Enforcing Usage Constraints on Credentials for Web Applications

Jinwei Hu, Heiko Mantel, and Sebastian Ruhleder

Department of Computer Science, TU Darmstadt, Germany
{hu,mantel}@mais.informatik.tu-darmstadt.de
sebastian.ruhleder@googlemail.com

Abstract For using credential-based access control effectively, recent work identified the need to enforce usage constraints also on credentials. The enforcement of such constraints has not yet been investigated for web applications, although it is relevant when credential-based access control is employed in a web application. This article proposes an approach suitable for enforcing usage constraints on credentials in web applications. More concretely, we present a novel algorithm and an implementation of this algorithm that construct constraint-compliant proofs for credential-based access control policies. We proved that our solution is correct and showed that it is also efficient through extensive experiments.

1 Introduction

Many web applications control access to their resources using policies that state which attributes a client must have in order to obtain access. For example, an online svn system in a university might allow a client to view a directory if some employee of the university owns the directory and if the employee nominates the client as a collaborator. One promising approach to enforce a policy like this is credential-based access control (CBAC) [1].

In CBAC, credentials are used to encode attributes of clients. The representation of credentials may vary, but usually digitally-signed certificates are used as representation. Which credentials a client must provide in order to obtain a particular access is specified by an access control policy. Hence, before granting access, an enforcement mechanism needs to check that the provided credentials legitimate the desired access. This last step is known as proof construction.

Recent work on CBAC identified a need to also restrict the usage of credentials [2, 3], in particular, if CBAC is employed in open systems. In an open, distributed system, a credential might be issued with a particular purpose, but it is hardly possible for a credential issuer to exclude the possibility that the credential might be exploitable in other, unintended ways. The danger is that a credential is used in the construction of proofs that result in authorizations unforeseen by the issuer. Usage constraints on credentials enable credential issuers to better control the use of their credentials. They allow issuers to encode the purpose of a credential, thus, reducing the threat of misuse.

Usage constraints on credentials are obviously relevant for web applications that employ CBAC. However, the enforcement of usage constraints on credentials has not been investigated in this domain yet. Moreover, solutions in other domains cannot be transferred to web applications in a straightforward manner, as web applications pose new challenges to enforcing usage constraints on credentials. For instance, if a web application needs to respond to requests with low latencies, even under high workloads, then a web application developer is likely reluctant to adopt a usage constraint for credentials that causes substantial overhead. This is the challenge that we address in this article.

We present a novel algorithm for constructing proofs (1) that soundly certify compliance with a given CBAC policy and (2) that respect all usage constraints of credentials used during proof construction. Moreover, our algorithm is complete in the sense that it is able to generate all proofs that comply with (1) and (2). The feature of generating all constraint-compliant proofs is driven by the need to strategically manage credential disclosure to, for example, minimize the number or sensitivity of credentials sent to an application [4, 5]. With all proofs at hand, one could choose a proof that meets a strategy.

In the implementation of our solution, we followed the idea of proof-carrying authorization to web applications [6]. More concretely, we implemented a browser extension for deploying our proof-construction algorithm. A client adds this extension for requesting resources at a web application. A web application simply checks the validity of a proof received from a client which uses the browser extension. In this way, the application only needs to incorporate a proof-checking functionality.

To sum up, our main technical contribution is a novel algorithm for constructing constraint-compliant proofs. We proved the correctness of our algorithm and demonstrated its effectiveness and efficiency in experiments.

The rest of the article is organized as follows. We present preliminaries and our problem statement in Section 2. In Section 3, we present our algorithm, including the algorithm description and the correctness theorem. We describe our implementation in Section 4, followed by the performance evaluation of our algorithm in Section 5. Finally, we discuss related work and conclude in Section 6.

