An Automatic Inference of Minimal Security Types

Dominik Bollmann, Steffen Lortz, Heiko Mantel, and Artem Starostin

Computer Science Department, TU Darmstadt, Germany
<lastname>@mais.informatik.tu-darmstadt.de

Abstract Type-based information-flow analyses provide strong end-to-end confidentiality guarantees for programs. Yet, such analyses are not easy to use in practice, as they require all information containers in a program to be annotated with security types, which is a tedious and error-prone task — if done manually. In this article, we propose a new algorithm for inferring such security types automatically. We implement our algorithm as an Eclipse plug-in, which enables software engineers to use it for verifying confidentiality requirements in their programs. We experimentally show our implementation to be effective and efficient.
We also analyze theoretical properties of our security-type inference algorithm. In particular, we prove it to be sound, complete, minimal, and of linear time-complexity in the size of the program analyzed.

1 Introduction

We present a solution for verifying confidentiality requirements in Java programs. Our solution consists of a type system for verifying information-flow security, a language for annotating sources, sinks, and other information containers, and an algorithm for inferring such annotations. We implement our solution as an Eclipse plug-in, and our experimental evaluation shows that it significantly outperforms prior solutions. We prove that our solution is sound and minimal.

Our solution runs in $O(n)$ time, where n is the size of the input program. It requires annotations of sources to be fixed, while allowing annotations of sinks and all other information containers to be flexible. Other solutions that run in $O(n)$ time require either annotations of all information containers to be fixed (see, e.g., [30]), or at least annotations of all sources and sinks to be fixed (see, e.g., [10]). On the other side of the spectrum, principal types [12, 13, 29] provide enough information for verifying a program against arbitrary annotations of sources and sinks. A disadvantage of principal types is that their construction requires $O(nv^3)$ time, where n is the size of the input program and v is the number of its variables [13]. A conceptual novelty of our solution is that, despite it runs in $O(n)$ time, it achieves minimality, similarly to principal types.

The soundness of our security analysis might not be a distinctive feature because there are other information-flow analyses that have been proven sound (e.g., [1, 2, 12, 29–31]), but it is an important one. However, there are also well-known information-flow analyses for which no soundness result exists (e.g.,

Published in:
S. Jajodia and C. Mazumdar (Eds.): ICISS 2015, LNCS 9478, pp. 395-415, 2015

The final authenticated version is available online at
https://doi.org/10.1007/978-3-319-26961-0_24

[4, 8, 18]). We consider soundness a crucial attribute, because without it, the guarantees established by a security analysis are unclear.

We implemented our solution as an Eclipse plug-in Adele (Assistant for Developing Leak-free Programs). It supports developers in writing Java programs with secure information flow. Adele analyzes the source code in the background, fully automatically, and reports detected information leaks. Due to the minimality result, Adele provides developers with an overview of all potential sinks to which the confidential information flows. This overview enables an informed navigation in the decision space for refactoring the program into a leak-free one.

We experimentally evaluated our solution at a spectrum of Java programs. We observed that for a single manually annotated information container, our algorithm infers security types for up to 128 other containers. Hence, our algorithm reduces the burden of manual security-type annotation by up to two orders of magnitude. Regarding performance, our experiments suggest that our solution needs, on average, less than 0.02 ms to analyze a line of source code. We also wanted to compare, in practice, the performance of our solution with that of principal types. Unfortunately, we did not find any implementation of principal types that we could have used in an experimental comparison. The other most flexible sound algorithm for inferring security types [27], that we are aware of, is implemented in SecJ [26]. Hence, we used it as a point of comparison. We experimentally compared the performance of our solution and SecJ, which revealed ours to be two orders of magnitude faster (in addition to being more flexible).

In summary, the novelties of this article are both conceptual and practical. Conceptually, we show how to achieve minimality without having to use principal types. Practically, we present a solution for the verification of confidentiality requirements in Java programs that is sound and flexible, and we experimentally demonstrate it to be effective and efficient.

The article is structured as follows. In Section 2, we define the Java subset that we focus on. In Section 3, we present our language for annotating information containers, our type system for verifying information-flow security, and a soundness result for the type system. In Section 4, we introduce our type-inference algorithm. In Section 5, we provide soundness, completeness, minimality, and complexity results for our algorithm. In Section 6, we present the implementation of our solution. In Section 7, we experimentally evaluate our solution. After a discussion of related work in Section 8, we conclude in Section 9.

Adele, its source code, and our benchmark programs are available for download under the MIT license at `www.mais.informatik.tu-darmstadt.de/adele`.

2 Programming Language

We focus on a sequential object-oriented fragment of Java with recursive method calls. Let underspecified sets $\mathcal{C}$, $\mathcal{M}$, $\mathcal{F}$, and $\mathcal{X}$ denote the sets of class, method, field, and variable names, respectively. Let $\mathcal{M} \cap \mathcal{F} = \emptyset$, let $this, result \in \mathcal{X}$, and let $Object \in \mathcal{C}$. We define the sets of data types T, expressions E, statements S, method definitions M, and class definitions C by the BNF in Figure 1, where

$C, D \in \mathcal{C}$, $x \in \mathcal{X}$, $f \in \mathcal{F}$, $m \in \mathcal{M}$, and overlined terms, e.g., $\overline{\mathsf{T}\ x}$, denote arbitrarily but finitely many repetitions of the term. We define a program as $P \subseteq \mathsf{C}$.

A data type is the primitive type `boolean` or a class name from $\mathcal{C}$. An expression is a literal expression `null`, `true`, or `false`, a variable access x, a field access $e.f$, an equality check e_1 == e_2, a type check e `instanceof` C, or a cast $((C)\ e)$. A statement is a field assignment $e_1.f = e_2$, variable assignment $x = e$, instance creation $x =$ `new` $C()$, method call $x = e.m(e_1, \ldots, e_n)$, variable declaration $T\ x = e;\ S$, conditional branching `if` (e) { S_1; } `else` { S_2; }, or sequential composition $S_1; S_2$. In a method definition $T\ m(T_1\ x_1,\ \ldots,\ T_n\ x_n)$ { T *result*; S; `return` *result*; }, m denotes the method name, $T_1\ x_1,\ \ldots,\ T_n\ x_n$ denote the formal parameters with their data types, S denotes the method body, and T denotes the data type of the return value. In a class definition `class` C `extends` D { $T_1\ f_1;\ \ldots;\ T_i\ f_i;\ M_1\ \ldots\ M_j$ }, C denotes the class name, D denotes the name of the immediate superclass of C, $T_1\ f_1, \ldots, T_i\ f_i$ denote the field declarations of C with their data types, and $M_1, \ldots, M_j$ denote the method definitions of C.

$$
\begin{aligned}
\mathsf{T} &::= \texttt{boolean} \mid C \\
\mathsf{E} &::= \texttt{null} \mid \texttt{true} \mid \texttt{false} \mid x \mid \mathsf{E}.f \mid \mathsf{E} == \mathsf{E} \\
&\quad \mid \mathsf{E}\ \texttt{instanceof}\ C \mid ((C)\ \mathsf{E}) \\
\mathsf{S} &::= \mathsf{E}.f = \mathsf{E} \mid x = \mathsf{E} \mid x = \texttt{new}\ C() \mid x = \mathsf{E}.m(\overline{\mathsf{E}}) \\
&\quad \mid \mathsf{T}\ x = \mathsf{E};\ \mathsf{S} \mid \texttt{if}\ (\mathsf{E})\ \{\ \mathsf{S};\ \}\ \texttt{else}\ \{\ \mathsf{S};\ \} \mid \mathsf{S}; \mathsf{S} \\
\mathsf{M} &::= \mathsf{T}\ m(\overline{\mathsf{T}\ x})\ \{\ \mathsf{T}\ \mathit{result};\ \mathsf{S};\ \texttt{return}\ \mathit{result};\ \} \\
\mathsf{C} &::= \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{\mathsf{T}\ f};\ \ \overline{\mathsf{M}}\ \}
\end{aligned}
$$

Figure 1: Programming language syntax.

