance.

For the experimental evaluation, we used three Intel Quad-Core (i5-4590 3.3GHz) machines with 32 GB RAM. The machines ran Ubuntu 14.04.2 LTS with a 3.13.0 kernel and OpenJDK 7 for running the FTP servers. We measured page fetch times as well as the encapsulation performance on an Intel Quad-Core (i7-6600U 2.6GHz) machine with 16 GB RAM, a 4.4.26 kernel, OpenJDK 8, and AspectJ 1.8.10, using curl 7.52.1 with `trace-time` option.

The configuration of the targets is the same for AnomicFTPD and for simple-ftpd and is as described in the following. The configuration provides the files for both the effectiveness test cases and the run-time performance evaluation. The target consists of three agents (FTP servers), each running on a separate computer. Two users have accounts on each of the target's agents, "user0" and "user1". There are 1001 COI classes, named "coi0" to "coi1000". For class "coi0", there are two files of proprietor "p0", named "`coi0:p0:f0`" and "`coi0:p0:f1`" and one file of proprietor "p1", named "`coi0:p1:f0`". For the other COI classes, there are one file for each of the two proprietors, named "`coiX:pY:f0`", where X is the number of the COI class and Y the number of the proprietor. Each of these files contain random data of size 100 kilobytes. Moreover, there are one file each for file sizes 0 kilobytes to 1000 kilobytes in steps of 100 kilobytes, named "`coi0:p0:sN`" where N is the number of kilobytes. All of the aforementioned files are stored in the same directory and on each agent of the target.

Effectiveness of the Enforcement

In this section, we show by means of several test cases that ChESt is effective in enforcing the Chinese Wall Security Policy in storage systems, particularly under race conditions.

We select test cases from two categories: tests without race conditions and tests with race conditions. Table 7.1 on the following page displays the list of test cases, along with their expected results and their empirically observed results.

The test cases are split into two groups, separated by a horizontal rule in the table. Test cases 7.1 and 7.2 cover the two conceptual possibilities for a single user to download distinct files in compliance with the CWSP. Test case 7.3 covers the one possibility for a user to violate the CWSP and Test case 7.4 covers same situation except that the downloads are conducted by two distinct users and, as such, compliant with the CWSP. Finally, Test case 7.5 modifies Test case 7.3 to several in which the user attempts to circumvent ChESt through race conditions. For each test case, the table provides the test itself as well as the expected result from an effective enforcement mechanism for CWSP.

#	test case	expectation	result
7.1	user successively downloads two distinct files from same proprietor	both downloads succeed	✔
7.2	user successively downloads two files from different COI classes	both downloads succeed	✔
7.3	user successively downloads two files from same COI classes but different proprietor	first download succeeds, second is denied	✔
7.4	two users successively download one file each from same COI classes but different proprietor	both downloads succeed	✔
7.5	for 1000 distinct COI classes, a user concurrently downloads two files with different proprietor from this COI class	for each of the COI classes, one of the downloads succeeds and the other one is denied	✔

Table 7.1.: Test cases for the empirical evaluation of ChESt's effectiveness

We conducted each of Test cases 7.1–7.4 manually for each of AnomicFTPD and simple-ftpd, which were both encapsulated with ChESt. Each of the two downloads were performed from distinct encapsulated agents. The test cases were conducted independently from each other by restarting the encapsulated target between the tests. Test case 7.5 was conducted in an automated fashion through a script. In the script, we also measured at which points in time the concurrent downloads are requested from the servers for gathering evidence whether the downloads are indeed started sufficiently close together for discovering potential flaws. The observed results are presented in the rightmost column of Table 7.1. A checkmark (✔) indicates that the respective observed result equaled the formulated expectation for both AnomicFTPD and simple-ftpd each of the times the test was conducted; a cross (✘) indicates that observed and expected result differed at least once for at least one of the two targets. As the table shows, in all conducted test cases the observed results match the expected results. We are confident that the automation for Test case 7.5 was indeed suitable for discovering potential flaws, because the points in time of the concurrent downloads were often only few microseconds apart.

Overall, the empirical evaluation of ChESt's effectiveness based on the presented test cases provides strong evidence that ChESt indeed effectively enforces the CWSP in storage systems, even in presence of race conditions.

Run-time Performance

ChESt, like CReDiC, is an enforcement mechanism and, as such, introduces a certain amount of overhead at the run-time of its target. In the following, we evaluate the amount of overhead in our experimental setup by comparing the performance of ChESt to the performance of the target. We perform this comparison for the two targets of ChESt, AnomicFTPD and simple-ftpd.

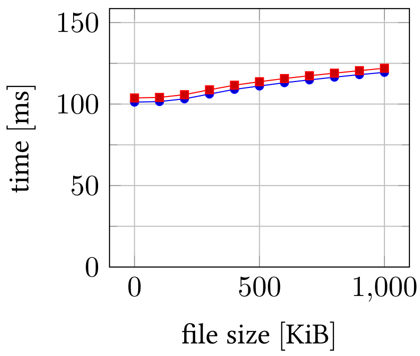
For the performance evaluation, we selected two experiments. In both experiments, we essentially perform Test case 7.1, i.e., permitted downloads, except that always the same file is downloaded. The first experiment varies the size of the file being downloaded as a parameter whose choice is expected to be irrelevant for the performance of ChESt. In this experiment, we choose to perform the download at an agent whose unit is also the responsible unit for this download event. That is, in the experiment, no delegation takes place. The second experiment uses a fixed file size (100 kilobytes) and varies the number of hops taken in the cooperation among units. For this, we introduced a variation in the routing of ChESt (i.e., in method getNext) such that the units establish a ring topology and ensured for a varying number of agents that the responsible unit differs from the unit that intercepts the access event. This way, the number of hops taken equals the number of agents in the storage system. Our hypotheses for the experiments are that the file size does not influence the overhead caused by ChESt and that the overhead increases linearly in the number of hops.

We conducted the experiments by performing FTP file downloads from the agents of the storage service and measuring the duration until the file was received. In the measured durations, we included the whole time frame from sending the FTP request until the receipt of the complete file. The measured durations do not include further parts of the FTP connection, such as logging in, changing the directory to the directory of the respective file, and logging out. We expect these durations to adequately reflect the durations and, consequentially, also overheads as they would be perceived by an actual user of the respective target. The measurings were performed and recorded by the curl tool, which supports downloading of files from FTP servers and recording time stamps with microseconds precision.

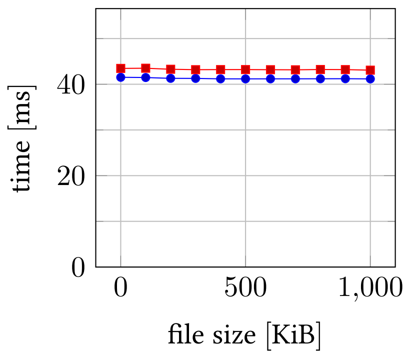
Figure 7.4 on the next page shows our experimental results. For each of the two servers, the figure contains three plots. The first plots, (a) and (b), show the absolute times that the downloads of files took, without ChESt (blue circle marks) and with ChESt (red box marks). The second plots, (c) and (d), show the absolute (blue circle marks) and relative (red box marks) overhead induced by ChESt. The absolute overhead is the difference between the first two plots, and the relative overhead is the absolute overhead divided by the absolute time without ChESt. Both plots show results for ChESt in which no cooperation takes place. The third plots, (e) and (f), show the absolute overhead induced by ChESt for different numbers of hops in the cooperation between units.

Without ChESt, downloads of files from AnomicFTPD took from 101.2ms to 119.5ms, depending on the file size. Downloads of files from simple-ftpd took between 41.2ms and 41.5ms. With ChESt, downloads from AnomicFTPD took from 103.8ms to 122.0ms and downloads from simple-ftpd took between 43.1ms and 43.5ms. For both FTP servers, the resulting absolute overhead ranged between 2.44ms and 2.64ms (AnomicFTPD) and, respectively, 1.91ms and 2.04ms (simple-ftpd). The relative overhead dropped from 2.54% to 2.11% for AnomicFTPD and varied between 4.63% and 4.94% for simple-ftpd. For two hops in the cooperation, the absolute overhead for downloading a 100 kilobytes file was 6.07ms for AnomicFTPD and 5.00ms for simple-ftpd. For three hops, the overhead was 8.26ms and, respectively, 7.25ms.

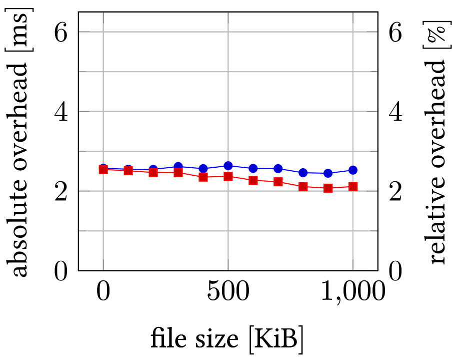
The experiments show that the two FTP servers differ in their efficiency, presumably due to their internal use of different buffering techniques and buffer sizes for implementing the



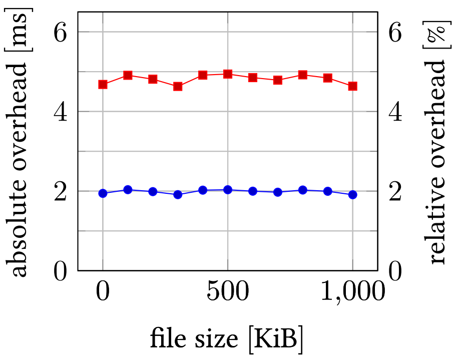
(a) AnomicFTPD, by file size



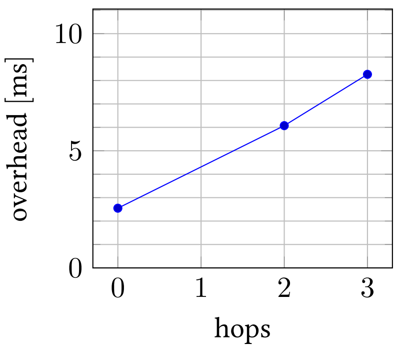
(b) simple-ftpd, by file size



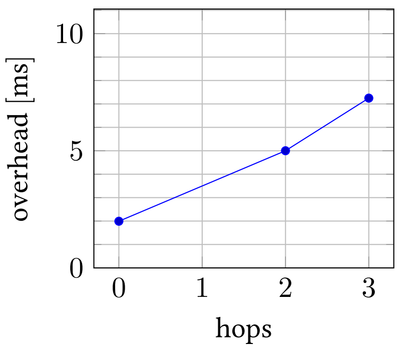
(c) AnomicFTPD, by file size



(d) simple-ftpd, by file size



(e) AnomicFTPD, by path length



(f) simple-ftpd, by path length

Figure 7.4.: Run-time performance evaluation results for ChESt

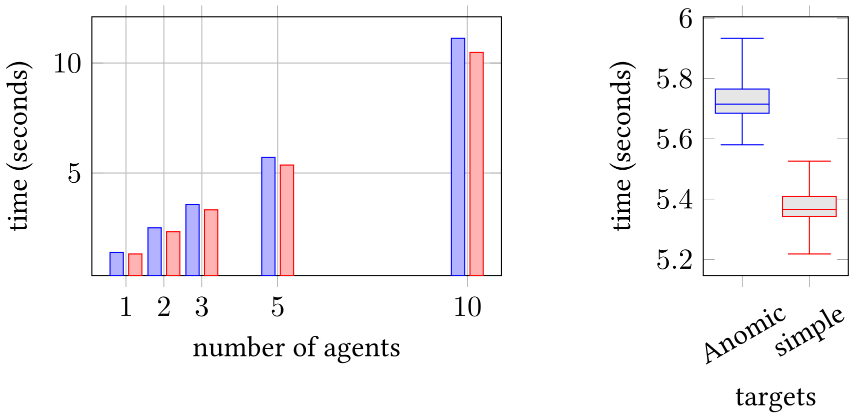


Figure 7.5.: Encapsulation performance evaluation results for ChESt

protocol. We verified particularly for simple-ftpd that downloads were correctly conducted in the experiments. The run-time overhead introduced by enforcing the CWSP with ChESt shows to be independent of the sizes of downloaded files, as expected. Moreover, the experiments reveal a roughly linear increase of the overhead in number of hops by around 2ms per hop. Overall, the results confirm that ChESt efficiently enforces the CWSP in the distributed storage service.

Encapsulation Performance

In order to use ChESt, CliSeAu must be used to encapsulate the target before the target is started. This step has to be performed only once before the encapsulated target is started. In the following, we evaluate for a varying number of agents of the two targets – AnomicFTPD and simple-ftpd – the time taken by CliSeAu to encapsulate the target.

In the evaluation, we used the same configuration for each agent and its unit except that each unit had a distinct unit identifier. The CoDSPL policies provided to CliSeAu in the individual experiments, thus, varied only in the lists of unit identifiers. For each number of agents encapsulated, we conducted 100 measurings. The plots in Figure 7.5 show the results of the evaluation. The horizontal axes of both plots indicate the number of agents encapsulated in the respective experiment. The vertical axes indicate the time taken, in seconds. The plot on the left of the figure shows two bars for each number of agents, depicting the mean of the lower 90th% of the 100 conducted measurements [Oak14, p. 28] for AnomicFTPD (left, blue) and simple-ftpd (right, red), respectively. For 5 agents, the plot on the right of the figure contains two box plots, one for AnomicFTPD and one for simple-ftpd. The kind of data displayed by each box plot – minimum, first quartile, median, third quartile, and maximum – are the same as in Figure 6.5 on page 105.

The left plot in Figure 7.5 shows the results of measuring the time taken by CliSeAu for applying ChESt to each of the two FTP servers. For AnomicFTPD, the time grows from 1.39s for 1 agent to 11.12s for 10 agents. For simple-ftpd, the time grows from 1.32s for 1 agent to 10.48s for 10 agents. With both servers, as the figure shows, the time grows about linearly with the number of agents. The instrumentation that is part of

CliSeAu's encapsulation increased the JAR file of AnomicFTPD from 32.6KB to 38.1KB and the JAR file of simple-ftpd from 18.4KB to 22.9KB. The box plots of the figure shows that for encapsulating 5 agents, the time taken varied for AnomicFTPD and, respectively, simple-ftpd by 0.35s and 0.31s between the minimal and maximal time, and by 0.08s and 0.07s between the lower and upper quartile of measurings. That is, the variation in the time taken is small.

7.5. Optimizations

Static delegation enables effective enforcement while retaining a conceptual simplicity. Optimizations in terms of the number of delegations are possible when simplicity is sacrificed to some extent.

Regarding the amount of delegations performed for enforcing a particular security property, static delegation leaves room for optimizations under certain conditions. For instance, with certain security properties, once a particular decision about an event has been made, all subsequent occurrences of the same event could soundly and transparently be handled according to the same decision. An abstract example of such security properties are properties that are monotonous over time in the set of permissible events. A concrete instance would be the Chinese Wall Security Policy, in which the set of permissible events is monotonically decreasing over time. Hence, such security properties would not require delegation for the occurrence of an event for which a prior occurrence has already received the particular decision. The extent to which such augmentations of static delegation lend themselves for soundly and transparently enforcing security, however, depends on the respective security property.

Another optimization concerns knowledge about the possible runs of the target. In our conjectures, we do not make use of knowledge about the possible runs of the target. Such knowledge about the possible runs of the target could be obtained through static analysis of the target. Rather, the condition quantifies over all possible event sequences – an over-approximation of the target's runs. The use of a more precise set would yield weaker but yet sufficient conditions.

In our discussions of the possible knowledge of a unit, we deliberately abstract from any orderings of events. However, the observations by a unit would allow the unit to establish an ordering at least between some of the observed events – for instance such events intercepted and delegated by one and the same unit. Capturing the knowledge of such orderings would reduce the set of possible worlds for a unit to consider in decision-making and, thus, enable the unit to become authoritative for a broader class of security properties than order-insensitive ones. Formally such partial orders among observed events could be captured, e.g., through *strand spaces* [THG99], a model used for verification of cryptographic protocols.

7.6. Summary

We introduced the static delegation technique for the cooperation between the units of a distributed enforcement mechanism. The technique specifies that a unit must delegate the decision-making for an event always to the responsible unit for the event, except when the intercepting unit already is responsible for the event. We defined two classes of security properties, partitionable security properties and order-insensitive security properties properties. For properties from the intersection of the two classes, we show when an assignment of events to responsible units makes the units authoritative and thereby enables them to effectively enforce the security property. For the specification of policies that employ static delegation, we presented an extension library for CoDSPL that consists of an abstract class from which local policy classes can inherit as well as classes for delegation requests and responses. We presented ChESt, two CoDSPL policies for enforcing a Chinese Wall Security Policy on a distributed storage service built from one of two FTP servers. In empirical evaluations, we show that ChESt effectively and efficiently enforces security in the scenario when applied with CliSeAu.

Static delegation eliminates time-of-check to time-of-use flaws for partitionable order-insensitive security properties essentially by eliminating the time from between check and use: Assigning all events from an equivalence class of a partitionable security property to the same responsible unit ensures that decisions about potentially interfering intercepted events are made by the same unit; A responsible unit can treat a decision like it was already enforced because the security property is order-insensitive such that it does not matter when it's actually enforced. Since static delegation enables the effective enforcement of security properties even under race conditions, the technique is particularly suitable for scenarios such as the conflict of interest scenario, in which attackers can trigger race conditions. After our initial proposal of an instance of static delegation for enforcing the Chinese Wall Security Policy [GMS12], similar instances based on responsible units have been adopted in the literature [DLJ15]. However, we are the first to describe the technique in the presented generality and the first to classify security properties for effective enforcement in distributed programs.

A security property that is partitionable into a large partition of events allows static delegation to be used in a way that distributes the responsibilities for the sets of events in the partition over the units of a distributed enforcement mechanism. The units of the mechanism can then cooperate in a decentralized fashion. Particularly, one can distribute the responsibilities such that expected frequencies of event occurrences for the individual sets in a partition match the capacities available at the responsible units. Such a decentralized enforcement avoids bottlenecks in the cooperation among units. Decentralized enforcement has been studied before [SVA+04; OBM10; KP15] but we are the first to study it in combination with effectively enforcing security properties.

Static delegation simplifies the design of policies for enforcing security properties in distributed programs as it obviates the need for specifying how units of a distributed enforcement mechanism cooperate. Moreover, static delegation for a partitionable order-insensitive security property makes units responsible for classes of potentially interfering events and, thereby, enables the decision-making at each individual unit to be realized like in a non-distributed scenario. Both, simplified delegation and simplified decision-making, allow for less complex policy specifications and, in consequence, a reduced chance of flaws in the policy specification.

Chapter

8

A Formal Cooperation Model for CliSeAu

8.1. Introduction

Formal models of enforcement mechanisms enable unambiguous formulations of desired properties and allow for rigorous proofs that the properties are satisfied, even in presence of race conditions. While for non-distributed enforcement mechanisms various such models have been proposed in the literature (e.g., [Sch00; HMS06a; BOS07; ADG09; HT09; LBW09]), models for distributed enforcement mechanisms and the cooperation adopted by the mechanisms have not been proposed.

In this chapter, we propose a formal model of CliSeAu's generic enforcement capsules, particularly including the cooperation between units and the communication between the components within units. The model is specified in Hoare's CSP [Hoa85] and captures the modular architecture of CliSeAu's generic enforcement capsules. The model's parameters reflect the CoDSPL policies that can be input to CliSeAu. In the model capture also how individual agents and units are composed such that intercepting events and imposing countermeasures is possible. For formally verifying whether a given security property from the class of safety properties is soundly enforced by an instance of the model, this chapter provides a formal definition of sound enforcement in terms of our model. Picking up the application scenario of Section 7.4, we use our model to formalize ChESt and use our definition of sound enforcement to formally verify that the cooperative enforcement performed by ChESt is indeed sound.

The formal model provided in this chapter allows one to capture one's CoDSPL policy in a formal language (CSP) whose semantics is more amenable to formal analysis than Java. In particular, by faithfully modeling one's CoDSPL policy in our formal model, one can analyze one's CoDSPL policy with regard to sound enforcement. Our model of a distributed enforcement mechanism that captures the concurrent operation of agents and units as well as the cooperation adopted by the mechanism's units. That is, the model can in particular capture race conditions caused by concurrent security-relevant operations of agents, such that possibilities for race conditions are adequately covered when verifying sound enforcement.

Structure This chapter is structured as follows: Section 8.2 introduces syntax and semantics of the sub-language of CSP we use in this chapter. In Section 8.3, we provide a definition of the formal model, its parameters, and how the modeled enforcement mechanism is combined with the model of a target. Section 8.4 defines a notion of soundness for this model. The applicability of the model for proving sound enforcement is demonstrated in Section 8.5 with an example instance. Section 8.6 summarizes the contributions presented in this chapter.

8.2. A Primer on CSP

Hoare's algebra of communicating sequential processes (*CSP*) is a process algebra with a rich set of operators and several semantics [Hoa85]. The formal models we present in this chapter are built on a sub-language and the trace semantics of the process algebra. In this section, we introduce the syntax and semantics of this sub-language of CSP.

Definition 8.1. A *process expression* is an element of the language $\mathcal{L}(P)$, defined by the following BNF:

$$\begin{aligned} P = {}& \text{`}\mathsf{STOP}\text{'}_E \mid \mathsf{NAME} \mid e\,\text{`}\rightarrow\text{'}\,P \mid x\,\text{`}{:}\text{'}\,E\,\text{`}\rightarrow\text{'}\,P \mid x\,\text{`}\rightarrow\text{'}\,P \mid P\,\text{`}\backslash\text{'}E \\ & \mid P\,\text{`}\Box\text{'}\,P \mid P\,\text{`}\sqcap\text{'}\,P \mid P\,\text{`}\|\text{'}\,P. \end{aligned}$$

where E ranges over sets, NAME ranges over identifiers (called *process names*), e ranges over elements (called events[1]), and x ranges over *event variables*. $\Diamond$

Intuitively, the primitives and operators of the language model *systems* exposing the following behaviors.

- STOP_E models a system that immediately terminates.
- NAME models a system that behaves as the system with name NAME.
- $e \rightarrow P$ models a system that first performs event e and afterwards behaves according to the process expression P.
- $x\colon E \rightarrow P$ models a system that first participates in some event e from set E and afterwards behaves according to the process expression P where all free occurrences[2] of x are substituted by e.
- $x \rightarrow P$ models the same as $e \rightarrow P$, except that the concrete event e that takes place is determined by a preceding $x\colon E$.
- $P \setminus E$, called *hiding*, models a system that behaves as P but all events in the set E are hidden to the environment of the system.
- $P \Box Q$, called *external choice*, models a system that behaves as either P or Q, depending on the choice of the environment of the system.
- $P \sqcap Q$, called *internal choice*, models a system that behaves as either P or Q where it is up to the system to choose between P and Q.

[1]For the sake of consistency with the literature, we use the term "event" for these elements in CSP and also, as introduced earlier in this thesis, more broadly for atomic actions of agents.

[2]For the sake of brevity we refrain from formally defining free or bound occurrences of variables as well as substitution, as we use them analogous to, e.g., first order logic.

- $P \parallel Q$, called *parallel composition*, models a system resulting from running the systems modeled by P and Q in parallel such that the two systems synchronize on the events that they can both in principle participate in.

Beyond the syntax introduced in Definition 8.1, we use some short-hand notation to ease the legibility and simplify the specification of process expressions. Firstly, we lift the binary operators $\Box$, $\sqcap$, and $\parallel$ to n-ary operators over non-empty finite sets. For the external choice, we use $\Box_{i \in I} P(i)$ to abbreviate $P(a)$ if $I = \{a\}$ and to abbreviate $P(a) \Box \left(\Box_{i \in I \setminus \{a\}} P(i) \right)$ if $a \in I$ and $I \setminus \{a\} \neq \emptyset$. Where the set E of events is unambiguous from the context, we also use the n-ary operators over empty sets to denote STOP_E. For the internal choice ($\sqcap_{i \in I} P(i)$) and parallel composition ($\parallel_{i \in I} P(i)$), we use the analogous notation.

Secondly, we use structured events to model communication over channels. A structured event is of the form $c.m$, where c is the *channel* over which *message* m is communicated. For making explicit that a message is sent over a channel, we write $c!m \to P$ instead of $c.m \to P$. For making explicit that a message is received over a channel, we write $c?x\colon M \to P$ instead of $x\colon \{c.m \mid m \in M\} \to P$. Finally, when messages to be received over a channel are themselves structured in the form of tuples of values, we write $c?(x_1, \ldots, x_n)\colon M_1 \times \ldots \times M_n \to P$ instead of writing $c?x\colon M_1 \times \ldots \times M_n \to P$ and using selector functions for the individual elements of x in P.

In this thesis, we use the trace semantics of CSP, which we introduce in the following definitions. The models specified by process expressions are defined below.

Definition 8.2. A *process* is a tuple (E, Tr) consisting of a set E of events and a non-empty, prefix-closed set of finite sequences over E, $Tr \subseteq E^*$. We call E the *alphabet* of the process and Tr the set of *possible traces* of the process. We use *PROC* to denote the set of all processes. $\Diamond$

The alphabet of a process contains all events in which the process could in principle engage. The set of possible traces of a process contains all sequences of events that the process can in principle perform. Note that in the standard literature on CSP [Hoa85; Ros05], what we call process expression is typically referred to as "process". What we call process does not have a named counterpart in the works of Hoare and Roscoe but the components of processes – alphabet and possible traces – are described individually there.

For modeling recursive behavior of systems, the syntax of process expressions contains process names. The meaning of a process name is assigned through an equation of the following form.

Definition 8.3. A *process equation* is an equation $\mathsf{NAME} \overset{\text{def}}{=}_E P$, where NAME is a process name, P is a process expression, and E is a set of events. $\Diamond$

Intuitively, a process equation declares a process name NAME and defines that this process name models the system modeled by process expression P. The set of events in which the process could in principle engage is declared to be E. We occasionally omit the set of events when it is clear from the respective process expression.

In this thesis, we use process names in the form of simple identifiers and also in a parameterized form, such as $\mathsf{NAME}(x_1, \ldots, x_n)$. In this parameterized form, we view the whole term including the arguments as the process name. Correspondingly, when we use process equations to define such parameterized process names, we typically provide a set of process equations, one for each instance of the parameters. When we include such parameterized process names in process expressions, we often use sub-expressions in place of the parameters. In such cases, we consider the result of evaluating the sub-expressions to be part of the process expression rather than the sub-expressions themselves. For instance, we might write $\mathsf{NAME}(A \cup \{x\})$ to refer to the process name $\mathsf{NAME}(B)$, where B is the set resulting from $A \cup \{x\}$.

For defining the trace semantics, we first define the semantics of a process expression under a set of process equations and a given function that provides semantics – in terms of processes – to the individual process names defined by the process equations. In a subsequent definition, we clarify that this function corresponds to fixed points for the semantics of process names under the set of process equations. In this semantics, we follow Hoare [Hoa85] and Roscoe [Ros05, p. 37].

Definition 8.4. Let $\mathcal{EQ}$ be a set of process equations in which no left-hand side occurs more than once and let N be the set of process names declared by the process equations in $\mathcal{EQ}$. Let $\mathit{fp} : N \to \mathit{PROC}$ be a total function assigning processes to process names. Then the process specified by a process expression P under $\mathcal{EQ}$ and fp is $(\alpha(P), \mathit{traces}(P))$, where α and traces are defined recursively over process expressions as follows.

- for every set E of events:

$$\begin{aligned}\alpha(\mathsf{STOP}_E) =& E\\ \mathit{traces}(\mathsf{STOP}_E) =& \{\langle\,\rangle\}\end{aligned}$$

- for each process name $\mathsf{NAME} \in N$ and $(E, \mathit{Tr}) = \mathit{fp}(\mathsf{NAME})$:

$$\begin{aligned}\alpha(\mathsf{NAME}) =& E\\ \mathit{traces}(\mathsf{NAME}) =& \mathit{Tr}\end{aligned}$$

- for process expression P and event $e \in \alpha(P)$:

$$\begin{aligned}\alpha(e \to P) =& \alpha(P)\\ \mathit{traces}(e \to P) =& \{\langle\,\rangle\} \cup \{\langle e\rangle.\mathit{tr} \mid \mathit{tr} \in \mathit{traces}(P)\}\end{aligned}$$

- for process expression P, set $E \subseteq \alpha(P)$, and event variable x:

$$\begin{aligned}\alpha(x\colon E \to P) =& \alpha(P)\\ \mathit{traces}(x\colon E \to P) =& \{\langle\,\rangle\} \cup \{\langle e\rangle.\mathit{tr} \mid e \in E \wedge \mathit{tr} \in \mathit{traces}(P[x/e])\}\end{aligned}$$

  where $P[x/e]$ is the process expression P with all free occurrences of x substituted by e.
- for process expression P and event variable x:

$$\alpha(x \to P) = \alpha(P)$$

  while $\mathit{traces}(x \to P)$ is not defined.

- for process expression P and set E of events:

$$\begin{aligned}\alpha(P \setminus E) =& \alpha(P) \setminus E \\ \mathit{traces}(P \setminus E) =& \{\mathit{tr} \upharpoonright \alpha(P \setminus E) \mid \mathit{tr} \in \mathit{traces}(P)\}\end{aligned}$$

- for process expressions P and Q with $\alpha(P) = \alpha(Q)$:

$$\begin{aligned}\alpha(P \,\Box\, Q) =& \alpha(P \sqcap Q) = \alpha(P) = \alpha(Q) \\ \mathit{traces}(P \,\Box\, Q) =& \mathit{traces}(P \sqcap Q) = \mathit{traces}(P) \cup \mathit{traces}(Q)\end{aligned}$$

- for process expressions P and Q:

$$\begin{aligned}\alpha(P \parallel Q) =& \alpha(P) \cup \alpha(Q) \\ \mathit{traces}(P \parallel Q) =& \left\{ \mathit{tr} \in \alpha(P \parallel Q)^* \;\middle|\; \begin{array}{l} \mathit{tr} \upharpoonright \alpha(P) \in \mathit{traces}(P) \wedge \\ \mathit{tr} \upharpoonright \alpha(Q) \in \mathit{traces}(Q) \end{array} \right\}\end{aligned} \qquad \Diamond$$

Note that the definition does not provide a process for every process expression. Firstly, no process is provided when a process name is used for which no process equation is contained in the given set of process equations. Secondly, no process is provided when the process expression contains free occurrences of event variables.

Recall that the previous definition does not demand a connection between the processes assigned to process names by function *fp* and the right-hand side of the corresponding process equations. We establish this connection in the following definition.

Definition 8.5. Let $\mathcal{EQ}$ be a set of process equations in which no left-hand side occurs more than once and let $\mathit{fp} : N \to \mathit{PROC}$ be a total function assigning processes to process names. Then *fp* is a *fixed point* of $\mathcal{EQ}$ if for each process equation $(\mathsf{NAME} \overset{\text{def}}{=}_E P) \in \mathcal{EQ}$, it holds that $\mathit{fp}(\mathsf{NAME}) = (\alpha(P), \mathit{traces}(P))$. $\Diamond$

Throughout this thesis, the process expressions we specify obey the property that each occurrence of a process name is *guarded*, i.e., sequentially preceded by an event. For a more rigorous definition of guardedness, we refer to Hoare [Hoa85, p. 79]. As observed by Hoare [Hoa85, pp. 77 sqq.], sets of process equations in which all occurrences of process names are guarded have a unique fixed point. We make use of this in the following.

Definition 8.6. Let $\mathcal{EQ}$ be a set of process equations in which no left-hand side occurs more than once and that has a unique fixed point $\mathit{fp} : N \to \mathit{PROC}$. Then the *trace semantics* of a process expression P under set $\mathcal{EQ}$ of process equations is the process $(\alpha(P), \mathit{traces}(P))$. $\Diamond$

In this thesis, when we omit an explicit specification of the set of process equations, we implicitly consider the set of all process equations defined up to the respective point in the thesis. Since we ensure to not use multiple process equations for the same process name, in this set of process equations no left-hand side occurs more than once. Moreover, when we use the functions α and *traces*, we leave the fixed point to the respective set of process equations implicit.

We model properties of process expressions by unary predicates on traces.