2 Preliminaries and Problem Statement

Credential-based access control. We use the policy language RT_0 [7], a role-based trust management language, to express credentials. RT_0 has four types of credentials:

- *Simple membership* $A.R \leftarrow D$: principal D is assigned to role $A.R$.
- *Simple containment* $A.R \leftarrow B.R_1$: principals assigned to role $B.R_1$ are also assigned to role $A.R$.
- *Linking containment* $A.R \leftarrow A.R_1.R_2$: principals assigned to role $B.R_2$ with B being assigned to role $A.R_1$ are also assigned to role $A.R$.

$$\frac{A.R \leftarrow D}{D \textbf{ in } A.R}\ sm \qquad \frac{D \textbf{ in } B.R_1 \quad A.R \leftarrow B.R_1}{D \textbf{ in } A.R}\ sc$$

$$\frac{B \textbf{ in } A.R_1 \quad D \textbf{ in } B.R_2 \quad A.R \leftarrow A.R_1.R_2}{D \textbf{ in } A.R}\ lc$$

$$\frac{D \textbf{ in } B_1.R_1 \quad \cdots \quad D \textbf{ in } B_k.R_k \quad A.R \leftarrow B_1.R_1 \cap \ldots \cap B_k.R_k}{D \textbf{ in } A.R}\ ic$$

Figure 1: The inference rules for RT_0 credentials.

– *Intersection containment* $A.R \leftarrow B_1.R_1 \cap \ldots \cap B_n.R_n$: a principal is assigned to role $A.R$ if the principal is also assigned to all the roles $B_1.R_1$, ..., and $B_n.R_n$.

The semantics of RT_0 credentials is formalized by the inference rules in Figure 1, where D **in** $A.R$ denotes the principal D's membership of role $A.R$. For a credential of the form $X \leftarrow Y$, we say X is the head and Y is the body of the credential. Readers are referred to [7] for more details of RT_0.

In RT_0, roles are used to model particular accesses to particular resources. For example, one could introduce a role $Univ.network$, where $Univ$ is a principal and $network$ is a role name, to model access to networks of $Univ$. In turn, to access a particular resource one needs to have a membership in the role to which this resource corresponds.

Role memberships are granted based on credentials. For example, the credential $Univ.network \leftarrow Alice$ grants the membership in $Univ.network$ to $Alice$. For another example, the credentials in the following set jointly grant to Alice the membership in the role $Univ.network$. In both cases, one actually derives a proof of the membership using the provided credentials.

$$Univ.network \leftarrow Univ.guest \qquad Univ.guest \leftarrow Univ.Prof.collaborator$$
$$Univ.Prof \leftarrow Bob \qquad Bob.collaborator \leftarrow Alice$$

Definition 1 (Proofs). *Given a principal p, a role r, and a set CS of credentials, we say a tuple $(p, r, c, \emptyset)$ is a* base proof *of p **in** r based on CS if c is a simple membership credential $r \leftarrow p$. We say (p, r, c, s) is a* proof *of p **in** r based on CS if either it is a base proof or s is a set of proofs based on CS, called* sub-proofs*, such that for any $(p_i, r_i, c_i, s_i) \in s$, (1) there exists $c \in CS$ such that*

$$\frac{p_1 \textbf{ in } r_1 \quad \ldots \quad p_k \textbf{ in } r_k \quad c}{p \textbf{ in } r}\ l$$

for some $l \in \{sm, sc, lc, ic\}$ and (2) (p_i, r_i, c_i, s_i) is also a proof based on CS.

A proof (p, r, c, s) shows the membership of p in r. Thus, upon seeing the proof, a system deploying credential-based access control allows the access of p

to the resource represented by r. We say a proof (p, r, c, s) *uses* a credential if it is c or it is used by the sub-proofs in s.

Usage constraints on credentials. As discussed in [2, 3], credential issuers are in need of specifying constraints on how their credentials are used, in addition to the language support (e.g., RT). Typical example constraints include delegation depth and final-usage constraints. Following [3], we define a constraint as a deterministic finite automaton (DFA) $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of input symbols, $\delta\colon Q \times \Sigma \mapsto Q$ is a transition function, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. The input set Σ of a constraint shall be instantiated with the set of roles in the set CS of credentials when constructing proofs based on CS.

To define the semantics of constraints, we let $words(P)$ be *words* of a proof $P = (p, r, c, s)$ such that $words(P) = \{r\}$ if $s = \emptyset$ and, otherwise, $words(P) = \{r; w \in \Sigma^* \mid \exists P' \in s : w \in words(P')\}$. A proof (p, r, c, s) based on CS *satisfies* a constraint if all words of the proof are accepted by the constraint. Informally, a constraint restricts which roles may appear in a proof and where they may appear in the proof.