Class definitions specify the inheritance hierarchy: For all classes $C, D \in \mathcal{C}$ defined in a program P, C is a subclass of D, written $C \leq_P D$, if and only if $D = C$ or another class $D' \in \mathcal{C}$ is defined, such that D' is the immediate superclass of C and $D' \leq_P D$. A subclass C of a class D inherits all field declarations and method definitions from D. If C defines a method with the same name as in D, then the method is overridden by the new definition from C. We assume that *Object* is the common superclass of all classes in a program, and that it does not declare any fields or define any methods.

We call a program *well-formed* if (1) it satisfies type-safety conditions commonly imposed by Java compilers, (2) each class has a unique name, fields and methods have unique names within each class, and local variables and formal parameters have unique names within each method, and (3) field names declared in a class are not reused in field declarations of its subclasses, and methods are only overridden by methods that declare the same formal parameters with the same data types. In this article, we assume all programs to be well-formed.

The uniqueness of names allows identifying classes within a program P by elements of $\mathcal{C}$, fields by elements of $\mathsf{FID} = \mathcal{F} \times \mathcal{C}$, methods by elements of $\mathsf{MID} = \mathcal{M} \times \mathcal{C}$, and variables by elements of $\mathsf{VID} = \mathcal{X} \times \mathcal{M} \times \mathcal{C}$. For all $C \in \mathcal{C}$ and $f \in \mathcal{F}$, the partial function $\mathsf{fieldsof}_P : \mathcal{C} \rightharpoonup \mathcal{P}(\mathcal{F})$ is defined, such that $C \in \mathit{dom}(\mathsf{fieldsof}_P)$ if and only if P contains a definition of class C, and $f \in \mathsf{fieldsof}_P(C)$ if and only if class C in P declares or inherits field f. For all $C \in \mathcal{C}$ and $m \in \mathcal{M}$, the partial function $\mathsf{methodsof}_P : \mathcal{C} \rightharpoonup \mathcal{P}(\mathcal{M})$ is defined, such that $C \in \mathit{dom}(\mathsf{methodsof}_P)$ if

and only if P contains a definition of class C, and $m \in \mathsf{methodsof}_P(C)$ if and only if class C in P defines or inherits method m. For all $C \in \mathcal{C}$, $m \in \mathcal{M}$, and $x \in \mathcal{X}$, the partial function $\mathsf{varsof}_P : \mathsf{MID} \rightharpoonup \mathcal{P}(\mathcal{X})$ is defined, such that $(m, C) \in dom(\mathsf{varsof}_P)$ if and only if $m \in \mathsf{methodsof}_P(C)$, $x \in \mathsf{varsof}_P(m, C)$ if and only if formal parameter x is declared by method m defined or inherited by class C in P, or local variable x is declared by method m defined by class C in P. The set of *defined identifiers* in P is $\mathsf{names}_P = \{(x, m, C) \in \mathsf{VID} \mid x \in \mathsf{varsof}_P(m, C)\} \cup \{(f, C) \in \mathsf{FID} \mid f \in \mathsf{fieldsof}_P(C)\} \cup \{(m, C) \in \mathsf{MID} \mid m \in \mathsf{methodsof}_P(C)\}$.

The semantics of the language in Figure 1 corresponds to that of a syntactically equivalent Java subset.

3 A Type System for Verifying Information-Flow Security

We define a security type system in the spirit of [1] for the language from Section 2. This type system ensures that confidential information does not flow to untrusted sinks during a program execution. Which containers store confidential and which store public information is specified by security-type annotations.

3.1 An Annotation Language and Information-Flow Policy

To specify between which information containers information may flow, every information container in a program may be annotated with a security-type annotation `@High` or `@Low`. Such annotations induce an information-flow policy.

An information-flow policy (brief: policy) defines a set of security domains $\mathcal{D}$, an interference relation $\sqsubseteq \subseteq \mathcal{D} \times \mathcal{D}$, and a domain assignment $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$. The security domains (brief: domains) from the set $\mathcal{D}$ denote abstract levels of confidentiality. The interference relation is a partial order on security domains that specifies between which domains information may flow. The domain assignment associates some information containers in a program with a security domain. A policy defines the permitted flows of information between the information containers: For any two containers $a, b \in \mathsf{VID} \cup \mathsf{FID}$ with $\mathsf{da}(a) = d$ and $\mathsf{da}(b) = d'$, information from a may be written into b if and only if $d \sqsubseteq d'$. We assume a two-level information-flow policy $(\mathcal{D}, \mathsf{da}, \sqsubseteq)$ with the security domains $\mathcal{D} = \{low, high\}$ and the interference relation $\sqsubseteq = \{(low, low), (low, high), (high, high)\}$. This policy allows expressing that confidential information must not leak to untrusted sinks of a program. While we focus on the two-level policy, an extension to arbitrary lattices is straightforward.

The domain assignment is induced from the security-type annotations of a concrete program as follows. For any program P with security-type annotations, the *annotation-induced domain assignment* $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$ is defined, such that for all $x \in \mathcal{X}$, $m \in \mathcal{M}$, $C \in \mathcal{C}$, and $f \in \mathcal{F}$: (1) $\mathsf{da}(f, C)$ is defined if and only if program P contains class C that declares field f, and the declaration is annotated with either `@High` or `@Low`, (2) if $\mathsf{da}(f, C)$ is defined, $\mathsf{da}(f, C) = high$ if the declaration of field (f, C) in program P is annotated with `@High`, and $\mathsf{da}(f, C) = low$ otherwise, (3) $\mathsf{da}(x, m, C)$ is defined if and only if program P contains class C that defines method m, and in the definition, the declaration of

variable x is annotated with either `@High` or `@Low`, (4) if $\mathsf{da}(x, m, C)$ is defined, $\mathsf{da}(x, m, C) = \mathit{high}$ if the declaration of variable (x, m, C) is annotated with `@High`, and $\mathsf{da}(x, m, C) = \mathit{low}$ otherwise. An information-flow policy $(\mathcal{D}, \mathsf{da}, \sqsubseteq)$ with an annotation-induced domain assignment intuitively requires for an annotated program that information obtained from information containers annotated with `@High` shall not flow to those annotated with `@Low`.

Due to inheritance and overriding, certain identifiers in names_P can be aliases of the same information container. To ensure that a domain assignment does not associate different security domains with such identifiers, we require any domain assignment for P to be *consistent for* P. For any set X, a partial function $g : \mathsf{names}_P \rightharpoonup X$ is consistent for P if and only if for all $C, D \in \mathcal{C}$ with $C \leq_P D$ it holds: (1) for all $f \in \mathsf{fieldsof}_P(D)$, if $(f, C) \in \mathit{dom}(g)$ and $(f, D) \in \mathit{dom}(g)$ then $g(f, C) = g(f, D)$, (2) for all $m \in \mathsf{methodsof}_P(D)$, if $(m, C) \in \mathit{dom}(g)$ and $(m, D) \in \mathit{dom}(g)$, then $g(m, C) = g(m, D)$, (3) for all $m \in \mathsf{methodsof}_P(D)$ and $x \in \{x \mid (x_1, \ldots, x_n) = \mathsf{pars}_P(m, D) \wedge \exists i \in \{1, \ldots, n\}.x = x_i\}$, if $(x, m, C) \in \mathit{dom}(g)$ and $(x, m, D) \in \mathit{dom}(g)$, then $g(x, m, C) = g(x, m, D)$, where the partial function $\mathsf{pars}_P : \mathsf{MID} \rightharpoonup \mathcal{X}^*$ is defined for $T\ m(T_1\ x_1, \ldots, T_n\ x_n)\ \{\ldots\}$ in the definition of any class $C \in \mathcal{C}$ in P, such that $\mathsf{pars}_P(m, C) = (x_1, \ldots, x_n)$, and $\mathsf{pars}_P(m, C) = \mathsf{pars}_P(m, D)$ if C inherits m from superclass $D \in \mathcal{C}$.

3.2 A Security Type System

A domain assignment assigns security domains to a subset of fields and variables in a program. Our security type system requires the domain assignment to be extended, so that all defined identifiers of fields, methods, and variables are associated with a security domain. A *complete typing* (brief: typing) of a program P is a function $\mathsf{t} : \mathsf{names}_P \to \mathcal{D}$ that is consistent for P. Intuitively, a typing of a program associates all variables, fields, and methods of the program with security domains, such that all identifiers that could be aliases of the same field, method, or variable are assigned the same domain. We call typing t *compatible* with domain assignment da if and only if for all $a \in \mathit{dom}(\mathsf{da})$ it holds $\mathsf{t}(a) = \mathsf{da}(a)$.