Definition 8.7. A process expression P *satisfies* a unary predicate φ on $traces(P)$, denoted by P sat φ, if and only if $\varphi(tr)$ holds for each $tr \in traces(P)$. $\Diamond$

We abuse notation and write P sat A, for a set A of sequences, to denote P sat φ_A where φ_A is the characteristic predicate for A.

Definition 8.8. Let P be a process expression with set $\mathcal{EQ}_P$ of process equations and let Q be a process expression with set $\mathcal{EQ}_Q$ of process equations. We say that the two process expressions P and Q are *equivalent* under their respective sets of process equations, denoted $P\,_{\mathcal{EQ}_P}{\equiv}_{\mathcal{EQ}_Q}\,Q$, if and only if $\alpha(P) = \alpha(Q)$ and $traces(P) = traces(Q)$ holds. Where the same sets of process equations shall be used for both process expressions and they are clear from the context, we also omit them. $\Diamond$

Definition 8.9. Let $\mathcal{EQ}$ be a set of process equations and let N be the set of process names declared by the process equations in $\mathcal{EQ}$.

For a process expression P and trace $tr \in traces(P)$, we denote by $P \,/\, tr$, the process expression recursively defined on the structure of P and the elements in tr as follows.

- for every set E of events: $\mathsf{STOP}_E \,/\, \langle\rangle = \mathsf{STOP}_E$
- for each process name $\mathsf{NAME} \in N$: $\mathsf{NAME} \,/\, tr = Q \,/\, tr$, where Q is the process expression for NAME according to $\mathcal{EQ}$
- for process expression P and event $e \in \alpha(P)$: $(e \to P) \,/(e.t) = P \,/\, t$.
- for process expression P, set $E \subseteq \alpha(P)$, and event variable x: $(x\colon E \to P) \,/(e.t) = P[x/e] \,/\, t$ if $e \in E$.
- for process expression P and event variable x: $(x \to P) \,/\, t$ is undefined
- for process expressions P and Q with $\alpha(P) = \alpha(Q)$: $(P \,\Box\, Q) \,/\, tr = P \,/\, tr$ if $tr \in traces(P) \setminus traces(Q)$; $(P \,\Box\, Q) \,/\, tr = Q \,/\, tr$ if $tr \in traces(Q) \setminus traces(P)$; and $(P \,\Box\, Q) \,/\, tr = (P \,/\, tr) \sqcap (Q \,/\, tr)$ if $tr \in traces(Q) \cap traces(P)$
- for process expressions P and Q with $\alpha(P) = \alpha(Q)$: $(P \sqcap Q) \,/\, tr = P \,/\, tr$ if $tr \in traces(P) \setminus traces(Q)$; $(P \sqcap Q) \,/\, tr = Q \,/\, tr$ if $tr \in traces(Q) \setminus traces(P)$; and $(P \sqcap Q) \,/\, tr = (P \,/\, tr) \sqcap (Q \,/\, tr)$ if $tr \in traces(Q) \cap traces(P)$
- for process expressions P and Q: $(P \parallel Q) \,/\, tr = (P \,/\, tr) \parallel (Q \,/\, tr)$
- for process expression P and set E of events: $(P \setminus E) \,/\, tr = (\sqcap_{t \in T_{tr}} P \,/\, t) \setminus E$ where $T_{tr} = traces(P) \cap \{t \mid t \upharpoonright \alpha(P) \setminus E = tr\}$ if T_{tr} is finite and $tr \in traces(P \setminus E)$. $\Diamond$

Intuitively, $P \,/\, tr$ models a system that behaves as P after it has performed the sequence tr of events. The following recapitulates some basic properties of process expressions provided by Hoare [Hoa85].

Lemma 8.1. *Let P, Q be process expressions, E, E' be sets of events, t_1, t_2, tr be sequences of events, $e \in \alpha(P)$ be an event, and f be an injective function on alphabets.*

(a) $f(P \parallel Q) \equiv f(P) \parallel f(Q)$ *[Hoa85, p. 65 (L3)]*
(b) $P \setminus (E \cup E') \equiv (P \setminus E) \setminus E'$ *[Hoa85, p. 92 (L2)]*
(c) $(P \parallel Q) \setminus E \equiv (P \setminus E) \parallel (Q \setminus E)$ *if* $\alpha(P) \cap \alpha(Q) \cap E = \emptyset$ *[Hoa85, p. 92 (L6)]*
(d) $f(P \setminus E) \equiv f(P) \setminus f(E)$ *[Hoa85, p. 92 (L7)]*
(e) $P \,/(t_1.t_2) \equiv (P \,/\, t_1) \,/\, t_2$ *[Hoa85, p. 32 (L2)]*

$\Diamond$

8.3. The Model in CSP

In Chapter 5, we have seen the modular architecture of CliSeAu's generic ECs and how these generic ECs cooperate. This architecture provides fixed components – the interceptor, the coordinator, and the enforcer – as well as parametric components – the event factory, the local policy, and the enforcer factory (see Section 3.2 on page 28). In this section, we provide a formal model in the process algebra CSP that captures the essential aspects of CliSeAu's generic ECs: Concretely, the model captures the parametricity in components that specify the cooperative decision-making, captures the possibility of units to cooperate, and captures the interface of the local policy component to the coordinator. The focus on the interface of the local policy is due to our focus on particularly capturing the cooperation faithfully.

The model we present comprises the modular model of units, which exhibits the same modular architecture as the service automata concept (see Section 2.8), which also CliSeAu's generic ECs implement. Two of the components are fixed and the remaining two components are parametric. In the following, we first introduce the fixed components in Section 8.3.1 and subsequently specify the interface for the parametric components in Section 8.3.2. Finally, in Section 8.3.3 we present how a distributed target is encapsulated by an instance of the formal model.

8.3.1. Fixed Components

Our formal model of CliSeAu's generic ECs comprises two fixed components, one modeling the coordinator of CliSeAu and one modeling the interceptor of CliSeAu. These two components also define the interface to the two parametric components, which consists of a collection of channels. We first introduce these channels and subsequently zoom into the individual fixed components.

Figure 8.1 on the following page shows the four components of an EC model, where the fixed components are displayed in white boxes and the parametric components in shaded boxes. Beyond the EC model in focus, the figure furthermore shows the agent encapsulated by the unit and other units, which encapsulate other agents of a distributed target. An arrow from one component to another indicates that the former communicates to the latter. Where an arrow is absent, no communication takes place. The labels of the arrows show the names of the channels used for the communication as well as the domain of the messages communicated via these channels. Where no channels are used, i.e., between the agent and the unit, only the domain of communicated events is placed as labels. In the following, we introduce the individual channels.

The interceptor communicates *intercepted events*, i.e., events that the agent cannot perform without the unit halting the agent until a decision is made and is implemented, to the coordinator via channel icpt. We use *IE* as meta-variable for sets of intercepted events. This models the communication of event objects by CliSeAu's generic ECs from the interceptor to the coordinator through a network socket.

The coordinator has two channels to the local policy: lreq and rreq. Through the former, intercepted events are communicated, and through the latter, delegation requests and delegation responses – together modeled by the set *DR* – are communicated. In the reverse

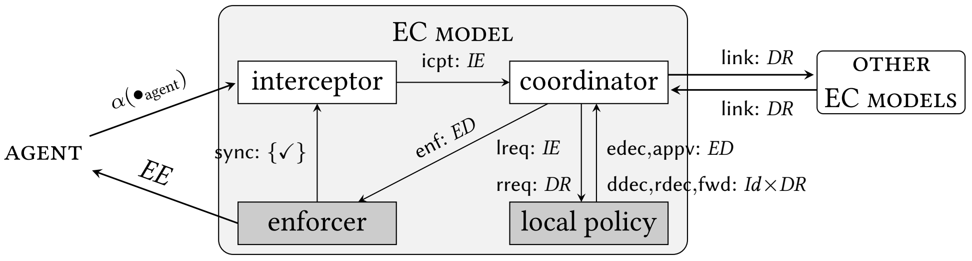


Figure 8.1.: Components and channels of an EC model

direction, five channels allow the local policy to communicate to the coordinator. The local policy can return a decision to the coordinator via the channels edec (decisions made locally) and appv (decisions obtained after delegation). We use *ED* as meta-variable for sets of decisions. The local policy can also return a delegation request or delegation response together with the identifier of the unit to which the delegation shall be performed via the channels ddec (delegation for locally intercepted event), rdec (decision for remotely intercepted and delegated event), and fwd (further delegation for a previously delegated event). We use *Id* as meta-variable for sets of unit identifiers. The five channels between the coordinator and the local policy model the two methods (localRequest and remoteRequest), including actual arguments and return values, that a subclass of LocalPolicy in CoDSPL must implement. The actual exchange of delegation requests and delegation responses between units takes place via the link channels, one channel for each pair of units and communication direction. These channels model the network sockets used by the coordinators of CliSeAu to communicate with each other.

The enforcer has one channel, enf, for obtaining decisions from the coordinator. This models the communication of decision objects by CliSeAu's generic ECs from the coordinator to the enforcer through a network socket. Moreover, the enforcer has one channel, sync for synchronizing with the interceptor by sending the symbol $\checkmark$. This synchronization is used to model that in CliSeAu's generic ECs, the interceptor and the enforcer ensure that the agent is blocked from the time an operation is intercepted until the enforcer has performed a countermeasure. Finally, the enforcer can, without dedicated channel, impose events on the agent. We use *EE* as meta-variable for such sets of *effectable events*, i.e., events that an enforcer can effectuate. This models that an enforcer can itself perform operations in the context of the agent as part of enforcing a security property.

The interceptor The interceptor is the component of a unit that observes security-relevant operations of an agent for a subsequent decision-making. Upon observing a security-relevant operation, the interceptor blocks all further operations of the agent until a countermeasure has been performed. It triggers the decision-making by passing the intercepted event to the coordinator. We capture these properties of the interceptor as follows.

Definition 8.10. Let $ii = (E, IE)$ be a tuple where E is a set of all events of the agent and IE is a set of intercepted events. Then the *interceptor model* for ii is the process expression

INT_{ii} along with the process equation

$$\begin{aligned}\mathsf{INT}_{ii} \overset{\text{def}}{=}_{E^{\mathsf{INT}}_{ii}} \quad & x\colon (E \cap \mathit{IE}) \to \mathsf{icpt}!x \to \mathsf{sync}?y\colon \{\checkmark\} \to \mathsf{INT}_{ii}\\ & \Box x\colon (E \setminus \mathit{IE}) \to \mathsf{INT}_{ii}\end{aligned}$$

where $E^{\mathsf{INT}}_{ii} = E \cup \{\mathsf{icpt}.e \mid e \in E \cap \mathit{IE}\} \cup \{\mathsf{sync}.\checkmark\}$. We denote the set containing this single process equation by $\mathcal{EQ}^{\mathsf{INT}}_{ii}$. We call *ii* an *interceptor instance*. ◊

The interceptor model is parametric in two sets, *E* and *IE*, capturing all events of the agent and the security-relevant ones. It observes the security-relevant events of the agent by synchronizing on all events in $E \cap \mathit{IE}$ and sends them to the coordinator via the icpt channel, as formalized in the first line of the process equation. Secondly, the interceptor model also synchronizes on the events that are not security-relevant (in $E \setminus \mathit{IE}$) such that these events can only occur when the interceptor model is not blocked. That is, while the interceptor model is blocked, all events of the agent are blocked, too. As soon as a countermeasure for a security-relevant event has been performed and the interceptor model is unblocked by receiving ✓ on channel sync, the agent is unblocked again. Overall, the interceptor model, thus, faithfully captures the desired properties of the interceptor.

The coordinator The coordinator is the component that takes over the communication between the local policy and the interceptor, the enforcer, and remote units. Concretely, the coordinator forwards events from the local interceptor and delegation requests and responses from remote units to the local policy; it forwards decisions as well as delegation requests and responses of the local policy to the enforcer or, respectively, remote units. Moreover, the coordinator in CliSeAu's generic ECs is stateless and is non-blocking in the sense that it does not wait for the result of a delegation request until processing further incoming intercepted events or delegation requests and responses. We capture these properties of the coordinator as follows.

Definition 8.11. Let $ci = (\mathit{Id}, i, \mathit{IE}, \mathit{ED}, \mathit{DR})$ be a tuple where *Id* is a set of unit identifiers, $i \in \mathit{Id}$ is a unit identifier, *IE* is a set of intercepted events, *ED* is a set of decisions, and *DR* is a set of delegation requests and responses. Then the *coordinator model* for *ci* is the process expression COR_{ci} along with the process equation

$$\begin{aligned}\mathsf{COR}_{ci} \overset{\text{def}}{=}_{E^{\mathsf{COR}}_{ci}} & \mathsf{icpt}?e\colon \mathit{IE} \to \mathsf{lreq}!e\\ & \to \Big(\quad \big(\mathsf{edec}?ed\colon \mathit{ED} \to \mathsf{enf}!ed \to \mathsf{COR}_{ci}\big)\\ & \qquad \Box\big(\mathsf{ddec}?(k, dr)\colon (\mathit{Id}\setminus\{i\}) \times \mathit{DR} \to \mathsf{link}_{i,k}!dr \to \mathsf{COR}_{ci}\big)\Big)\\ \Box\Box_{j\in \mathit{Id}\setminus\{i\}} & \mathsf{link}_{j,i}?dr\colon \mathit{DR} \to \mathsf{rreq}!dr\\ & \to \Big(\quad \big(\mathsf{fwd}?(k, dr')\colon (\mathit{Id}\setminus\{i\}) \times \mathit{DR} \to \mathsf{link}_{i,k}!dr' \to \mathsf{COR}_{ci}\big)\\ & \qquad \Box\big(\mathsf{rdec}?(k, dr')\colon (\mathit{Id}\setminus\{i\}) \times \mathit{DR} \to \mathsf{link}_{i,k}!dr' \to \mathsf{COR}_{ci}\big)\\ & \qquad \Box\big(\mathsf{appv}?ed\colon \mathit{ED} \to \mathsf{enf}!ed \to \mathsf{COR}_{ci}\big)\Big)\end{aligned}$$

where

$$E^{\mathsf{COR}}_{ci} = \left\{ \begin{array}{l|l} \mathsf{icpt}.e, \mathsf{enf}.ed, \mathsf{link}_{i,k}.dr, \mathsf{link}_{k,i}.dr, & e \in \mathit{IE}, ed \in \mathit{ED}, \\ \mathsf{lreq}.e, \mathsf{rreq}.dr, \mathsf{edec}.ed, \mathsf{appv}.ed, & dr \in \mathit{DR}, \\ \mathsf{ddec}.(k, dr), \mathsf{rdec}.(k, dr), \mathsf{fwd}.(k, dr) & k \in \mathit{Id} \setminus \{i\} \end{array} \right\}$$

We denote the set containing this single process equation by $\mathcal{EQ}^{\mathsf{COR}}_{ci}$. We call *ci* a *coordinator instance*. $\Diamond$

The coordinator model is parametric in several sets and an identifier. The sets *IE*, *ED*, and *DR* determine the possible messages that can be communicated via the channels that the coordinator uses. The set *Id* specifies which units belong to the distributed enforcement mechanism and thereby which other units incoming delegation requests and responses the coordinator shall receive from and send to. In the first and fourth line of the process equation, the process waits for incoming intercepted events (channel icpt) and, respectively delegation requests and responses (channel $\mathsf{link}_{j,k}$). It subsequently passes its input to the local policy. The results received from the local policy through the channels in the subsequent lines of the process equation are then further delivered to the enforcer or to a remote unit. That is, the coordinator takes over the communication between the local policy and the interceptor, the enforcer, and remote units. Moreover, the coordinator model is non-blocking with regard to delegation as evident from the third, fifth, and sixth line of the process equation: After receiving on a channel from the local policy a message dr or dr' to be delegated, the coordinator sends out this message on a link channel and afterwards returns to its initial receiving state again. However, despite this implicit state established through the external choices and the sequential steps, the coordinator does not carry any state. In particular, the coordinator does not record information about the history of processed intercepted events or delegation requests and responses. Overall, the coordinator model, thus, faithfully captures the intended behavior and properties of the coordinator.

8.3.2. Parametric Components

In this section, we define the two parametric components of our formal model by their interfaces to the fixed components. We capture the interface of a parametric component via the alphabet of the process that instantiates the component, i.e., via the set of events in which the process can participate and, thus, potentially synchronize with other components. We provide examples to illustrate our definitions.

The enforcer In CliSeAu as well as in our formal model, the enforcer serves the purpose of implementing the decisions made by the enforcement mechanism. What it means to implement a decision can vary depending on the security property being enforced as well as the target. Therefore, our model aims to impose as little constraints as possible on the enforcer model, demanding only that the enforcer model interfaces with the remaining components of the formal model.

Definition 8.12. An *enforcer model* for a set *ED* of decisions and a set *EE* of effectable events is a process expression *P* along with a set $\mathcal{EQ}$ of process equations such that

$$\alpha(P) = \{\mathsf{sync}.\checkmark\} \cup \{\mathsf{enf}.ed \mid ed \in ED\} \cup EE.$$

holds. ◊

Our definition of an enforcer model specifies the interface to the remaining components of the formal model by fixing the alphabet of a process expression that could be used for instantiating this parametric component. Concretely, alphabet fixes which channels can be used (sync and enf), which messages may be communicated via these channels, and which events the enforcer can effectuate. Note that through the parameterized set *EE*, further channels could be introduced to the alphabet. We eliminate this possibility in Section 8.3.3, after all the individual components have been introduced.

Generally, a wide range of possible enforcer models are imaginable. In the following, we demonstrate by example that our definition of enforcer models is sufficiently broad to capture the countermeasures supported by two of the most acknowledged models of generic enforcement mechanisms: security automata [Sch00] and edit automata [LBW09] [LBW09].

The first example of an enforcer model is one that conceptually supports two possible decisions: permitting the occurrence of an event and terminating the respective agent of the target. These two possibilities coincide with the countermeasures of the security automata model.

Example 8.1. The *terminating enforcer* for a set *EE* of effectable events is the enforcer model TERM_{EE} for $ED = EE \times \{perm, term\}$ and *EE* together with the process equation

$$\begin{aligned}\mathsf{TERM}_{EE} \overset{\text{def}}{=}_{E_{\mathsf{TERM}}} \quad & \mathsf{enf}?(e,x) \colon EE \times \{perm\} \to e \to \mathsf{sync}!\checkmark \to \mathsf{TERM}_{EE} \\ & \Box \mathsf{enf}?(e,x) \colon EE \times \{term\} \to \mathsf{STOP}_{E_{\mathsf{TERM}}}\end{aligned}$$

where the alphabet $E_{\mathsf{TERM}} = \{\mathsf{sync}.\checkmark\} \cup \{\mathsf{enf}.ed \mid ed \in ED\} \cup EE$ satisfies the constraint of an enforcer model for *ED* and *EE*. ◊

In the terminating enforcer, decisions are modeled as the cross product of events and a set of two symbols modeling permission and, respectively, termination. When the terminating enforcer receives a decision via the enf channel, and the decision is to permit an event (first line of the process equation), then it executes the respective event and subsequently unblocks the interceptor by sending ✓ over channel sync. Afterwards, the terminating enforcer returns again to its initial configuration in which it is ready to receive further decisions. When the terminating enforcer receives a decision to terminate (second line of the process equation), then it neither performs the respective event nor unblocks the interceptor and instead terminates itself. By terminating itself, the terminating enforcer particularly prevents to receive and implement further decisions for the local agent.

The edit automata model allows, in addition to termination, also to *suppress* a security-violating event, i.e., to skip the event but to allow the agent to resume afterwards. In the following second example of an enforcer model, we capture suppression as the only possible countermeasure.

Example 8.2. The *suppressing enforcer* for a set *EE* of effectable events is the enforcer model SUPP_{EE} for $ED = EE \times \{perm, supp\}$ and *EE* together with the process equation

$$\begin{aligned}\mathsf{SUPP}_{EE} \overset{\text{def}}{=}_{E_{\mathsf{SUPP}}} \quad & \mathsf{enf}?(e,x)\colon EE \times \{perm\} \to e \to \mathsf{sync}!\checkmark \to \mathsf{SUPP}_{EE}\\ & \Box \mathsf{enf}?(e,x)\colon EE \times \{supp\} \to \mathsf{sync}!\checkmark \to \mathsf{SUPP}_{EE}\end{aligned}$$

where the alphabet $E_{\mathsf{SUPP}} = \{\mathsf{sync}.\checkmark\} \cup \{\mathsf{enf}.ed \mid ed \in ED\} \cup EE$ satisfies the constraint of an enforcer model for *ED* and *EE*. ◊

The suppressing enforcer is modeled very similar to the terminating enforcer. The case of permitting an event (first line of the process equation) is treated in the same way as for the terminating enforcer. The difference is in the second line, in which the decision to suppress an event is implemented by immediately unblocking the interceptor via the sync channel, without allowing the event to happen before. This way, only the intercepted event is suppressed and the agent is allowed to resume afterwards.

In addition to suppressing events, the edit automata model finally also allows to *insert* events, i.e., to perform events that the target itself would not have performed at the respective point in the target's execution. In the following final example, we combine the possible countermeasures of terminating, suppressing, and inserting.

Example 8.3. The *replacing enforcer* for a set *IE* of intercepted events and a set *EE* of effectable events is the enforcer model $\mathsf{REPLACE}_{IE,EE}$ for $ED = IE \times (EE \cup \{term\})^*$ and *EE* together with the process equations

$$\begin{aligned}\mathsf{REPLACE}_{IE,EE} &\overset{\text{def}}{=}_{E_{\mathsf{REPL}}} \mathsf{enf}?(e,t)\colon IE \times (EE \cup \{term\})^* \to \mathsf{REPL}_{IE,EE}(t)\\ \mathsf{REPL}_{IE,EE}(\langle\,\rangle) &\overset{\text{def}}{=}_{E_{\mathsf{REPL}}} \mathsf{sync}!\checkmark \to \mathsf{REPLACE}_{IE,EE}\\ \mathsf{REPL}_{IE,EE}(\langle term\rangle.t) &\overset{\text{def}}{=}_{E_{\mathsf{REPL}}} \mathsf{STOP}_{E_{\mathsf{REPL}}}\\ \mathsf{REPL}_{IE,EE}(\langle e\rangle.t) &\overset{\text{def}}{=}_{E_{\mathsf{REPL}}} e \to \mathsf{REPL}_{IE,EE}(t)\end{aligned}$$

for each $t \in (EE \cup \{term\})^*$ and $e \in EE$. In the process equations, the alphabet E_{REPL} is defined by $E_{\mathsf{REPL}} = \{\mathsf{sync}.\checkmark\} \cup \{\mathsf{enf}.ed \mid ed \in ED\} \cup EE$, The alphabet satisfies the constraint of an enforcer model for *ED* and *EE*. ◊

In the replacing enforcer, the decisions are tuples of an intercepted event and a sequence of effectable events and the special symbol *term*. Intuitively, a decision (e,t) means that instead of *e*, the enforcer shall perform the sequence *t* of events. If *t* contains the symbol *term*, then the agent shall be terminated after inserting the prefix of *t* to this point. Otherwise, if *t* does not contain *term*, then the agent may resume after *t* has been inserted. In the process equations, this intuition is mainly captured through the auxiliary processes with process names $\mathsf{REPL}_{IE,EE}(t)$. These processes recursively perform the events in *t* in sequential order (fourth process equation) until either *term* is encountered (third process equation) or *t* has been processed completely (second process equation). The former case is analogous to the terminating enforcer, in which no unblocking of the interceptor takes place and the process terminates itself. The latter case is like the "*perm*" cases in the terminating enforcer and the suppressing enforcer: the interceptor is unblocked and the enforcer becomes ready to receive further decisions. Overall, the replacing enforcer, thus, subsumes the countermeasures terminating ($t = \langle term \rangle$), suppressing ($t = \langle\,\rangle$), and inserting ($t \in EE^*$). It also supports countermeasures such as "terminate after insertion", which could be used, e.g., for logging the reason for termination.

As demonstrated by Examples 8.1 to 8.3, our definition of enforcer model and the process algebra CSP are powerful enough to capture the countermeasures of security automata and edit automata. On the other hand, as described earlier, the interface provided to enforcer models allows them to model also the countermeasures supported by CliSeAu's generic ECs.

The local policy In CliSeAu, the local policy is the component that makes decisions for intercepted operations of an agent as well as for delegation requests of other units, and creates delegation requests and delegation responses when a local decision cannot be made by the local policy. How a local policy proceeds to fulfill this purpose can vary significantly depending on the security property to be enforced. Like for the enforcer model, our model of a local policy constrains only the interface to the remaining components and not the internal behavior.

Definition 8.13. Let $ci = (Id, i, IE, ED, DR)$ be a coordinator instance (see Definition 8.11 on page 137). A *local policy model* for ci is a process expression P along with a set $\mathcal{EQ}$ of process equations such that

$$\alpha(P) = \left\{ \begin{array}{l|l} \mathsf{lreq}.e, \mathsf{rreq}.dr, \mathsf{edec}.ed, \mathsf{appv}.ed, & e \in IE, ed \in ED, \\ \mathsf{ddec}.(k, dr), \mathsf{rdec}.(k, dr), \mathsf{fwd}.(k, dr) & dr \in DR, k \in Id \setminus \{i\} \end{array} \right\}$$

holds. $\diamondsuit$

The definition specifies the allowed interface of a local policy model to its environment by fixing the alphabet of its process expression. The seven channels and the respective messages that must be contained in the alphabet correspond to the interface described in Section 8.3.1 and illustrated in Figure 8.1. All these channels are shared with the coordinator model. That is, we have $\alpha(P) \subseteq \alpha(\mathsf{COR}_{ci})$. Note that this constraint on the alphabet of a local policy model still allows a local policy model to make use of further events internally, as long as they are hidden from its environment. The possibility of internal events allows a local policy model to be composed of several communicating components. We demonstrate such modularity of a local policy model in Section 8.5.

In the following, we provide a small example of a local policy model. To show the expressive power, we specify a local policy model that simulates a security automaton, which is defined by Schneider [Sch00] as follows.

Definition 8.14. A *security automaton* is a tuple (Q, Q_0, IE, δ) where Q is a countable set of states, $Q_0 \subseteq Q$ is a countable set of initial states, IE is a set of intercepted events, and $\delta : Q \times IE \to \mathcal{P}(Q)$ is a (total) transition function. In the following, we lift δ to sets of states, using $\delta(Q', e)$ to denote $\bigcup_{q \in Q'} \delta(q, e)$. $\diamondsuit$

Intuitively, a security automaton starts in a set of states Q_0. Upon an intercepted event $e \in IE$, it makes a transition to a new set of states that is determined by the transition function δ. As long as this new set of states is non-empty, the respective intercepted events are permitted. If this new set of states at some point is empty, then this models that the target shall be terminated. Security automata are a model of an enforcement mechanism for non-distributed targets and, as such, do not make use of cooperation. Our local policy model simulating a security automaton is defined as follows.

Example 8.4. Let $(Q, Q_0, I\!E, \delta)$ be a security automaton with a finite number of states (a restriction also applied by Erlingsson [Erl04]). Let Id be a non-empty set of unit identifiers and $i \in Id$ be the identifier of the unit in which the security automaton shall be simulated for the decision-making. Let $ci = (Id, i, I\!E, ED, DR)$ for $ED = I\!E \times \{perm, term\}$ and $DR = \emptyset$ be a coordinator instance.

We specify the security automaton as a local policy model for ci by the process expression $\mathsf{SecAut}(Q_0)$ along with the set containing the following process equation for each $Q' \subseteq Q$.

$$\begin{aligned}\mathsf{SecAut}(Q') \overset{\text{def}}{=}_{E_{\mathsf{SecAut}}} & \mathsf{lreq}?e\colon \{e' \mid \delta(Q', e') \neq \emptyset\} \to \mathsf{edec}!(e, perm) \to \mathsf{SecAut}(\delta(Q', e)) \\ & \Box \mathsf{lreq}?e\colon \{e' \mid \delta(Q', e') = \emptyset\} \to \mathsf{edec}!(e, term) \to \mathsf{SecAut}(\emptyset)\end{aligned}$$

where

$$E_{\mathsf{SecAut}} = \left\{ \begin{array}{l|l} \mathsf{lreq}.e, \mathsf{rreq}.dr, \mathsf{edec}.ed, \mathsf{appv}.ed, & e \in I\!E, ed \in ED, \\ \mathsf{ddec}.(k, dr), \mathsf{rdec}.(k, dr), \mathsf{fwd}.(k, dr) & dr \in DR, k \in Id \backslash \{i\} \end{array} \right\}. \qquad \Diamond$$

The process defined in the example receives inputs on channel lreq and on no other channels. Notably, delegation requests and responses are not received on channel rreq, following that security automata do not perform cooperation. Along this line, the set DR of delegation requests and responses is defined to be empty. For events received on channel lreq, the process distinguishes two cases: events that lead to a non-empty successor set of states (first line of the process equation) and events that lead to an empty set (second line). In the former case, the decision to permit the respective event is sent via channel edec and the process continues with an updated set of states. In the latter case, the decision to terminate the agent is sent and the process also continues with the updated set of states (where the empty set causes all subsequent events to result in the decision to terminate as well). This faithfully captures the provided intuitive semantics of a security automaton. Moreover, as the alphabet of the process is defined, the process is indeed a local policy model for the coordinator instance ci used.

The example of the local policy model for a security automaton shows that our definition is expressive with respect to non-cooperative decision-making. A concrete example that also demonstrates the means of a local policy model to employ delegation, is shown in Section 8.5.

8.3.3. Encapsulated Target

In this section, we define how the four components introduced so far are composed to a model of a CliSeAu's generic EC and how an instance of such generic EC encapsulates an agent of a target in terms of CSP operators. We also define how multiple such EC models are composed to a model of an encapsulated target.

The generic EC CliSeAu's generic ECs are generic in the sense that they are parametric in the enforcer and the local policy as well as in the agent of a target. That is, the generic

EC is parametric in components that specify functionality. The generic EC can also be viewed as parametric in the agent whose operations the generic EC intercepts and which it blocks until a decision is made. We capture this as follows.

Definition 8.15. The *EC model* for a coordinator instance $ci = (Id, i, IE, ED, DR)$ is the parametric process expression defined by

$$EC_{ci}(\bullet_{\text{agent}}, \bullet_{\text{pol}}, \bullet_{\text{enf}}) = \left[\begin{array}{lcl} \big(\bullet_{\text{agent}} \parallel \mathsf{INT}_{(\alpha(\bullet_{\text{agent}}),IE)} \big) \setminus IE & \parallel & \mathsf{COR}_{ci} \\ \parallel \quad \bullet_{\text{enf}} & \parallel & \bullet_{\text{pol}} \end{array} \right] \setminus H_{ci},$$

where

$$H_{ci} = \left\{ \begin{array}{l|l} \mathsf{icpt}.e, \mathsf{sync}.\checkmark, \mathsf{enf}.ed, & e \in IE, ed \in ED, \\ \mathsf{lreq}.e, \mathsf{rreq}.dr, \mathsf{edec}.ed, \mathsf{appv}.ed, & dr \in DR, \\ \mathsf{ddec}.(k, dr), \mathsf{rdec}.(k, dr), \mathsf{fwd}.(k, dr) & k \in Id \setminus \{i\} \end{array} \right\} \qquad \Diamond$$

The EC model is defined as a parametric process expression. Syntactically, the parameters of the EC model are split into functional ones (specified in parentheses) and non-functional ones (specified as indices) to expose their different nature. The functional parameters, i.e., the agent, the local policy, and the enforcer, are captured as "holes" in the process expression, indicated by the "$\bullet$", into which the respective instances can be plugged. This is to emphasize that the functional parameters do not constitute variables for process expressions as first class entities in CSP, which would not be included in our CSP fragment. The EC model composes all four components as well as the agent's process in parallel, as the process expression shows. Note that the structure of the process expression in the definition reflects the architecture depicted in Figure 8.1 on page 136.