For example, suppose that $Univ$ wants to prevent its credential $c1 = Univ.network \leftarrow Univ.guest$ from being used for proving memberships in role $Univ.internal$, regardless of whichever credentials have been issued or may be issued later. This is a final usage constraint on $c1$; $Univ$ could specify the DFA in Figure 2 as a constraint and attach the constraint to $c1$. This constraint excludes any proof of the form $(p, Univ.internal, c, s)$. Receiving words of such a proof, the automaton transits from state q_0 to q_1 with the input $Univ.internal$; if $s \neq \emptyset$, the automaton stays at state q_1 with input r_i for any sub-proofs $(p_i, r_i, c_i, s_i) \in s$. Hence, the automaton could not reach a final state, which means that the proof does not satisfy the constraint. On the other hand, for a proof of the form (p, r, c, s) where $r \neq Univ.internal$, its words are accepted by the automaton and thus the proof satisfies the constraint.

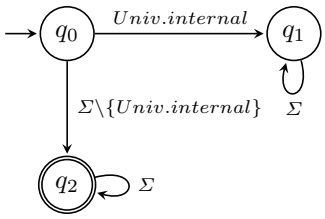


Figure 2: An example constraint.

In principle, each credential may come along with a constraint specifying its usage. We say a proof is *constraint-compliant* if it satisfies all constraints that are attached to the credentials used in the proof.

Definition 2 (Problem statement). *Given a set CS of credentials, a principal p, and a role r, the problem is to find all constraint-compliant proofs of p **in** r based on CS.*

```
SAGE-PROVE(credentials, r, p)
    let nodes = CREATE-NODES(credentials) and cons = ∅
    return CONSTRUCT-PROOFS(r, p, nodes, (), cons)

CONSTRUCT-PROOFS(r, p, nodes, rolepath, cons)
 1  r' = EXTEND-ROLE-PATH(rolepath, r)
 2  for every constraint con ∈ cons
 3       if con cannot reach a final state after the input r'
 4            return ∅
 5  if nodes contains no node for r
 6       return ∅
 7  let node = node for r and proofs = ∅
 8  for each {c, p'} ∈ node.members with p' == p
 9       let proofs = proofs ∪ NEW-PROOF(r, p, c, ∅)
10  for each credential c ∈ node.credentials
11       let sets = ∅ and cons' = cons ∪ {c's constraint}
12       if c is a simple containment credential
13            let sets = HANDLE-SC(c, p, nodes, r', cons')
14       elseif c is an intersection containment credential
15            let sets = HANDLE-IC(c, p, nodes, r', cons')
16       elseif c is a linking containment credential
17            let sets = HANDLE-LC(c, p, nodes, r', cons')
18       for each set of subproofs s ∈ sets
19            let proofs = proofs ∪ NEW-PROOF(r, p, c, s)
20  return FILTER-VALID-PROOFS(proofs)
```

Figure 3: The Sage algorithm for constructing constraint-compliant proofs.

3 Algorithm

This section presents our algorithm Sage that solves the problem in Definition 2. We first describe the algorithm and then discuss its correctness.

Overview. The Sage algorithm, as shown in Figure 3, is essentially a depth-first search algorithm. It makes use of a dependency between c and s in a proof $P = (p, r, c, s)$: The inference rule used to conclude the role membership p **in** r must take the credential c and all sub-proofs $s_i \in s$ as its premise; further, the rule must match one of the four inference rules *sm*, *sc*, *ic*, and *lc*. Consequently, the type of the credential c directly determines what proofs must be present in s for this to be possible. The Sage algorithm utilizes this dependency between c and s to construct proofs recursively: When a role membership p **in** r shall be proven, (1) select all credentials with r as the head; (2) for every such credential, (2i) identify the role memberships that must be proven in order to satisfy the premise of an appropriate inference rule, and (2ii) then call the Sage algorithm again

with the same principal for every identified role membership. If all necessary role memberships can be proven, construct (and later return) a proof with the recursively constructed proofs and the associated, previously selected credential.

Main procedure. The entry point of Sage is the SAGE-PROVE function: It takes a set of credentials, a role, and a principal as input and returns a set of proofs showing the membership of the principal in the role. Sage consists mainly of two parts: initialization and proof construction.