Our type system uses a function type_P to determine data types of information containers and expressions in a given program P.The definition of type_P relies on the partial functions $\mathsf{ftype}_P : \mathsf{FID} \rightharpoonup \mathsf{T}$ and $\mathsf{vtype}_P : \mathsf{VID} \rightharpoonup \mathsf{T}$ to determine data types of fields and local variables, respectively. $\mathsf{ftype}_P(f, C)$ is defined if and only if $f \in \mathsf{fieldsof}_P(C)$, and $\mathsf{ftype}_P(f, C) = T$ if f is declared with data type T in C, and otherwise $\mathsf{ftype}_P(f, C) = \mathsf{ftype}_P(f, D)$, where D is the immediate superclass of C. $\mathsf{vtype}_P(x, m, C)$ is defined if and only if $x \in \mathsf{varsof}_P(m, C)$, and $\mathsf{vtype}_P(x, m, C) = T$ if x is declared with data type T in method m defined by C, and otherwise $\mathsf{vtype}_P(x, m, C) = \mathsf{vtype}_P(x, m, D)$, where D is the immediate superclass of C.

$$
\begin{aligned}
\mathsf{type}_P(x, m, C) &= \mathsf{vtype}_P(x, m, C) \\
\mathsf{type}_P(e_1.f, m, C) &= \mathsf{ftype}_P(f, \mathsf{type}_P(e_1, m, C)) \\
\mathsf{type}_P(((T)\ e_1), m, C) &= T \\
\mathsf{type}_P(\mathtt{null}, m, C) &= \mathit{Object} \\
\mathsf{type}_P(e, m, C) &= \mathtt{boolean}, \text{ for all other } e
\end{aligned}
$$

Figure 2: Data types of expressions.

Finally, the partial function $\mathsf{type}_P : \mathsf{E} \times \mathsf{MID} \rightharpoonup \mathsf{T}$ is defined in Figure 2, where $e, e_1, e_2 \in \mathsf{E}$, $T \in \mathsf{T}$, $x \in \mathcal{X}$, $f \in \mathcal{F}$, $m \in \mathcal{M}$, and $C \in \mathcal{C}$.

For a given program P and function $\gamma : \mathsf{names}_P \to Y$, we use *method signatures* $\mathsf{msig}_P^\gamma : \mathsf{MID} \rightharpoonup Y^*$ to denote the values that γ associates with a method's formal parameters, return value, and heap effect, e.g., in the signature $\mathsf{msig}_P^{\mathsf{t}}(m, C) = \langle d_t, (d_1, \ldots, d_n) \xrightarrow{d_h} d_r \rangle$ of method (m, C) wrt. typing t, d_t and d_r denote the security domains associated with *this* and *result*, respectively, $d_1, \ldots, d_n$ denote the domains associated with the method's parameters, and d_h denotes the domain associated with the method's heap effect.

Whether a program is typable wrt. a typing is defined by a set of security typing rules. A selection of our security typing rules corresponding to object-oriented features is presented in Figure 3. In these rules, the judgment for expressions is denoted by $m, C, P; \mathsf{t} \vdash e : d$, where m, C, P denote the context in which the expression e is evaluated, and d denotes the security domain of the value the expression evaluates to wrt. typing t of P. The judgment for statements is denoted by $m, C, P; \mathsf{t} \vdash S : (d', \kappa')$, where $S \in \mathsf{S}$ denotes a statement and $d', \kappa' \in \mathcal{D}$ denote security domains. The judgment for method definitions is denoted by $C, P; \mathsf{t} \vdash M$, where C denotes the class of program P in which the method is defined. The judgment for typing program P wrt. complete typing t of P is denoted by $\mathsf{t} \vdash P$. It is derivable if the judgment for method definitions is derivable wrt. t, for all method definitions in all class definitions in P. We say that program P is *accepted* by our security type system wrt. complete typing $\mathsf{t} : \mathsf{names}_P \to \mathcal{D}$ for P if and only if the judgment $\mathsf{t} \vdash P$ is derivable.

If a program is accepted by the type system wrt. a complete typing of the program, the typing is an approximation of the possible distribution of confidential information during program's execution. Intuitively, (1) each security domain associated by the typing with an information container is an upper bound on the security domains of containers from which information may flow into this one, and (2) each security domain associated by the typing with a method is a lower bound on all security domains of fields that the method may write.

$$\frac{m, C, P; \mathsf{t} \vdash e_1 : d_1 \quad d = \mathsf{t}(f, \mathsf{type}_P(e_1, m, C)) \quad d_1 \sqsubseteq d' \quad d \sqsubseteq d'}{m, C, P; \mathsf{t} \vdash e_1.f : d'}$$

$$\frac{m, C, P; \mathsf{t} \vdash e_1 : d_1 \quad m, C, P; \mathsf{t} \vdash e_2 : d_2 \quad d = \mathsf{t}(f, \mathsf{type}_P(e_1, m, C)) \quad d_1 \sqsubseteq d \quad d_2 \sqsubseteq d \quad \kappa' \sqsubseteq d}{m, C, P; \mathsf{t} \vdash e_1.f = e_2 : (d', \kappa')}$$

$$\frac{\begin{array}{c} m, C, P; \mathsf{t} \vdash e_1 : d_1 \quad \ldots \quad m, C, P; \mathsf{t} \vdash e_n : d_n \\ \mathsf{msig}_P^{\mathsf{t}}(m_2, \mathsf{type}_P(e_1, m, C)) = \langle d_t', (d_2', \ldots, d_n') \xrightarrow{d_h'} d_r' \rangle \\ d_x = \mathsf{t}(x, m, C) \qquad \forall i \in \{2, \ldots, n\}.d_i \sqsubseteq d_i' \\ d_r' \sqsubseteq d_x \quad d_1 \sqsubseteq d_t' \quad d_1 \sqsubseteq d_h' \quad d_1 \sqsubseteq d_x \quad d' \sqsubseteq d_x \quad \kappa' \sqsubseteq d_h' \end{array}}{m, C, P; \mathsf{t} \vdash x = e_1.m_2(e_2, \ldots, e_n) : (d', \kappa')}$$

$$\frac{m, C, P; \mathsf{t} \vdash S : (d', \kappa') \quad \mathsf{t}(m, C) \sqsubseteq \kappa'}{C, P; \mathsf{t} \vdash T_r\ m(\ldots)\ \{\ T_r\ \mathit{result};\ S;\ \mathtt{return}\ \mathit{result};\}}$$

Figure 3: Selected security typing rules.

3.3 Soundness of the Security Type System

We prove the soundness of our security type system wrt. a security property in the style of Noninterference [9]. For an execution of a single method, our noninterference-like security property intuitively requires that the information stored in *low* return values and *low* object fields on the resulting heap is independent from the information stored in *high* formal parameters and *high* object fields on the initial heap. Which information containers are *low* or *high* is given by a typing. If all executions of a method respect our noninterference-like security property, we call such a method *noninterfering wrt. a typing*. A program is *noninterfering wrt. a typing*, if all its methods are noninterfering wrt. the typing.

Theorem 1 (Soundness of the Security Type System). Let $P \subseteq \mathsf{C}$ be a program and $\mathsf{t} : \mathsf{names}_P \to \mathcal{D}$ be a complete typing for P. If $\mathsf{t} \vdash P$ is derivable, then P is noninterfering wrt. t.

4 Our Security-Type Inference Algorithm

The type system from Section 3 requires a complete typing of a program for verification of the program's information-flow security. In this section, we define our security type inference algorithm to automatically infer, for a given program and a domain assignment, a complete typing for the program that is compatible with the domain assignment. The algorithm consists of four steps: (1) *Assignment of security type variables*: Associate each information container and method in the program not associated with a security domain by the domain assignment with a type variable. (2) *Derivation of constraints*: Derive constraints from the program that an inferred typings has to satisfy, so that the program is accepted wrt. the inferred typing by the security type system. (3) *Constraint solving*: Assign a domain to each type variable, so that all constraints are satisfied. (4) *Inferring a typing*: If constraint solving was successful, output a typing, and an error value indicating failure, otherwise. Section 4.1 to Section 4.4 present Step 1 to Step 4, respectively.