A particular technical novelty in our EC model is how the agent is combined with the components of the EC model. The agent ($\bullet_{\text{agent}}$) is composed in parallel to the interceptor model, with $\alpha(\bullet_{\text{agent}})$ as shared alphabet, around which all intercepted events (IE) are hidden. Only around this hiding, the remaining components of the EC model are composed in parallel. The parallel composition of the agent and the interceptor together with the hiding of intercepted events enables the interceptor to learn about the next event of the agent without making intercepted events visible to the environment. Such events can only become visible to the environment, if and when they are performed by the enforcer, which is composed outside the hiding operator. That is, this approach to combine an agent with a unit is rendered possible through the modular architecture of CliSeAu's generic ECs, which is reflected in our EC model. The approach allows the EC model to intercept security-relevant events for cooperative decision-making without the need for a prior transformation of the agent (e.g., by renaming the events in IE). We discuss how this approach relates to other process-algebraic specifications of enforcement mechanisms in Section 9.7.

Note that the alphabet of an EC model indeed establishes the narrow interface between an EC model and its environment as depicted in Figure 8.1. All unit-internal communication events of the interceptor and the enforcer (first line in the definition of H_{ci}) as well as of the local policy (second and third line in the definition) are hidden from the environment of the EC model. That is, the local policy and the interceptor cannot communicate directly with the environment of the EC model; the coordinator can only communicate via the

$\mathsf{link}_{i,k}$ channels; and the enforcer can only communicate via the events contained in its parameter EE, i.e., the events that are also intended to be visible to the environment of the EC model.

Remark 8.1. In CliSeAu, an enforcer can perform operations on the agent that have side-effects and can thereby influence the future behavior of the agent. Our model does not allow the enforcer to use events from set IE to influence the agent, because the hiding of IE prevents any synchronization. That is, an enforcer model cannot alter the future behavior of the agent, e.g., by inserting events from IE. In this regard, our model resembles the edit automata model [LBW09], which also does not capture feedback of the enforcement mechanism on the agent through inserted events. ◊

Remark 8.2. In CliSeAu, it is possible to intercept operations of an agent that send data to another agent as well as to intercept operations that receive data from another agent. Our model allows this only for the sending agent, unless the communication shall always be prohibited and could, thus, be excluded from EE. That is, events of an agent that model input from the environment must not be in IE. This is because at the receiving agent, the communication event would synchronize with the enforcer model and the latter cannot influence the agent via events in IE, as pointed out in Remark 8.1. A simple way around this limitation is to not intercept the input events directly but rather events that model the further processing of the inputs. ◊

Remark 8.3. Our EC model comprises two parametric functional components. This deviates in the parametricity of CliSeAu's generic ECs in two regards. Firstly, we deliberately do not include a counterpart to the CliSeAu's event factory. This is because the purpose of this component in CliSeAu is a design that separates an agent's structure of intercepted operations from the event objects used by the local policy. In the formal model, such separation would not offer any advantage as the agents can be modeled suitably for the respective local policy in the first place. Secondly, in our formal model, the enforcer model captures what in CliSeAu is split into the enforcer factory and the countermeasure objects. We do not introduce this split explicitly in our model, as our model shall primarily serve to faithfully capture the cooperation and as where desired, the enforcer models could be composed of two components that model the enforcer factory and, respectively, the countermeasure objects. ◊

The target In our formal model, we capture distributed targets as processes that are composed of the individual agents of the target. Formally, we capture this as follows.

Definition 8.16. A *decomposed target model* is a family $(AGENT_i)_{i \in I}$ of process expressions along with a set $\mathcal{EQ}$ of process equations. Each $AGENT_i$ models an agent of the target. The *target model* for a decomposed target model is the parallel composition $\|_{i \in I} AGENT_i$ of its agents. ◊

The definition makes the individual agents of a distributed target explicit. The parallel composition of the agents is the natural choice in CSP to specify autonomous components that, yet, have the possibility to communicate with each other.

The encapsulated target We now lift the encapsulation from a single agent to a distributed target. As a preparatory step, we first combine the parameters of the EC models to a model of a CoDSPL policy.

A CoDSPL policy specifies the target whose agents shall be encapsulated as well as the parameters of the units that encapsulate these agents. We capture the corresponding entity for our formal model as follows.

Definition 8.17. A *policy model* of our formal model is a tuple

$$pm = (Id, (AGENT_i)_{i \in Id}, (IE_i, EE_i, ED_i, POL_i, ENF_i)_{i \in Id}, DR, \mathcal{EQ}),$$

where Id is a finite set of all identifiers of monitored components, $(AGENT_i)_{i \in Id}$ is a decomposed target model, and for each unit identifier $i \in Id$, IE_i is a set of intercepted events, EE_i is a set of effectable events, ED_i is a set of decisions, POL_i is a local policy model for (Id, i, IE_i, ED_i, DR), ENF_i is an enforcer model for ED_i and EE_i. Finally, DR is a set of possible delegation requests and responses and $\mathcal{EQ}$ is a set of process equations defining all process names that occur in $AGENT_i$, POL_i, and ENF_i. ◊

The policy model subsumes all parameters of the individual components of an EC model. It shares with CoDSPL policies the specification of unit identifiers, agents, local policies, and enforcers. The set of process equations conceptually corresponds to the classpaths in CoDSPL policies and the set of intercepted events corresponds to the pointcuts in CliSeAu. The sets of effectable events, decisions, and delegation requests and responses have no counterpart in CoDSPL policies: In CoDSPL, a universe for these sets is established through abstract classes and interfaces that are part of CoDSPL's definition. Conversely, the policy model does not comprise IP addresses and network ports, as the coordinator models uses identifiers to unambiguously select the communication channels to other EC models.

The policy model, as introduced in Definition 8.17, allows some of its components to be defined in a way that violates the architecture of EC models introduced in Figure 8.1 on page 136. We capture those policy models that adhere to the architecture as follows.

Definition 8.18. We say that a policy model

$$pm = (Id, (AGENT_i)_{i \in Id}, (IE_i, EE_i, ED_i, POL_i, ENF_i)_{i \in Id}, DR, \mathcal{EQ})$$

is *proper*, if and only if for each $i \in Id$ the following the following conditions hold:

(a) $(\alpha(AGENT_i) \cup EE_i) \cap (H_{(Id,i,IE_i,ED_i,DR)} \cup \{\mathsf{link}_{j,k}.dr \mid j,k \in Id \wedge dr \in DR\}) = \emptyset$
(b) $IE_i \subseteq \alpha(AGENT_i)$
(c) $EE_i \supseteq IE_i$ ◊

Definition 8.18 (a) captures that the events of an agent may not interfere with the internal events of an EC model (introduced in Definition 8.15) as well as with the communication events between EC models (link channels). It furthermore captures that also the effectable events, EE_i, of the enforcer model may not interfere with the internal events of the EC model. Definition 8.18 (b) constrains that the set of intercepted events must indeed be events of the respective agent. This ensures, together with Definition 8.18 (a) that the hiding operator around the agent and the interceptor in the EC model does not hide the channels of the interceptor model to the coordinator model and the enforcer model. The two conditions imposed by proper policy models suffice to exclude undesired interferences that would violate the architecture of EC models. In particular, the sets ED_i and DR need not be constrained, as they are only used for messages on channels that belong to the architecture. Finally, Definition 8.18 (c) ensures that the alphabet of every EC model subsumes the alphabet of the encapsulated agent, which preserves the shared alphabet between the (encapsulated) agent and its environment.

With the above definitions, we now have all the ingredients to model an encapsulated target.

Definition 8.19. Let a proper policy model

$$pm = (Id, (AGENT_i)_{i \in Id}, (IE_i, EE_i, ED_i, POL_i, ENF_i)_{i \in Id}, DR, \mathcal{EQ})$$

be given. Then the *encapsulated target model* for pm is the process expression

$$ET_{pm} = \mathop{\|}_{i \in Id} EC_{(Id,i,IE_i,ED_i,DR)}(AGENT_i, POL_i, ENF_i)$$

along with the set

$$\mathcal{EQ}_{pm} = \mathcal{EQ} \cup \bigcup_{i \in Id} \mathcal{EQ}^{\mathsf{COR}}_{(Id,i,IE_i,ED_i,DR)} \cup \bigcup_{i \in Id} \mathcal{EQ}^{\mathsf{INT}}_{(\alpha(AGENT_i),IE_i)}$$

of process equations. $\Diamond$

An encapsulated target model is defined as the parallel composition of the individual EC models, instantiated with a given policy model. The parallel composition captures that the individual EC models run autonomously and can yet communicate via shared events (established by the link channels). The set of process equations combines all process equations of the policy model with the equations belonging to the fixed components of our formal model. That is, all process names used by the encapsulated target model are also defined by an equation in this set.

Overall, our formal model captures the modular architecture and the parametricity of CliSeAu's generic ECs. The EC models support cooperation via channels between their coordinator models, like with CliSeAu's generic ECs. Finally, regarding our focus on the cooperation, we particularly modeled the interface between the coordinator model and the local policy model to closely capture the interface in CliSeAu's generic ECs.

8.4. Soundness Notion in CSP

In earlier chapters, we have already captured security properties as sets of security-compliant sequences of events. The model of encapsulated targets introduced in Section 8.3 provides the remaining entity for a formal verification of sound enforcement of a security property. In this section, we make use of our the models of encapsulated target and of security properties to formalize soundness.

We have previously introduced soundness informally as the condition that the security property is never violated when the enforcement mechanism is applied to the target. Since in our formal model the encapsulated target is unambiguously determined by a policy model, we formalize soundness in terms of policy models.

Definition 8.20. Let

$$pm = (Id, (AGENT_i)_{i \in Id}, (IE_i, EE_i, ED_i, POL_i, ENF_i)_{i \in Id}, DR, \mathcal{EQ})$$

be a policy model and let $P \subseteq \alpha(ET_{pm})^*$ be a security property. We say that *pm* is a *sound policy model* for *P* if and only if ET_{pm} sat *P* holds under set $\mathcal{EQ}_{pm}$ of process equations. ◊

In the definition, security properties are modeled as sets of security-compliant sequences of events. Concretely, security properties are expressed on the events of the whole encapsulated target model, i.e., the events of the target's agents, effectable events of enforcers that are not events of the agents, and events used for the cooperation among EC models. This model of security properties, firstly, allows the security property to capture security requirements on the distributed target as a whole, including localizable as well as concerted security properties; Secondly it, allows the security property to speak about events introduced by the units of the distributed enforcement mechanism. Soundness with respect to a security property is then mainly captured through the "satisfies" predicate ET_{pm} sat *P*. That is, all traces of the encapsulated target model must be included in *P* and, thus, be security-compliant. Conversely, the security property is not violated by any of the possible runs of the encapsulated target model for the policy model. That is, the definition faithfully captures the condition of when the encapsulated target model for a policy model soundly enforces a security property.

Note that, by definition of the trace semantics of CSP, the set of possible traces of a process specified in CSP is prefix-closed. In consequence security properties that are safety properties can be captured well by the satisfies predicate in trace semantics, as also pointed out, e.g., by Roscoe [Ros05, p. 45].

Remark 8.4. According to Roscoe [Ros05], the trace semantics of CSP is inadequate for reasoning about liveness properties of process expressions. Where liveness properties are of concern, Roscoe advocates the failures/divergences semantics of CSP to give "the most satisfactory representation of a process" [Ros05, p. 211]. These semantics as well as liveness properties, however, are outside the scope of this chapter. ◊

Remark 8.5. In this chapter, our goal is a model that allows one to verify sound enforcement. As captured in Definition 8.20, soundness is concerned only with the outcomes of the enforcement mechanism, not the behaviors of the target that leads to these outcomes. In that regard, soundness differs from transparency [LBW09], i.e., the property that the enforcement mechanism does not alter the behavior of the target when it behaves security-compliant. Transparency, thus, is concerned with a relationship between the target's behaviors and the respective outcomes of the enforcement mechanism. Due to this inherent difference between soundness and transparency, transparency cannot be captured similarly to soundness in our model. In particular, $traces(\|_{i \in Id} AGENT_i) \cap P \subseteq traces(ET_{pm})$ constitutes only a necessary condition for transparency, as all security-compliant runs of the target are still possible. However, this condition is not sufficient in general, as it permits the enforcement mechanism to alter the security-compliant runs as long as the collection of all security-compliant runs of the target is preserved. ◊

Along with the semantics of CSP introduced in Section 8.2, the definition of a sound policy model provided in this section enables one to formally verify soundness (Requirement (Req-2)). We demonstrate this with a concrete example in the following section.

8.5. Application of the Model

In this section, we apply the formal model introduced in this chapter for modeling a CoDSPL policy and formally verifying that the resulting policy model is sound. The application scenario we employ for this purpose is the same as in Section 7.4: a distributed storage service in which a Chinese Wall Security Policy shall be enforced.

We proceed as follows. Firstly, we introduce our model of the target and our formalization of the security property. Secondly, we provide a model of the CoDSPL policy of ChESt. Finally, we show that our policy model indeed is sound.

8.5.1. Target and Security Property

We model the target, a distributed storage service as follows. The storage service consists of a set of $m \in \mathbb{N}$ agents, $SP = \{1, \ldots, m\}$, which we refer to as the service providers. It offers its services to the $k \in \mathbb{N}$ users $U = \{u_1, \ldots, u_k\}$ who can store files at the storage service and retrieve files from the service.

When a user requests a file from one of the service providers, this service provider first checks whether the file exists and the user has sufficient permissions to retrieve the file. If this is the case, then the service provider returns the file to the user. Otherwise, the service provider signals to the user that the access to the file is denied. We model the successful accesses by events of the form $e = (u, sp, o)$, where $u \in U$ is the requesting user, $sp \in SP$ is the service provider, and $o \in O$ is the object representing the requested file's content. We model the unsuccessful responses by events of the form $e = (u, sp, denied)$, where $denied \notin O$ is a special symbol that is distinct from any file content. We capture all the response events involving a service provider sp by the set $AE_{sp} = U \times \{sp\} \times O$ and denote the service provider for an event e by $sp(e) = U \times SP \times O$. We denote the union of these sets for all service providers by AE and refer to the events in the set as *access events*.

We model the individual service providers by the underspecified process expressions $(SP_{sp})_{sp \in SP}$ and set $\mathcal{EQ}_{sp}$ of process equations with $AE_{sp} \subseteq \alpha(SP_{sp})$ and such that the internal and external channels of the EC model (see Figure 8.1) are not used by the processes that model the service providers.

The files in our scenario may belong to competing companies. We say that such files are in conflict (of interest) and capture the conflicts between files by an irreflexive and symmetric relation $COI \subseteq O \times O$ on objects, called the *conflict of interest relation*. We lift the conflict relation on objects to access events with the relation $\otimes \subseteq AE \times AE$ defined by $(u, sp, o) \otimes (u', sp', o')$ if and only if $u = u'$ and $(o, o') \in COI$. Based on the previously introduced concepts, we formalize the Chinese Wall Security Policy [BN89] as follows.

Definition 8.21. For a given set $E \supseteq AE$, the *Chinese Wall security property* is the security property $ChW(E)$ defined by

$$ChW(E) = \{tr \in E^* \mid \neg\exists e_1, e_2 \in AE : (e_1 \lhd tr \wedge e_2 \lhd tr \wedge e_1 \otimes e_2)\} \qquad \Diamond$$

We capture the Chinese Wall Security Policy by a set of sequences over a parametric set of possible events. The condition for a sequence to be in this set constrains only the access events contained in the sequence: No two access events that are in conflict may be contained within one and the same sequence. All non-access events, such as user authentication or directory browsing, which we do not model explicitly, are not constrained by the property.

8.5.2. Policy Model

We develop a policy model for enforcing the Chinese Wall security property on the distributed storage service. Our goal is a policy model by which each service provider in encapsulated with an EC model. We begin by specifying the enforcer model and subsequently specify the local policy model.

Enforcer model We aim to enforce the security property by replacing access events that would violate the Chinese Wall security property by the corresponding denying events. That is, if the occurrence of an access event (u, sp, o) violates the security property, it shall be replaced by the access event (u, sp, denied). We capture this replacement by total function $\mathit{deny} : AE \to AE$, defined by $\mathit{deny}((u, sp, o)) = (u, sp, \mathit{denied})$ and $\mathit{deny}((u, sp, \mathit{denied})) = (u, sp, \mathit{denied})$. That is, denied accesses remain unchanged by the function. Note that for any $sp \in SP$ and $e \in AE_{sp}$, we have $\mathit{deny}(e) \in AE_{sp}$.

For implementing the replacement, we use a replacing enforcer of Example 8.3 on page 140 at each service provider. That is, for each service provider $sp \in SP$, we use $\mathsf{REPLACE}_{AE_{sp},AE_{sp}}$ with set $ChWD_{sp} = AE_{sp} \times (AE_{sp} \cup \{\mathit{term}\})^*$ of decisions and set AE_{sp} of effectable events. We denote the union of these sets of decisions for all service providers by $ChWD$. While in principle these enforcer models support replacement with sequences of events and support termination, we do not make use of these possibilities in the local policy model we define next.

Local policy model For enforcing the Chinese Wall security property, we specify a local policy model that applies static delegation and that exhibits a modular architecture. The modularity follows Chapter 7 with separation of concerns for delegation, decision-making and routing. We capture this modularity by three separate components that constitute our local policy model.

For static delegation, the responsibility function is a central element (see Section 7.2). We capture the responsibility function by function $\mathit{resp} : AE \to SP$, which we leave partially underspecified, requiring only that $\mathit{resp}(e) = \mathit{resp}(e')$ holds for $e \otimes e'$. As we observed in Conjecture 7.4 on page 120, the Chinese Wall security property is partitionable into sets of access events that share the same user and COI class. This partition refines the induced event partition of resp and therefore enables the responsible units to be authoritative. We model delegation in the local policy model for the EC model with identifier $i = sp$ at service provider $sp \in SP$ by the process expression DEL_i under the respective process equations provided in Figure 8.2 on page 151. The alphabet E_i^{DEL} is defined by

$$E_i^{\mathsf{DEL}} = \left\{ \begin{array}{l|l} \mathsf{lreq}.e, \mathsf{lereq}.e, & e \in AE_i, \\ \mathsf{rtreq}.(k, e'), & e' \in AE, k \in SP \setminus \{i\} \end{array} \right\}.$$

Intuitively, this processes receive locally intercepted events (on channel lreq) and distinguish whether this EC model is the responsible unit for the event or not. In the former case, the event is sent to the decision-making component (via channel lereq) and in the latter case, the event is sent to the routing component, together with the identifier of the event's responsible unit (via channel rtreq).

The decision-making component for the EC model with identifier $i \in SP$ is modeled by the process expressions $\mathsf{DEC}_i(q)$, where $q \in \mathcal{P}(AE)$, under the respective process equations provided in Figure 8.2 on the next page. The parameter q denotes the state of the decision-making component and models the access events that the decision-making component has previously permitted. The alphabet E_i^{DEC} is defined by

$$E_i^{\mathsf{DEC}} = \left\{ \begin{array}{l} \mathsf{lereq}.e, \mathsf{edec}.ed, \\ \mathsf{rereq}.e', \mathsf{rtrsp}.(k, ed') \end{array} \middle| \begin{array}{l} e \in AE_i, ed \in ChWD_i, \\ e' \in AE, ed' \in ChWD, k \in SP \setminus \{i\} \end{array} \right\}.$$

Intuitively, the processes receive events for decision-making based on the respective state. Events intercepted from the local agent are received via channel lereq and events from remote agents are received via channel rereq. For each event received on one of the two channels, the processes distinguish whether the event is in conflict with any of the previously permitted events or not. For this distinction, in the process equations for DEC_i, we make use of function $conf : \mathcal{P}(AE) \to \mathcal{P}(AE)$ defined by $conf(q) = \{e \in AE \mid \exists e' \in q \colon e \otimes e'\}$. For an event e that is in conflict with a previously permitted event, the process produces the decision $(e, \langle deny(e)\rangle)$, capturing that the event shall be replaced by the corresponding denial event $deny(e)$. Otherwise, the process produces the decision $(e, \langle e\rangle)$, capturing that the event shall occur (i.e., be replaced by itself). If the decision is for an event from the local agent, the process sends the decision to the coordinator model via channel edec. Otherwise, if the decision is for a remote agent $sp(e)$, it sends the decision to the routing component via channel rtrsp for being routed to the EC model of that agent.

The routing component serves the purpose of determining the next EC model to send messages with a particular destination unit. We model a static routing, in which the next unit towards a destination does not depend on prior routing choices. We capture this form of routing by a function $nxt : SP \times SP \to SP$, where $nxt(i, j) = k$ models that from unit i the next unit to destination j is k. For our application scenario, we define the function by $nxt(i, j) = j$ for each $i, j \in SP$, that is, the destination unit is directly connected to the source unit and the routing makes use of this. Based on the routing, we define the set of all delegation requests and responses by $ChWDR = SP \times (AE \cup ChWD)$, where the first component models the identifier of the destination unit and the second models the request or, respectively, response. The routing component for the EC model with identifier $i \in SP$ is modeled by the process expressions SRP_i, under the respective process equations provided in Figure 8.2 on the facing page.

The alphabet E_i^{SRP} is defined by

$$E_i^{\mathsf{SRP}} = \left\{ \begin{array}{l} \mathsf{rereq}.e, \mathsf{rtreq}.(k, e), \\ \mathsf{appv}.ed, \mathsf{rtrsp}.(k, ed'), \mathsf{rreq}.dr, \\ \mathsf{fwd}.(k, dr), \mathsf{ddec}.(k, dr), \mathsf{rdec}.(k, dr) \end{array} \middle| \begin{array}{l} e \in AE, k \in SP \setminus \{i\}, \\ ed \in ChWD_i, ed' \in ChWD, \\ dr \in ChWDR \end{array} \right\}.$$

Intuitively the process receives three kinds of inputs: remote requests coming from the coordinator model (via channel rreq), requests from the delegation component to route

$$
\begin{aligned}
\mathsf{DEL}_i &\overset{\text{def}}{=}_{E_i^{\mathsf{DEL}}} \mathsf{lreq}?e\colon \{e' \in AE_i \mid resp(e') = i\} \to \mathsf{lereq}!e \to \mathsf{DEL}_i \\
&\quad \Box \mathsf{lreq}?e\colon \{e' \in AE_i \mid resp(e') \neq i\} \to \mathsf{rtreq}!(resp(e), e) \to \mathsf{DEL}_i \\
\mathsf{DEC}_i(q) &\overset{\text{def}}{=}_{E_i^{\mathsf{DEC}}} \mathsf{lereq}?e\colon AE_i \cap conf(q) \to \mathsf{edec}!(e, \langle deny(e)\rangle) \to \mathsf{DEC}_i(q) \\
&\quad \Box \mathsf{lereq}?e\colon AE_i \setminus conf(q) \to \mathsf{edec}!(e, \langle e\rangle) \to \mathsf{DEC}_i(q \cup \{e\}) \\
&\quad \Box \mathsf{rereq}?e\colon conf(q) \to \mathsf{rtrsp}!(sp(e), (e, \langle deny(e)\rangle)) \to \mathsf{DEC}_i(q) \\
&\quad \Box \mathsf{rereq}?e\colon AE \setminus conf(q) \to \mathsf{rtrsp}!(sp(e), (e, \langle e\rangle)) \to \mathsf{DEC}_i(q \cup \{e\}) \\
\mathsf{SRP}_i &\overset{\text{def}}{=}_{E_i^{\mathsf{SRP}}} \mathsf{rreq}?(k, e)\colon \{i\} \times AE \to \mathsf{rereq}!e \to \mathsf{SRP}_i \\
&\quad \Box \mathsf{rreq}?(k, ed)\colon \{i\} \times ChWD \to \mathsf{appv}!ed \to \mathsf{SRP}_i \\
&\quad \Box \mathsf{rreq}?(k, x)\colon \{(k', x') \in ChWDR \mid k' \neq i\} \\
&\qquad \to \mathsf{fwd}!(nxt(i, k), (k, x)) \to \mathsf{SRP}_i \\
&\quad \Box \mathsf{rtreq}?(k, e)\colon SP \times AE \to \mathsf{ddec}!(nxt(i, k), (k, e)) \to \mathsf{SRP}_i \\
&\quad \Box \mathsf{rtrsp}?(k, ed)\colon SP \times ChWD \to \mathsf{rdec}!(nxt(i, k), (k, ed)) \to \mathsf{SRP}_i \\
\mathsf{ChWLP}_i &\overset{\text{def}}{=} (\mathsf{DEL}_i \parallel \mathsf{DEC}_i(\emptyset) \parallel \mathsf{SRP}_i) \setminus H_i^{\mathrm{pol}}
\end{aligned}
$$

Figure 8.2.: Components of the local policy model for enforcing the Chinese Wall security property

delegation requests (via channel rtreq), and requests from the decision-making component to route delegation responses (via channel rtrsp). Incoming delegation requests and delegation responses for the local unit (i.e., where $k = i$) are processed by sending the event to the decision-making component (via channel rereq) or, respectively, sending the decision to the coordinator model for being enforced (via channel appv). All other inputs lead the process to send a delegation request or response to the coordinator model, using function *nxt* to compute the next unit.

Note that separation of concerns between delegation, decision-making, and routing manifests itself in the process expressions. The responsibility function *resp* is only used by processes DEL_i. The routing function *nxt* is only used by processes SRP_i. And the conflicts of interest are only used by processes $\mathsf{DEC}_i(q)$.

We define the local policy model ChWLP_i for the Chinese Wall security property as the parallel composition of the three previously introduced components. In this composition, the three components communicate with each other via their shared channels, which are hidden from the environment through the hiding of the set H_i^{pol}, defined by which hides the internal communication with the set

$$
H_i^{\mathrm{pol}} = \left\{ \begin{array}{l|l} \mathsf{lereq}.e, \mathsf{rereq}.e', & e \in AE_i, e' \in AE \\ \mathsf{rtreq}.(k, e'), \mathsf{rtrsp}.(k, ed) & ed \in ChWD, k \in SP \setminus \{i\} \end{array} \right\}.
$$

The events that are exempt from the hiding establish the communication interface between the local policy model and the coordinator model.

Policy model Based on the ingredients introduced before in this section as well as Section 8.5.1, we define the policy model *ChWgp* for our application scenario by the tuple

$$ChWgp = \left(\begin{array}{l} SP, (SP_i)_{i \in SP}, \\ (AE_i, AE_i, ChWD_i, \mathsf{ChWLP}_i, \mathsf{REPLACE}_{AE_i,AE_i})_{i \in SP}, \\ ChWDR, \mathcal{EQ}_{ChW} \end{array} \right),$$

where the set $\mathcal{EQ}_{ChW}$ of process equations is defined as the union of the process equations of the target, $\mathcal{EQ}_{ChW}$, the process equations of the replacing enforcer (see Example 8.3 on page 140), and the process equations in Figure 8.2 for all $i \in SP$ and $q \in \mathcal{P}(AE)$.

Finally, we show that the our policy model *ChWgp* is proper. Technically, this allows us to apply Definition 8.19 for obtaining an encapsulated target model for *ChWgp*. Intuitively, this confirms that *ChWgp* does not involve undesired interactions between components that violate the modular architecture of EC models.

Theorem 8.1. *ChWgp is a proper policy model.* ◊

Proof. Firstly, we show that *ChWgp* indeed is a policy model as it is defined in Definition 8.17 on page 145. Since the number of elements in the tuple *ChWgp* as well as their basic mathematical type (set or process expression) are correct, what remains to be shown are the conditions that relate these elements. Let $i \in SP$ be arbitrary but fixed in the following. Then the conditions for the EC model with identifier i are the following.

1. ChWLP_i is a local policy model for $(SP, i, AE_i, ChWD_i, ChWDR)$.

   By definition of a local policy model, this condition is satisfied if

   $$\alpha(\mathsf{ChWLP}_i) = \left\{ \begin{array}{l|l} \mathsf{lreq}.e, \mathsf{rreq}.dr, \mathsf{edec}.ed, \mathsf{appv}.ed, & e \in IE, ed \in ED, \\ \mathsf{ddec}.(k, dr), \mathsf{rdec}.(k, dr), \mathsf{fwd}.(k, dr) & dr \in DR, k \in SP \backslash \{i\} \end{array} \right\}$$

   holds for $IE = AE_i$, $ED = ChWD_i$, and $DR = ChWDR$. The definition of ChWLP_i gives $\alpha(\mathsf{ChWLP}_i) = (E_i^{\mathsf{DEL}} \cup E_i^{\mathsf{DEC}} \cup E_i^{\mathsf{SRP}}) \setminus H_i^{\mathsf{pol}}$, which by the definitions of the four sets satisfies the condition.

2. $\mathsf{REPLACE}_{AE_i,AE_i}$ is an enforcer model for $ChWD_i$ and AE_i.

   By its definition in Example 8.3, the replacing enforcer $\mathsf{REPLACE}_{AE_i,AE_i}$ is an enforcer model for $AE_i \times (AE_i, \{term\})^*$ and AE_i. The former of the two sets coincides with $ChWD_i$ by its definition on Page 149. Therefore, the condition is satisfied.

3. $\mathcal{EQ}_{ChW}$ contains a process equation for each process name that occurs in SP_i, ChWLP_i, and $\mathsf{REPLACE}_{AE_i,AE_i}$.

   This condition is satisfied because $\mathcal{EQ}_{ChW}$ is defined as the union of the corresponding three sets of process equations.

Secondly, we show that *ChWgp* is proper, by showing that *ChWgp* satisfies the individual conditions in Definition 8.18. For this, let $i \in SP$ be arbitrary but fixed in the following.

On 8.18 (a): The intersection in the condition is empty by our definitions of SP_i (Page 148) and AE_i (Page 148).

On 8.18 (b): Condition $AE_i \subseteq \alpha(SP_i)$ holds by definition of SP_i.

On 8.18 (c): This condition holds because by definition of *ChWgp*, intercepted events and effectable events are both equal AE_i. □

Overall, the policy model we have defined models the CoDSPL policy of ChESt (described in Section 7.4) for enforcing the Chinese Wall security property on a distributed storage service. Concretely, the model captures several key aspects. Firstly, it captures static delegation, i.e., delegation with a static assignment of intercepted events to responsible unit, in the particular form that two conflicting events share the same responsible unit. Secondly, the model captures the separation of concerns delegation, decision-making, and routing. In some of the details, the model deviates from the CoDSPL policy of ChESt. Firstly, the ChESt uses hash maps for storing, for each user and each COI class, the name of the company whose files the user has accessed from this COI class. In the model, where performance and storage complexity is not our concern, we simplify this to storing the sets of all permitted access events. Secondly, deviating from the ChESt, the delegation requests in our model include the identifier of the responsible unit. This merely simplifies our local policy model a bit, as it allows the routing component to identify whether a delegation request must be forwarded to another unit or not without invoking the delegation component. However, since the responsibility function is known to each unit, including the responsible unit does not provide any additional information to the cooperating units. That is, despite the named deviations, we consider our model to faithfully capture the CoDSPL policy of ChESt.

8.5.3. Soundness Result

In the previous section, we formally modeled the components for enforcing the Chinese Wall security property in a distributed storage service with static delegation. We can now show that this instance of our formal model indeed soundly enforces the Chinese Wall security property.

Theorem 8.2. *ChWgp is a sound policy model for* $ChW(\alpha(ET_{ChWgp}))$ ◊

Proof sketch. Suppose, Theorem 8.2 does not hold. Then there exists a trace *tr* of ET_{ChWgp} that contains two conflicting events e_1 and e_2 – i.e., with $e_1 \otimes e_2$. The replacing enforcer ensures that an access event is performed only after it has received a corresponding "permitting" decision. We can show that the permission decisions for e_1 and e_2 must have been made by their respective responsible units. Since e_1 and e_2 are in conflict, they have the same responsible unit by the definition of function *resp*. This unit must hence have permitted both events e_1 and e_2. However, this contradicts the decision-making component DEC of that unit together with the definition of function *conf*. Hence, *tr* cannot contain both conflicting events e_1 and e_2 and Theorem 8.2 holds.

A full formal proof of the theorem can be found in Appendix A.2. □

The soundness result shows how our definitions related to soundness for our CSP model of encapsulated targets can be instantiated for a concrete example. Moreover, the soundness result provides strong evidence even beyond the model. Since we captured in the model the same cooperation technique we used for enforcing the Chinese Wall security property in the case study of Chapter 7, the soundness result further supports our soundness observations in that chapter.