Initialization. Sage first converts the credentials into nodes. Sage uses *nodes*, each of which is a container for credentials with the same head role. Sage stores every credential $c \in CS$ in exactly one node, i.e. the node associated to its head role. The use of nodes provides a simplified way to access different credentials in the set CS. Sage initializes the constraints as an empty set, as no credential has been used for proofs yet.

Proof construction. Sage calls the CONSTRUCT-PROOFS function to compute the set of proofs. Function CONSTRUCT-PROOFS consists of three stages: pre-processing (Lines 1 - 7), sub-proof construction (Lines 8 - 19), and post-processing (Line 20). The pre-processing stage checks whether any proofs may be constructed; the sub-proof construction stage proceeds to construct sub-proofs with different types of credentials. The post-processing stage checks additional constraints that proofs should comply with.

Pre-processing. Line 1 extends the role path with the provided role r. Given a proof (p, r, c, s), a *role path* is a sequence of roles, starting with r and being followed by a role path of a proof in s. Note that a proof may have multiple role paths. This extended role path serves as a trace at which point the algorithm is in relation to the overall structure of the constructed proofs. In lines 2 - 4, this extended role path is used to check whether all constraints in the set *constraints* can still reach a final state from it. If not, any further proofs do not comply with at least one of the constraints. Line 5 checks whether there is any credential available to construct further proofs. If so, the associated node is then assigned to the variable *node* and the set *proofs* is initialized as an empty set. The set *proofs* serves as a container for proofs.

Sub-proof construction. Lines 8 - 19 construct sub-proofs for each type of credentials. First, lines 8 - 9 traverse all simple member credentials of *node* and add to the set *proofs* a proof for each credential $r \leftarrow p$. Lines 10 - 19 traverse the set of credentials of *node* to construct proofs for the three remaining types of credentials (simple, intersection, or linking containment). Depending on the type of credentials, a helper function is invoked (lines 12 - 17). Each function returns a collection of sets of sub-proofs that can be used to construct proofs in combination with c. Note that the helper functions also call CONSTRUCT-PROOFS for proof construction. Please see Appendix A for the helper functions. Line 19 combines the returned sets of sub-proofs with credential c to a proof of p's membership in r.

Post-processing. Line 20 (FILTER-VALID-PROOFS) filters proofs that satisfy all remaining constraints. Recall that the filtering at lines 2 - 4 checks constraints of a selected credential; this check does not concern constraints introduced posterior to the selected credential. FILTER-VALID-PROOFS checks those latter constraints. In combination, these two checks ensure that Sage returns only constraint-compliant proofs.

Algorithm correctness. Given a set of credentials CS, a principal p, and a role r, we let $\mathit{Proofs}(C, r, p)$ be the set of constraint-compliant proofs of p **in** r. That is, $\mathit{Proofs}(C, r, p)$ contains all the proofs that show the membership of p in r, according to the semantics of RT_0 and constraints. Let SAGE-PROVE(C, r, p) be the set of proofs returned by the algorithm in Figure 3.

Theorem 1. *Given a set of credentials CS, a principal p, and a role r,*

$$\text{SAGE-PROVE}(C, r, p) = \mathit{Proofs}(C, r, p).$$

Proof sketch: We prove two lemmas:

1. SAGE-PROVE$(C, r, p) \subseteq \mathit{Proofs}(C, r, p)$ (i.e., any proof returned by the algorithm is constraint-compliant), and
2. SAGE-PROVE$(C, r, p) \supseteq \mathit{Proofs}(C, r, p)$ (i.e., any constraint-compliant proof will be generated by the algorithm).

We first map a proof (p, r, c, s) to a tree where the root is the proof itself and its children are the sub-proofs in the set s. The height of a proof is then defined as the height of this tree. We then prove the two lemmas by induction on the height. The full proof of the theorem is available on the authors' website.

4 Implementation

To implement the proposed approach for web applications, we first adapt the communication process between a client and a web application. As shown in Figure 4, the communication takes four steps: (1) The client sends a request for a resource to the application. (2) Upon receiving a request, the application returns a policy for the client to prove. (3) The client constructs a proof of the policy and sends the proof and the credentials together to the application. (4) The application sends the response to the client. If the proof is checked constraint-compliant, the resource is sent to the client; otherwise not.