4.1 Assignment of Security Type Variables

Let $\mathcal{V}$ denote the infinite *set of type variables*. Each information container and method in a given program, not associated with a domain by a given domain assignment, is associated with a type variable by the security context of the program and domain assignment. Let $\mathsf{typevar} : \mathsf{VID} \cup \mathsf{FID} \cup \mathsf{MID} \to \mathcal{V}$ be an arbitrary but fixed injective function assigning type variables to identifiers of variables, fields, and methods. A *security context* for program P and domain assignment da is a function $\sigma : \mathsf{names}_P \to \mathcal{D} \cup \mathcal{V}$, such that:

- for all $(f, C) \in \mathsf{FID} \cap \mathsf{names}_P$ it holds that (1) if $D \in \mathcal{C}$ exists, so that $(f, D) \in dom(\mathsf{da})$ and $C \leq_P D \vee D <_P C$, then $\sigma(f, C) = \mathsf{da}(f, D)$, else (2) if $D \in \mathcal{C}$ exists, so that $f \in \mathsf{fieldsof}_P(D)$ and $C <_P D$, then $\sigma(f, C) = \sigma(f, D)$, and (3) $\sigma(f, C) = \mathsf{typevar}(f, C)$, otherwise,

– for all $(m, C) \in \mathsf{MID} \cap \mathsf{names}_P$ it holds that (1) if $D \in \mathcal{C}$ exists, so that $m \in \mathsf{methodsof}_P(D)$ and $C <_P D$, then $\sigma(m, C) = \sigma(m, D)$, and (2) $\sigma(m, C) = \mathsf{typevar}(m, C)$, otherwise, and
– for all $(x, m, C) \in \mathsf{VID} \cap \mathsf{names}_P$ it holds that (1) if $(x, m, C) \in dom(\mathsf{da})$, then $\sigma(x, m, C) = \mathsf{da}(x, m, C)$, else (2) if $D \in \mathcal{C}$ exists, so that $(x, m, D) \in dom(\mathsf{da})$, $C <_P D \vee D <_P C$, and $x \in \{x \mid (x_1, \ldots, x_n) = \mathsf{pars}_P(m, D) \wedge \exists i \in \{1, \ldots, n\}.x = x_i\}$, then $\sigma(x, m, C) = \mathsf{da}(x, m, D)$, else (3) if $D \in \mathcal{C}$ exists, so that $m \in \mathsf{methodsof}_P(D)$, $C <_P D$, and $x \in \{x \mid (x_1, \ldots, x_n) = \mathsf{pars}_P(m, D) \wedge \exists i \in \{1, \ldots, n\}.x = x_i\}$, then $\sigma(x, m, C) = \sigma(x, m, D)$, and (4) $\sigma(x, m, C) = \mathsf{typevar}(x, m, C)$, otherwise.

The first condition requires the security context to assign to each field identifier (1) the same security domain that da assigns to an alias of the field, (2) the same security type variable the security context assigns to the same field in a super class, or (3) a unique security type variable if the field is declared in the class denoted by the identifier. The second and third conditions impose similar requirements for method identifiers and variable identifiers, respectively. The third condition distinguishes between formal parameters and local variables, since only parameters can be aliases of each other, whereas local variables are only accessible within the declaring method definition.

A security context agrees with the corresponding domain assignment for all field and variable identifiers, for which the domain assignment is defined, by construction. All identifiers that are not associated with a security domain based on the domain assignment are assigned a security type variable. The set of type variables in the range of σ is denoted by $\mathsf{typevars}_\sigma = \{\alpha \in \mathcal{V} \mid \alpha \in rng(\sigma)\}$. The set $\mathsf{typevars}_\sigma$ denotes the set of type variables for which constant security domains have to be inferred to obtain a complete typing of the program P.

4.2 Derivation of Constraints

In the second step of our security-type inference algorithm, constraints on type variables are derived that a typing of a program has to satisfy, so that the program is accepted wrt. the typing by the security type system from Section 3. We use the notation for constraints and derivations rules in the spirit of [27].

Constraints. We denote constraints on type variables by constraint formulas. A *constraint formula* (brief: constraint) is a term $\lambda \preceq \lambda'$, where $\preceq$ is a binary relation symbol and $\lambda, \lambda' \in \mathcal{D} \cup \mathcal{V}$ are either security domains or type variables. The set $\mathcal{K}_V$ of all constraint formulas over some set of type variables $V \subseteq \mathcal{V}$ is defined by $\mathcal{K}_V = \{\lambda \preceq \lambda' \mid \lambda, \lambda' \in \mathcal{D} \cup V\}$. Intuitively, a constraint formula $\lambda \preceq \lambda'$ requires that information is permitted to flow from the security domain denoted by λ to the security domain denoted by λ'. A *constraint scheme* is a pair (V, K) of a finite set of type variables $V \subseteq \mathcal{V}$ and a set of constraint formulas $K \subseteq \mathcal{K}_V$ over the type variables in V. The set $\mathcal{S}$ of all constraint schemes is defined by $\mathcal{S} = \{(V, K) \mid V \subseteq \mathcal{V} \wedge |V| \in \mathbb{N}_0 \wedge K \subseteq \mathcal{K}_V\}$.

Constraint derivation rules. We define a set of derivation rules that analyze the possible flow of information through the program and generate a constraint

$$\frac{m, C, P; \sigma; V_0 \vdash e_1 : \alpha_1 \rightsquigarrow (V_1, K_1) \qquad \lambda = \sigma(f, \mathsf{type}_P(e_1, m, C)) \\ \alpha \in \mathcal{V} \setminus (V_0 \cup V_1) \qquad V = V_1 \cup \{\alpha\} \qquad K = K_1 \cup \{\alpha_1 \preceq \alpha, \lambda \preceq \alpha\}}{m, C, P; \sigma; V_0 \vdash e_1.f : \alpha \rightsquigarrow (V, K)}$$

$$\frac{m, C, P; \sigma; V_0 \vdash e_1 : \alpha_1 \rightsquigarrow (V_1, K_1) \qquad m, C, P; \sigma; V_0 \cup V_1 \vdash e_2 : \alpha_2 \rightsquigarrow (V_2, K_2) \\ \alpha, \beta \in \mathcal{V} \setminus (V_0 \cup V_1 \cup V_2) \qquad V = V_1 \cup V_2 \cup \{\alpha, \beta\} \\ \lambda = \sigma(f, \mathsf{type}_P(e_1, m, C)) \qquad K = K_1 \cup K_2 \cup \{\alpha_1 \preceq \lambda, \alpha_2 \preceq \lambda, \beta \preceq \lambda\}}{m, C, P; \sigma; V_0 \vdash e_1.f = e_2 : (\alpha, \beta) \rightsquigarrow (V, K)}$$

$$\frac{\begin{array}{c} m_1, C, P; \sigma; V_0 \vdash e_1 : \alpha_1 \rightsquigarrow (V_1, K_1) \\ \vdots \\ m_1, C, P; \sigma; \bigcup_{i \in \{0,\ldots,n-1\}} V_i \vdash e_n : \alpha_n \rightsquigarrow (V_n, K_n) \\ \langle \lambda_t, (\lambda_2, \ldots, \lambda_n) \xrightarrow{\lambda_h} \lambda_r \rangle = \mathsf{msig}^{\sigma}_P(m_2, \mathsf{type}_P(e_1, m_1, C)) \qquad \lambda_x = \sigma(x, m_1, C) \\ \alpha, \beta \in \mathcal{V} \setminus (\bigcup_{i \in \{0,\ldots,n\}} V_i) \qquad V = \bigcup_{i \in \{1,\ldots,n\}} V_i \cup \{\alpha, \beta\} \\ K = K_1 \cup \bigcup_{i \in \{2,\ldots,n\}} (K_i \cup \{\alpha_i \preceq \lambda_i\}) \cup \\ \{\alpha_1 \preceq \lambda_t, \alpha_1 \preceq \lambda_h, \alpha_1 \preceq \lambda_x, \alpha \preceq \lambda_x, \beta \preceq \lambda_h, \lambda_r \preceq \lambda_x\} \end{array}}{m_1, C, P; \sigma; V_0 \vdash x = e_1.m_2(e_2, \ldots, e_n) : (\alpha, \beta) \rightsquigarrow (V, K)}$$

$$\frac{m, C, P; \sigma; V_0 \vdash S : (\alpha_1, \beta_1) \rightsquigarrow (V_1, K_1) \qquad \lambda_h = \sigma(m, C) \qquad K = K_1 \cup \{\lambda_h \preceq \beta_1\}}{C, P; \sigma; V_0 \vdash T\ m(\ldots)\ \{\ T\ result;\ S;\ \texttt{return}\ result;\}\ \rightsquigarrow (V_1, K)}$$

Figure 4: Selected constraint derivation rules.

scheme with constraints imposed on an acceptable typing of the program. Most of the rules impose constraints on auxiliary type variables that are not in the security context. To ensure the uniqueness of these type variables, all rules, except the rule for programs take a set V_0 of already used type variables, to exclude when selecting a new auxiliary type variable. Selected constraint derivation rules for object-oriented features of our language are given in Figure 4.