8.6. Summary

We introduced a formal model of an enforcement mechanism for distributed targets in the process algebra CSP. The model captures the modular architecture of CliSeAu's generic enforcement capsules, including the communication within and between the generic ECs. The model is parametric and its parameters reflect the CoDSPL policies that can be input to CliSeAu. In the model, we captured also how the units of the enforcement mechanism are combined with the agents of the target by a specific combination of parallel composition and hiding. We provided a formal definition of sound enforcement for instances of our model based on CSP's "satisfies" predicate. By an example instance of the formal model, we demonstrated for enforcing a At the example of the Chinese Wall Security Policy in a distributed storage service, we demonstrated how the formal model can be instantiated and how instances of the model can be proven to soundly enforce security.

Technically, our proposed technique for combining the models of units and agents enables the units to intercept events without any effects on the environment. Previously proposed process-algebraic models of enforcement mechanisms [BOS07; BBK12] employ a single parallel composition between the target and the whole mechanism. As a result, interception, decision-making, and the imposition of countermeasures take place instantaneously such that the only countermeasure is to block security-violating events. Our technique allows for a wide range of countermeasures – including termination, suppression, insertion, and replacement – to be modeled, as demonstrated in our examples. Moreover, our technique enables units to communicate internally as well as with other units between the point in time an event is intercepted and the point in time a countermeasure is applied.

By faithfully capturing the architecture of CliSeAu's generic ECs and capturing instances of the parametric model as through models of CoDSPL policies, the formal model provided in this chapter allows one to analyze modeled CoDSPL policies with regard to sound enforcement. Our model makes the concurrent operation of agents and units as well as the communication between units explicit. This explicitness allows the model to capture how information is disseminated between units while the agents keep running and particularly allows the model to expose the effects of race conditions on the enforcement. Providing means for the formal verification of proposed enforcement mechanisms is in line with Anderson's principles that an enforcement mechanism shall be subject to analysis and testing [And72]. We are not the first to use formal modeling for distributed enforcement [KP14; Kel16]. However, our model is the first formal model of a cooperating distributed enforcement mechanism.

Chapter

9

Related Work

9.1. Overview

In this chapter, we compare the contributions presented in Chapters 3 to 8 with tools, techniques, and formal models proposed in the literature. We first compare with policy languages and instrumentation techniques used by other distributed as well as non-distributed generic enforcement mechanisms in Sections 9.2 and 9.3. We then compare specifically with other distributed enforcement mechanisms and particularly with delegation for cooperation among the units of distributed enforcement mechanisms in Sections 9.4 and 9.5. Approaches to separation of concerns pursued in other enforcement mechanisms are discussed in Section 9.6 and a comparison with other formal models of enforcement mechanisms is provided in Section 9.7. Finally, in Section 9.8, we compare with related works on the application scenarios used in this thesis.

9.2. Policy Languages for Run-time Enforcement

The languages used in run-time enforcement that are conceptually closest to CoDSPL are the languages used by Polymer and JavaMOP. The language used by Polymer [BLW09] essentially consists of two parts: an *action declaration file*, which specifies a set of patterns that match Java methods and thereby declares the operations of the target that Polymer shall intercept, and a policy definition, which specifies a Java class derived from a Policy base class in a variant of Java that augments Java with a **switch**-like construct for distinguishing program operations by action patterns. A particularity of Polymer's policies is that the policies separate decisions from their enforcement, for the purpose of making policies composable. Polymer's action declaration file serves the same purpose as CoDSPL's specification of security-relevant program operations. For both tools, the relevant operations are methods. CoDSPL's patterns for specifying Java operations, being based on AspectJ, are more expressive than the ones of Polymer as they allow, for instance, specifying constraints on the calling context. Polymer's policies have in common with

CoDSPL policies that they are specified in the form of Java code with a certain mandatory low-level architecture. The counterpart to Polymer's Policy class is CoDSPL's LocalPolicy class. Like CoDSPL, Polymer's policy language also supports abstracting from concrete program operations [BLW09, Section 2.2] and also distinguishes the decision-making from the enforcement of decisions [BLW09, Section 2.1]. There are two main conceptual differences between CoDSPL and the policy language of Polymer. Firstly, CoDSPL policies can specify cooperation between units, whereas Polymer does not aim at distributed targets and, hence, does not foresee cooperation. Secondly, Polymer's policies include means for hierarchically composing multiple policies through logical combinators. CoDSPL does not particularly foresee policy composition.

A JavaMOP policy [MJG+12] consists of three parts: a specification of relevant program operations, a specification of the property to be monitored by JavaMOP, and specification of how JavaMOP shall act upon violation or satisfaction of the specified property. The relevant program operations are specified through a syntax close to the syntax of AspectJ pointcuts. The property to enforce can be specified in several ways, including finite-state machines and LTL formulae. This property is bound to references of objects of the running target or to combinations of object references. That is, individual objects can violate or satisfy the given property. The actions to be performed upon violation or, respectively, satisfaction of the property are specified as a Java code block. CoDSPL shares with JavaMOP that security-relevant program operations are specified in a syntax close to the syntax of AspectJ pointcuts. Both tools also use the semantics of AspectJ pointcuts. Moreover, CoDSPL shares with JavaMOP that countermeasures are specified as Java code. The difference between CoDSPL and JavaMOP is in the decision-making, in which JavaMOP's language allows very concise specifications but limits the expressiveness. For instance, conditions on the value of String objects cannot be expressed. In the design of CoDSPL, the focus is more on the expressiveness, even though it comes at the expense of less concise specifications.

The policy language used by Clara [BH12] shows similarity to the language used by JavaMOP. The conceptual differences are that Clara's language keeps the specification of operations and the corresponding countermeasure code together [BH12, Figure 4]. The key difference between JavaMOP and Clara is that Clara uses static analysis for more precisely identifying join point shadows based on the specified property. CliSeAu, as the tool that applies CoDSPL policies, does not provide such a static analysis.

Automata-like specifications of enforcement mechanism behavior can be found in other policy languages than the languages of JavaMOP and Clara. *SAL* [ES00b], PSLang [Erl04], and ConSpec [AN08] are specification languages in which policy state can be captured through a set of typed variables from a pre-defined range of types. Transitions are labeled with program operations and the state resulting from a transition is specified via a code block in a domain-specific language that re-assigns the state variables. While the use of domain-specific languages for operational policy specifications generally yields more concise policies, it also reduces expressiveness. The development from SAL to PSLang indicates that moving towards a more expressive policy language involves adding data types known also from general-purpose programming languages.

Mechanisms for monitoring targets with respect to properties have been proposed [SVA+04; HPB+07; BGT09; BHK+12; HOW14; BCE+16], where the desired property is specified in a variant of linear-time temporal logic (LTL). These mechanisms aim at the detection of property violations, not their prevention. *PT-DTL* [SVA+04] is a temporal logic with operators for remote sub-formulas, i.e., formulas that are expected to be satisfied at a remote agent. The logic allows a rather declarative way of specifying the cooperation between units than CoDSPL's operational approach. The semantics of PT-DTL, though, is defined in a way that cooperation takes place if and when agents communicate such that both false positives as well as false negatives cannot be prevented. The logics MFOTL [BHK+12; BCE+16], MTL [HOW14], and PTLTL^{FO} [BGT09], in contrast to PT-DTL, allow universal quantification but do not provide explicit constructs for cooperation or distributed targets. *OSL* [HPB+07] is a temporal logic specifically aiming at usage control without quantifiers but with implicit universal quantification over free variables [HPB+07, Section 3.4].

9.3. Instrumentation-based Enforcement Mechanisms

Aspect-oriented programming forms the technical basis of several instrumentation-based enforcement mechanisms. JavaMOP [MJG+12] and Clara [BH12] use AspectJ internally as a back-end for instrumenting the bytecode of Java targets. JavaMOP is a generic mechanism for monitoring and enforcement of properties on Java programs. Its policy language incorporates parts of the AspectJ language for the specification of program operations at the application-level [MJG+12, Figure 2]. Given a policy, JavaMOP creates an AspectJ aspect that a user of JavaMOP can than weave into a target using the AspectJ compiler [MJG+12, Section 2.2]. Clara uses a different policy language than JavaMOP but its policy language also incorporates parts of the AspectJ language [BH12, Fig. 5]. Like JavaMOP, Clara also generates an AspectJ aspect for use with the AspectJ compiler. CliSeAu follows a similar approach as JavaMOP and Clara: CliSeAu's syntax for specifying security-relevant program operations, introduced in Section 3.4.1 on page 30, builds on AspectJ's pointcut language and CliSeAu's encapsulation tool internally generates an AspectJ aspect. A technical difference is that the encapsulation tool rather than its user invokes the AspectJ compiler and thereby makes AspectJ more transparent to its users than JavaMOP an Clara do. The main architectural difference between CliSeAu and JavaMOP as well as Clara is that the latter weave all parts of the mechanism into the agent while CliSeAu's cross-lining places part of the mechanism into a separate program and generates the mechanisms in a way that establishes the interfaces between the parts.

Early versions of JavaMOP [CR07] supported the generation of so-called *outline monitors*, a variant of JavaMOP monitors whose interception functionality is inlined into the code of the target but whose decision-making and acting functionality is placed into a "standalone process or thread" [CR07, Section 4.1]. The communication between the two parts of the mechanism is implemented via message passing. Notably, the feature is rarely documented and is no longer supported in recent versions of JavaMOP, particularly including the most recent version, JavaMOP 4 [Khe]. Due to the choice of placing the acting functionality outside the scope of the target, an outline monitor of JavaMOP "cannot access the internal state of the monitored system, limiting its capability for error recovery" [CR07]. CliSeAu's cross-lining technique places the enforcer into the target, which enables countermeasures that modify the target's internal state.

Other enforcement mechanisms in the literature have been proposed with dedicated instrumentation techniques. SASI is an enforcement mechanism for targets in Java bytecode as well as for x86 bytecode [ES00b]. In the following exposition, we focus on the variant for Java. SASI implements the concept of security automata by instrumenting Java targets such that automata transitions are performed before security-relevant bytecode instructions. Erlingsson and Schneider call this technique *inlined reference monitors* (*IRMs*). Erlingsson also proposes the *Policy Enforcement Toolkit* (PoET), an enforcement mechanism that augments SASI in multiple directions, including support for instrumenting dynamically loaded code and instrumenting at a more coarse granularity of program operations. Compared to CliSeAu, PoET supports instrumenting Java bytecode at a finer granularity of program operations and generally applies more sophisticated techniques for ensuring that the mechanism's code cannot be circumvented by the target. In the design and implementation of CliSeAu, the focus is a different one: providing sufficient granularity for capturing security-relevant program operations and providing generic means for cooperation.

Another instrumentation technique specifically developed for an enforcement mechanism is the technique employed by Polymer [BLW09]. Conceptually, Polymer's instrumentation technique differs from CliSeAu's in two ways. Firstly, Polymer uses the Apache BCEL library for explicitly specifying how the target's Java bytecode is instrumented [BLW09, Section 3.1]. This aims at more precise control over how the instrumented target behaves, particularly that the enforcement mechanism is invoked at the specified operations and is not circumvented by the target. Efforts in this direction have also been incorporated into *Mobile* [HMS06a], an enforcement mechanism for Microsoft's CIL language. In the design of CliSeAu, AOP tools are employed instead of custom instrumentation techniques mainly because these tools already have been used and stabilized for several years and error-prone re-implementations of bytecode rewriting could be avoided. Secondly, Polymer instruments the bytecode of a target from within a custom class loader at run-time, when the code of the target is loaded. This enables Polymer to instrument code that is not available before the target executes. CliSeAu does not support instrumentation of dynamically loaded code.

The enforcement mechanisms discussed in this section have in common that they involve some form of automatic modification of the target's code. Beyond this approach, there are also mechanisms built on modifications of the system in which the target is executed [EGC+10; OBM10] or built on manual modifications of specific targets [KP14] or libraries used by targets [SVA+04; BDE+08]. Such approaches are outside the scope of this thesis.

In her work, Mazaheri [Maz12] uses an earlier version of CliSeAu and provides an extension for supporting Ruby targets. This extension is based on a JRuby program that mediates the communication between inlined interceptor and enforcer components, implemented in Ruby, and the decider program, implemented in Java. The support for Ruby targets in CliSeAu presented in this thesis is built on an architecture that does not require such a mediation component and rather uses JSON as interchange format for objects between Ruby components and Java components. Our architecture of CliSeAu for Ruby targets is, thus, simpler and enabled a more efficient implementation by avoiding delays caused by the mediator component.

9.4. Enforcement Mechanisms for Distributed Targets

In the direction of enforcement mechanisms for distributed targets, the line of work possibly closest to this thesis is the usage control by Kelbert and Pretschner [KP12; KP13; KP14; KP15; Kel16]. This line of work contributes a formal model of cross-system data flow tracking [KP13; KP14], an enforcement mechanism implementing the concepts behind the model [KP15], and empirical evaluations of the enforcement mechanism [KP15]. The provided enforcement mechanism features a modular architecture [KP15, Fig. 3] consisting of one or more policy enforcement points (PEP), a policy decision point (PDP), a policy information point (PIP), and a *Cassandra node*. The PEP combines functionality of what in CliSeAu's architecture is interceptor and enforcer. The line of work builds on PEPs specialized for particular targets or components like operating systems and interpreters [Kel16, Section 2.2.2]. That is, how the PEP and, by transitivity, the enforcement mechanism is applied to a given target or system is not part of this line of work. CliSeAu allows its users to apply an enforcement mechanism to given Java and Ruby targets and to select via the policy which operations to intercept. This enables users to enforce security properties that are naturally formulated at an application-level granularity of operations (e.g., "download of a file over the FTP protocol") without implementing a specialized PEP or breaking the operations down into low-level operations of, e.g., an operating system for which a PEP has been implemented. PDP, PIP, and Cassandra node of Kelbert et al. together realize a functionality conceptually similar to CliSeAu's coordinator and local policy components: making decisions, capturing information relevant for decision-making, and cooperation. The approach by Kelbert et al. relies on cooperation based on information exchange via an instance of the distributed database Cassandra [LM10]. Since Cassandra is a general-purpose database, it trades availability and consistency of data; Configuring Cassandra to enforce consistency, as pointed out by Kelbert [Kel16, p. 90], incurs performance penalties through synchronization among all nodes of the distributed database, which the effectively corresponds to a centralized database. By having the cooperation specified as part of the policy, CliSeAu enables its users to tailor the cooperation to the respective security property to be enforced and thereby achieve consistency without centralization, as we demonstrate, e.g., in Chapter 7.

The *law-governed architecture for distributed systems* (*LGAD*) by Minsky [Min91] and its implementation by the Moses toolkit [MU00] control distributed programs with a particular focus on the interaction between agents. The Moses toolkit supports controlling the interaction between agents of distributed Java programs. Technically, Moses is implemented as a middleware that agents of the controlled target are expected to use for all cooperation among themselves [Min05]. Moses aims at enforcing properties, called *laws*, on the cooperation between agents. Moses enforces laws at the level of agent communication by delivering, blocking, or modifying exchanged messages. CliSeAu differs from Moses in three main aspects. First, CliSeAu can intercept communication operations of agents, as Moses can, but can additionally intercept computation operations of agents. Second, CliSeAu can impose countermeasures on a target that change the communication behavior between agents, as Moses can, but can additionally impose countermeasures that alter internal operations of an agent. Third, CliSeAu can be applied to Java programs even if they have not been programmed to use a particular middleware like Moses. Overall, this enables CliSeAu to be applied to a wider range of targets and to enforce a wider range of security requirements than Moses.

DiAna [SVA+04] is a tool for monitoring temporal properties on snapshots of distributed Java programs. These programs are assumed to be built on a monitoring library that is provided by DiAna. In this sense, DiAna is similar to Moses. The DiAna intercepts the communication operations between the agents of the target and its distributed components exchange information among each other piggy-backed on the messages exchanged between the agents. DiAna signals detected violations of a given property but does not impose countermeasures. That is, DiAna performs cooperative decentralized monitoring. The conceptual difference between CliSeAu and DiAna subsumes the three aspects in which CliSeAu and Moses differ. Further noteworthy differences are that DiAna supports declarative property specifications in a distributed variant of past-time temporal logic, PT-DTL, while CliSeAu supports operational policy specifications.

Porscha [OBM10] is a mechanism for enforcing digital rights management policies in the middleware of Android smartphones. Porscha operates as a proxy for content like SMS and e-mail and constrains access to such content to a policy-determined set of local or remote applications. For transmission of content, Porscha utilizes encryption to protect the content. Moreover, Porscha transmits together with the content also the policies that apply to the content. That is, Porscha cooperates across smartphones by transmitting policies. CliSeAu and Porscha have in common that they support cooperative decentralized enforcement. A key difference is that CliSeAu's units can cooperate even when the respective encapsulated agents are not communicating. This allows CliSeAu to enforce security properties in which local operations (e.g., the printing of a document) can have an impact on remote agents (e.g., the document may be printed one times less there).

Somewhat related to enforcement for distributed targets is the usage control approach by Lovat et al. [PLB11; Lov15; LOP16]. The approach aims at performing usage control within a single non-distributed system, across multiple layers of abstraction such as operating system, window manager, and database. The approach makes use of a modular enforcement mechanism consisting of a PEP, a PDP, and a PIP component for each layer, i.e., similar to the architecture of Kelbert and Pretschner [KP15]. The layers' mechanisms exchange information about their state upon each intercepted event [Lov15, Algorithm 1]. The main difference between Lovat et al.'s approach and CliSeAu is that the former is based on cooperation via shared memory while CliSeAu cooperates via message passing. How the conceptually orthogonal approaches compare has not yet been investigated. Similar to the approach of Kelbert et al., the work by Lovat et al. builds on PEPs specialized for particular targets or components like operating systems and interpreters. That is, how the enforcement mechanism is applied to a given target or system is not in the scope of this line of work.

9.5. Cooperation by Delegation

Notions of delegation are wide-spread in computer science. The notions probably closest to the delegation in this thesis are those from multi-agent systems and software engineering. In multi-agent systems, the notion of delegation is used for "requests for action and requests for solutions" [Fer99, p. 332]. At a more technical level, our delegation among units also matches notions of delegation from software engineering, such as "an implementation mechanism in which an object forwards [...] a request to another object. This delegate carries out the request on behalf of the original object." [GHJ$^+$95, p. 360] At this level, the local policies of units represent the forwarding objects and these local policies carry out requests by making decisions.

Note that prevalent notions of delegation in computer security – e.g., "one entity authorizes a second entity to perform functions on its behalf" [Bis03, p. 7] – are not in the focus of our work. In our work, when units delegate, functional aspects dominate: making a decision about an intercepted operation by involving other units that can contribute information. For instance, the units of CReDiC, our enforcement mechanism for controlled re-sharing in Diaspora*, trust each other in terms of making sound and transparent decisions, retaining the information contained in delegation requests, and not abusing their capacity to make decisions for denial of service attacks. We therefore do not build our units on authorizations, e.g., for decision-making or for accessing units' state. Effective cooperation in case of distrust among units would be a topic of future work.

Enforcement mechanisms for distributed targets proposed in the literature take approaches for cooperation that focus more on exchange of information than on the delegation of decision-making. The mechanism by Kelbert and Pretschner [KP15], for instance, use a distributed database for exchanging information, and The approach of DiAna [SVA$^+$04] for cooperation is based on information piggy-backed on the communication between the agents of the target. Moses [MU00] is a rather generic toolkit for governing the interaction between agents in a distributed target and its rule-based policy language could, as we presume, be used for specifying delegation. However, to the best of our knowledge, examples of using Moses comprise policies based on information exchange rather than delegation. We are not aware of other enforcement mechanisms than CliSeAu in which cooperation is realized through delegation.

Static delegation We published an instance of static delegation initially in 2012 for enforcing the Chinese Wall Security Policy in a decentralized fashion and formally proved soundness of the instance [GMS12]. Technically, in ChESt, we assign responsibilities by hash values of user names and COI classes. We published this determination of responsibilities for the CWSP for the first time as part of the initial publication of the CliSeAu implementation in 2014 [GHM14]. This approach was afterwards adopted by Decat, Lagaisse, and Joosen [DLJ15], who propose a decentralized architecture for enforcing security properties specified in a variant of XACML. The architecture proposed by Decat et al. consists of several coordinator components, one for each agent of the distributed target. Concretely, Decat et al. propose that "each subject and resource is assigned to exactly one responsible coordinator and this coordinator manages this subject/resource for all policy evaluations related to it. [...] [T]he responsible coordinators are determined based on a hash of the id of the subject and resource in question." This is close to what we implement in ChESt, except that we use the COI class in the computation of the hash value rather than the resource (i.e., the file).

As discussed on on this page, other approaches for cooperative enforcement in distributed targets use different means than delegation for the cooperation among units.

This includes the usage control mechanism by Kelbert and Pretschner [KP15] as well as DiAna [SVA+04]. Several mechanisms for usage control in distributed targets particularly control agent-internal events on policy-relevant data as well as events that transfer such data between the agents of a distributed target [PHB06; ZSS08]. In these mechanisms, agent-internal events are decided by the unit of the respective agent and transfer events are decided by the unit of the sending agent. This approach by the mechanisms can be seen as a "local" variant of static delegation, provided that the information about the agent from which an event was intercepted is part of the event.

Most closely related to our definition of order-insensitive security properties (Definition 7.5) might be the properties enforceable with shallow history automata. *Shallow history automata* [Fon04] are a variant of security automata [Sch00], whose states record the set of all granted events. That is, shallow history automata do not record the ordering or multiplicity of granted events. In the following, we refer to the class of security properties that are enforceable with a shallow history automaton as *shallow history properties*. The classes of order-insensitive security properties and of shallow history properties intersect, but none is subsumed by the other. In the intersection of the classes are properties that are not just insensitive to ordering but also insensitive to multiplicities of occurrences. Order-insensitive but not shallow history are properties in which the multiplicity of past events makes a difference for the permissibility of subsequent events. An example for such properties is $P = \{\langle\,\rangle, \langle e\rangle, \langle e, e\rangle, \langle e, e, e\rangle\}$, in which the event e may occur at most 3 times. Shallow history properties that are not order-insensitive exist as well. An example for such properties is $P = \{t \in \{e, e'\}^* \mid t = t'.\langle e'\rangle.t'' \multimap t' \upharpoonright \{e\} \neq \langle\,\rangle\}$ in which event e' may occur after and only after event e has occurred at least once. Another example is the quota policy we discussed in Example 7.3 on page 113. That is, while order-insensitive security properties intuitively bear similarity to Fong's shallow history properties, the two classes of properties are different and none of them is subsumed by the other.

9.6. Separation of Concerns

Separation of concerns is commonly used in software engineering for reducing conceptual complexity and for increasing maintainability of software [BME+07, pp. 13–14]. To the best of our knowledge, separation of concerns for enforcement mechanisms has been described explicitly only in few works of the research area. Polymer's central feature of policy composition is based on a separation between decision-making and the implementation of decisions made. This separation also separates the decision-making from state-keeping, is part of the implementation of decisions. The separation proposed by Polymer has been adopted also by other enforcement mechanisms in the literature [RHN+13]. A similar separation between decision-making and state-keeping is also proposed by the XACML reference architecture through its distinction of the components PDP and PIP. This architecture, along with its underlying separation, has been adopted by several mechanisms for usage control [KP15; LMM12; LOP16]. We adopted this separation between decision-making and state-keeping in our framework for modular local policies. For our purposes, we further refined the separation, distinguishing decision-making into subgoal selection, subgoal realization, and routing. Enforcement mechanisms for distributed targets, such as DiAna, Moses or the mechanism by Kelbert and Pretschner [KP15], as far as literature about them suggests, do not provide a more fine-grained separation of concerns than between decision-making (PDP) and state-keeping (PIP).

9.7. Formal Models of Enforcement Mechanisms

We compare with related works in two directions. Firstly, we compare our contributions with other models of enforcement mechanisms for distributed targets. Secondly, we compare our contributions with other formalizations of enforcement mechanisms that utilize the process algebra CSP.

Formal models for distributed enforcement of security Martinelli and Matteucci [MM08] propose a framework for synthesizing centralized and, respectively decentralized mechanisms for enforcing security properties in distributed targets. Concretely, the security properties for distributed targets are specified in a particular variant of modal logic. These properties are then first decomposed into properties of the target's components and, second, the result is subject to one of the two proposed syntheses – a centralized one or a decentralized one. The approach is instantiated at the example of the Chinese Wall security property. The centralized approach relies on a single mechanism and therefore does not involve communication between mechanisms. The decentralized approach involves multiple mechanisms – one for every component of the target –, but the individual mechanisms cannot communicate with each other. In consequence, concerted security properties such as, e.g, the Chinese Wall Security Policy as specified in Example 2.2 cannot be enforced unless one accepts severe cuts in the effectiveness of the mechanism as, e.g., pointed out by Martinelli and Matteucci [MM08].

Kelbert and Pretschner [KP14] and Kelbert [Kel16] propose a model for decentralized distributed usage control. In the model, systems are captured by the sets of possible traces of the systems' distributed subsystems (agents in the terminology of this thesis). The properties enforced in the proposed model are specified in OSL, a temporal logic. The authors formalize which of the distributed enforcement mechanisms' coordination is required for evaluating whether a given OSL formula is satisfied or violated at a point in time. For computing this set of mechanisms, the authors make the assumption that the target's clocks are synchronized and that the distribution of data to the target's agents is known among all the mechanisms. In our model, we do not rely on the assumption of such dynamically changing information to be known at the units. Moreover, the line of work by Kelbert et al. uses formal modeling for the targets as well as for the properties, but not for the enforcement mechanism itself and does not formally verify sound enforcement.

Mazaheri [Maz12] uses a previously presented version of our model [GMS12] for formalizing a security property and an enforcement mechanism for the scenario of the case study presented in Chapter 6. Based on the formal model, Mazaheri shows that the mechanism does not soundly enforce the security property and proposes a weaker variant of the property that can be soundly enforced. That is, Mazaheri's work provides further evidence that our model can be used for modeling units as well as for proving and disproving sound enforcement.

Formal models of enforcement mechanisms in CSP The process algebra CSP has traditionally been used for the verification of security protocols with regard to properties such as confidentiality and authenticity (e.g., [Sch96; LR97; DR10] and [Ros05, pp. 468–488]). However, we are not the first to use CSP for modeling enforcement mechanisms and their composition with models of targets.

Basin, Olderog, and Sevinç [BOS07] use CSP-OZ [Fis97], a variant of CSP combined with Object Z, for formalizing enforcement mechanisms and verifying properties of concrete instances. Technically, the formalized enforcement mechanisms are CSP process expressions like in our model. Basin et al. propose to specify security properties as process expressions and consider a given combination of target and enforcement mechanism to be sound, if the latter is a trace refinement of the former. For checking this notion of soundness, they use the FDR model checker. In our formal model, we formalize security properties as predicates on traces – without the indirection through processes. This allows us to specify security properties in a more declarative rather than operational fashion, as we exemplify with the Chinese Wall security property. Basin, Burri, and Karjoth [BBK12] propose an approach for synthesizing enforcement mechanisms in CSP for *separation of duties* [PP06, pp. 250–251] constraints expressed in a formal language called *separation of duty algebra* (*SoDA*). The authors show that the synthesized mechanism is correct in the sense that all it allows precisely those traces that satisfy the respective SoDA formula.

With both techniques, the enforcement mechanisms are combined with targets through parallel composition [BOS07; BBK12]. This allows the enforcement mechanism to let permitted event happen (by synchronizing on them), to indefinitely block security-violating events (by not synchronizing on them), and to insert sequences of events that are not shared between the target and the mechanism. When the events shared between the target and the mechanism subsume the whole alphabet of the target, then the indefinite blocking is equivalent to termination countermeasure. Suppressing selected events and replacing an intercepted event with a sequence of events is not supported by the model. This is due to the parallel composition without hiding, which effects the interception, decision-making, and the permission or blocking to take place instantaneously: Once an event is synchronized on, it immediately becomes effective in the sense of being visible to the environment. The only way, thus, to block an event is to not synchronize on it in the first place. First intercepting the event and then making a decision is not possible with this form of composition between the target and the mechanism. The proposed model does not particularly aim at distributed targets or decentralization and correspondingly do not foresee cooperation between enforcement mechanisms.

9.8. Application Scenarios

The focus of this thesis is on a generic enforcement mechanism for distributed targets, and we demonstrate in Chapters 6 to 8 that our generic tools, techniques, and models are indeed applicable in concrete application scenarios. However, we are not the first to apply generic as well as specialized approaches in these or similar application scenarios. We discuss related works on the application scenarios in the remainder of this section.

9.8.1. Controlled Re-sharing in DOSNs

We use the case study presented in this chapter to demonstrate that delegation can be used with CliSeAu in a modular fashion for effectively and efficiently enforcing security requirements in distributed targets. In the following, we discriminate our contributions made in the case study from other works on enforcing users' privacy is OSNs as well as on trust models.

Probably closest to our contribution of CReDiC is the work of Mazaheri [Maz12]. Mazaheri proposes a formal model in CSP for controlled re-sharing and an implementation of this model for Diaspora* using CliSeAu. In her work, Mazaheri uses a trust model based on scalar trust values in which a re-share path consist of the users who shared and re-shared the respective post. In the model, trust values are assigned to users and these trust values are multiplied for computing path trust. This is conceptually similar to our model. The main difference between her model and ours is that her model leaves out the notion of categories. In that regard, our model more closely captures how DOSNs like Diaspora* maintain relationships between users. Moreover, as part of our model, we provide a semantics of user's privacy policies, which we capture as compliance of re-sharing in Definition 6.6. Our definition of this notion is independent of a DOSN system model. This is in contrast to Mazaheri's work, in which compliance is expressed as a trace property of a partially underspecified DOSN. Our notion of compliance therefore achieves to be of lower conceptual complexity and, at the same time, supports a wider design space for the recency of users' privacy policies.

The mechanism provided by Mazaheri [Maz12] implements the trust model discussed above using CliSeAu. The logic proposed with Mazaheri [Maz12]'s local policy implementation for an intercepted re-share operation – first querying the re-share path, then the sensitivity value of the post, and finally the users' trust values along the re-share path – served as basis for the logic in CReDiC. The distinctive feature of CReDiC is its modular architecture, which separates the individual steps into micro-policies of lower conceptual complexity. Technically, our implementation of CReDiC re-uses the set of Diaspora* pointcuts and the event factory developed by Mazaheri but ports both to a newer version of Diaspora* and substantially extends them to a trust model based on categories.

Underlying our model of trust, presented in Section 6.4.1, are two main design decisions. Firstly, we model trust as scalar values ranging from 0 to 1, which can be found also elsewhere in the literature [BBK94; Gol05; KGG+06]. Alternatives found in the literature are qualitative models of trust between users based on one or more binary relations (e.g., [Fon11]) as well as quantitative models based on vectors of scalars (e.g., [Jøs98; HWS09]). We build our trust model on scalar trust values, such as to give users some means for quantifying their relationships and yet to take into account that specifying trust vectors might be a burden users might refrain to take. Secondly, our model utilizes a particular notion of trust concatenation and selects a single path – the re-share path – for capturing trust of an author in a re-sharing user. Alternative models for concatenation of scalar trust values have been proposed [Gol05; KGG+06]. These models, however, were proposed for selecting from all paths between the respective users. Further models for trust concatenation are based on trust vectors (e.g., [HWS09]). In defining compliance with users' privacy policies based on the re-share path and no further paths between users, we see two advantages: reduced complexity and context-dependence. By context-dependence we mean that we consider the trust along the list of users who have actually seen, read, and re-shared the post. We thus find that this choice of trust value reflects well the notion of decision trust, which by definition is associated with a situational context.

The focus in our case study is controlled re-sharing in decentralized OSNs. In the following, we discuss approaches proposed to control sharing and re-sharing for centralized OSNs. Fong et al. [BFS+12; Fon11; FAZ09] propose a model of OSNs, a ReBAC model, and a language for expressing ReBAC policies. In line of work, relationships between users are modeled as binary relations on users. The policy language is a variant of modal logic on the relationships of the OSN that allows specifying constraints on sharing, re-sharing, and subsequent distribution. The ReBAC mechanism for OSNs by Carminati et al. [CFP09] enforces privacy policies of authors that can specify the maximum length of re-share paths, the minimal concatenated trust value, or relationship categories. The actual access control is shared between the requesting user, who provides a proof of being authorized to access the resource, and the resource provider, who checks the proof. *Virtual Private Social Networks* [BCP+14; CHC13] are OSNs that build on centralized OSNs like Facebook, but achieve privacy of user information at the client-side via a browser extension. This line of work focuses on controlling the sharing of posts but not their subsequent distribution through re-sharing. *SCUTA* [KPP+11] is a usage control mechanism for centralized OSNs. The goal of this mechanism is not to control sharing or re-sharing in the OSN but to control, at the client-side, what operations users can perform on received posts – including, e.g., viewing, saving, and printing.