Among the four steps, step (2) employs the Sage algorithm proposed in Section 3. We implement the algorithm in a browser extension. As such, users at the client need not interact with CBAC when requesting resources. At the web application side, we implement a reference monitor which checks the proof and returns the check results to the application.

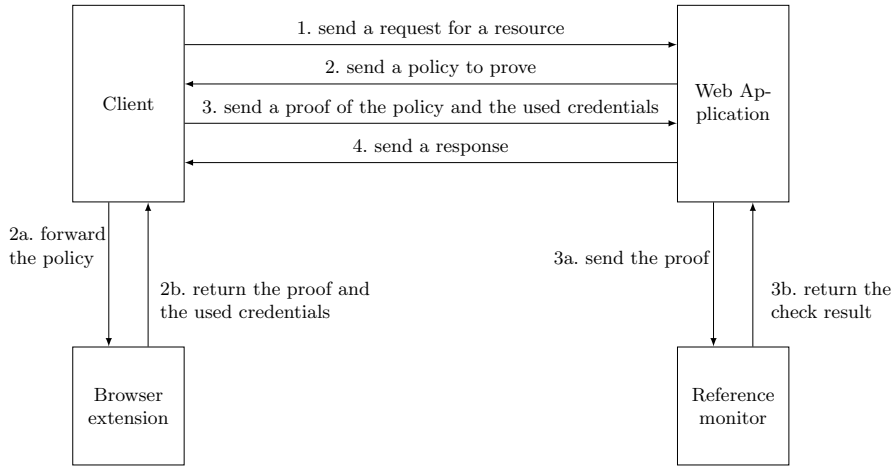


Figure 4: The proposed architecture of CBAC in web applications.

4.1 Browser extension

We developed a Chrome[1] extension. This extension intercepts policies from web applications at step (2) in Figure 4. With the policy, the extension then invokes a library *Sage.js*, which is a JavaScript implementation of Sage. Obtaining the proofs, the extension selects one proof and re-sends the request to the web application. Note that we leave the investigation of proof selection as future work.

Alternatively, one may write client-side scripts to construct proofs. This will remove the need of a browser extension. However, based on our experiences, a browser extension could use more computational resources and thus lead to better performance. Hence, we chose the extension over client-side scripts.

4.2 Reference monitor

We chose to implement a reference monitor for Ruby on Rails[2] (or Rails in short) applications. This choice is motivated by the use of model-view-controller pattern of Rails applications. In such a pattern, the notion of resources are directly linked to that of models; each resource is associated with a model. This logical connection between resources and models provides an entry point for incorporating a reference monitor. Also, at the code level, the separation of the view of resources from the model enables us to introduce custom methods to intercept resource requests.

Function-wise, our reference monitor maintains a relation between resources and policies that should be proved in order to access the respective resources. The monitor also enforces the policies by checking proofs of the policies provided by

[1] `https://www.google.com/chrome/browser/`
[2] `http://rubyonrails.org/`

each requester. To make the monitor effective, we modify Rails controllers which handle authorization logics. We implemented a reference monitor for BrowserCMS[3] as a case study.

5 Experiments

We evaluated the performance of our Javascript implementation of the algorithm Sage with synthetic credentials. The experimental results demonstrate the efficiency of Sage. We first describe a generator we used to synthesize credentials for experiments. Then, we present the experimental results.

5.1 Credential generator

In order to generate credentials, we make use of a template of credentials available for proving p **in** r for randomly chosen principal p and role r. In a template, leaf nodes are labeled "sm", indicating simple membership credentials shall be generated, and non-leaf nodes are labeled "sc", "ic", or "lc", indicating the other three types of credentials shall be generated, respectively. A template with a root node rn means that a proof (p, r, c, s) can be obtained where c is a type rn credential and rn has a child node ch corresponding to a sub-proof (p', r', c', s') in s such that c' is a type ch credential. Then random, concrete credentials are generated whenever a node is reached when traversing a template. In addition, we also generate some "noisy" credentials which are useless for proving p **in** r.