The judgment for deriving constraint schemes from expressions is of the form $m, C, P; \sigma; V_0 \vdash e : \alpha \rightsquigarrow (V, K)$. It denotes that expression e in method (m, C) of program P under security context σ is associated with type variable α, and constraint scheme (V, K) specifies requirements on the security domain that α denotes. Intuitively, the constraint scheme derived from an expression requires that the domain denoted by α is an upper bound on the domains of all information containers from which information flows into the expression's value.

The judgment for deriving constraint schemes from statements is of the form $m, C, P; \sigma; V_0 \vdash S : (\alpha, \beta) \rightsquigarrow (V, K)$. It denotes that statement S in method (m, C) of program P under security context σ is associated with type variables α and β, and imposes constraints specified by constraint scheme (V, K). Intuitively, the derived constraints require that the auxiliary type variable α denotes a lower bound on all security domains of variables that the statement may write, β denotes a lower bound on all security domains of fields that the statement may write, and all security domains of information containers that the statement may write denote upper bounds on the security domains of the information written into the respective information container.

The judgment for deriving constraint schemes from method definitions is of the form $C, P; \sigma; V_0 \vdash T\ m(\dots)\ \{\ T\ result;\ S;\ \texttt{return}\ result;\} \rightsquigarrow (V, K)$. It denotes that the definition of method (m, C) in program P under security context σ imposes the constraints specified by constraint scheme (V, K). This constraint scheme contains the constraints derived from the body of the method and one additional constraint $\lambda_h \preceq \beta_1$, requiring that the security domain of the method's heap effect is a lower bound on the domains of the fields the execution of the method's body may write.

The constraint scheme derived from the definition of a class is comprised of the constraints imposed by the definitions of the methods of this class. The constraint scheme derived from a program is the union of all constraints of all defined methods of all classes in the program.

4.3 Constraint Solving

In the third step of our security-type inference algorithm, the constraints in a derived constraint scheme are solved. The objective is to determine a *variable valuation* associating the type variables in the constraints with security domains, so that all constraints are satisfied if interpreting the binary relation $\preceq$ as $\sqsubseteq$. A variable valuation is a total function $\mathrm{I} : V \to \mathcal{D}$, where $V \subseteq \mathcal{V}$ and V is finite. We denote the set of all variable valuations by $\mathcal{I}$. A variable valuation I is lifted to $\widehat{\mathrm{I}} : V \cup \mathcal{D} \to \mathcal{D}$ so that for all $\lambda \in V \cup \mathcal{D}$ that $\widehat{\mathrm{I}}(\lambda) = \lambda$ if $\lambda \in \mathcal{D}$ and $\widehat{\mathrm{I}}(\lambda) = \mathrm{I}(\lambda)$, otherwise. Hence, $\widehat{\mathrm{I}}$ associates all domains with themselves.

A constraint formula $\lambda \preceq \lambda' \in \mathcal{K}_V$ is satisfied by a variable valuation I, denoted by $\mathrm{I} \models \lambda \preceq \lambda'$, if and only if $\widehat{\mathrm{I}}(\lambda) \sqsubseteq \widehat{\mathrm{I}}(\lambda')$. For a set of constraint formulas $K \subseteq \mathcal{K}_V$, we write $\mathrm{I} \models K$ to denote that $\mathrm{I} \models \lambda \preceq \lambda'$ for all $\lambda \preceq \lambda' \in K$. Intuitively, constraint formula $\lambda \preceq \lambda'$ is satisfied by variable valuation I, if the interference relation $\sqsubseteq$ permits flows from the domain that I associates with the left operand to the domain that I associates with the right operand. A variable valuation satisfies a set of constraints if it satisfies all constraints in the set.

To solve a constraint scheme, we adopt a constraint solving algorithm of Rehof and Mogensen [21]. The algorithm takes a constraint scheme and either computes variable valuation I satisfying all constraints, or it determines that the constraint set is not satisfiable and outputs an error value $\bot$. We model this algorithm by the function $\mathsf{solve} : \mathcal{S} \to \mathcal{I} \cup \{\bot\}$.

4.4 Inferring a Typing

In the fourth step of our security-type inference algorithm, the results from the other steps are combined to infer a typing based on a program and a domain assignment. Our security-type inference algorithm takes a program and a domain assignment for the program as input, and either outputs a complete typing of the program, or an error value denoting that no typing could be inferred. We model our algorithm by the function $\mathsf{infer} : \mathcal{P}(\mathsf{C}) \times (\mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}) \to (\mathsf{names}_P \to \mathcal{D}) \cup \{\bot\}$ that is defined for any $P \subseteq \mathsf{C}$ and $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$ for P by

(1) $\mathsf{infer}(P, \mathsf{da}) = \widehat{\mathrm{I}} \circ \sigma$ if $(V, K) \in \mathcal{S}$ and $\mathrm{I} : V \to \mathcal{D}$ exist so that $\mathsf{solve}(V, K) = \mathrm{I}$ and $\sigma \vdash P \rightsquigarrow (V, K)$ is derivable under the security context $\sigma : \mathsf{names}_P \to \mathcal{D} \cup \mathcal{V}$ for P and da, and by
(2) $\mathsf{infer}(P, \mathsf{da}) = \bot$, otherwise.

5 Soundness, Completeness, Minimality, Complexity

In this section, we present the soundness, completeness, minimality and complexity results for our security-type inference algorithm.

5.1 Soundness

If our security-type inference algorithm infers a typing for a given program and domain assignment, then the program is accepted wrt. the inferred typing by the security type system from Section 3.

Lemma 1 (Correctness). Let $P \subseteq \mathsf{C}$ be a program, and $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$ be a domain assignment for P. If a complete typing $\mathsf{t} : \mathsf{names}_P \to \mathcal{D}$ of P exists, such that $\mathsf{infer}(P, \mathsf{da}) = \mathsf{t}$, then $\mathsf{t} \vdash P$ is derivable.

In order to express the soundness of our security-type inference algorithm, we need to lift our notion of noninterference from Section 3 to one wrt. a domain assignment. For a program P and a domain assignment $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$ for P, P is *noninterfering* wrt. da if and only if a complete typing $\mathsf{t} : \mathsf{names}_P \to \mathcal{D}$ exists, such that t is compatible with da and P is noninterfering wrt. t. If a program is noninterfering wrt. a typing t and t is compatible with a domain assignment da, then all outputs of the program into information containers that da associates with *low* are independent from information stored in containers that da associates with *high*. This holds because t agrees with da on all identifiers for which da is defined, and a noninterfering program wrt. t is guaranteed to have no flows of information from information containers that t associates with *high* to containers that t associates with *low*. If our algorithm infers a typing for a given program and domain assignment, then the program is noninterfering wrt. the domain assignment.

Theorem 2 (Soundness). Let $P \subseteq \mathsf{C}$ be a program, and $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$ be a domain assignment for P. If a complete typing $\mathsf{t} : \mathsf{names}_P \to \mathcal{D}$ exists, such that $\mathsf{infer}(P, \mathsf{da}) = \mathsf{t}$, then P is noninterfering wrt. da.