In the direction of decentralized OSNs, Carminati et al. [AC14] propose an access control mechanism for cloud-based OSNs. Like for [CFP09], the actual access control is performed jointly by requester and provider of a resource. The proposed mechanism supports re-sharing, but introduces centralized components, called KMS and RMS, into the DOSN for storing keys and access rules. While the mechanism makes use of encryption for users' keys and access rules transmitted to KMS and RMS, colluding KMS and RMS can reveal the plain data. Our approach aims at a completely decentralized architecture which has the benefit of avoiding a monopoly in the DOSN. *Safebook* [CMS09a; CMS09b] and *PeerSoN* [BSV+09; BKB14] are DOSNs aiming at protecting the privacy of user data by means of cryptography. Both DOSNs include a mechanism for controlling the sharing of posts. Controlling re-sharing and subsequent distribution of posts is not in their scope.

D-FOAF [KGG+06] is a distributed identity management system on top of the trust relationships between users in multiple OSNs. The system is proposed to be used for distributed authentication of users to services external to the OSNs as well as for the delegation of access rights at external services. For computing the trust between two users, D-FOAF gathers all information about the paths between requester and owner at one location. This is different from CReDiC in that CReDiC computes trust between two users based on a single path – the re-share path – as defined by the notion of path trust. Moreover, CReDiC computes path trust in a distributed fashion to keep users' privacy policies decentralized.

9.8.2. Chinese Wall Enforcement

Distributed enforcement We are not the first to study the enforcement of Chinese Wall Security Policies in distributed targets. In the following, we compare our chosen design and implementation of ChESt with other works proposing enforcement mechanisms for Chinese Walls.

Martinelli and Matteucci [MM08] propose a technique for synthesizing controllers for enforcing security properties and apply this technique for synthesizing centralized controllers for the Chinese Wall Security Policy. As pointed out by Martinelli et al., Chinese Walls cannot be enforced in a decentralized fashion without coordination, unless some restricting assumptions are made. Martinelli et al., for instance, synthesize controllers for a system in which some agents are forbidden to perform some operations from the start, thus performing an overly conservative (intransparent) enforcement.

Minsky [Min04] shows how Chinese Wall Security Policies can be enforced in a completely decentralized approach, i.e., without coordination among the distributed components of the enforcement mechanism. For this, Minsky imposes a restrictive assumption on the interactions of users with the system: It assumes that users only interact with the system via a single agent. This enables the distributed components of the mechanism proposed by Minsky to determine locally whether an access can soundly and transparently be granted. By utilizing cooperation, our ChESt does not require such an assumption and, thus, allows users to interact with the system through changing client nodes (e.g., a desktop and multiple mobile devices).

We published the core concept of ChESt, enforcing the Chinese Wall by establishing authoritative responsible units for all operations by the same user, in 2012 [GMS12]. This concept was adopted afterwards in further work [FS13]. Fairweather and Shin [FS13] propose a decentralized architecture for enforcing Chinese Walls in an infrastructure-as-a-service (IaaS) cloud and implement this architecture for the IaaS platform Eucalyptus. The decentralized architecture includes, decision-making and state-keeping components for domains of possibly multiple users. In our approach, we do not explicitly establish a notion of users' domains, but implicitly establish such domains through the sets of users whose operations are assigned to the same responsible unit. Fairweather et al. confirm the effectiveness of their approach by testing.

To the best of our knowledge, we were the first to propose sound and transparent enforcement of the Chinese Wall Security Policy in a distributed target and in a decentralized fashion.

Non-distributed enforcement The enforcement of Chinese Walls in non-distributed targets has been subject to several works since the work of Brewer and Nash [BN89]. Since the focus of this thesis is on distributed enforcement, we discuss here only two selected works. Fong [Fon04] shows that the Chinese Wall Security Policy as formulated by Brewer et al. belongs to the class of enforceable security properties. More precisely, Fong shows, that the policy is enforceable by shallow history automata, i.e., enforcement mechanisms that store in their state which events were granted in the past but not their order of occurrence. In our case study, the local policy we propose is also oblivious of the ordering of event occurrences. Our local policy is even oblivious of the actual files accessed by the events and rather records only the proprietor and its conflict of interest class. We compare the class of security properties enforceable with shallow history automata to our order-insensitive security properties in Section 9.5.

Hussein, Meredith, and Rosu [HMR12] show in an example how the JavaMOP tool can be used for enforcing a Chinese Wall. In their example, the authors take a technically slightly different approach than we do regarding the selection of security-relevant program operations. Hussein et al. intercept events that initiate and end code that is executed in the context of a particular user being active. That is, the enforcement mechanism in their example does not rely on the active user being accessible to the enforcement mechanism when an access operation is intercepted by the mechanism. In ChESt, we chose to access the user behind an access operation at the time the operation is intercepted, for two reasons: Firstly, the information about the user is available in both targets we considered and, hence, intercepting further operations is not required for effectively enforcing the Chinese Wall. Last but not least, reducing the number of operations intercepted by the enforcement mechanism improves the efficiency of the mechanism.

Chapter 10

Conclusion and Outlook

10.1. Conclusions

We presented a framework for enforcing security in distributed programs that integrates tools and techniques for the specification, enforcement, and verification of security policies for distributed programs. The framework builds on run-time enforcement and on generic enforcement mechanisms whose distributed units can cooperate with each other. With the framework, we lift the state of the art regarding generic enforcement mechanisms from non-distributed programs to distributed programs. The framework can be used by service providers for securing their distributed services against malicious users.

For the specification of security policies, the framework provides the policy language CoDSPL (Chapter 3). A CoDSPL policy specifies which units shall constitute the distributed enforcement mechanism, which operations are security-relevant to the individual units, and how decisions for security-relevant operations shall be made, possibly with cooperation. Using Java as a sub-language for deciding, CoDSPL enables policies to express a wide range of decision-making algorithms. In CoDSPL, the policy can specify also which units cooperate and when they cooperate using the expressive power of Java. This expressive power enables policies to avoid cooperation when it is not needed, to cooperate in a centralized fashion, or to cooperate in a decentralized fashion. With static delegation, we presented a technique for a simple decentralized cooperation that can be used for effectively enforcing order-insensitive partitionable security properties (Chapter 7). For specifying more complex forms of cooperation, we presented an extension of CoDSPL that allows for modular policy specifications through separation of concerns (Chapter 6).

For enforcing security requirements on distributed programs, the framework provides the tool CliSeAu (Chapter 5). CliSeAu consists of an implementation of a distributed enforcement mechanism and a tool that can apply the mechanism to a given distributed target implemented in Java or in Ruby. The mechanism is parametric and is instantiated by CliSeAu for enforcing a given CoDSPL policy. The modular design of CliSeAu follows principles of object-oriented design, and the implementation of CliSeAu has successfully been checked by static code analysis tools against incorrect and inefficient code. For applying a distributed enforcement mechanism to a distributed target, CliSeAu builds on the cross-lining technique (Chapter 4). The technique employs, on the one hand, program instrumentation for enabling the mechanism to use programming-language and application-level abstractions for controlling a distributed target at run-time. On the other hand, cross-lining establishes a part of the mechanism's units to run in parallel to the agents of the distributed target such that a unit can cooperate even when its agent is idle, terminated, or performing security-irrelevant actions.

For verifying distributed enforcement mechanisms that use cooperation, the framework provides a formal model of the framework's enforcement mechanism in the process algebra CSP (Chapter 8). The model captures the modular architecture of the mechanism, the parametric components of the mechanism, and the cooperation between the mechanism's units. The model is complemented by a notion of sound enforcement that allows one to verify whether a given CoDSPL policy indeed enforces a given safety property. Since our model makes the concurrent operation of agents and units as well as the cooperation among units explicit, the verification can particularly also capture potential race conditions between security-relevant actions of a target. In a non-trivial example, we demonstrated the instantiation of the model for a concrete policy as well as the formal verification of sound enforcement.

In two concrete case studies, we used the framework for developing the enforcement mechanisms CReDiC (Chapter 6) and ChESt (Chapter 7), which enforce security in two distributed programs. CReDiC enforces users' privacy policies in the re-sharing of messages in the decentralized online social network Diaspora*. ChESt enforces a Chinese Wall Security Policy in a distributed storage service. Both mechanisms are specified as CoDSPL policies and applied to the respective system with CliSeAu. Experiments with selected test cases suggest that the mechanisms effectively enforce the respective security requirements of the case study. Performance evaluations show that both of our mechanisms enforce the respective security requirements with moderate performance overhead of below 13ms even when the mechanisms use cooperation.

10.2. Outlook

The contributions presented in this thesis constitute a basis for further investigating the enforcement of security in distributed programs. In the following, we discuss several directions.

Broadening the range of stakeholders In this thesis, we focus on the requirements of service providers as the users of our proposed framework. With the focus on this stakeholder, we assume that the design and implementation of the distributed target has already completed when the mechanism is applied. However, CliSeAu could also be employed already by the developers of the distributed target as a tool for achieving separation between the functionality of the target and security. Such separation could simplify the design and implementation of the target and make the target more flexible with respect to changes in the security requirements after the initial deployment of the target. Alternatively, instead of using CliSeAu themselves for securing a target, the developers could also design and implement the target in a way that facilitates the subsequent use of CliSeAu, e.g., by service providers.

Broadening the range of adversaries In the design of our framework, the focus was on malicious users as the adversaries against whom security shall be enforced. For the sake of this focus, techniques against other adversaries that were already investigated in the literature about run-time enforcement, were not incorporated into the framework. For a security enforcement tool against a broader range of adversaries, possibilities for integrating such techniques into the framework could be investigated. For instance, when an adversary has (partial) control over the code of the target, then more fine-grained control over the instrumentation of the target might be necessary such that attempts of the target to disable or circumvent the mechanism could be defeated. Enforcement mechanisms like Polymer already pursue this approach by restricting, for instance, how the target can use Java's reflection API. Orthogonally, when an adversary has control over the computers on which some of the target's agents run, then the adversary could not only disable the CliSeAu-generated units at these computers but also, by manipulating the cooperation among units before encryption takes place, tamper with the enforcement of other units. Such an adversary could be an insider at the service provider [PHN07; PHG+10]. As technical means for countering such an adversary, an investigation of trusted computing technology could be promising.

Composing security policies In the design of our framework's enforcement mechanism, the separation between decisions and their realization in the form of countermeasures was inspired by Polymer [BLW09]. In Polymer, this separation serves the purpose of enabling the composition of policies through so-called combinators. These combinators select a decision from the suggested decisions of the combinators' sub-policies and only the selected decision is subsequently implemented. We expect that such combinators could be transferred from Polymer's non-distributed enforcement mechanism to the distributed enforcement mechanism of our framework in a naïve fashion with moderate implementation effort. However, such a naïve approach easily leads to inefficient policies. For instance, a conjunctive combinator of many sub-policies, of which each can perform cooperation for providing a suggested decision, might lead to a high communication overhead. An understanding of efficient policy combinators for distributed enforcement mechanisms would, thus, be desirable.

Domain-specific policy language support In CoDSPL, essential parts of a policy are specified in the general-purpose programming language Java. This provides the developer of a security policy for CliSeAu with a rich language for specifying, e.g., what data structures are used for state-keeping and how the delegation is implemented. Moreover, as Java is one of today's most popular programming languages, CoDSPL might be quickly comprehensible to many programmers. However, being a general-purpose language, Java might require more specification effort than necessary for security requirements in specific domains. In unpublished work, preliminary support for *d3log*, a distributed variant of datalog [JS01], has already been developed and evaluated [Sch13] for an earlier version of CliSeAu and showed that this language allows rather concise policy specifications. Variants of temporal logics, such as the language PT-DTL used by the DiAna mechanism, might be further promising candidates. We expect that support for additional policy languages could be realized even without modification of CoDSPL or CliSeAu in two alternative forms: through static translation into CoDSPL policies or through interpretation at run-time by an extension library for CoDSPL, in the shape of the extensions we propose in Chapters 6 and 7.

Bibliography

[AC14] Davide Alberto Albertini and Barbara Carminati. "Relationship-based Information Sharing in Cloud-based Decentralized Social Networks". In: *Proceedings of the Fourth Conference on Data and Application Security and Privacy.* Ed. by Elisa Bertino, Ravi S. Sandhu, and Jaehong Park. ACM, 2014, pp. 297–304.

[ADG09] Irem Aktug, Mads Dam, and Dilian Gurov. "Provably Correct Runtime Monitoring". In: *Journal of Logic and Algebraic Programming* 78(5), 2009: pp. 304–339.

[AHM+08] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25(5), 2008: pp. 22–29.

[AN08] Irem Aktug and Katsiaryna Naliuka. "ConSpec: A Formal Language for Policy Specification". In: *First International Workshop on Run Time Enforcement for Mobile and Distributed Systems.* Vol. 197(1). ENTCS. 2008, pp. 45–58.

[And72] James P. Anderson. *Computer Security Technology Planning Study.* Tech. rep. ESD-TR-73-51, Vol. II. Electronic Systems Division, Air Force Systems Command, 1972.

[Ano09] *AnomicFTPD v0.94.* 2009. URL: `http://anomic.de/AnomicFTPServer/` (visited on 07/2014).

[Aqu] *Aquarium. Aspect-Oriented Programming in Ruby.* URL: `http://aquarium.rubyforge.org/index.html` (visited on 03/2017).

[AS85] Bowen Alpern and Fred B. Schneider. "Defining Liveness". In: *Information Processing Letters* 21, 1985: pp. 181–185.

[AS87] Bowen Alpern and Fred B. Schneider. "Recognizing Safety and Liveness". In: *Distributed Computing* 2(3), 1987: pp. 117–126.

[BBG+60] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. "Report on the Algorithmic Language ALGOL 60". In: *Communications of the ACM* 3(5), 1960: pp. 299–314.

[BBK12] David A. Basin, Samuel J. Burri, and Günter Karjoth. "Dynamic Enforcement of Abstract Separation of Duty Constraints". In: *ACM Transactions on Information and System Security* 15(3), 2012: 13:1–13:30.

[BBK94] Thomas Beth, Malte Borcherding, and Birgit Klein. “Valuation of Trust in Open Networks”. In: *Proceedings of the Third European Symposium on Research in Computer Security.* LNCS 875. Springer, 1994, pp. 3–18.

[BCE⁺16] David Basin, Germano Caronni, Sarah Ereth, Matús Harvan, Felix Klaedtke, and Heiko Mantel. “Scalable Offline Monitoring of Temporal Specifications”. In: *Formal Methods in System Design* 48(1), 2016: pp. 1–34.

[BCP⁺14] Filipe Beato, Mauro Conti, Bart Preneel, and Dario Vettore. “VirtualFriendship: Hiding interactions on Online Social Networks”. In: *Proceedings of the Conference on Communications and Network Security.* IEEE, 2014, pp. 328–336.

[BD06] Nadia Belblidia and Mourad Debbabi. “Formalizing AspectJ Weaving for Static Pointcuts”. In: *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods.* IEEE Computer Society, 2006, pp. 50–59.

[BD15] Elaine Barker and Quynh Dang. “Recommendation for Key Management – Part 3: Application-Specific Key Management Guidance, Revision 1”. In: *NIST Special Publication* 800(57), 2015.

[BDE⁺08] Arnar Birgisson, Mohan Dhawan, Úlfar Erlingsson, Vinod Ganapathy, and Liviu Iftode. “Enforcing Authorization Policies Using Transactional Memory Introspection”. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security.* Alexandria, Virginia, USA: ACM, 2008, pp. 223–234.

[BFS⁺12] Glenn Bruns, Philip W. L. Fong, Ida Siahaan, and Michael Huth. “Relationship-Based Access Control: its Expression and Enforcement Through Hybrid Logic”. In: *Proceedings of the Second Conference on Data and Application Security and Privacy.* Ed. by Elisa Bertino and Ravi S. Sandhu. ACM, 2012, pp. 117–124.

[BGH⁺14] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. “AppGuard - Fine-Grained Policy Enforcement for Untrusted Android Applications”. In: *Revised Selected Papers of the 6th International Workshop on Data Privacy Management and Autonomous Spontaneous Security (2013).* Ed. by Joaquín García-Alfaro, Georgios V. Lioudakis, Nora Cuppens-Boulahia, Simon N. Foley, and William M. Fitzgerald. LNCS 8247. Springer, 2014, pp. 213–231.

[BGT09] Andreas Bauer, Rajeev Goré, and Alwen Tiu. “A First-Order Policy Language for History-Based Transaction Monitoring”. In: *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing.* Ed. by Martin Leucker and Carroll Morgan. LNCS 5684. Springer, 2009, pp. 96–111.

[BH12] Eric Bodden and Laurie J. Hendren. “The Clara Framework for Hybrid Typestate Analysis”. In: *International Journal on Software Tools for Technology Transfer* 14(3), 2012: pp. 307–326.

[BHK⁺12] David A. Basin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. "MONPOLY: Monitoring Usage-Control Policies". In: *Revised Selected Papers of the Second International Conference on Runtime Verification (2011).* Ed. by Sarfraz Khurshid and Koushik Sen. LNCS 7186. Springer, 2012, pp. 360–364.

[Bis03] Matt Bishop. *Computer Security: Art and Science.* Addison-Wesley, 2003.

[BJK⁺13] David A. Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zalinescu. "Enforceable Security Policies Revisited". In: *ACM Transactions on Information and System Security* 16(1), 2013: 3:1–3:26.

[BKB14] Oleksandr Bodriagov, Gunnar Kreitz, and Sonja Buchegger. "Access Control in Decentralized Online Social Networks: Applying a Policy-Hiding Cryptographic Scheme and Evaluating Its Performance". In: *2014 International Conference on Pervasive Computing and Communication Workshops.* IEEE, 2014, pp. 622–628.

[BLW09] Lujo Bauer, Jay Ligatti, and David Walker. "Composing Expressive Runtime Security Policies". In: *Transactions on Software Engineering and Methodology* 18(3), 2009: 9:1–9:43.

[BME⁺07] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Connallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications.* 3rd ed. Addison Wesley Pub Co Inc, 2007.

[BN89] David F.C. Brewer and Michael J. Nash. "The Chinese Wall Security Policy". In: *Proceedings of the IEEE Symposium on Security and Privacy.* 1989, pp. 206–214.

[BOS07] David A. Basin, Ernst-Rüdiger Olderog, and Paul E. Sevinç. "Specifying and Analyzing Security Automata Using CSP-OZ". In: *ACM Symposium on Information, Computer and Communications Security.* Ed. by Feng Bao and Steven Miller. ACM, 2007, pp. 70–81.

[Bou] Legion of the Bouncy Castle. *The Bouncy Castle Java Cryptography APIs.* URL: `http://www.bouncycastle.org/java.html` (visited on 07/2016).

[BSV⁺09] Sonja Buchegger, Doris Schiöberg, Le-Hung Vu, and Anwitaman Datta. "PeerSoN: P2P Social Networking: Early Experiences and Insights". In: *Proceedings of the Second EuroSys Workshop on Social Network Systems.* ACM, 2009, pp. 46–52.

[CFP09] Barbara Carminati, Elena Ferrari, and Andrea Perego. "Enforcing Access Control in Web-based Social Networks". In: *Transactions on Information and System Security* 13(1), 2009.

[CHC13] Mauro Conti, Arbnor Hasani, and Bruno Crispo. "Virtual Private Social Networks and a Facebook Implementation". In: *Transactions on the Web* 7(3), 2013: 14:1–14:31.

[Che] The Checkstyle Project. *checkstyle.* URL: `http://checkstyle.sourceforge.net/` (visited on 11/2016).

[CL85] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Transactions on Computer Systems* 3(1), 1985: pp. 63–75.

[CM04] Brian Chess and Gary McGraw. "Static Analysis for Security". In: *IEEE Security & Privacy* 2(6), 2004: pp. 76–79.

[CMJ+09] Feng Chen, Patrick O'Neil Meredith, Dongyun Jin, and Grigore Rosu. "Efficient Formalism-Independent Monitoring of Parametric Properties". In: *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 383–394.

[CMS09a] Leucio Antonio Cutillo, Refik Molva, and Thorsten Strufe. "Safebook: A privacy-preserving online social network leveraging on real-life trust". In: *Communications Magazine* 47(12), 2009: pp. 94–101.

[CMS09b] Leucio Antonio Cutillo, Refik Molva, and Thorsten Strufe. "Safebook: Feasibility of Transitive Cooperation for Privacy on a Decentralized Social Network". In: *10th International Symposium on a World of Wireless, Mobile and Multimedia Networks*. IEEE, 2009, pp. 1–6.

[CR07] Feng Chen and Grigore Roşu. "MOP: An Efficient and Generic Runtime Verification Framework". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, 2007, pp. 569–588.

[CS10] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *Journal of Computer Security* 18(6), 2010: pp. 1157–1210.

[DBV+10] Anwitaman Datta, Sonja Buchegger, Le-Hung Vu, Thorsten Strufe, and Krzysztof Rzadca. "Decentralized Online Social Networks". In: *Handbook of Social Network Technologies and Applications*. Ed. by Borko Furht. Springer, 2010, pp. 349–378.

[DDL+01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. "The Ponder Policy Specification Language". In: *Proceedings of the 2001 International Workshop on Policies for Distributed Systems and Networks*. Ed. by Morris Sloman, Jorge Lobo, and Emil Lupu. LNCS 1995. Springer, 2001, pp. 18–38.

[DLJ15] Maarten Decat, Bert Lagaisse, and Wouter Joosen. "Scalable and Secure Concurrent Evaluation of History-based Access Control Policies". In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 281–290.

[DR10] Tien Tuan Anh Dinh and Mark Ryan. "Verifying Security Property of Peer-to-Peer Systems Using CSP". In: *Proceedings of the 15th European Symposium on Research in Computer Security*. Ed. by Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou. LNCS 6345. Springer, 2010, pp. 319–339.

[EAC99] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. "History-Based Access Control for Mobile Code". In: *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*. Ed. by Jan Vitek and Christian Damsgaard Jensen. LNCS 1603. Springer, 1999, pp. 413–431.

[Eas05] Donald E. Eastlake 3rd. *Additional XML Security Uniform Resource Identifiers (URIs)*. RFC 4051. RFC Editor, 2005.

[EFT94] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical Logic*. 2nd ed. Undergraduate Texts in Mathematics. Springer, 1994.

[EGC$^+$10] William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. Ed. by Remzi H. Arpaci-Dusseau and Brad Chen. USENIX Association, 2010, pp. 393–407.

[Erl04] Úlfar Erlingsson. "The Inlined Reference Monitor Approach to Security Policy Enforcement". PhD thesis. Cornell University, 2004.

[ES00a] Úlfar Erlingsson and Fred B. Schneider. "IRM Enforcement of Java Stack Inspection". In: *Proceedings of the 2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000, pp. 246–255.

[ES00b] Úlfar Erlingsson and Fred B. Schneider. "SASI Enforcement of Security Policies: A Retrospective". In: *Proceedings of the 2nd New Security Paradigms Workshop*. Caledon Hills, Ontario, Canada: ACM, 2000, pp. 87–95.

[ET99] David Evans and Andrew Twyman. "Flexible Policy-Directed Code Safety". In: *1999 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1999, pp. 32–45.

[FAZ09] Philip W. L. Fong, Mohd M. Anwar, and Zhen Zhao. "A Privacy Preservation Model for Facebook-Style Social Network Systems". In: *Proceedings of the 14th European Symposium on Research in Computer Security*. LNCS 5789. Springer, 2009, pp. 303–320.

[Fer99] Jacques Ferber. *Multi-Agent Systems – An Introduction to Distributed Artificial Intelligence*. Addison-Wesley-Longman, 1999.

[Fis97] Clemens Fischer. "CSP-OZ: A Combination of Object-Z and CSP". In: *Formal Methods for Open Object-based Distributed Systems: Volume 2*. Ed. by Howard Bowman and John Derrick. Springer US, 1997, pp. 423–438.

[FMH$^+$03] Ronald Fagin, Yoram Moses, Joseph Y Halpern, and Moshe Y Vardi. *Reasoning About Knowledge*. MIT Press, 2003.

[Fon04] Philip W. L. Fong. "Access Control By Tracking Shallow Execution History". In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2004, pp. 43–55.

[Fon11] Philip W. L. Fong. "Relationship-Based Access Control: Protection Model and Policy Language". In: *Proceedings of the First Conference on Data and Application Security and Privacy*. Ed. by Ravi S. Sandhu and Elisa Bertino. ACM, 2011, pp. 191–202.

[Fow04] Martin Fowler. *Inversion of Control Containers and the Dependency Injection Pattern*. 2004. URL: `http://martinfowler.com/articles/injection.html` (visited on 05/2017).

[FS13] Ying Fairweather and Dongwan Shin. "Towards Multi-policy Support for IaaS Clouds to Secure Data Sharing". In: *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Ed. by Elisa Bertino, Dimitrios Georgakopoulos, Mudhakar Srivatsa, Surya Nepal, and Alessandro Vinciarelli. ICST, 2013, pp. 31–39.

[Gat07] Carrie E. Gates. "Access Control Requirements for Web 2.0 Security and Privacy". In: *Workshop on Web 2.0 Security & Privacy*. 2007.

[GBJ+08] Gurvan Le Guernic, Anindya Banerjee, Thomas P. Jensen, and David A. Schmidt. "Automata-Based Confidentiality Monitoring". In: *Proceedings of the 11th Asian Computing Science Conference 2006*. Ed. by Mitsu Okada and Ichiro Satoh. LNCS 4435. Springer, 2008, pp. 75–89.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[GHM+17a] Richard Gay, Jinwei Hu, Heiko Mantel, and Sogol Mazaheri. "Relationship-Based Access Control for Resharing in Decentralized Online Social Networks". In: *Post-Proceedings of the 10th International Symposium on Foundations & Practice of Security*. LNCS 10723. to appear. Springer, 2017.

[GHM+17b] Richard Gay, Jinwei Hu, Heiko Mantel, and Johannes Schickel. "Towards Accelerated Usage Control based on Access Correlations". In: *Proceedings of the 22nd Nordic Conference on Secure IT Systems*. Ed. by Helger Lipmaa, Aikaterini Mitrokotsa, and Raimundas Matulevičius. LNCS 10674. to appear. Springer, 2017.

[GHM14] Richard Gay, Jinwei Hu, and Heiko Mantel. "CliSeAu: Securing Distributed Java Programs by Cooperative Dynamic Enforcement". In: *Proceedings of the 10th International Conference on Information Systems Security (ICISS)*. Ed. by Atul Prakash and Rudrapatna K. Shyamasundar. LNCS 8880. Springer, 2014, pp. 378–398.

[GJS+14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Programming Language, Java SE 8 Edition*. Addison-Wesley, 2014.

[GM82] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1982, pp. 11–20.

[GMS12] Richard Gay, Heiko Mantel, and Barbara Sprick. "Service Automata". In: *Proceedings of the 8th International Workshop on Formal Aspects of Security and Trust (2011)*. Ed. by Gilles Barthe, Anupam Datta, and Sandro Etalle. LNCS 7140. Springer, 2012, pp. 148–163.

[GMS13] Richard Gay, Heiko Mantel, and Henning Sudbrock. "An Empirical Bandwidth Analysis of Interrupt-Related Covert Channels". In: *2nd International Workshop on Quantitative Aspects in Security Assurance*. 2013.

[GMS15] Richard Gay, Heiko Mantel, and Henning Sudbrock. "An Empirical Bandwidth Analysis of Interrupt-Related Covert Channels". In: *International Journal of Secure Software Engineering* 6(2), 2015. Ed. by Alessandro Aldini, Fabio Martinelli, and Neeraj Suri: pp. 1–22.

[Gol05] Jennifer Ann Golbeck. "Computing and Applying Trust in Web-based Social Networks". PhD thesis. University of Maryland, 2005.

[Gooa] Google. *Google Java Style Guide*. URL: `http://google.github.io/styleguide/javaguide.html` (visited on 10/2016).

[Goob] Google. *GSON library*. URL: `https://github.com/google/gson` (visited on 07/2016).

[GSS+] Daniel Grippi, Maxwell Salzberg, Raphael Sofaer, and Ilya Zhitomirskiy. *The Diaspora* Project*. URL: `http://diasporafoundation.org/` (visited on 02/2016).

[GT14] Hendra Gunadi and Alwen Tiu. "Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System". In: *Proceedings of the 19th International Symposium on Formal Methods*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. LNCS 8442. Springer, 2014, pp. 296–311.

[Ham16] Tobias Hamann. "CliSeAu for Android Applications: Design, Case Studies and Evaluation". Master Thesis. TU Darmstadt, 2016.

[HMR12] Soha Hussein, Patrick O'Neil Meredith, and Grigore Rosu. "Security-Policy Monitoring and Enforcement with JavaMOP". In: *Proceedings of the 2012 Workshop on Programming Languages and Analysis for Security*. Ed. by Sergio Maffeis and Tamara Rezk. ACM, 2012.

[HMS06a] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. "Certified Inlined Reference Monitoring on .NET". In: *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*. Ed. by Vugranam C. Sreedhar and Steve Zdancewic. ACM, 2006, pp. 7–16.

[HMS06b] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. "Computability Classes for Enforcement Mechanisms". In: *Transactions on Programming Languages and Systems* 28(1), 2006: pp. 175–205.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[HOW14] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. "Online Monitoring of Metric Temporal Logic". In: *Proceedings of the 5th International Conference on Runtime Verification.* Ed. by Borzoo Bonakdarpour and Scott A. Smolka. LNCS 8734. Springer, 2014, pp. 178–192.

[HP09] Matús Harvan and Alexander Pretschner. "State-Based Usage Control Enforcement with Data Flow Tracking Using System Call Interposition". In: *Third International Conference on Network and System Security.* Ed. by Yang Xiang, Javier Lopez, Haining Wang, and Wanlei Zhou. IEEE Computer Society, 2009, pp. 373–380.

[HPB+07] Manuel Hilty, Alexander Pretschner, David A. Basin, Christian Schaefer, and Thomas Walter. "A Policy Language for Distributed Usage Control". In: *12th European Symposium on Research in Computer Security.* 2007, pp. 531–546.

[HT09] Marieke Huisman and Alejandro Tamalet. "A Formal Connection between Security Automata and JML Annotations". In: *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering.* Ed. by Marsha Chechik and Martin Wirsing. LNCS 5503. Springer, 2009, pp. 340–354.

[HWS09] Chung-Wei Hang, Yonghong Wang, and Munindar P. Singh. "Operators for Propagating Trust and Their Evaluation in Social Networks". In: *Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems.* Vol. 2. IFAAMAS, 2009, pp. 1025–1032.

[JFM+04] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor". In: *Proceedings of the 13th USENIX Security Symposium.* Ed. by Matt Blaze. USENIX, 2004, pp. 179–194.

[JIB07] Audun Jøsang, Roslan Ismail, and Colin Boyd. "A Survey of Trust and Reputation Systems for Online Service Provision". In: *Decision Support Systems* 43(2), 2007: pp. 618–644.

[Jøs98] Audun Jøsang. "A Subjective Metric of Authentication". In: *Proceedings of the 5th European Symposium on Research in Computer Security.* LNCS 1485. Springer, 1998, pp. 329–344.

[JS01] Trevor Jim and Dan Suciu. "Dynamically Distributed Query Evaluation". In: *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* Ed. by Peter Buneman. ACM, 2001.

[Jür02] Jan Jürjens. "UMLsec: Extending UML for Secure Systems Development". In: *Proceedings of the 5th International Conference on The Unified Modeling Language.* Ed. by Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook. LNCS 2460. Springer, 2002, pp. 412–425.

[KAB07] Christian Kästner, Sven Apel, and Don S. Batory. "A Case Study Implementing Features Using AspectJ". In: *Proceedings of the 11th International Conference on Software Product Lines.* IEEE Computer Society, 2007, pp. 223–232.