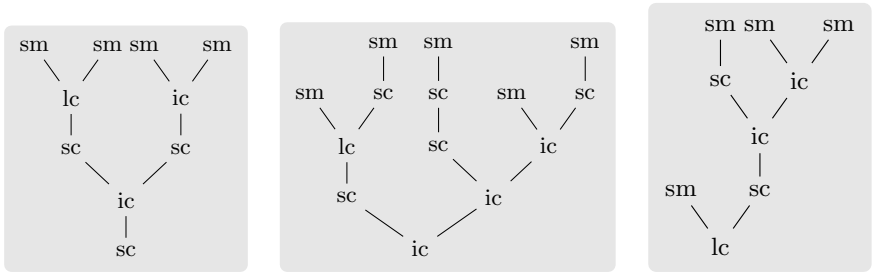
The generator takes as input four parameters: (1) a tree template, (2) the height of a tree, (3) variant: the number of credentials that shall be created for each type of credentials at each node of the template, and (4) the number of noisy credentials. The generator outputs, in addition to credentials, the following parameters: the size of a generated credential set (i.e., the number of credentials in the set) and the number of credentials of each type in a credential set.

5.2 Experimental credential sets

We generated credentials by letting, all uniformly, the template be one of templates in Figure 6, height range from 1 to 5, the number of variant be 1 or 2, and the number of noisy credentials range from 0 to 20. We obtained 945 generated credential sets in total and used them as input to Sage. Note that the templates cover example policies like "Co-workers can see all photos and music" in the case studies of [8].

Figure 7 shows the distribution of the size of the generated credential sets, when the size is smaller than 200. The most of the generated credentials sets have a size smaller than 80. On the other hand, for the ranges between 80 and 180, each range contains 2-10 credentials sets. Also, there are 147 generated credential sets whose size is larger than 200.

[3] `http://www.browsercms.org/`



Template #1 Template #2 Template #3

Figure 6: Templates that were used to generate credentials

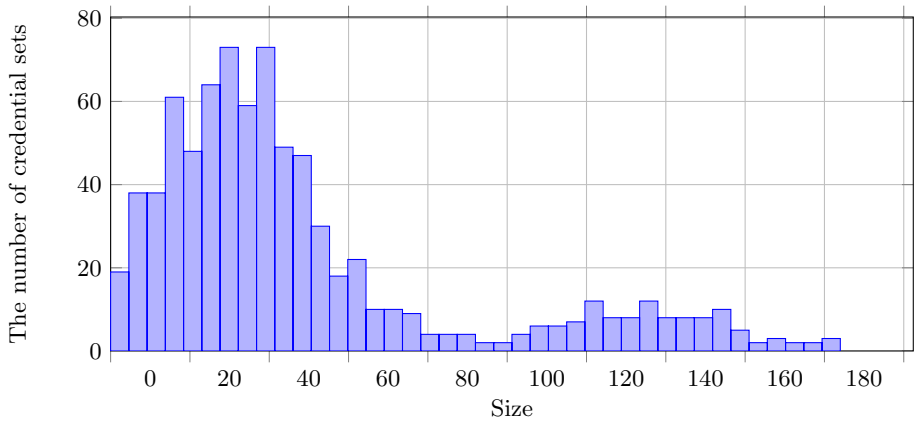


Figure 7: The size distribution of the generated credential sets of size smaller than 200

5.3 Experimental results

For each of the 945 generated credential sets, we invoked Sage, which is implemented as a JavaScript library, to compute all constraint-compliant proofs of p **in** r, where p and r were chosen when generating the credential set. For each credential set, we attached usage constraints to 30% of the credentials in the set: half are delegation depth constraints and the other half are final-usage constraints. The experiments were performed on a machine with 8 GB 1600 MHz DDR3 memory and Intel Core i5-3570 3.4 GHz RAM.

Figure 8 shows the time Sage took to return proof sets for each generated credential set. When the size of the credential sets is smaller than 100, the time is less than 1 ms, with four exceptions. When the size is smaller than 200, the time is always less than 10 ms. And when the size is smaller than 1000, the time is always less than 100 ms. In all cases, the time is less than 1 second. The computation time grows exponentially with the size of the credential sets. However, assuming the size of credential sets used in practice is smaller than 1000, the overhead, being less than 100 ms, is moderate.