5.2 Completeness

The completeness result guarantees that our security-type algorithm always outputs a typing for a program and domain assignment, if the domain assignment can be extended to a complete typing of the program, such that the program is accepted wrt. the typing by our security type system.

Theorem 3 (Completeness). Let $P \subseteq \mathsf{C}$ be a program, and $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$ be a domain assignment for P. If $\mathsf{t} : \mathsf{names}_P \to \mathcal{D}$ exists, so that t is a complete typing of P, t is compatible with da, and $\mathsf{t} \vdash P$ is derivable, then $\mathsf{infer}(P, \mathsf{da}) \neq \bot$.

5.3 Minimality

In order to define the minimality of typings, we first introduce the interference relation $\sqsubseteq_P \subseteq (\mathsf{names}_P \to \mathcal{D}) \times (\mathsf{names}_P \to \mathcal{D})$ on typings of a program P, that is defined, such that for all typings $\mathsf{t}, \mathsf{t}' : \mathsf{names}_P \to \mathcal{D}$, $\mathsf{t} \sqsubseteq_P \mathsf{t}'$ if and only if $\mathsf{t}(a) \sqsubseteq \mathsf{t}'(a)$ for all $a \in \mathsf{names}_P$. For program P and domain assignment $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$ for P, a complete typing $\mathsf{t} : \mathsf{names}_P \to \mathcal{D}$ is *minimal* for P and da, if and only if for all typings $\mathsf{t}' : \mathsf{names}_P \to \mathcal{D}$, such that t' is a complete typing of P, t' is compatible with da, and $\mathsf{t}' \vdash P$ is derivable, it holds that $\mathsf{t} \sqsubseteq_P \mathsf{t}'$. Intuitively, a typing is minimal for a program and domain assignment, if it is a lower bound on all typings of the program that are compatible to the domain assignment and under which the program is accepted by our security type system. Our inference algorithm only infers typings that are minimal.

Theorem 4 (Minimality). Let $P \subseteq \mathsf{C}$ be a program, and $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$ be a domain assignment for P. If a typing $\mathsf{t} : \mathsf{names}_P \to \mathcal{D}$ exists such that $\mathsf{infer}(P, \mathsf{da}) = \mathsf{t}$, then t is minimal for P and da.

Intuitively, an inferred typing for a given program and domain assignment associates a domain with each information container that is a least upper bound on all domains of containers from which information may flow into this container. This offers two appealing opportunities for using our security-type inference algorithm (1) to explore where the confidential information in a program may flow, and (2) to verify a program against arbitrary annotations of sinks.

Exploring flows of confidential information. To use our security-type inference algorithm for exploring where confidential information in a program may flow, one annotates sources of confidential information, i.e., information containers from which confidential information is read, in the program with `@High`. Then the security-type inference algorithm infers the security domain *high* for all information containers to which these confidential inputs may flow, and *low* for all other information containers.

Verifying a program against arbitrary annotations of sinks. As long as the sources annotated with `@High` remain the same, one inferred typing for a program allows to verify the program against different annotations of sinks. Given an annotation-induced domain assignment $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$ for a program P and an inferred typing $\mathsf{t} : \mathsf{names}_P \to \mathcal{D}$ for a domain assignment that associates the same identifiers of sources with *high* as da, P is noninterfering wrt. da if $\mathsf{t}(a) \sqsubseteq \mathsf{da}(a)$ for all identifiers $a \in \mathit{dom}(\mathsf{da})$.

5.4 Computational Complexity

Our security-type inference algorithm and security type system analyze a program with a worst case time-complexity that is linear in the size of the program. As the *size of a program*, we consider the number of nodes in the program's abstract syntax tree.

Theorem 5 (Complexity). For any program $P \subseteq \mathsf{C}$ and domain assignment $\mathsf{da} : \mathsf{VID} \cup \mathsf{FID} \rightharpoonup \mathcal{D}$ for P, given a precomputed security context $\sigma : \mathsf{names}_P \to \mathcal{D} \cup \mathcal{V}$ for P and da, the security-type inference and security type checking take $O(n)$ time, where n is the size of the program.

6 Implementation as an Eclipse Plug-in

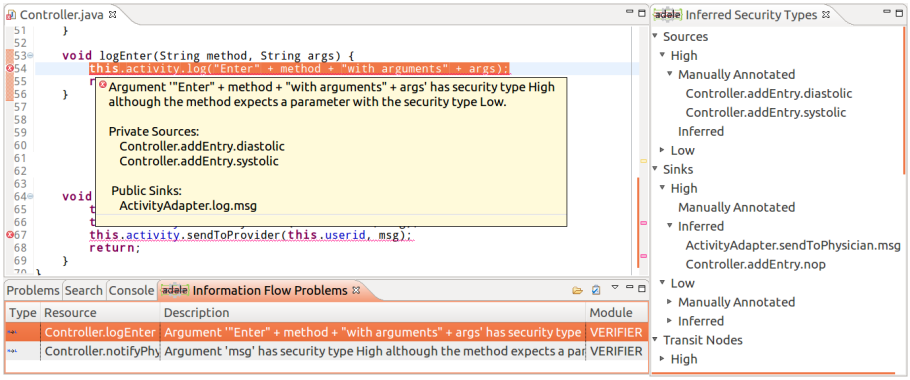
We implemented our solution as an Eclipse plug-in Adele (Assistant for Developing Leak-free Programs). It leverages our security-type inference algorithm and security type system for the development of Java programs with secure information flow. Adele integrates into the Eclipse IDE, analyzes the source code in the background, fully-automatically, and reports detected information leaks.

User interface: input. Adele allows its user to control two parameters of the analysis: the location of the source code to analyze and the information-flow policy. The location of the source code can be specified by selecting a source directory or a package containing Java source files within the current workspace of Eclipse. Selecting a package within a larger program allows focusing the analysis on a security-critical part of a given program. The information-flow policy is specified directly in the source code with Java annotations `@High` and `@Low`. The usage and semantics of these annotations are as described in Section 3.

User interface: output. The output of Adele consists of (1) a report on detected information leaks, and (2) inferred security types for information containers. Adele displays this information in the views "Information Flow Problems" and "Inferred Security Types", respectively. The view "Information Flow Problems" (see Figure 5 (a)) lists detected leaks together with information that could be helpful for mitigating them, e.g., the location of the leak in the code, and sources and sinks relevant for the leak. The detected leaks are also marked in the source code editor of Eclipse. In the view "Inferred Security Types" (see Figure 5 (b)), information containers are structured into categories "Sources", "Sinks", and "Transit Nodes". "Sources" groups information containers from which information is read but not written to. "Sinks" groups information containers to which flows within the program exist, but which are never read. "Transit Nodes" contains information containers that are read and written. Within these three categories, the identifiers are grouped by whether they are manually annotated or have an inferred security type, and by their security types.

7 Evaluation

The experimental evaluation of our solution has the goal of answering the following three questions: (i) What is the ratio between manually annotated and automatically inferred types, in practice? (ii) What is the performance of our solution, in practice? (iii) What is the relationship between the performance of our solution and that of SecJ [26], an implementation of a security-type inference algorithm [27] for a programming language similar to the one that we use?



(a) Reporting detected information leaks.

(b) Exploring inferred security types.

Figure 5: User interface of ADELE.

Our benchmark applications. We conduct our evaluation on four conceptual Java applications that we developed ourselves, inspired by real-world applications with similar functionality. We decided to develop applications ourselves in order to introduce information leaks into some of them, purposely, and investigate how the implementation of our solutions detects these leaks. Application "Blood Pressure History" (short: BPH) allows its user to record blood pressure values and to view previously recorded values. The application automatically informs a physician if the measured values are critical. The security concern is that the user's blood pressure values leak to third parties. Application "Company Strategy" (short: CS) allows a company to send resource requests to a supplier in order to pursue an internal strategy with certain resource requirements. The security concern is that confidential details about the company's internal strategy leak to the supplier. Application "Job Finder" (short: JF) searches a database for jobs that match the user's keywords. The security concern is that the user's keywords leak to an employer. Application "Online Shop" (short: OS) allows its users to maintain a wish list that their friends can use for selecting gifts. The security concern is that confidential information about user's purchases leaks to friends. Applications BPH and CS are analyzed in three variants each with modifications of the code that affect their security.