[Kel16] Florian Manuel Kelbert. “Data Usage Control for Distributed Systems”. Dissertation. München: Technische Universität München, 2016.

[KGG+06] Sebastian Ryszard Kruk, Slawomir Grzonkowski, Adam Gzella, Tomasz Woroniecki, and Hee-Chul Choi. “D-FOAF: Distributed Identity Management with Access Rights Delegation”. In: *Proceedings of the First Asian Semantic Web Conference*. 2006, pp. 140–154.

[Khe] Ali Kheradmand. *JavaMOP4 Syntax*. URL: `http://fsl.cs.illinois.edu/index.php/JavaMOP4_Syntax` (visited on 08/2016).

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. “An Overview of AspectJ”. In: *Proceedings of the 15th ECOOP*. LNCS 2072. Springer, 2001, pp. 327–353.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-Oriented Programming”. In: *Proceedings of the 11th European Conference on Object-Oriented Programming*. Ed. by Mehmet Aksit and Satoshi Matsuoka. Springer-Verlag, 1997, pp. 220–242.

[KN06] Gerwin Klein and Tobias Nipkow. “A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler”. In: *Transactions on Programming Languages and Systems* 28(4), 2006: pp. 619–695.

[KP12] Florian Kelbert and Alexander Pretschner. “Towards a Policy Enforcement Infrastructure for Distributed Usage Control”. In: *17th ACM Symposium on Access Control Models and Technologies*. Ed. by Vijay Atluri, Jaideep Vaidya, Axel Kern, and Murat Kantarcioglu. ACM, 2012, pp. 119–122.

[KP13] Florian Kelbert and Alexander Pretschner. “Data Usage Control Enforcement in Distributed Systems”. In: *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*. Ed. by Elisa Bertino, Ravi S. Sandhu, Lujo Bauer, and Jaehong Park. ACM, 2013, pp. 71–82.

[KP14] Florian Kelbert and Alexander Pretschner. “Decentralized Distributed Data Usage Control”. In: *Proceedings of the 13th Conference on Cryptology and Network Security*. Ed. by Dimitris Gritzalis, Aggelos Kiayias, and Ioannis Askoxylakis. LNCS 8813. Springer, 2014, pp. 353–369.

[KP15] Florian Kelbert and Alexander Pretschner. “A Fully Decentralized Data Usage Control Enforcement Infrastructure”. In: *13th International Conference on Applied Cryptography and Network Security*. Ed. by Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis. LNCS 9092. Springer, 2015, pp. 409–430.

[KPP+11] Prachi Kumari, Alexander Pretschner, Jonas Peschla, and Jens-Michael Kuhn. “Distributed Data Usage Control for Web Applications: a Social Network Implementation”. In: *Proceedings of the First Conference on Data and Application Security and Privacy*. San Antonio, TX, USA: ACM, 2011, pp. 85–96.

[KV01] Orna Kupferman and Moshe Y. Vardi. “Synthesizing Distributed Systems”. In: *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society, 2001, pp. 389–398.

[LAB+92] Butler W. Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. “Authentication in Distributed Systems: Theory and Practice”. In: *ACM Transactions on Computer Systems* 10(4), 1992: pp. 265–310.

[Lam78] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21(7), 1978: pp. 558–565.

[LBD02] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. “SecureUML: A UML-Based Modeling Language for Model-Driven Security”. In: *Proceedings of the 5th International Conference on The Unified Modeling Language.* Ed. by Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook. LNCS 2460. Springer, 2002, pp. 426–441.

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. “Edit Automata: Enforcement Mechanisms for Run-time Security Policies”. In: *International Journal of Information Security* 4(1–2), 2005: pp. 2–16.

[LBW09] Jay Ligatti, Lujo Bauer, and David Walker. “Run-Time Enforcement of Non-safety Policies”. In: *Transactions on Information and System Security* 12(3), 2009: 19:1–19:41.

[Lie04] Karl J. Lieberherr. “Controlling the Complexity of Software Design”. In: *Proceedings of the 26th International Conference on Software Engineering.* Ed. by Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum. IEEE Computer Society, 2004, pp. 2–11.

[LL05] V. Benjamin Livshits and Monica S. Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis”. In: *Proceedings of the 14th USENIX Security Symposium.* Ed. by Patrick D. McDaniel. USENIX Association, 2005.

[LM10] Avinash Lakshman and Prashant Malik. “Cassandra: A Decentralized Structured Storage System”. In: *Operating Systems Review* 44(2), 2010: pp. 35–40.

[LMM12] Aliaksandr Lazouski, Fabio Martinelli, and Paolo Mori. “A Prototype for Enforcing Usage Control Policies Based on XACML”. In: *Proceedings of the 9th International Conference on Trust, Privacy and Security in Digital Business.* Ed. by Simone Fischer-Hübner, Sokratis K. Katsikas, and Gerald Quirchmayr. LNCS 7449. Springer, 2012, pp. 79–92.

[LOP16] Enrico Lovat, Martín Ochoa, and Alexander Pretschner. “Sound and Precise Cross-Layer Data Flow Tracking”. In: *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems.* Ed. by Juan Caballero, Eric Bodden, and Elias Athanasopoulos. LNCS 9639. Springer, 2016, pp. 38–55.

[Lov15] Enrico Lovat. “Cross-layer Data-centric Usage Control”. Dissertation. Technische Universität München, 2015.

[LR97] Gavin Lowe and A. William Roscoe. “Using CSP to Detect Errors in the TMN Protocol”. In: *IEEE Transactions on Software Engineering* 23(10), 1997: pp. 659–669.

[LYB+14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition.* Addison-Wesley, 2014.

[Man03] Heiko Mantel. “A Uniform Framework for the Formal Specification and Verification of Information Flow Security”. PhD thesis. Universität des Saarlandes, 2003.

[Maz12] Sogol Mazaheri. “Race Conditions in Distributed Enforcement at the Example of Online Social Networks”. Bachelor Thesis. TU Darmstadt, 2012.

[Min04] Naftaly H. Minsky. “A Decentralized Treatment of a Highly Distributed Chinese-Wall Policy”. In: *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks.* IEEE Computer Society, 2004, pp. 181–184.

[Min05] Naftaly Minsky. *Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism (An Introduction, and a Reference Manual).* Tech. rep. Rutgers University, 2005.

[Min91] Naftaly H. Minsky. “The Imposition of Protocols Over Open Distributed Systems”. In: *IEEE Transactions on Software Engineering* 17(2), 1991: pp. 183–195.

[MJG+12] Patrick O’Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. “An Overview of the MOP Runtime Verification Framework”. In: *International Journal on Software Tools for Technology Transfer* 14(3), 2012: pp. 249–289.

[MKD03] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. “A Compilation and Optimization Model for Aspect-Oriented Programs”. In: *Proceedings of the 12th International Conference on Compiler Construction.* Ed. by Görel Hedin. LNCS 2622. Springer, 2003, pp. 46–60.

[MM08] Fabio Martinelli and Ilaria Matteucci. “Synthesis of Local Controller Programs for Enforcing Global Security Properties”. In: *3rd International Conference on Availability, Reliability and Security.* IEEE Computer Society, 2008, pp. 1120–1127.

[MS03] Heiko Mantel and Andrei Sabelfeld. “A Unifying Approach to the Security of Distributed and Multi-Threaded Programs”. In: *Journal of Computer Security* 11(4), 2003: pp. 615–676.

[MS12] Christopher Mann and Artem Starostin. “A Framework for Static Detection of Privacy Leaks in Android Applications”. In: *Proceedings of the ACM Symposium on Applied Computing.* Ed. by Sascha Ossowski and Paola Lecca. ACM, 2012, pp. 1457–1462.

[MSK11] Huina Mao, Xin Shuai, and Apu Kapadia. “Loose Tweets: An Analysis of Privacy Leaks on Twitter”. In: *Proceedings of the 10th Annual ACM workshop on Privacy in the Electronic Society*. Ed. by Yan Chen and Jaideep Vaidya. ACM, 2011, pp. 1–12.

[MU00] Naftaly H. Minsky and Victoria Ungureanu. “Law-governed Interaction: a Coordination and Control Mechanism for Heterogeneous Distributed Systems”. In: *ACM Transactions on Software Engineering Methodology* 9(3), 2000: pp. 273–305.

[NSC$^+$08] Srijith Krishnan Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. “A Virtual Machine Based Information Flow Control System for Policy Enforcement”. In: *Electronic Notes in Theoretical Computer Science* 197(1), 2008: pp. 3–16.

[Oak14] Scott Oaks. *Java Performance - The Definitive Guide: Getting the Most Out of Your Code*. O’Reilly, 2014.

[OAS13] OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0*. Candidate OASIS Standard 01. 2013.

[OBM10] Machigar Ongtang, Kevin R. B. Butler, and Patrick Drew McDaniel. “Porscha: Policy Oriented Secure Content Handling in Android”. In: *26th Annual Computer Security Applications Conference*. Ed. by Carrie Gates, Michael Franz, and John P. McDermott. ACM, 2010, pp. 221–230.

[Ora] Oracle. *Class Properties – load*. URL: `http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html#load(java.io.InputStream)` (visited on 10/2015).

[PBS14] Thomas F. J.-M. Pasquier, Jean Bacon, and Brian Shand. “FlowR: Aspect Oriented Programming for Information Flow Control in Ruby”. In: *13th International Conference on Modularity*. Ed. by Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld. ACM, 2014, pp. 37–48.

[PFS14] Thomas Paul, Antonino Famulari, and Thorsten Strufe. “A Survey on Decentralized Online Social Networks”. In: *Computer Networks* 75, 2014: pp. 437–452.

[PHB06] Alexander Pretschner, Manuel Hilty, and David A. Basin. “Distributed Usage Control”. In: *Communications of the ACM* 49(9), 2006: pp. 39–44.

[PHG$^+$10] Christian W. Probst, Jeffrey Hunker, Dieter Gollmann, and Matt Bishop. “Aspects of Insider Threats”. In: *Insider Threats in Cyber Security*. Ed. by Christian W. Probst, Jeffrey Hunker, Dieter Gollmann, and Matt Bishop. Vol. 49. Advances in Information Security. Springer, 2010, pp. 1–15.

[PHN07] Christian W. Probst, René Rydhof Hansen, and Flemming Nielson. “Where Can an Insider Attack?” In: *Revised Selected Papers of the Fourth International Workshop on Formal Aspects in Security and Trust (2006)*. Ed. by Theodosis Dimitrakos, Fabio Martinelli, Peter Y. A. Ryan, and Steve A. Schneider. LNCS 4691. Springer, 2007, pp. 127–142.

[PLB11] Alexander Pretschner, Enrico Lovat, and Matthias Büchler. "Representation-Independent Data Usage Control". In: *6th International Workshop on Data Privacy Management and Autonomous Spontaneus Security*. Ed. by Joaquín García-Alfaro, Guillermo Navarro-Arribas, Nora Cuppens-Boulahia, and Sabrina De Capitani di Vimercati. LNCS 8813. Springer, 2011, pp. 122–140.

[PMD] PMD. *PMD*. URL: `http://pmd.github.io/` (visited on 05/2016).

[PP06] Charles P. Pfleeger and Shari Lawrence Pfleeger. *Security in Computing*. 4th ed. Prentice Hall, 2006.

[PR90] Amir Pnueli and Roni Rosner. "Distributed Reactive Systems Are Hard to Synthesize". In: *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1990, pp. 746–757.

[PS04] Jaehong Park and Ravi S. Sandhu. "The UCON$_{\mathrm{ABC}}$ Usage Control Model". In: *ACM Transactions on Information and System Security* 7(1), 2004: pp. 128–174.

[RBG$^+$15] Vineet Rajani, Abhishek Bichhawat, Deepak Garg, and Christian Hammer. "Information Flow Control for Event Handling and the DOM in Web Browsers". In: *IEEE 28th Computer Security Foundations Symposium*. Ed. by Cédric Fournet, Michael W. Hicks, and Luca Viganò. IEEE, 2015, pp. 366–379.

[RHN$^+$13] Gregor Richards, Christian Hammer, Francesco Zappa Nardelli, Suresh Jagannathan, and Jan Vitek. "Flexible Access Control for JavaScript". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes. ACM, 2013, pp. 305–322.

[Ros05] A. William Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, Inc., 2005.

[Rus92] John M. Rushby. *Noninterference, Transitivity, and Channel-Control Security Policies*. Tech. rep. CSL-92-02. SRI International, 1992.

[Sal74] Jerome H. Saltzer. "Protection and the Control of Information Sharing in Multics". In: *Communications of the ACM* 17(7), 1974: pp. 388–402.

[Sch00] Fred B. Schneider. "Enforceable Security Policies". In: *Transactions on Information and System Security* 3(1), 2000: pp. 30–50.

[Sch04] Bruce Schneier. *Secrets & Lies – Digital Security in a Networked World. With new information about post-9/11 security*. Wiley, 2004.

[Sch13] Dominic Scheurer. "Enforcing Datalog Policies with Service Automata on Distributed Version Control Systems". Bachelor Thesis. TU Darmstadt, 2013.

[Sch16] Johannes Schickel. "Using File-Correlation to Accelerate Decision-Making in a Decentralized Cooperative Security Enforcement". Master Thesis. TU Darmstadt, 2016.

[Sch96] Steve Schneider. "Security Properties and CSP". In: *Proceedings of the 1996 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1996, pp. 174–187.

[Sea05] Robert C Seacord. *Secure Coding in C and C++*. Pearson Education, 2005.

[Sim10] *simple-ftpd*. 2010. URL: `https://github.com/rath/simple-ftpd` (visited on 07/2014).

[SO05] Guttorm Sindre and Andreas L. Opdahl. "Eliciting Security Requirements with Misuse Cases". In: *Requirements Engineering* 10(1), 2005: pp. 34–44.

[Som16] Ian Sommerville. *Software Engineering*. 10th ed. Pearson Education, 2016.

[SVA+04] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Roşu. "Efficient Decentralized Monitoring of Safety in Distributed Systems". In: *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 418–427.

[Tea05] The AspectJ Team. *The AspectJ™ 5 Development Kit Developer's Notebook, Appendix A: A Grammar for the AspectJ 5 Language*. 2005. URL: `http://www.eclipse.org/aspectj/doc/next/adk15notebook/grammar.html` (visited on 04/2016).

[THG99] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. "Strand Spaces: Proving Security Protocols Correct". In: *Journal of Computer Security* 7(1), 1999: pp. 191–230.

[Tie15] Moritz Tiedje. "Design and Evaluation of Profiling Methods for the Distributed Enforcement Mechanism CliSeAu". Bachelor Thesis. TU Darmstadt, 2015.

[TS14] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems – Principles and Paradigms*. 2nd ed. Pearson Education, 2014.

[Uni02] United States Code. *Sarbanes-Oxley Act of 2002, PL 107-204, 116 Stat 745*. Codified in Sections 11, 15, 18, 28, and 29 USC. 2002.

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis". In: *Journal of Computer Security* 4(2/3), 1996: pp. 167–188.

[War67] Willis H. Ware. "Security and Privacy in Computer Systems". In: *Proceedings of the American Federation of Information Processing Societies (AFIPS) '67 Spring Joint Computer Conference*. Vol. 30. AFIPS Conference Proceedings. AFIPS, 1967, pp. 279–282.

[WCS+02] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. "Linux Security Modules: General Security Support for the Linux Kernel". In: *Proceedings of the 11th USENIX Security Symposium*. Ed. by Dan Boneh. USENIX, 2002, pp. 17–31.

[Wir77] Niklaus Wirth. "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?" In: *Communications of the ACM* 20(11), 1977: pp. 822–823.

[WKD04] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. "A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming". In: *ACM Transactions on Programming Languages and Systems* 26(5), 2004: pp. 890–910.

[WLA$^+$93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation". In: *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*. Ed. by Andrew P. Black and Barbara Liskov. ACM, 1993, pp. 203–216.

[YLL$^+$09] Ching-man Au Yeung, Ilaria Liccardi, Kanghao Lu, Oshani Seneviratne, and Tim Berners-Lee. "Decentralization: The Future of Online Social Networking". In: *W3C Workshop on the Future of Social Networking*. 2009.

[ZSS08] Xinwen Zhang, Jean-Pierre Seifert, and Ravi Sandhu. "Security Enforcement Model for Distributed Usage Control". In: *Proceedings of the 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*. IEEE Computer Society, 2008, pp. 10–18.

[ZTE13] Bin Zeng, Gang Tan, and Úlfar Erlingsson. "Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors". In: *22th USENIX Security Symposium*. 2013.

Appendix

A Proofs

A.1. Cooperative Enforcement with Authoritative Delegates

In this appendix, we present formal proofs for Chapter 7 that are omitted in the main part of the thesis.

Theorem 7.1. *Let E be a set of events and let P be a security property over E.*

(a) P is partitionable into $(E_i)_{i\in\{\top\}}$ where $E_\top = E$.

(b) Let $\mathcal{E} = (E_i)_{i\in I}$ and $\mathcal{E}' = (E'_j)_{j\in\mathcal{J}}$ be partitions of E such that $\mathcal{E}$ is a refinement of $\mathcal{E}'$ and P is partitionable into $\mathcal{E}$. Then P is also partitionable into $\mathcal{E}'$. ◊

Proof. Let E be an arbitrary but fixed set of events and let P be an arbitrary but fixed security property over E. In the following, we show the individual parts of the theorem.

On 7.1 (a): Let $E_\top = E$. We will show that P is partitionable into $(E_i)_{i\in\{\top\}}$. For this, it suffices to show that there exists a family of security properties $(P_i)_{i\in\{\top\}}$ such that

$$P = \{t \in E^* \mid \forall i \in \{\top\} : ((t \upharpoonright E_i) \in P_i)\}$$

holds. To show this constructively, let $P_\top = P$. Then we obtain

$$\begin{aligned}\{t \in E^* \mid \forall i \in \{\top\} : ((t \upharpoonright E_i) \in P_i)\} =& \{t \in E^* \mid (t \upharpoonright E_\top) \in P_\top\}\\ =& \{t \in E^* \mid (t \upharpoonright E) \in P\}\\ =& \{t \in E^* \mid t \in P\} = P\end{aligned}$$

as it was to be shown.

On 7.1 (b): Let $\mathcal{E} = (E_i)_{i\in I}$ and $\mathcal{E}' = (E'_j)_{j\in\mathcal{J}}$ be partitions of E such that $\mathcal{E}$ is a refinement of $\mathcal{E}'$ and P is partitionable into $\mathcal{E}$.

We will show that P is also partitionable into $\mathcal{E}'$. We show this constructively, by defining the family $(P'_j)_{j\in\mathcal{J}}$ of security properties as

$$P'_j = \{t \in (E'_j)^* \mid \forall i \in I : ((f(i) = j) \rightarrow (t \upharpoonright E_i) \in P)\},$$

where $f : I \to \mathcal{J}$ is defined by $f(i) = j$ if $E_i \subseteq E'_j$. The function f is well-defined (because $\mathcal{E}'$ is a partition) and total (because $\mathcal{E}$ is a refinement of $\mathcal{E}'$).

To complete the proof that P is partitionable into $\mathcal{E}'$, For this, it suffices to show that

$$P = \{t \in E^* \mid \forall j \in \mathcal{J} : ((t \upharpoonright E'_j) \in P'_j)\}$$

holds. We provide the evidence through the following sequence of equations:

$$\begin{aligned}
&\{t \in E^* \mid \forall j \in \mathcal{J} : ((t \upharpoonright E'_j) \in P'_j)\}\\
=\ &\text{by definition of } P'_j\\
&\left\{ t \in E^* \middle| \forall j \in \mathcal{J} : (t \upharpoonright E'_j) \in \left\{ t \in (E'_j)^* \middle| \begin{array}{r} \forall i \in I : ((f(i) = j) \\ \Rightarrow (t \upharpoonright E_i) \in P) \end{array} \right\} \right\}\\
=\ &\text{by simplification of the nested intensional set}\\
&\{t \in E^* \mid \forall j \in \mathcal{J} : \forall i \in I : ((f(i) = j) \Rightarrow ((t \upharpoonright E'_j) \upharpoonright E_i) \in P)\}\\
=\ &\text{utilizing that by definition of } f, f(i) = j \text{ implies } E_i \subseteq E'_{f(i)}\\
&\{t \in E^* \mid \forall j \in \mathcal{J} : \forall i \in I : ((f(i) = j) \Rightarrow (t \upharpoonright E_i) \in P)\}\\
=\ &\text{by the fact that } f \text{ is a total function}\\
&\{t \in E^* \mid \forall i \in I : ((t \upharpoonright E_i) \in P)\}\\
=\ &\text{by the prerequisite that } P \text{ is partitionable into } \mathcal{E}\\
&P
\end{aligned}$$

as it was to be shown. □

A.2. A Formal Cooperation Model for CliSeAu

In this appendix, we present formal proofs for Chapter 8 that are omitted in the main part of the thesis. The proof for the soundness theorem can be found at the end of this appendix on page 200. To increase the readability of the proof, we show major steps of the proof in separate lemmas:

- Definition A.1 on the facing page introduces several abbreviations used in the proofs.
- Definition A.2 on the next page specifies a renaming on events and an encapsulated target model without hiding. This removal of the hiding simplifies the reasoning about the events that are hidden in ET_{ChWgp}.
- Lemma A.1 on page 192 states that the renaming is injective, a property we exploit in the proofs whenever we make use of Lemma 8.1. Lemma A.2 on page 193 states that the hiding in the definitions of the EC models and our local policy model can be moved outside of the encapsulated target model, if local channels are renamed to be unique. The prerequisites of both lemmas are covered in the proof of Theorem 8.2.
- Lemma A.3 on page 194 states that certain events cannot occur in traces of the encapsulated target model: decisions are not sent on channel ddec, delegation requests are not sent on channel rdec, delegation requests and decisions are not

forwarded, remote decisions are not approved for locally decidable events, and local enforcement decisions are not sent to an enforcer for events that have a remote responsible node.

- Lemma A.4 on page 196 states that the replacing enforcer performs an effectable event only after it has received a corresponding decision.
- Lemma A.5 on page 197 states that every performed access event must have ultimately been permitted by its responsible unit.
- Lemma A.6 on page 199 states that no decision-making component permits two conflicting events in a single trace.

In the remainder of this appendix, we make use of some short-hand notation to improve the legibility of the lemmas and proofs.

Definition A.1. For each $i \in SP$, we abbreviate

(a) $\mathsf{COR}_{(SP,i,AE_i,ChWD_i,ChWDR)}$ by COR_i,
(b) $H_{(SP,i,AE_i,ChWD_i,ChWDR)}$ by H_i,
(c) $\mathsf{INT}_{(\alpha(SP_i),AE_i)}$ by INT_i,
(d) $\mathsf{REPLACE}_{AE_i,AE_i}$ by $\mathsf{REPLACE}_i$,
(e) $\mathsf{REPL}_{AE_i,AE_i}(t)$ by $\mathsf{REPL}_i(t)$,
(f) $EC_{(SP,i,AE_i,ChWD_i,ChWDR)}(SP_i, \mathsf{ChWLP}_i, \mathsf{REPLACE}_i)$ by EC_i, and
(g) the set of all denying access events by $DENY = \{e \in AE \mid deny(e) = e\}$. $\lozenge$

Definition A.2.

(a) For each identifier $i \in SP$ of a unit in our policy model, let $\rho_i : RE_i \to (RE_i \cup \{c_i.m \mid c.m \in H_i \cup H_i^{\text{pol}}\})$ be the *renaming function* on

$$\begin{aligned} RE_i =& \alpha((SP_i \parallel \mathsf{INT}_i) \setminus AE_i) \cup \alpha(\mathsf{COR}_i) \\ & \cup \alpha(\mathsf{REPLACE}_i) \cup \alpha(\mathsf{DEL}_i) \cup \alpha(\mathsf{DEC}_i(\emptyset)) \cup \alpha(\mathsf{SRP}_i) \end{aligned}$$

defined by

$$\rho_i(e) = \begin{cases} c_i.m & \text{if } e \in H_i \cup H_i^{\text{pol}} \text{ with } e = c.m, \\ e & \text{otherwise.} \end{cases}$$

We lift the renaming functions from events to three further entities:

- Firstly, we abuse notation and lift the renaming functions from events to sets of events.
- Secondly, we lift the renaming functions from events to process expressions as in [Hoa85, Section 2.6], by applying the renaming to all events occurring in the respective process expression. In lifting a renaming function ρ_i to process expressions, in addition to renaming all events in the process expression, we furthermore translate each occurring process name NAME to a process name uniquely identified by NAME and i.
- Based on the previous liftings of the renaming functions, we finally lift the renaming functions also to sets of process equations by applying the renaming functions to the respective left-hand side, right-hand side, and alphabet of each process equation in the set.

(b) The set of all *renamed hidden events* is defined as $H_{EI} = \bigcup_{i \in SP} \rho_i(H_i \cup H_i^{\mathrm{pol}})$.

(c) The *encapsulated target with internals* is the process expression EI, defined by

$$EI = \mathop{\Vert}_{i \in SP} \left(\begin{array}{c} \rho_i((SP_i \parallel \mathsf{INT}_i) \setminus AE_i) \parallel \rho_i(\mathsf{COR}_i) \\ \parallel \rho_i(\mathsf{REPLACE}_i) \parallel \rho_i(\mathsf{DEL}_i) \parallel \rho_i(\mathsf{DEC}_i(\emptyset)) \parallel \rho_i(\mathsf{SRP}_i) \end{array} \right)$$

along with the set $\mathcal{EQ}_{EI} = \bigcup_{i \in SP} \rho_i(\mathcal{EQ}_{ChW})$ of process equations. Note that, from how we lifted the renaming functions to process equations, the set union in the definition of $\mathcal{EQ}_{EI}$ does not introduce contradictory process equations. In the remainder of this proof, unless explicitly mentioned otherwise, we implicitly use $\mathcal{EQ}_{EI}$ as the set of process equations that determines the semantics of process expressions. $\diamondsuit$

Lemma A.1. *For each identifier $i \in SP$, if $\alpha(SP_i) \cap \rho_i(H_i \cup H_i^{\mathrm{pol}}) = \emptyset$ holds, then ρ_i is an injective function.* $\diamondsuit$

Proof. Let $i \in SP$ be arbitrary but fixed. We show that ρ_i is injective, i.e., for each $e, e' \in RE_i$, if $\rho_i(e) = \rho_i(e')$, then $e = e'$.

Firstly, by the definitions of RE_i, H_i (on Page 191 and, as H_{ci}, on Page 143), and H_i^{pol} (on Page 151), it holds that

$$RE_i = H_i \cup H_i^{\mathrm{pol}} \cup \{\mathsf{link}_{i,k}.dr, \mathsf{link}_{k,i}.dr \mid dr \in ChWDR \wedge k \in SP \setminus \{i\}\} \cup \alpha(SP_i). \quad \text{(A.1)}$$

Now let $e, e' \in RE_i$ be arbitrary but fixed such that $\rho_i(e) = \rho_i(e')$ holds. In the following, to complete the proof, we show that $e = e'$ holds. With the following three cases, we perform a complete case distinction:

1. $e, e' \in H_i \cup H_i^{\mathrm{pol}}$:

   By the definitions of H_i and H_i^{pol}, we have that $e = c.m$ and $e' = c'.m'$ for channels c and c' and messages m and m'. By the prerequisite $\rho_i(e) = \rho_i(e')$, we get $c_i.m = c'_i.m'$ and, hence, $c_i = c'_i$ and $m = m'$, which, in turn, shows that $e = e'$ holds.

2. $e, e' \notin H_i \cup H_i^{\mathrm{pol}}$:

   In this case, by definition of ρ_i, we have $\rho_i(e) = e$ and $\rho_i(e') = e'$. By the prerequisite $\rho_i(e) = \rho_i(e')$ we, thus, immediately get $e = e'$.

3. $e \in H_i \cup H_i^{\mathrm{pol}}$ and $e' \notin H_i \cup H_i^{\mathrm{pol}}$ (inverse case analogous):

   In this case, by definition of ρ_i, we have $\rho_i(e') = e'$ and, thus, $\rho_i(e) = e'$. By Equation (A.1), we distinguish two cases for e':

   - $e' \in \{\mathsf{link}_{i,k}.dr, \mathsf{link}_{k,i}.dr \mid dr \in ChWDR \wedge k \in SP \setminus \{i\}\}$: This case cannot satisfy the condition $\rho_i(e) = e'$ because by the definitions of H_i and H_i^{pol}, no channel in $H_i \cup H_i^{\mathrm{pol}}$ can be renamed to then coincide with link.
   - $e' \in \alpha(SP_i)$: This case cannot satisfy the condition $\rho_i(e) = e'$ because of the prerequisite $\alpha(SP_i) \cap \rho_i(H_i \cup H_i^{\mathrm{pol}}) = \emptyset$.

Hence, for all possible cases of $\rho_i(e) = \rho_i(e')$ we have shown that $e = e'$ holds. Thus, ρ_i is injective. $\square$

Lemma A.2. *If $\alpha(SP_i) \cap (H_{EI} \cup H_i^{\text{pol}}) = \emptyset$ holds for each identifier $i \in SP$, then $EI \setminus H_{EI}\ {}_{\mathcal{EQ}_{EI}}{\equiv}_{\mathcal{EQ}_{ChW}}\ ET_{ChWgp}$.* $\Diamond$

Proof. We first make two observations that we make use of afterwards

1. For each identifier $j \in SP$, the definition of each process expression
$$P \in \{\mathsf{COR}_i, \mathsf{DEL}_i, \mathsf{DEC}_i(\emptyset), \mathsf{SRP}_i \mid i \in SP \setminus \{j\}\}$$
under set $\mathcal{EQ}_{ChW}$ of process equations ensures $\alpha(P) \cap \rho_j(H_j \cup H_j^{\text{pol}}) = \emptyset$.
2. For each $j \in SP$ and each $Q \in \{\mathsf{REPLACE}_i, (SP_i \parallel \mathsf{INT}_i) \setminus AE_i \mid i \in SP \setminus \{j\}\}$, the precondition $\alpha(SP_i) \cap (H_{EI} \cup H_i^{\text{pol}}) = \emptyset$ together with $AE_i \subseteq \alpha(SP_i)$ from Theorem 8.1 and the definitions of the process names $\mathsf{REPLACE}_i$ and INT_i under set $\mathcal{EQ}_{ChW}$ of process equations implies $\alpha(Q) \cap \rho_j(H_j \cup H_j^{\text{pol}}) = \emptyset$.

We denote the above two observations as (†) in the following sequence of equivalence transformations that shows the claim made by the lemma.

$$EI \setminus H_{EI}$$

$\equiv$ by Definition A.2 (c) and Lemma 8.1 (a)

$$\left(\mathop{\Big\|}_{i \in SP} \rho_i \left(\begin{array}{c} (SP_i \parallel \mathsf{INT}_i) \setminus AE_i \parallel \mathsf{COR}_i \\ \parallel \mathsf{REPLACE}_i \parallel \mathsf{DEL}_i \parallel \mathsf{DEC}_i(\emptyset) \parallel \mathsf{SRP}_i \end{array} \right) \right) \setminus H_{EI}$$

$\equiv$ by Definition A.2 (b), Lemma 8.1 (c), and observations (†)

$$\mathop{\Big\|}_{i \in SP} \left(\rho_i \left(\begin{array}{c} (SP_i \parallel \mathsf{INT}_i) \setminus AE_i \parallel \mathsf{COR}_i \\ \parallel \mathsf{REPLACE}_i \parallel \mathsf{DEL}_i \parallel \mathsf{DEC}_i(\emptyset) \parallel \mathsf{SRP}_i \end{array} \right) \setminus \rho_i(H_i \cup H_i^{\text{pol}}) \right)$$

$\equiv$ by Lemma 8.1 (d)

$$\mathop{\Big\|}_{i \in SP} \rho_i \left(\left(\begin{array}{c} (SP_i \parallel \mathsf{INT}_i) \setminus AE_i \parallel \mathsf{COR}_i \\ \parallel \mathsf{REPLACE}_i \parallel \mathsf{DEL}_i \parallel \mathsf{DEC}_i(\emptyset) \parallel \mathsf{SRP}_i \end{array} \right) \setminus (H_i \cup H_i^{\text{pol}}) \right)$$

${}_{\mathcal{EQ}_{EI}}{\equiv}_{\mathcal{EQ}_{ChW}}$ by Definition A.2 (a), which gives $\rho_i(e) = e$ for all $e \notin H_i \cup H_i^{\text{pol}}$

$$\mathop{\Big\|}_{i \in SP} \left(\left(\begin{array}{c} (SP_i \parallel \mathsf{INT}_i) \setminus AE_i \parallel \mathsf{COR}_i \\ \parallel \mathsf{REPLACE}_i \parallel \mathsf{DEL}_i \parallel \mathsf{DEC}_i(\emptyset) \parallel \mathsf{SRP}_i \end{array} \right) \setminus (H_i \cup H_i^{\text{pol}}) \right)$$

$\equiv$ by Lemma 8.1 (b); by precondition $\alpha(SP_i) \cap H_i^{\text{pol}} = \emptyset$;
by the fact that for each $i \in SP$ and $P \in \{\mathsf{INT}_i, \mathsf{REPLACE}_i, \mathsf{COR}_i\}$,
$\alpha(P) \cap H_i^{\text{pol}} = \emptyset$ holds according to the process equation for P;
and by definition of ChWLP_i

$$\mathop{\Big\|}_{i \in SP} \left(\left(\begin{array}{c} (SP_i \parallel \mathsf{INT}_i) \setminus AE_i \parallel \mathsf{COR}_i \\ \parallel \mathsf{REPLACE}_i \parallel \mathsf{ChWLP}_i \end{array} \right) \setminus H_i \right)$$

$\equiv$ by Definitions 8.15 and A.1 (f)

$$\mathop{\Big\|}_{i \in SP} \mathsf{EC}_i$$

$\equiv$ by Definition 8.19

$$ET_{ChWgp}$$ $\square$

Remark A.1. In the following, we repeatedly make use of the following patterns of reasoning.

- Let $tr \in traces(EI)$ be a trace, P and Q be process expressions such that $EI \equiv P \parallel Q$ holds, and $e \in \alpha(Q)$ be an event. Then from $e \lhd tr$ we can follow that $e \lhd tr \upharpoonright \alpha(Q) \wedge tr \upharpoonright \alpha(Q) \in traces(Q)$. We indicate this reasoning by $\overset{(*)}{\Longrightarrow}$.
- Let Q be a process expression, $e \in \alpha(Q)$ be an event that never occurs at the beginning of a trace of Q, tr_Q be a sequence, and $E \subseteq \alpha(Q)$ be the set of immediate predecessors of e in process expression Q. Then from $e \lhd tr_Q \wedge tr_Q \in traces(Q)$ we can conclude that $\bigvee_{e' \in E}(e' \lhd tr_Q)$ holds. We use $\overset{(**)}{\Longrightarrow}$ to indicate this reasoning.
- We often combine the above steps in the form $e \lhd tr \overset{(*)}{\Longrightarrow} e \lhd tr_Q \wedge tr_Q \in traces(Q) \overset{(**)}{\Longrightarrow} \bigvee_{e' \in E}(e' \lhd tr_Q) \Longrightarrow \bigvee_{e' \in E}(e' \lhd tr)$ for $tr_Q = tr \upharpoonright \alpha(Q)$. The last implication trivially holds. In the remainder of this appendix, we abbreviate such chains of reasoning by $e \lhd tr \overset{Q}{\Rightarrow} \bigvee_{e' \in E}(e' \lhd tr)$. ◊

Lemma A.3. *Let $AE_i^{ld} = \{e \in AE_i \mid sp(e) = resp(e)\}$ be the set of "locally decidable" access events at service provider $i \in SP$. Then for all traces $tr \in EI$, service providers $i, j, k \in SP$ with $j \neq i$, access events $e \in AE$, and decisions $ed \in ChWD$, the following holds:*

(a) $\mathsf{ddec}_i.(j, (k, ed)) \ntriangleleft tr$
(b) $\mathsf{rdec}_i.(j, (k, e)) \ntriangleleft tr$
(c) $\mathsf{fwd}_j.(i, (i, e)) \ntriangleleft tr$ *and* $\mathsf{fwd}_j.(i, (i, ed)) \ntriangleleft tr$
(d) if $ed = (e', t)$ *for some* $e' \in AE_i^{ld}$ *and* t, *then* $\mathsf{appv}_i.ed \ntriangleleft tr$
(e) if $ed = (e', t)$ *for some* $e' \in AE_i \setminus AE_i^{ld}$ *and* t, *then* $\mathsf{edec}_i.ed \ntriangleleft tr$ ◊

Proof. Let the trace $tr \in traces(EI)$ be arbitrary but fixed.

On A.3 (a): Let identifiers $i, j, k \in SP$ with $i \neq j$ and decision $ed \in ChWD$ be arbitrary but fixed. We show the claim by contradiction and assume that $\mathsf{ddec}_i.(j, (k, ed)) \lhd tr$ holds.

$$\begin{aligned}
&\overset{(*)}{\Longrightarrow} \mathsf{ddec}_i.(j, (k, ed)) \lhd tr \upharpoonright \alpha(\rho_i(\mathsf{SRP}_i)) \\
&\qquad \wedge tr \upharpoonright \alpha(\rho_i(\mathsf{SRP}_i)) \in traces(\rho_i(\mathsf{SRP}_i)) \\
&\Longrightarrow \text{by definition of } \mathsf{SRP}_i \text{ (line 4)} \\
&\qquad (k, ed) \in SP \times AE \\
&\Longrightarrow \text{by the definitions of } AE \text{ and } ChWD\text{, which imply } AE \cap ChWD = \emptyset \\
&\qquad (k, ed) \notin SP \times ChWD
\end{aligned}$$

This contradicts the preconditions $k \in SP$ and $ed \in ChWD$. Hence, the assumption of $\mathsf{ddec}_i.(j, (k, ed)) \lhd tr$ is wrong and $\mathsf{ddec}_i.(j, (k, ed)) \ntriangleleft tr$ holds.

On A.3 (b): The proof goes along the lines of the one for the previous part, with $ChWD$ exchanged by AE and ddec exchanged by rdec.

On A.3 (c): We show the claim by contradiction and assume that there exist $x \in AE \cup$

$ChWD$ and $i, j \in SP$ with $j \neq i$, such that $\mathsf{fwd}_j.(i,(i,x)) \lhd tr$ holds.

$$\xrightarrow{\rho_j(\mathsf{SRP}_j)} \mathsf{rreq}_j.(i,x) \lhd tr$$

$$\xrightarrow{\rho_j(\mathsf{COR}_j)} \mathsf{link}_{k,j}.(i,x) \lhd tr \text{ for some service provider } k \neq j$$

$$\xrightarrow{\rho_k(\mathsf{COR}_k)} \mathsf{ddec}_k.(j,(i,x)) \lhd tr \vee \mathsf{rdec}_k.(j,(i,x)) \lhd tr \vee \mathsf{fwd}_k.(j,(i,x)) \lhd tr$$

$$\begin{aligned}
\overset{(*)}{\Longrightarrow} & (\mathsf{ddec}_k.(j,(i,x)) \lhd tr \upharpoonright \alpha(\rho_k(\mathsf{SRP}_k)) \\
& \wedge tr \upharpoonright \alpha(\rho_k(\mathsf{SRP}_k)) \in traces(\rho_k(\mathsf{SRP}_k))) \\
& \vee \ (\mathsf{rdec}_k.(j,(i,x)) \lhd tr \upharpoonright \alpha(\rho_k(\mathsf{SRP}_k)) \\
& \quad \wedge tr \upharpoonright \alpha(\rho_k(\mathsf{SRP}_k)) \in traces(\rho_k(\mathsf{SRP}_k))) \\
& \vee \ (\mathsf{fwd}_k.(j,(i,x)) \lhd tr \upharpoonright \alpha(\rho_k(\mathsf{SRP}_k)) \\
& \quad \wedge tr \upharpoonright \alpha(\rho_k(\mathsf{SRP}_k)) \in traces(\rho_k(\mathsf{SRP}_k)))
\end{aligned}$$

$\Longrightarrow$ by definition of SRP_k

$j = nxt(k,i)$

$\Longrightarrow$ by definition of nxt

$j = nxt(k,i) = i$

The last equation contradicts the initial assumption of $i \neq j$. Hence, the assumption $\mathsf{fwd}_j.(i,(i,x)) \lhd tr$ cannot hold, i.e., $\mathsf{fwd}_j.(i,(i,x)) \not\lhd tr$ holds for all $j \neq i$.

On A.3 (d): We show the claim by contradiction and assume there exists an identifier $i \in SP$ and a decision $ed = (e', t) \in ChWD$ for some $e' \in AE_i^{\mathrm{ld}}$ such that $\mathsf{appv}_i.ed \lhd tr$ holds.

$$\xrightarrow{\rho_i(\mathsf{SRP}_i)} \mathsf{rreq}_i.(i,ed) \lhd tr \xrightarrow{\rho_i(\mathsf{COR}_i)} \mathsf{link}_{j,i}.(i,ed) \lhd tr \text{ for some identifier } j \neq i$$

$$\xrightarrow{\rho_j(\mathsf{COR}_j)} \mathsf{ddec}_j.(i,(i,ed)) \lhd tr \vee \mathsf{fwd}_j.(i,(i,ed)) \lhd tr \vee \mathsf{rdec}_j.(i,(i,ed)) \lhd tr$$

$\Longrightarrow$ by Lemma A.3 (a) and Lemma A.3 (c)

$\mathsf{rdec}_j.(i,(i,ed)) \lhd tr$

$$\xrightarrow{\rho_j(\mathsf{SRP}_j)} \mathsf{rtrsp}_j.(i,ed) \lhd tr \xrightarrow{\rho_j(\mathsf{DEC}_j(\emptyset))} \mathsf{rereq}_j.e' \lhd tr \xrightarrow{\rho_j(\mathsf{SRP}_j)} \mathsf{rreq}_j.(j,e') \lhd tr$$

$$\xrightarrow{\rho_j(\mathsf{COR}_j)} \mathsf{link}_{k,j}.(j,e') \lhd tr \text{ for some identifier } k \neq j$$

$$\xrightarrow{\rho_k(\mathsf{COR}_k)} \mathsf{ddec}_k.(j,(j,e')) \lhd tr \vee \mathsf{fwd}_k.(j,(j,e')) \lhd tr \vee \mathsf{rdec}_k.(j,(j,e')) \lhd tr$$

$\Longrightarrow$ by Lemma A.3 (b) and Lemma A.3 (c)

$\mathsf{ddec}_k.(j,(j,e')) \lhd tr$

$$\xrightarrow{\rho_k(\mathsf{SRP}_k)} \mathsf{rtreq}_k.(j,e') \lhd tr$$

$$\begin{aligned}
\overset{(*)}{\Longrightarrow} & \ \mathsf{rtreq}_k.(j,e') \lhd tr \upharpoonright \alpha(\rho_k(\mathsf{DEL}_k)) \\
& \wedge tr \upharpoonright \alpha(\rho_k(\mathsf{DEL}_k)) \in traces(\rho_k(\mathsf{DEL}_k))
\end{aligned}$$

$\Longrightarrow$ by definition of DEL_k, according to which $\mathsf{rtreq}.(j,e')$ can only occur if $e' \in \{e'' \in AE_k \mid k \neq resp(e'')\}$ holds

$$\begin{aligned}
&e' \in AE_k \wedge k \neq resp(e') \\
\Longrightarrow\ & \text{by definition of function } sp \text{ on access events} \\
&sp(e') \neq resp(e) \\
\Longrightarrow\ & \text{by definition of } AE_i^{\mathrm{ld}} \\
&e' \notin AE_i^{\mathrm{ld}}
\end{aligned}$$

Hence, the assumption $\mathsf{appv}_i.ed \lhd tr$ leads to a contradiction to the precondition $e' \in AE_i^{\mathrm{ld}}$. Therefore, $\mathsf{appv}_i.ed \not\lhd tr$ holds, as it was to be shown.

On A.3 (e): Let identifier $i \in SP$ and decision $ed = (e', t) \in ChWD$ with $e' \in AE_i \setminus AE_i^{\mathrm{ld}}$ be arbitrary but fixed. We show the claim by contradiction and assume $\mathsf{edec}_i.ed \lhd tr$ holds.

$$\begin{aligned}
&\xRightarrow{\rho_i(\mathsf{DEC}_i(\emptyset))} \mathsf{lereq}_i.e' \lhd tr \\
&\overset{(*)}{\Longrightarrow} \mathsf{lereq}_i.e' \lhd tr \upharpoonright \alpha(\rho_i(\mathsf{DEL}_i)) \wedge tr \upharpoonright \alpha(\rho_i(\mathsf{DEL}_i)) \in traces(\rho_i(\mathsf{DEL}_i)) \\
&\Longrightarrow \text{by definition of } \mathsf{DEL}_i\text{, according to which } \mathsf{lereq}.e' \\
&\qquad \text{can only occur if } e' \in \{e'' \in AE_i \mid i = resp(e'')\} \\
&\quad i = resp(e') \\
&\Longrightarrow \text{by the precondition that } e' \in AE_i \text{ and by definition of } AE_i^{\mathrm{ld}} \\
&\quad e' \in AE_i^{\mathrm{ld}}
\end{aligned}$$

Hence, the assumption $\mathsf{edec}_i.ed \lhd tr$ leads to a contradiction to the precondition $e' \notin AE_i^{\mathrm{ld}}$. Therefore, $\mathsf{edec}_i.ed \not\lhd tr$ holds, as it was to be shown. □

Lemma A.4. *For each access event $e \in AE$ and each trace $tr \in traces(EI)$ with $e \lhd tr$, there exists a decision $ed = (e', t) \in ChWD_{sp(e)}$ with $e \lhd t$ and $\mathsf{enf}_{sp(e)}.(e', t) \lhd tr$.* ◊

Proof. Let $e \in AE$ be an access event and $tr \in traces(EI)$ be a trace with $e \lhd tr$. Let $i = sp(e)$.

$$\begin{aligned}
&\overset{(*)}{\Longrightarrow} \text{with } e \in AE_{sp(e)} = AE_i \text{ by definition of } sp, \\
&\qquad AE_i \subseteq \alpha(\mathsf{REPLACE}_i) \text{ by definition of } \rho_i(\mathsf{REPLACE}_i), \\
&\qquad \text{and } \rho_i(e) = e \text{ according to Definition A.2 (a)} \\
&\quad e \lhd tr \upharpoonright \alpha(\rho_i(\mathsf{REPLACE}_i)) \wedge tr \upharpoonright \alpha(\rho_i(\mathsf{REPLACE}_i)) \in traces(\rho_i(\mathsf{REPLACE}_i)) \\
&\Longleftrightarrow \text{with } tr_r = tr \upharpoonright \alpha(\rho_i(\mathsf{REPLACE}_i)) \\
&\quad e \lhd tr_r \wedge tr_r \in traces(\rho_i(\mathsf{REPLACE}_i)) \\
&\overset{(\dagger)}{\Longrightarrow} \mathsf{enf}_i.(e', t) \lhd tr_r \quad \text{for some } (e', t) \in ChWD_i \text{ with } e \lhd t \\
&\Longrightarrow \text{with } i = sp(e) \text{ and } tr_r = tr \upharpoonright \alpha(\rho_i(\mathsf{REPLACE}_i)) \\
&\quad \mathsf{enf}_{sp(e)}.(e', e) \lhd tr
\end{aligned}$$

Below, we prove the correctness of the implication labeled (†) above. For this, we first show by induction over the length of tr_2 that if $tr_1.tr_2 \in traces(\rho_i(\mathsf{REPLACE}_i))$ holds for $tr_2 \in {AE_i}^* \setminus \{\langle\rangle\}$ and tr_1 that does not end with an event from AE_i, then $tr_1 = tr'_1.\langle \mathsf{enf}_i.(e', tr_2.t')\rangle$ and $\rho_i(\mathsf{REPLACE}_i) / (tr_1.tr_2) \equiv \rho_i(\mathsf{REPL}_i(t'))$ for some event $e' \in AE_i$ and some sequence t'.

base case ($|tr_2| = 1$): In this case, we have $tr_2 = \langle e\rangle$ for some $e \in AE_i$. From the definitions of $\rho_i(\mathsf{REPLACE}_i)$ and $\rho_i(\mathsf{REPL}_i)$, where there is only one possible position in the process expressions at which access event e can occur, it follows that $\rho_i(\mathsf{REPLACE}_i) / tr_1 \equiv \rho_i(\mathsf{REPL}_i(tr_2.t'))$ for some t'. By the prerequisite that tr_1 does not end with an event from AE_i, it follows from the definition of $\rho_i(\mathsf{REPLACE}_i)$ that $tr_1 = tr'_1.\langle \mathsf{enf}_i.(e', tr_2.t')\rangle$ for some $e' \in AE_i$ and some t'. Moreover, by Lemma 8.1 (e) it follows that $\rho_i(\mathsf{REPLACE}_i) / (tr_1.tr_2) \equiv \rho_i(\mathsf{REPL}_i(t'))$.

step case ($|tr_2| > 1$): In this case, we have $tr_2 = tr'_2.\langle e\rangle$ for some $e \in AE_i$. The induction hypothesis for tr'_2 gives, firstly, $tr_1 = tr'_1.\langle \mathsf{enf}_i.(e', tr'_2.t')\rangle$ and, secondly, $\rho_i(\mathsf{REPLACE}_i) / (tr_1.tr'_2) \equiv \rho_i(\mathsf{REPL}_i(t'))$ for some event $e' \in AE_i$ and some sequence t'. From the process equivalence together with $tr_1.tr'_2.\langle e\rangle \in traces(\rho_i(\mathsf{REPLACE}_i))$ and Lemma 8.1 (e) it follows that $\langle e\rangle \in traces(\rho_i(\mathsf{REPL}_i(t')))$. By definition of $\rho_i(\mathsf{REPL}_i)$, it then has to hold that $t' = \langle e\rangle.t''$. We therefore conclude that $tr_1 = tr'_1.\langle \mathsf{enf}_i.(e', tr_2.t'')\rangle$ and $\rho_i(\mathsf{REPLACE}_i) / (tr_1.tr_2) \equiv \rho_i(\mathsf{REPL}_i(t''))$ for some event $e' \in AE_i$ and some sequence t''.

Secondly, we make use of the previously shown claim to show the implication labeled (†). From $e \lhd tr_r$ and $tr_r \in traces(\rho_i(\mathsf{REPLACE}_i))$ if follows that there are sequences t_1, t'_1, t_2 with $t'_1 \in AE_i$ such that $tr_r = t_1.t'_1.\langle e\rangle.t_2$ holds. It follows that $\mathsf{enf}_i.(e', tr_2.t) \lhd t_1$ for some event $e' \in AE_i$ and some sequence t. Hence, in particular $\mathsf{enf}_i.(e', tr_2.t) \lhd tr_r$ □

Lemma A.5. *Let $e \in AE \setminus DENY$ be a successful access event and $tr \in traces(EI)$ with $e \lhd tr$ be a trace. Then $\mathsf{edec}_{resp(e)}.(e, \langle e\rangle) \lhd tr$ or $\mathsf{rtrsp}_{resp(e)}.(sp(e), (e, \langle e\rangle)) \lhd tr$ holds true.* ◇

Proof. Let $e \in AE \setminus DENY$ be a successful access event and $tr \in traces(EI)$ with $e \lhd tr$ be a trace. Let $i = sp(e)$.

$$\begin{aligned}
&\Longrightarrow \text{by Lemma A.4}\\
&\quad \mathsf{enf}_i.(e', t) \lhd tr \text{ for some event } e' \in AE_i \text{ and sequence } t \text{ with } e \lhd t\\
&\xrightarrow{\rho_i(\mathsf{COR}_i)} \mathsf{edec}_i.(e', t) \lhd tr \lor \mathsf{appv}_i.(e', t) \lhd tr\\
&\Longrightarrow \text{by Lemmas A.3 (d) and A.3 (e)}\\
&\quad (\mathsf{edec}_i.(e', t) \lhd tr \land sp(e') = resp(e'))\\
&\quad \lor (\mathsf{appv}_i.(e', t) \lhd tr \land sp(e') \neq resp(e'))
\end{aligned}$$

In the following, we show the claim for both disjunct cases separately.

1. case $\mathsf{edec}_i.(e', t) \triangleleft tr \wedge sp(e') = resp(e')$:

$$\overset{(*)}{\Longrightarrow} \mathsf{edec}_i.(e', t) \triangleleft tr \upharpoonright \alpha(\rho_j(\mathsf{DEC}_i(\emptyset))) \wedge tr \upharpoonright \alpha(\rho_j(\mathsf{DEC}_i(\emptyset))) \in traces(\rho_j(\mathsf{DEC}_i(\emptyset)))$$

$\Longrightarrow$ by definition of $\mathsf{DEC}_i(\emptyset)$

$$\mathsf{edec}_i.(e', t) \triangleleft tr \wedge (t = \langle e' \rangle \vee t = \langle deny(e') \rangle)$$

$\Longrightarrow$ by $e \triangleleft t$ and the preconditions $e \in AE \setminus DENY$ and $deny(e') \in DENY$

$$\mathsf{edec}_i.(e', t) \triangleleft tr \wedge t = \langle e \rangle \wedge e' = e$$

$\Longrightarrow$ by the preconditions $i = sp(e)$ and $sp(e') = resp(e')$

$$\mathsf{edec}_{resp(e)}.(e, \langle e \rangle) \triangleleft tr$$

2. case $\mathsf{appv}_i.(e', t) \triangleleft tr \wedge sp(e') \neq resp(e')$: Let $ed = (e', t)$.

$$\xrightarrow{\rho_i(\mathsf{SRP}_i)} \mathsf{rreq}_i.(i, ed) \triangleleft tr$$

$$\xrightarrow{\rho_i(\mathsf{COR}_i)} \mathsf{link}_{j,i}.(i, ed) \triangleleft tr \text{ for some identifier } j \neq i$$

$$\xrightarrow{\rho_j(\mathsf{COR}_j)} \mathsf{ddec}_j.(i, (i, ed)) \triangleleft tr \vee \mathsf{fwd}_j.(i, (i, ed)) \triangleleft tr \vee \mathsf{rdec}_j.(i, (i, ed)) \triangleleft tr$$

$\Longrightarrow$ by Lemma A.3 (a) and Lemma A.3 (c)

$$\mathsf{rdec}_j.(i, (i, ed)) \triangleleft tr$$

$$\xrightarrow{\rho_j(\mathsf{SRP}_j)} \mathsf{rtrsp}_j.(i, ed) \triangleleft tr \qquad (\dagger)$$

$$\overset{(*)}{\Longrightarrow} \mathsf{rtrsp}_j.(i, ed) \triangleleft tr \upharpoonright \alpha(\rho_j(\mathsf{DEC}_j(\emptyset))) \wedge tr \upharpoonright \alpha(\rho_j(\mathsf{DEC}_j(\emptyset))) \in traces(\rho_j(\mathsf{DEC}_j(\emptyset)))$$

$\overset{(**)}{\Longrightarrow}$ by definition of $\mathsf{DEC}_j(\emptyset)$ and $ed = (e', t)$

$$\mathsf{rereq}_j.e' \triangleleft tr \upharpoonright \alpha(\rho_j(\mathsf{DEC}_j(\emptyset))) \wedge (t = \langle e' \rangle \vee t = \langle deny(e') \rangle)$$

$\Longrightarrow$ by $e \triangleleft t$ and the preconditions $e \in AE \setminus DENY$ and $deny(e') \in DENY$

$$\mathsf{rereq}_j.e' \triangleleft tr \upharpoonright \alpha(\rho_j(\mathsf{DEC}_j(\emptyset))) \wedge t = \langle e \rangle \wedge e' = e \qquad (\ddagger)$$

$$\Longrightarrow \mathsf{rereq}_j.e \triangleleft tr$$

$$\xrightarrow{\rho_j(\mathsf{SRP}_j)} \mathsf{rreq}_j.(j, e) \triangleleft tr$$

$$\xrightarrow{\rho_j(\mathsf{COR}_j)} \mathsf{link}_{k,j}.(j, e) \triangleleft tr \text{ for some identifier } k \neq j$$

$$\xrightarrow{\rho_k(\mathsf{COR}_k)} \mathsf{ddec}_k.(j, (j, e)) \triangleleft tr \vee \mathsf{fwd}_k.(j, (j, e)) \triangleleft tr \vee \mathsf{rdec}_k.(j, (j, e)) \triangleleft tr$$

$\Longrightarrow$ by Lemma A.3 (b) and Lemma A.3 (c)

$$\mathsf{ddec}_k.(j, (j, e)) \triangleleft tr$$

$$\xrightarrow{\rho_k(\mathsf{SRP}_k)} \mathsf{rtreq}_k.(j, e) \triangleleft tr$$

$$\overset{(*)}{\Longrightarrow} \mathsf{rtreq}_k.(j, e) \triangleleft tr \upharpoonright \alpha(\rho_k(\mathsf{DEL}_k)) \wedge tr \upharpoonright \alpha(\rho_k(\mathsf{DEL}_k)) \in traces(\rho_k(\mathsf{DEL}_k))$$

$\Longrightarrow$ by definition of DEL_k

$$j = resp(e)$$

$$\Longrightarrow \text{with } \mathsf{rtrsp}_j.(i,(e',t)) \lhd tr \text{ from (†) above, } e' = e \text{ and } t = \langle e\rangle \text{ from (‡) above, and precondition } i = sp(e)$$

$$\mathsf{rtrsp}_{resp(e)}.(sp(e),(e,\langle e\rangle)) \lhd tr \qquad \square$$

Lemma A.6. *Let trace $tr \in traces(EI)$, identifier $i \in SP$, and events $e, e' \in AE \setminus DENY$ with $e \otimes e'$ be given. Then at least one of the following holds:*

- $\mathsf{edec}_i.(e,\langle e\rangle) \ntriangleleft tr$ *and for all identifiers* $j \in SP$, $\mathsf{rtrsp}_i.(j,(e,\langle e\rangle)) \ntriangleleft tr$, *or*
- $\mathsf{edec}_i.(e',\langle e'\rangle) \ntriangleleft tr$ *and for all identifiers* $j \in SP$, $\mathsf{rtrsp}_i.(j,(e',\langle e'\rangle)) \ntriangleleft tr$. ◊

Proof. Let trace $tr \in traces(EI)$, identifier $i \in SP$, and events $e, e' \in AE \setminus DENY$ with $e \otimes e'$ be arbitrary but fixed. Let, for each $\bar{e} \in \{e, e'\}$, the set $D_{\bar{e}}$ be defined by $D_{\bar{e}} = \{\mathsf{edec}_i.(\bar{e},\langle\bar{e}\rangle), \mathsf{rtrsp}_i.(j,(\bar{e},\langle\bar{e}\rangle)) \mid j \in SP\}$. Note that $D_e \cap D_{e'} = \emptyset$ holds, as $e \neq e'$ by definition of $\otimes$.

To show that the lemma holds, we show that $tr \upharpoonright D_e = \langle\,\rangle$ or $tr \upharpoonright D_{e'} = \langle\,\rangle$ holds, which is equivalent to $(tr \upharpoonright D_e \neq \langle\,\rangle) \rightarrow (tr \upharpoonright D_{e'} = \langle\,\rangle)$. Thus, we assume $tr \upharpoonright D_e \neq \langle\,\rangle$ and show in the following that $tr \upharpoonright D_{e'} = \langle\,\rangle$ holds.

$$tr \upharpoonright D_e \neq \langle\,\rangle$$

$$\Longrightarrow \exists x \in D_e : (x \lhd tr)$$

$$\overset{(*)}{\Longrightarrow} x \lhd tr \upharpoonright \alpha(\rho_i(\mathsf{DEC}_i(\emptyset))) \land tr \upharpoonright \alpha(\rho_i(\mathsf{DEC}_i(\emptyset))) \in traces(\rho_i(\mathsf{DEC}_i(\emptyset)))$$

$$\Longleftrightarrow \text{with } tr_{\mathrm{dec}} = tr \upharpoonright \alpha(\rho_i(\mathsf{DEC}_i(\emptyset)))$$

$$x \lhd tr_{\mathrm{dec}} \land tr_{\mathrm{dec}} \in traces(\rho_i(\mathsf{DEC}_i(\emptyset)))$$

$$\Longrightarrow \text{with } x \in D_e \text{ and } D_e \cap D_{e'} = \emptyset$$

$$x \lhd tr_{\mathrm{dec}} \land tr_{\mathrm{dec}} \in traces(\rho_i(\mathsf{DEC}_i(\emptyset))) \land x \notin D_{e'}$$

$$\Longrightarrow \text{without loss of generality}$$

$$tr_{\mathrm{dec}} = t'.\langle x\rangle.t'' \quad \text{for some } t' \text{ and } t'' \text{ with } t' \upharpoonright D_{e'} = \langle\,\rangle$$

$$\overset{(1)}{\Longrightarrow} t'' \in traces(\rho_i(\mathsf{DEC}_i(q))) \quad \text{for some } q \in \mathcal{P}(AE) \text{ with } e \in q$$

$$\overset{(2)}{\Longrightarrow} t'' \upharpoonright D_{e'} = \langle\,\rangle$$

$$\Longrightarrow \text{with } t' \upharpoonright D_{e'} = \langle\,\rangle \text{ and } x \notin D_{e'} \text{ from above}$$

$$tr_{\mathrm{dec}} \upharpoonright D_{e'} = (t'.\langle x\rangle.t') \upharpoonright D_{e'} = \langle\,\rangle$$

$$\Longrightarrow \text{by preconditions } D_{e'} \subseteq \alpha(\rho_i(\mathsf{DEC}_i(\emptyset))) \text{ and } tr_{\mathrm{dec}} = tr \upharpoonright \alpha(\rho_i(\mathsf{DEC}_i(\emptyset)))$$

$$tr \upharpoonright D_{e'} = \langle\,\rangle$$

Below, we prove the correctness of the implications labeled (1) and (2) above.

1. First we show by induction over the length of t that for all states $q \in \mathcal{P}(AE)$ and non-empty traces $t \in traces(\rho_i(\mathsf{DEC}_i(q))) \setminus \{\langle\,\rangle\}$, there are a state $q' \in \mathcal{P}(AE)$ and sequences t_1, t_2 such that $t = t_1.t_2$, $t_2 \neq \langle\,\rangle$, $|t_2| \leq 2$, $q' \supseteq q$, and $\rho_i(\mathsf{DEC}_i(q)) \,/\, t_1 \equiv \rho_i(\mathsf{DEC}_i(q'))$ holds.

   base case ($0 < |t| \leq 2$): Let $q \in \mathcal{P}(AE)$ be arbitrary but fixed. The claim is trivially satisfied by $t_1 = \langle\,\rangle$, $t_2 = t$, and $q' = q$.

step case ($|t| > 2$): Let $q \in \mathcal{P}(AE)$ be arbitrary but fixed. Let t', t'' be sequences with $|t'| = 2$ such that $t = t'.t''$ holds. Then by the definitions of $\rho_i(\mathsf{DEC}_i(q))$ and "/" and by Lemma 8.1 (e), it follows that $\rho_i(\mathsf{DEC}_i(q)) \,/\, t' \equiv \rho_i(\mathsf{DEC}_i(q''))$ holds for some $q'' \in \mathcal{P}(AE)$ with $q'' \supseteq q$. By the definition of $\equiv$, it follows that $t'' \in \mathit{traces}(\rho_i(\mathsf{DEC}_i(q'')))$. Then the induction hypothesis for t'' (which is non-empty but strictly shorter than t) and q'' provides that there are a state $q' \in \mathcal{P}(AE)$ and sequences t_1'', t_2'' such that $t'' = t_1''.t_2''$, $t_2'' \neq \langle\,\rangle$, $|t_2''| \leq 2$, $q' \supseteq q''$, and $\rho_i(\mathsf{DEC}_i(q'')) \,/\, t_1'' \equiv \rho_i(\mathsf{DEC}_i(q'))$ holds. Then $t_1 = t'.t_1''$, $t_2 = t_2''$, and q' are the witnesses for the claim, which satisfy $t = t_1.t_2$, $t_2 \neq \langle\,\rangle$, $|t_2| \leq 2$, $q' \supseteq q$, and, by Lemma 8.1 (e) for t_1, $\rho_i(\mathsf{DEC}_i(q)) \,/\, t_1 \equiv \rho_i(\mathsf{DEC}_i(q'))$.