Our experimental setup. We run all our experiments on a typical laptop with Intel Core i7 CPU at 2.50GHz×4 and 8Gb of RAM. We use Ububtu 12.04 and Oracle Java Platform SDK in version 1.8.0_45 for 64-bit Linux.

7.1 Ratio Between Manually Annotated and Inferred Types

For evaluating the ratio between manually annotated and inferred security types, we annotated each of our benchmark applications with information-flow policies that reflect the aforementioned security concerns. This results in annotating one or several information containers that correspond to a source with `@High`, and

one or several containers that correspond to a sink with `@Low`. Altogether the number of such manually annotated information containers ranges from 2 to 5 in our experiments. Our solution infers security types for all remaining information containers. Table 1 presents the results of our experiments.

Our solution successfully verifies the information-flow security of applications "Blood Pressure History 1" (BPH 1) and "Company Strategy 3" (CS 3). All remaining applications are insecure, and our solution successfully detects information leaks in them. In Table 1, we observe that the ratio between manually annotated information containers and those containers for which security types are inferred by our solution varies between 1 : 17 and 1 : 128 in our experiments. This suggests that our security-type inference algorithm reduces the burden of manual security-type annotation by up to two orders of magnitude.

#	Application	LoC	Leak	$\mathcal{M}$	$\mathcal{I}$	$\lfloor \mathcal{M} : \mathcal{I} \rfloor$
1	BPH 1	135	no	4	89	1:22
2	BPH 2	135	yes, explicit	5	88	1:17
3	BPH 3	136	yes, explicit	4	89	1:22
4	CS 1	147	yes, explicit	4	89	1:22
5	CS 2	151	yes, implicit	4	88	1:22
6	CS 3	307	no	4	190	1:47
7	JF	311	yes, implicit	3	187	1:62
8	OS	410	yes, implicit	2	256	1:128

Table 1: Number of security types in our benchmark applications: $\mathcal{M}$ denotes the number of manually annotated information containers, $\mathcal{I}$ denotes the number of inferred security types for other information containers.

7.2 Performance

For evaluating the performance of our solution we use the same benchmark applications and information-flow policies as in Subsection 7.1. We collect 1000 samples of our solution's running time on each benchmark application, from which we compute the estimated mean running time. We measure the running time in the steady state of the JVM using `System.nanoTime()` timer. To reduce the interference of the garbage collection with the measurements, `System.gc()` is called before each run of the analysis. Table 2 presents the results of our performance evaluation (see section "ADELE" of the table).

The overall time corresponds to the running time of the analysis from parsing to reporting. It includes the time of the type inference, the sum of the times of constraint collecting and solving. By dividing the running times estimated during the analysis of each application by the corresponding number of the source code lines, we compute the running time per line of code for the overall analysis, and for the type inference. We observe: (1) the overall running time of our solution, averaged among the benchmark applications, lies below 0.02 ms per line of code, and (2) the running time required for the type inference, averaged among the benchmark applications, lies below 0.008 ms per line of code. Taking into account that our solution has a linear time-complexity in size of the analyzed program (see Theorem 5), our experimental results suggest that our solution shall also be efficient when analyzing significantly larger applications.

#	Application	LoC	Estimated mean running time					
			Overall	Inference	Collecting	Solving	Overall per LoC	Inference per LoC
ADELE								
1	BPH 1	135	2.6600	1.0302	0.9526	0.0776	0.0197	0.0076
2	BPH 2	135	2.6865	0.8723	0.7833	0.0890	0.0199	0.0065
3	BPH 3	136	2.8716	1.0279	0.9318	0.0961	0.0211	0.0076
4	CS 1	147	2.8797	1.1081	0.9704	0.1376	0.0196	0.0075
5	CS 2	151	2.7458	1.0189	0.8581	0.1608	0.0182	0.0067
6	CS 3	307	4.6359	2.0526	1.9852	0.0674	0.0151	0.0067
7	JF	311	5.1064	2.5773	2.2175	0.3598	0.0164	0.0083
8	OS	410	6.8985	3.6433	3.2686	0.3748	0.0168	0.0089
SECJ								
1	BPH 1	135	775.9983	24.1443	20.7841	3.3602	5.7481	0.1788
2	BPH 2	135	736.9718	24.3384	21.3163	3.0221	5.4591	0.1803
3	BPH 3	136	801.3648	26.8502	23.1533	3.6968	5.8924	0.1974
4	CS 1	147	745.5113	27.7995	24.2680	3.5315	5.0715	0.1891
5	CS 2	151	757.0713	30.7687	26.6143	4.1543	5.0137	0.2038
6	CS 3	307	1169.3359	130.2044	118.6230	11.5814	3.8089	0.4241
7	JF	311	1279.5666	160.5022	140.3807	20.1215	4.1144	0.5161
8	OS	410	1655.0694	284.5710	249.6445	34.9265	4.0368	0.6941

Table 2: Estimated mean running time of ADELE and SECJ, in milliseconds.

7.3 Relationship to SecJ wrt. Performance

For evaluating the relationship of our solution to SECJ wrt. performance, we run the experiments from Subsection 7.2 also for SECJ [26]. Table 2 presents the results of this performance evaluation (see section "SECJ" of the table). By comparing the running time values observed in our experiments for ADELE and SECJ, we conclude: (1) overall, our solution is two order of magnitude faster than SECJ, and (2) the implementation of our security-type inference algorithm is an order of magnitude faster than the type inference in SECJ.

Leak not detected by SECJ. During our experiments, we found that the information leak in the application "Job Finder" is not detected by SECJ. In the code snippet, the confidential result of a job search `job` is converted into an instance of `JobRecord` and written to untrusted sink `choice`. Hence, there is an information leak from `job` to `choice`. SECJ, however, accepts this example as secure. We inspected the implementation of SECJ and suspect an error in its constraint derivation for the type casting, which results in the undetected leak. It seems that the error is caused by a wrong type variable in the implementation.

```
public JobRecord makeChoice(JobList jobs) {
    @High Element job = jobs.getFirst();
    @Low JobRecord choice = (JobRecord)job;
    return choice;
}
```

8 Related Work

The certification of programs for secure information flow [6] is a long-standing line of research. Starting from the work of Volpano, Irvine, and Smith [30],

security type systems have attracted a lot of attention for such certification. Sabelfeld and Myers provide in [22] a comprehensive overview of this area until the beginning of 2000s. Since then, a notable branch of this area focused on making security type systems applicable for realistic object-oriented languages, like Java. We limit this paragraph to security type systems for such languages, as we focus on a subset of Java in this article. Strecker [24] formalizes a security type system for MicroJava in Isabelle/HOL. Banerjee and Naumann [1] propose a security type system for a Java-like programming language extended with access-control features. We drew inspiration from their work when we were defining our programming language and our security type system. Barthe et al. [2] propose a security type system for a Java-like language that supports exceptions. Rafnsson et al. [20] propose a security type system that addresses dynamic class loading and the initialization of static fields. The aforementioned security type systems have been proven sound in [24], [1], [2], and [20], respectively. There are also type-based information-flow analyses [4, 8, 16, 18] that target programs written in larger fragments of Java, some — even full Java. Yet, they are not accompanied by formal soundness proofs, to the best of our knowledge.

Security type systems require all information containers in a program to be annotated with security types. Doing such annotations manually is a tedious and error-prone task. Security-type inference algorithms have a goal of inferring such annotations automatically. Type inference, in general, has a long-standing tradition (see, e.g., [7,17,25]). Starting from Volpano and Smith's type inference algorithm [31] for the security type system from [30], there has been a growing interest for type-inference algorithms tailored to information-flow analyses [3, 5, 10–14, 19, 23, 27, 28, 32]. In Table 3, we list attributes of twelve well-known security-type inference algorithms and compare them to our algorithm.

type-inference algorithm of	imperative, object-oriented language	soundness result	completeness result	minimality result	time-complexity
Volpano/Smith [31]	no	yes	yes	no	–
Pottier/Simonet [19]	no	yes	yes	no	–
Sun et al. [27]	yes	yes	yes	no	$O(n)$
Deng/Smith [5]	no	yes	yes	no	$O(n^2)$
Hristova et al. [10]	no	no	no	no	$O(n)$
Hunt/Sands [12, 13]	no	yes	yes	yes	$O(nv^3)$
Smith/Thober [23]	yes	yes	yes	no	$O(n^{n^5})$
King et al. [14]	yes	no	no	no	–
Terauchi [28]	no	yes	no	no	polynomial
Bedford et al. [3]	no	yes	yes	no	–
Weijers et al. [32]	no	no	no	no	–
Huang et al. [11]	yes	no	no	no	$O(n^3)$
Our algorithm	yes	yes	yes	yes	$O(n)$

Table 3: Attributes of security-type inference algorithms. (A dash means that the respective article does not provide information on the attribute.)