Second, we use the property proved before to show that implication (1) holds. For this, let $tr_{\mathrm{dec}} = t'.\langle x\rangle.t'' \in \mathit{traces}(\rho_i(\mathsf{DEC}_i(\emptyset)))$. It first follows that $t'.\langle x\rangle \in \mathit{traces}(\rho_i(\mathsf{DEC}_i(\emptyset)))$. Hence, there are a state $q \in \mathcal{P}(AE)$ and sequences t_1', t_2' such that $t'.\langle x\rangle = t_1'.t_2'.\langle x\rangle$, $|t_2'| < 2$, $q' \supseteq \emptyset$, and $\rho_i(\mathsf{DEC}_i(\emptyset)) \,/\, t_1' \equiv \rho_i(\mathsf{DEC}_i(q))$ hold. By definition of $\equiv$ (on Page 134), it follows that $t_2'.\langle x\rangle.t'' \in \mathit{traces}(\rho_i(\mathsf{DEC}_i(q)))$ holds. By definition of $\rho_i(\mathsf{DEC}_i(q))$ – recall firstly that x must be one of $\mathrm{edec}_i.(\bar{e}, \langle\bar{e}\rangle)$ and $\mathrm{rtrsp}_i.(j, (\bar{e}, \langle\bar{e}\rangle))$ for some $j \in SP$, and, secondly, that $\bar{e} \notin DENY$ – it holds that $\rho_i(\mathsf{DEC}_i(q)) \,/ (t_2'.\langle\bar{e}\rangle) \equiv \rho_i(\mathsf{DEC}_i(q'))$ for $q' = q \cup \{e\}$ and, by definition of $\equiv$, $t'' \in \mathit{traces}(\rho_i(\mathsf{DEC}_i(q')))$, as it was to be shown.

2. We show by induction over the length of traces t that for all states $q' \in \mathcal{P}(AE)$ with $e \in q'$, it holds that $t \in \mathit{traces}(\rho_i(\mathsf{DEC}_i(q')))$ implies $t \upharpoonright D_{e'} = \langle\,\rangle$.

   base case ($|t| \leq 1$): By definition of $\mathsf{DEC}_i(q')$, events from $D_{e'}$ cannot be contained in such a trace t with length $|t| \leq 1$.

   step case ($|t| > 1$): Let t_1, t_2 be sequences with $|t_1| = 2$ such that $t = t_1.t_2$ holds. For all $q' \in \mathcal{P}(AE)$, the definition of $\rho_i(\mathsf{DEC}_i(q'))$ gives:
   - An event $x' \in D_{e'}$ cannot be contained in t_1 because of the definition of *conf* on Page 150 and the precondition $e' \otimes e$; hence, we have $t_1 \upharpoonright D_{e'} = \langle\,\rangle$.
   - $\mathsf{DEC}_i(q') \,/\, t_1 \equiv \mathsf{DEC}_i(q'')$ for a state $q'' \supseteq q'$; hence, the induction hypothesis can be applied on q'' and t_2 to obtain $t_2 \upharpoonright D_{e'} = \langle\,\rangle$.

   It follows that $t \upharpoonright D_{e'} = \langle\,\rangle$. □

Based on the preceding lemmas and the auxiliary definition, we can now conduct the proof that our policy model, *ChWgp*, is sound for the Chinese Wall security property, $\mathit{ChW}(\alpha(ET_{ChWgp}))$.

Theorem 8.2. *ChWgp is a sound policy model for* $\mathit{ChW}(\alpha(ET_{ChWgp}))$ ◇

Proof. We show that *ChWgp* is a sound policy model for $\mathit{ChW}(\alpha(ET_{ChWgp}))$. By definition of a sound policy model (Definition 8.20 on page 147), this is equivalent to show that ET_{ChWgp} sat $\mathit{ChW}(\alpha(ET_{ChWgp}))$ holds under set $\mathcal{EQ}_{ChWgp}$ of process equations. Substituting the definitions of sat and *ChW*, this is equivalent to proving that for all traces $tr \in \mathit{traces}(ET_{ChWgp})$ there do not exist events $e_1, e_2 \in AE$ such that $e_1, e_2 \lhd tr$ and $e_1 \otimes e_2$ hold.

We conduct the proof by contradiction and assume that *ChWgp* is *not* a sound policy model for $\mathit{ChW}(\alpha(ET_{ChWgp}))$. From this assumption follows that there exist a trace $tr \in \mathit{traces}(ET_{ChWgp})$ and events $e_1, e_2 \in AE$ such that $e_1, e_2 \lhd tr$ and $e_1 \otimes e_2$ hold true.

Since events e_1 and e_2 are in conflict, they cannot be denying events (by definition of $\otimes$ on Page 148). Secondly, the responsible unit for e_1 and e_2 must be the same, i.e., $resp(e_1) = resp(e_2)$ (by the definition of *resp* on Page 149). Let $k = resp(e_1)$ be this responsible unit.

In the following, the process expression *EI* (see Definition A.2 (c)) denotes the encapsulated target model with all CSP hiding operations removed and the internal channels renamed to not introduce additional synchronization with the removal of hiding. We assume, without loss of generality, that the service providers SP_{sp} do neither make use of internal channels of our local policy model nor make use of the renamed channels (if they would, we could choose a different renaming). Then by Lemma A.2, we obtain the equivalence $EI \setminus H_{EI}\ {}_{\mathcal{EQ}_{EI}}{\equiv}_{\mathcal{EQ}_{ChW}}\ ET_{ChWgp}$, where the hiding set H_{EI} is defined in Definition A.2 (b). It follows that there must be a trace $tr' \in traces(EI)$ such that $tr' \restriction \alpha(ET_{ChWgp}) = tr$. Particularly, we therefore have $e_1, e_2 \lhd tr'$.

Applying Lemma A.5 for each event $e \in \{e_1, e_2\}$, we get that in both cases either event $\mathrm{edec}_k.(e, \langle e \rangle)$ or event $\mathrm{rtrsp}_k.(sp(e), (e, \langle e \rangle))$ is contained in tr'. This contradicts Lemma A.6, which states that this can only hold for at most one of e_1 and e_2. Consequently, the assumption that *ChWgp* is not a sound policy model for $ChW(\alpha(ET_{ChWgp}))$ does not hold leads to a contradiction and, thus, cannot be true. Therefore, *ChWgp is* a sound policy model for $ChW(\alpha(ET_{ChWgp}))$. $\square$

Appendix B

Selected Code Excerpts

B.1. Modular Delegation-Based Security Policies

In this appendix, we present selected code fragments for Chapter 6.

Expressiveness of the framework Listing B.1 on the next page shows an implementation of a local policy based on the framework presented in Chapter 6. The implementation contains the class ExpressivenessProof class that exhibits the same functionality as implemented by class SomePolicy, no matter how the latter class is actually implemented. The reasons for this equivalence is as follows. We only discuss the case of an event object, as the case of a delegation object is analogous. When SomePolicy is invoked for an event object, its result is the return value of method localRequest on this event object. We argue that the return value is the same for our simulating class ExpressivenessProof: When ExpressivenessProof is invoked for the event object, it first uses the its micro-policy factory (here Simulator) for obtaining a micro-policy by invoking the factory method createFromEvent of the Simulator object on the event object. The factory method uses p, its SomePolicy object for computing the same result as SomePolicy would have returned. It then returns a reference to itself, as Simulator also implements the MicroPolicy interface. In consequence, the suggestPolicyResult method of the Simulator object is invoked and returns a MicroPolicyResult object that encapsulates the previously computed result of SomePolicy. By the implementation of ModularLocalPolicy, if the result of suggestPolicyResult is a decision object, it is returned – hence, in his case, equivalence to SomePolicy is given. If the result is a delegation object, is is first subject to routing (which here, in line 18, returns the destination identifier directly) and then to the delegation object to be returned. Hence, equivalence is given in both cases.

```java
import net.cliseau.lib.policy.modular.*;
import net.cliseau.runtime.javacor.*;

class SomePolicy extends LocalPolicy {
  public SomePolicy(String identifier) { super(identifier); }

  public LocalPolicyResponse localRequest(Event event) {
    return null; /* here could be any code */}
  public LocalPolicyResponse remoteRequest(DelegationReqResp delReqResp) {
    return null; /* here could be any code */}
}

public class ExpressivenessProof extends ModularLocalPolicy<Void,Void> {
  public ExpressivenessProof(String identifier) {
    super(identifier,
        new Simulator(new SomePolicy(identifier)), null,
        new RoutingPolicy() {
          public String getNext(String id) { return id; }
        });
  }
  static class Simulator implements MicroPolicyFactory<Void,Void>,MicroPolicy<Void,Void> {
    private final LocalPolicy p;
    private LocalPolicyResponse res;

    public Simulator(LocalPolicy p) { this.p = p; }
    public MicroPolicy<Void,Void> createFromEvent(Event event) {
      res = p.localRequest(event);
      return this;
    }
    public MicroPolicy<Void,Void> createFromDelegation(DelegationReqResp delegation) {
      res = p.remoteRequest(delegation);
      return this;
    }
    public MicroPolicyResult suggestPolicyResult(Void ignored) {
      if (res instanceof Decision)
        return new MicroPolicyResult((Decision)res);
      else
        return new MicroPolicyResult((DelegationLocPolReturn)res);
    }
    public void implementSuggestion(Void ignored) { }
  }
}
```

Listing B.1.: Local policy based on the modular framework, mimicking a local policy that is not based on the framework

List of Figures

List of Tables

List of Listings

List of Definitions

List of Theorems, Lemmas, and Conjectures

List of Examples

Mathematical Notation

B

basic concepts

$dom(f)$	the domain of function f
$f \oplus g$	the overriding of f by g, i.e., the function that maps $x \in dom(g)$ to $g(x)$ and $x \in dom(f) \setminus dom(g)$ to $f(x)$
$A \rightharpoonup B$	set of all partial functions from A to B
$A \to B$	set of all total functions from A to B
$f[x \mapsto y]$	the total function that maps x to y and all other values x' to $f(x')$
$(x_1, \ldots, x_n)$	the n-ary tuple consisting of elements x_1 to x_n
$A \cap B$	the intersection of the sets A and B
$\|A\|$	the number of elements in the (finite) set A
$\{x, x', \ldots\}$	the (extensional) set containing the given elements x, x', ..., and no further elements
$\{t(x_1, \ldots, x_n) \in A \mid \varphi\}$	the (intensional) set containing the terms $t(x_1, \ldots, x_n)$ from domain A satisfying condition φ, which may have free variables x_1 to x_n
$A_1 \times \ldots \times A_n$	the set containing all tuples $(x_1, \ldots, x_n)$ with $x_i \in A_i$
$A \cup B$	the union of the sets A and B
$\bigcup_{i \in I} A$	the finite union of all sets A_i for which $i \in I$
$\emptyset$	the empty set
$[x, x']$	the set of all real numbers between, including, x and x'
$[x, x')$	the set of all real numbers between x (included) and x' (excluded)
$A \setminus B$	the set A without the elements of set B
$A \subseteq B$	subset-or-equal relation on sets
δ_φ	the value 1 if formula φ is satisfied in the respective context of use, and the value 0 otherwise
$\mathcal{F}(\mathcal{S}, V)$	the set of formulas in first-order logic
$(x_i)_{i \in I}$	a family of values
$\varphi \wedge \psi$	the logical conjunction between two formulas
$\neg\varphi$	the logical negation of a formula

$\varphi \vee \psi$	the logical disjunction between two formulas
$\mathfrak{I} \models \varphi$	the satisfaction relation on interpretations and formulas
$t^{\mathfrak{I}}$	the valuation of a term under an interpretation
$\varphi \rightarrow \psi$	the logical implication between two formulas
$\varphi \leftrightarrow \psi$	the bidirectional logical implication of two formulas
$\varphi \Longrightarrow \psi$	meta-level implication (each interpretation satisfying φ also satisfies ψ)
$\varphi \Longleftrightarrow \psi$	meta-level equivalence (short-hand for $\varphi \Longrightarrow \psi$ and $\psi \Longrightarrow \varphi$)
$\forall x : \varphi$	the (classical) universal quantification in first-order logic
$\forall x \in A : \varphi$	universal quantification in first-order logic with domain
$\exists x : \varphi$	the (classical) existential quantification in first-order logic
$\exists x \in A : \varphi$	existential quantification in first-order logic with domain
$\max\{x \mid \varphi(x)\}$	the maximal element (according to a total order that is supposed to be clear from the context of use) from elements x satisfying φ
$\min\{x_1, \ldots, x_n\}$	the minimal element (according to a total order that is supposed to be clear from the context of use) from elements x_1 to x_n
$\mathbb{N}$	the set of natural numbers
$\mathcal{P}(A)$	the powerset of A, i.e., the set of all subsets of A
$\mathbb{R}$	the set of real numbers
$\mathcal{T}(\mathcal{S}, V)$	the set of terms in first-order logic

C

CSP

$\alpha(P)$	function that returns the alphabet for a given process expression P according to the trace semantics (Definition 8.6 on page 133)
$traces(P)$	function that returns the set of possible traces for a given process expression according to the trace semantics (Definition 8.6 on page 133)
STOP_E	the process with alphabet E that immediately terminates
$e \rightarrow P$	the “prefixing” of process expression P by event e
$P_1 \square \ldots \square P_n$	the external choice of process expressions P_1 to P_n
$\square_{i \in I} P_i$	the external choice over process expressions P_i
$P_1 \sqcap \ldots \sqcap P_n$	the internal choice of process expressions P_1 to P_n
$\sqcap_{i \in I} P_i$	the internal choice over process expressions P_i
$P\ {}_{\mathcal{EQ}_P}{\equiv}_{\mathcal{EQ}_Q}\ Q$	equivalence of process expressions P and Q under sets $\mathcal{EQ}_P$ and, respectively $\mathcal{EQ}_Q$ of process equations

$P_1 \parallel \ldots \parallel P_n$	the parallel composition of process expressions P_1 to P_n
$\parallel_{i \in I} P_i$	the parallel composition of process expressions P_i
$P \mathbin{/} tr$	CSP's “after” operator
$c.m$	structured event modeling message m on channel c
$c?x\colon M \to P$	reception of message, bound by x, from set M on channel c and then P
$x\colon E \to P$	choice of event, bound by x, from set E and then P
$c!m \to P$	sending of message m on channel c and then P
$P \setminus E$	the hiding of all events in E from the environment of process expression P
$\mathsf{NAME} \overset{\mathbf{def}}{=}_E P$	a process equation
$PROC$	the set of all processes
P sat φ	satisfaction of unary predicate φ by process expression P
$P[x/e]$	the process expression P with all free occurrences of x substituted by e [Ros05, p. 37]

CSP Model of CliSeAu

appv	channel for “approval” of decisions obtained after delegation between a local policy model and the coordinator
COR_{ci}	process expression of the coordinator model for ci
ddec	channel for delegation resulting from locally intercepted events between a local policy model and the coordinator model
E_{ci}^{COR}	alphabet of the coordinator model for ci
E_{ii}^{INT}	alphabet of the interceptor model for ii
$EC_{ci}(\bullet_{\mathrm{agent}}, \bullet_{\mathrm{pol}}, \bullet_{\mathrm{enf}})$	the parametric process expression modeling a generic EC for coordinator instance ci and the three parametric process expressions
edec	channel for decisions between a local policy model and the coordinator model
enf	channel for decisions to be enforced between the coordinator model and an enforcer model
$\mathcal{EQ}_{ci}^{\mathsf{COR}}$	set of process equations for the coordinator model
$\mathcal{EQ}_{pm}$	the set of process equations of an encapsulated target model for a policy model pm
$\mathcal{EQ}_{ii}^{\mathsf{INT}}$	set of process equations for the interceptor model
ET_{pm}	the process expression of an encapsulated target model for a policy model pm
fwd	channel for forwarding delegation requests and delegation responses between a local policy model and the coordinator model

H_{ci}	the set of internalized (hidden) events of an EC model for coordinator instance ci
icpt	channel for intercepted events between the interceptor model and the coordinator model
INT_{ii}	process expression of the interceptor model for ii
$\mathsf{link}_{i,j}$	channel between EC models with identifiers i and j
lreq	channel for local requests between the coordinator model and a local policy model
rdec	channel for (remote) decisions to delegation requests between a local policy model and the coordinator model
rreq	channel for remote requests between the coordinator model and a local policy model
sync	channel for synchronization of an enforcer model with the interceptor model
$\bullet_{\text{agent}}$	"hole" for an agent in the process expression of the EC model
$\bullet_{\text{enf}}$	"hole" for an enforcer model in the process expression of the EC model
$\bullet_{\text{pol}}$	"hole" for an local policy model in the process expression of the EC model
$\checkmark$	symbol used as message over channel sync

CSP Model of CliSeAu: Instances

AE	the set of all access events
AE_{sp}	the set of all access events of service provider sp
$\boldsymbol{ChW}(E)$	the Chinese Wall security property for set E of all events
$ChWD$	the joined set of all decisions of the EC models for all service providers
$ChWD_i$	the set of all decisions of the EC model with identifier i
$ChWDR$	the set of all delegation requests and delegation responses of the EC model
$ChWgp$	the policy model for enforcing the Chinese Wall Security Policy in the distributed storage service
ChWLP_i	process name of the local policy model at EC model i for the Chinese Wall security property
$COI \subseteq O \times O$	the binary relation modeling conflicts on objects
$e \otimes e'$	the binary relation on access events modeling conflicts between the accessed objects
$conf(q)$	the set of all access events in conflict with state q
$\mathsf{DEC}_i(q)$	process name of the decision-making component at EC model i for the Chinese Wall security property

DEL_i	process name of the delegation component at EC model i for the Chinese Wall security property
denied	the symbol indicating that an access was denied
$deny(e)$	function that maps an access event e to the set of possible traces for a given process expression according to the trace semantics (Definition 8.6 on page 133)
E_i^{DEC}	the alphabet of DEC_i
E_i^{DEL}	the alphabet of DEL_i
E_{REPL}	alphabet of the replacing enforcer
E_{SecAut}	alphabet of the local policy model of a security automaton
E_i^{SRP}	the alphabet of SRP_i
E_{SUPP}	alphabet of the suppressing enforcer
E_{TERM}	alphabet of the terminating enforcer
$\mathcal{EQ}_{ChW}$	the set of process equations of the policy model for enforcing the Chinese Wall Security Policy in the distributed storage service
$\mathcal{EQ}_{sp}$	the set of process equations of the service providers
H_i^{pol}	the alphabet of ChWLP_i
lereq	internal channel for local events to be decided, between DEL_i and DEC_i
$nxt(i, i')$	the next service provider on the route from i to i'
O	the set of all file objects in the application scenario
perm	symbol used by some example enforcer models to indicate that an event shall be permitted
$\mathsf{REPL}_{IE,EE}$	process name of an auxiliary process for the replacing enforcer
$\mathsf{REPLACE}_{IE,EE}$	process name of the replacing enforcer
rereq	internal channel for remote events to be decided, between SRP_i and DEC_i
$resp(e)$	function that maps an access event e to the responsible unit for e
rtreq	internal channel for routing delegation requests between DEL_i and SRP_i
rtrsp	internal channel for routing delegation responses between DEC_i and SRP_i
SecAut	process name of the local policy model of a security automaton
SP_{sp}	the process expression for the service provider sp
SP	the set of identifiers of service providers

$sp(e)$	function that maps an access event e to the service provider it belongs
SRP_i	process name of the routing component at EC model i for the Chinese Wall security property
SUPP_{EE}	process name of the suppressing enforcer
supp	symbol used by some example enforcer models to indicate that an event shall be suppressed
TERM_{EE}	process name of the terminating enforcer
term	symbol used by some example enforcer models to indicate that an agent shall be terminated
U	the set of all users in the application scenario

CSP Model of CliSeAu: Proofs

$AE_i^{\mathbf{ld}}$	the set of events that occur at and are decided at EC model i
COR_i	process expression of the coordinator model at EC model i
$DENY$	set of all denying access events
EC_i	process expression of the EC model i
EI	process expression of the encapsulated target model with renamed channels and without hiding
$\mathcal{EQ}_{EI}$	the set of process equations of the encapsulated target model with renamed channels without hiding, EI
H_i	the set of internalized (hidden) events of the EC model i
H_{EI}	the set of internalized (hidden) events of the encapsulated target model
INT_i	process expression of the interceptor model at EC model i
RE_i	set of events that are subject to renaming function ρ_i
REPL_i	auxiliary process expression of the replacing enforcer at EC model i
$\mathsf{REPLACE}_i$	process expression of the replacing enforcer at EC model i
ρ_i	the renaming function on internal events of the EC model i

L

languages

Σ	the alphabet implicitly underlying all languages presented in this thesis, containing alphanumeric symbols, punctuation symbols, whitespace symbols, control symbols (such as line breaks), and mathematical symbols
$\hookleftarrow \in \Sigma$	symbol modeling a line break
ε	the empty word over Σ^*
$w\ w'$	the word resulting from concatenating the words w and w'

$\mathcal{W}$	the set of all words ($\mathcal{W} = \Sigma^*$)
$\mathcal{W}_A$	the set of all words not containing a symbol from A, i.e., ($\mathcal{W}_A = (\Sigma \setminus A)^*$)
$\mathcal{L}(nt)$	the set of all words specified by the BNF non-terminal nt

M

meta-variables

$\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}'$, ...	meta-variables ranging over partitions of sets
A, B	meta-variables ranging over sets of underspecified type
$\mathfrak{A}$	meta-variable ranging over first-order logic structures
$\mathfrak{a}$	meta-variable ranging over maps for function and relation symbols in first-order logic structures
$AGENT$	meta-variable for agents of a target model
β	meta-variable ranging over assignments in first-order logic
C, C', C_1, ...	meta-variables ranging over sets of category names in DOSNs
c, c', ...	meta-variables ranging over category names in DOSNs
c, c', ...	meta-variables ranging over channels in CSP
cat, cat', ...	meta-variables ranging over categories of a DOSN user
$cdConfigs$	meta-variable ranging over functions mapping configuration file names in a decider program to the content of these files, as specified in Definition 5.3
ci	meta-variable ranging over coordinator instances
cs	meta-variable ranging over constant symbols
D	meta-variable ranging over domains of first-order logic structures
δ	meta-variable ranging over transition functions of a security automaton
do, do', ...	meta-variables ranging over decision objects, i.e., objects of class Decision
DR	meta-variable ranging over sets of delegation requests and delegation responses
dr, dr', ...	meta-variables ranging over delegation request objects and delegation response objects
dr, dr', ...	meta-variables ranging over delegation requests and delegation responses
$\mathcal{E}$, $\mathcal{E}'$, ...	meta-variables ranging over partitions of sets of events
E, E', E_1, ...	meta-variables ranging over sets of events
e, e', e_1, ...	meta-variables ranging over events

ED	meta-variable ranging over sets of decisions exchanged between coordinator models and enforcer models
$ed, ed', \ldots$	meta-variables ranging over $\mathcal{L}(policy)$
$ed, ed', \ldots$	meta-variables ranging over decisions
EE	meta-variable ranging over sets of effectable events
ENF	meta-variable for enforcer models
$EnF, EnF', \ldots$	meta-variables ranging over enforcer factory class names
$eo, eo', \ldots$	meta-variables ranging over event objects, i.e., objects of class Event
$\mathcal{EQ}$	meta-variable ranging over sets of process equations
$EvF, EvF', \ldots$	meta-variables ranging over event factory class names
f, g	meta-variables ranging over functions of underspecified type
$\varphi, \psi, \varphi', \ldots$	meta-variables ranging over formulas
$files$	meta-variable ranging over file maps of CoDSPL policies
fp	meta-variable ranging over fixed points mapping process names to processes
fs	meta-variable ranging over function symbols
$glob$	meta-variable ranging over global configurations
$I, \mathcal{J}$	meta-variables ranging over index sets of underspecified type
i, j, k	meta-variables ranging over unit identifiers
i, j	meta-variables ranging over index variables, i.e., elements of index sets
$\mathfrak{I}$	meta-variable ranging over first-order logic interpretations
Id	meta-variable ranging over sets of unit identifiers
$id, id', \ldots$	meta-variables ranging over unit identifiers
$idl, idl', \ldots$	meta-variables ranging over lists of identifiers
$Ids \subseteq \mathcal{L}(Id)$	meta-variable ranging over identifier sets
IE	meta-variable ranging over sets of intercepted events
ii	meta-variable ranging over interceptor instances
$k \in \mathcal{W}$	meta-variable ranging over words that syntactically occur as keys in key-value pairs of encapsulation descriptions
$kv \in \mathcal{L}(keyvalue)$	meta-variable ranging over syntactic key-value pairs of encapsulation descriptions
loc	meta-variable ranging over local configurations
$LP, LP', \ldots$	meta-variables ranging over local policy class names
$lp, lp', \ldots$	meta-variables ranging over local policy objects
M	meta-variable ranging over general sets of messages for channels in CSP

$m, m', \dots$	meta-variables ranging over messages on channels in CSP
$m, m', \dots$	meta-variables ranging over intercepted methods
N	meta-variable ranging over sets of process names
NAME	meta-variable ranging over process names
$o, o', \dots$	meta-variables ranging over file objects in the application scenario of Section 8.5
$o, o', \dots$	meta-variables ranging over objects
$P, Q, P', \dots$	meta-variables ranging over process expressions
$P, P', P_i, \dots$	meta-variables ranging over security properties formalized as unary predicates on sequences of events
$p, p', \dots$	meta-variables ranging over posts in DOSNs
$pcname, pcname', \dots$	meta-variables ranging over pointcut names
$v \in \mathcal{L}(PointcutSpec)$	meta-variable ranging over words from the pointcut specification language
pd	meta-variable ranging over words from the pointcut declaration language
pe	meta-variable ranging over words from the pointcut expression language
$\pi, \pi', \dots$	meta-variables ranging over re-share paths in DOSNs
pm	meta-variable ranging over policy models
POL	meta-variable for local policy models
$pol \in POL$	meta-variable ranging over well-formed CoDSPL policies
$pp, pp', \dots$	meta-variables ranging over privacy policies
$pps : \mathcal{USER} \rightharpoonup PP$	meta-variables ranging over families of users' privacy policies
Q, Q'	meta-variables ranging over sets of states of a security automaton
q	meta-variables ranging over states of a process expression DEC_i
q	meta-variable ranging over individual states of a security automaton
Q_0	meta-variable ranging over sets of initial states of a security automaton
R	meta-variable ranging over n-ary relations
rel	meta-variables ranging over category assignments of a DOSN user
$resp, resp', \dots$	meta-variables for responsibility functions in static delegation
rs	meta-variable ranging over relation symbols
$\mathcal{S}$	meta-variable ranging over signatures

$s, s', \ldots$	meta-variables ranging over sensitivity values of posts in DOSNs
sc	meta-variable ranging over sensitivity coefficients for the re-share operation in DOSNs
sf, sf'	meta-variable ranging over substitution functions, i.e., partial functions from words to words
$sp \in SP$	meta-variables ranging over identifiers of service providers
t	meta-variable ranging over terms in first-order logic
$t, t', t_1, \ldots$	meta-variables ranging over sequences
$Tr, Tr', Tr_1, \ldots$	meta-variables ranging over sets of possible traces of processes
$tr, tr', tr_1, \ldots$	meta-variables ranging over traces
tv	meta-variables ranging over a user's trust in categories
$u, u', \ldots$	meta-variables ranging over users in the application scenario of Section 8.5
$u, u', \ldots$	meta-variables ranging over identifiers of users in DOSNs
V	meta-variable ranging over sets of variable symbols
$v \in \mathcal{W}$	meta-variable ranging over words that syntactically occur as values in key-value pairs of encapsulation descriptions
v	meta-variables ranging over variable symbols in first-order logic
$w, u, v, w', \ldots$	meta-variables ranging over words
x, y	meta-variables ranging over event variables in process expressions
x, y, z	meta-variables ranging over elements of underspecified type
x	meta-variable ranging over variables for messages in CSP

P

privacy policies

$\mathcal{CAT}$	universe of all possible category names in a DOSN
$PATH$	set of all possible re-share paths in a DOSN
PP	set of all possible privacy policies in a DOSN
$PPS = \mathcal{USER} \rightharpoonup PP$	set of all families of users' privacy policies
$pt(pps, \pi, u)$	function computing the path trust
PC	relation capturing the connected re-share paths for a family of users' privacy policies
$\mathcal{USER}$	universe of all possible identifiers of users in a DOSN

S

semantics of CoDSPL policies

$[\![pol]\!]$	function mapping CoDSPL policies *pol* to encapsulated targets
$[\![ed]\!]$	function mapping words $ed \in \mathcal{L}(policy)$ to their semantics
advt	the advice template shown in Listing 5.1 on page 64
AspectJ	partial function capturing the functionality of AspectJ when applied to an aspect and a given JAR file
aspt	the aspect template shown in Listing 5.2 on page 64
cdFixed	function mapping the names of files belonging to the fixed implementation of the decider program, i.e., of the coordinator and its start-up, to the Java bytecode content of these files
$crossline(pol, id)$	partial function capturing cross-lining for a CoDSPL policy *pol* and a unit identifier *id*
EA	the set modeling all possible encapsulated agents
$genAspect(pol, id)$	partial function capturing the generation of an aspect specifying the interceptor and enforcer components of the unit with unit identifier *id* for CoDSPL policy *pol*
$genDecider(pol, id)$	partial function capturing the generation of a decider program for the unit identifier *id* in CoDSPL policy *pol*
$inst(w, sf)$	total function capturing the instantiation of a template w based on a substitution function *sf*
$jar(files)$	function taking a file map, *files*, and returning a JAR file containing all files in the domain of *files*, as implemented by the functionality of the jar program
$kvmap(ed, w)$	the function mapping all suffixes of keys in encapsulation description *ed* prefixed with word w to their value
POL	domain of well-formed CoDSPL policies
$idsl(idl)$	function that maps a list *idl* of identifiers to the set of identifiers in *idl*

sequences

$\langle x_1, \ldots, x_n \rangle$	the sequence consisting of the single elements x_1 to x_n
A^*	the set of all sequences over the set A
A^+	the set of all non-empty sequences over the set A, i.e., $A^* \setminus \{\langle\,\rangle\}$
$\langle\,\rangle$	the empty sequence, where the domain of $\langle\,\rangle$ is determined by the context in which the symbol is used
$\lvert t \rvert$	the length of sequence t
$t_1. \ldots .t_n$	the sequence resulting from concatenating the sequences t_1 to t_n

$x \lhd t$	short-hand notation for $\exists t_1, t_2 : (t = t_1.\langle x \rangle.t_2)$
$x \not\lhd t$	short-hand notation for $\neg(x \lhd t)$
$t \preceq t'$	the prefix relation on sequences (t is a prefix of t'
$t \upharpoonright A$	the projection of sequence t to set A
syntax of encapsulation descriptions	
addrkey	non-terminal capturing those keys in an encapsulation description for specifying units' network addresses
'`cfg.crypto`'	key for specifying whether communication between units shall be encrypted
'`cfg.destdir`'	key for holding the destination directory of the encapsulated distributed target in encapsulation descriptions
'`cfg.units`'	key for holding the list of unit identifiers in encapsulation descriptions
classkey	non-terminal capturing those keys in an encapsulation description for specifying class names
cpkey	non-terminal capturing those keys in an encapsulation description for specifying classpaths containing component implementations and their dependencies
enckey	non-terminal capturing those keys in an encapsulation description for specifying how in-memory objects are encoded when transmitted over network connections
FileContent	domain of file contents
filekey	non-terminal capturing those keys in an encapsulation description for referring to file names
Id	non-terminal capturing a single unit identifier in an encapsulation description
keyvalue	non-terminal capturing a key-value pair of an encapsulation description
'`pointcuts`'	key for holding the path to the file containing the specification of security-relevant program operations
policy	non-terminal capturing the language of encapsulation descriptions
portkey	non-terminal capturing those keys in an encapsulation description for specifying units' network ports
'`target`'	key for holding the path to the JAR file that holds the Java bytecode of a target agent's implementation
syntax of security-relevant program operations	
PointcutDecl	non-terminal capturing the language of individual pointcut declarations
PointcutExpr	non-terminal capturing the language of pointcut expressions

PointcutSpec	non-terminal capturing the language of pointcuts

Index