A conceptual novelty of our algorithm over other security-type inference algorithms is that it is accompanied by a formally proven minimality result without having to use principal types [12, 13, 29]. Hunt and Sands [12] show how to infer principal types for programs written in a simple while-language. Their principal types describe, for each variable, all possible flows of information through the variable. This description is so fine-grained that is provides enough information for checking a program's compliance with an arbitrary information-flow policy. In [13], Hunt and Sands provide an algorithm for computing principal types in $O(nv^3)$, where n is the size of an input program and v the number of its variables. In a recent work [29], their principal type system is lifted to support dynamic policies. Generally, the idea of extending principal types to support an object-oriented Java-like programming language seems rather appealing. Yet, at this time it is not clear how to achieve this at low computational costs.

The security-type inference algorithm of Sun et al. [27] is the closest to our algorithm, supporting a programming language with the same features. The algorithms differ, most notably, in the following two technical aspects: (1) The algorithm of Sun et al. [27] maintains a type environment to dynamically read and keep track of the security types of local variables and formal parameters. We use a predefined security context to access security types of all information containers. (2) The algorithm of Sun et al. [27] conducts data type inference for local variables and expressions in parallel to the derivation of constraints for security types. As a consequence, all constraint derivation rules for expressions have to capture also inference of data types, and the type environment has to store data types of local variables and formal parameters, in addition to their security types. In contrast, we use results of a separate data-type inference algorithm just in those rules that require it, i.e., rules for a field access, field assignment, and method call. Modelling both the type environment and the inference of data types by separate functions enables implementation of our algorithm in a clean, modular fashion.

Sun et al. [27] do not comment whether their algorithm infers minimal typings. We conjecture that it probably does, at least if no polymorphism is used. However, due to the additional complexity coming with polymorphic classes, we cannot intuitively assess the minimality of their full algorithm without having to conduct a formal proof.

9 Conclusion

We presented a new algorithm for inferring security types in Java programs. We proved it to be sound, complete, minimal, and of linear time-complexity in the size of the program analyzed. The minimality of our algorithm allows flexible security analyses, in the sense that programs can be analyzed wrt. information-flow policies that fix only the annotations of sources, while leaving the annotations of sinks flexible. Based on our algorithm, we developed a solution for verifying confidentiality requirements in Java programs. We implemented or solution as an Eclipse plug-in, and experimentally showed that it is effective and efficient.

As future work, we plan to deploy the presented algorithm, after necessary adaptations, in our information-flow analysis for Dalvik bytecode [15].

Acknowledgements We thank the anonymous reviewers for their valuable comments. We thank Patrick Metzler for his help in the implementation of Adele. This work has been partially funded by the BMBF within EC SPRIDE and by the DFG as part of project E2 within the CRC 1119 CROSSING.

References

1. A. Banerjee and D. A. Naumann. Stack-based Access Control and Secure Information Flow. *Journal of Functional Programming*, 15(2):131–177, 2005.
2. G. Barthe, T. Rezk, and D. A. Naumann. Deriving an Information Flow Checker and Certifying Compiler for Java. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*, pages 230–242. IEEE, 2006.
3. A. Bedford, J. Desharnais, T. G. Godonou, and N. Tawbi. Enforcing Information Flow by Combining Static and Dynamic Analysis. In *Proceedings of the 6th Symposium on Foundations and Practice of Security (FPS)*, LNCS 8352, pages 83–101. Springer, 2013.
4. N. Broberg, B. van Delft, and D. Sands. Paragon for Practical Programming with Information-Flow Control. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS)*, LNCS 8301, pages 217–232. Springer, 2013.
5. Z. Deng and G. Smith. Type Inference and Informative Error Reporting for Secure Information Flow. In *Proceedings of the 44th Annual Southeast Regional Conference (ACM-SE)*, pages 543–548. ACM, 2006.
6. D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commununications of the ACM*, 20(7):504–513, 1977.
7. D. Duggan and F. Bent. Explaining Type Inference. *Science of Computer Programming*, 27(1):37–83, 1996.
8. M. D. Ernst, R. Just, S. Millstein, W. M. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative Verification of Information Flow for High-Assurance App Store. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 1092–1104. ACM, 2014.
9. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy (S&P)*, pages 11–20. IEEE, 1982.
10. K. Hristova, T. Rothamel, Y. A. Liu, and S. D. Stoller. Efficient Type Inference for Secure Information Flow. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 85–94. ACM, 2006.
11. W. Huang, Y. Dong, and A. Milanova. Type-Based Taint Analysis for Java Web Applications. In *Proceedings of the 17th Conference on Fundamental Approaches to Software Engineering (FASE)*, LNCS 8411, pages 140–154. Springer, 2014.
12. S. Hunt and D. Sands. On Flow-Sensitive Security Types. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 79–90. ACM, 2006.
13. S. Hunt and D. Sands. From Exponential to Polynomial-Time Security Typing via Principal Types. In *Proceedings of the 20th European Symposium on Programming (ESOP)*, LNCS 6602, pages 297–316. Springer, 2011.

14. D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, LNCS 5352, pages 56–70. Springer, 2008.
15. S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 93–104. ACM, 2014.
16. A. Lux and A. Starostin. A Tool for Static Detection of Timing Channels in Java. *Journal of Cryptographic Engineering*, 1(4):303–313, 2011.
17. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
18. A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241. ACM, 1999.
19. F. Pottier and V. Simonet. Information Flow Inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.
20. W. Rafnsson, K. Nakata, and A. Sabelfeld. Securing Class Initialization in Java-like Languages. *IEEE Transactions on Dependable and Secure Computing*, 10(1):1–13, 2013.
21. J. Rehof and T. A. Mogensen. Tractable Constraints in Finite Semilattices. *Science of Computer Programming*, 35(2):191–221, 1999.
22. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
23. S. F. Smith and M. Thober. Improving Usability of Information Flow Security in Java. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 11–20. ACM, 2007.
24. M. Strecker. Formal Analysis of an Information Flow Type System for MicroJava. Technical report, Technische Universität München, 2003.
25. M. Sulzmann. A General Type Inference Framework for Hindley/Milner Style Systems. In *Proceedings of the 5th Symposium on Functional and Logic Programming (FLOPS)*, LNCS 2024, pages 248–263. Springer, 2001.
26. Q. Sun. *Constraint-Based Modular Secure Information Flow Inference for Object-Oriented Programs*. PhD thesis, Stevens Institute of Technology, 2008.
27. Q. Sun, A. Banerjee, and D. A. Naumann. Modular and Constraint-based Information Flow Inference for an Object-oriented Language. In *Proceedings of the 11th Static Analysis Symposium (SAS)*, pages 84–99. Springer, 2004.
28. T. Terauchi. A Type System for Observational Determinism. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, pages 287–300. IEEE, 2008.
29. B. van Delft, S. Hunt, and D. Sands. Very Static Enforcement of Dynamic Policies. In *Proceedings of the 4th Conference on Principles of Security and Trust (POST)*, LNCS 9036, pages 32–52. Springer, 2015.
30. D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
31. D. M. Volpano and G. Smith. A Type-Based Approach to Program Security. In *Proceedings of the 7th Conference on Theory and Practice of Software Development (TAPSOFT)*, LNCS 1214, pages 607–621. Springer, 1997.
32. J. Weijers, J. Hage, and S. Holdermans. Security Type Error Diagnosis for Higher-Order, Polymorphic Languages. *Science of Computer Programming*, 95(2):200–218, 2014.