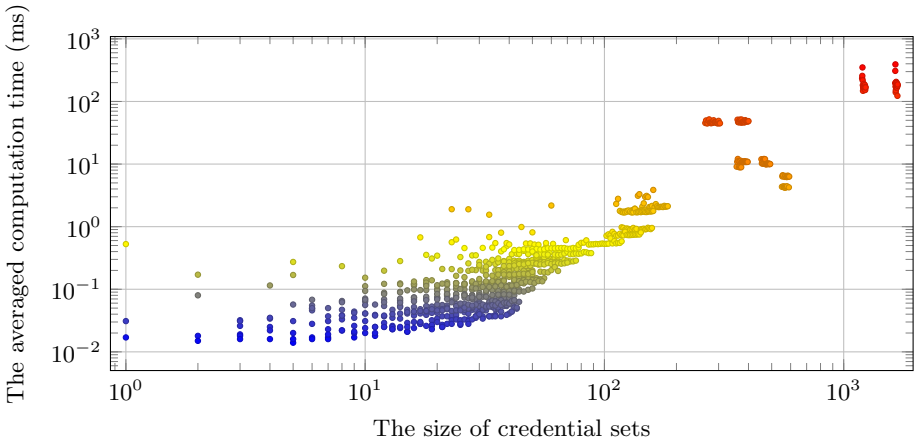


Figure 8: The time Sage took to return proof sets

Figure 9 depicts the time Sage took with respect to the size of the returned proof sets. The time increases exponentially with the size of the proof sets. However, when the size is smaller than 1000, the time is less than 100 ms, which we think is moderate. When the size is smaller than 10, the time is less than 1 ms except for two cases. When the size is smaller than 100, the time is less than 10 ms. In all cases, the computation time is less than 1 second.

Figure 10 shows the impact of noisy credentials on the computation time. In the figure, the computation time is the average of the time needed for the generated credential sets of the same size. The ratio of noisy credentials in the generated credential sets ranges from 0% to 50%. When the ratio is greater than 10%, the computation time is less than 100 ms. In all cases of the ratios, the time increases along with the growth of the size of the credential sets. Comparing the ratios, however, it appears that the higher the ratio is, the less computation time Sage took on average; the reason for this remains unclear to us.

6 Related Work and Conclusion

Seamons et al. [9] define two variants of the compliance checking problem: type-1 and type-2. The type-1 problem is to determine whether a policy is entailed by a set of credentials. The type-2 problem is to find a proof of a policy together with the used credentials in the proof. Lee and Winslett [4] propose a type-3 compliance checking problem – find all minimal proofs of a policy for a given set of credentials. While an algorithm for the type-2 problem shall be more efficient than an algorithm for the type-3 problem, the latter enables to apply strategies to proof and credential disclosure. This is also the main reason why we address a variant of the type-3 problem.

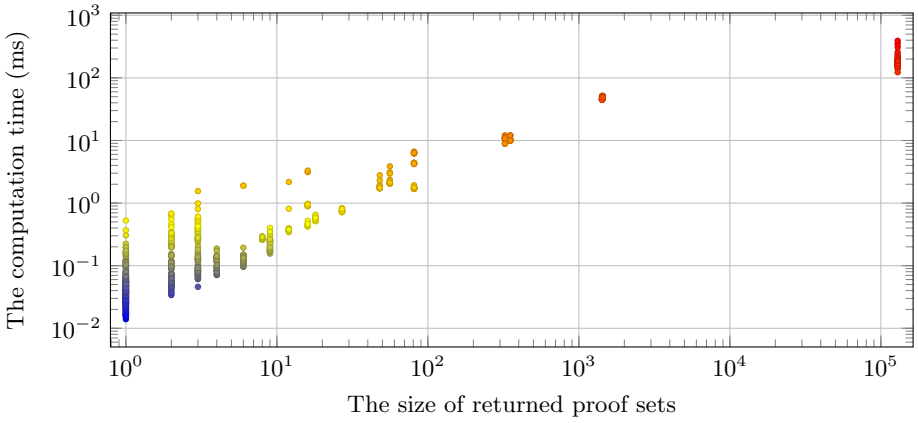


Figure 9: The time Sage took when the size of returned proof sets varies

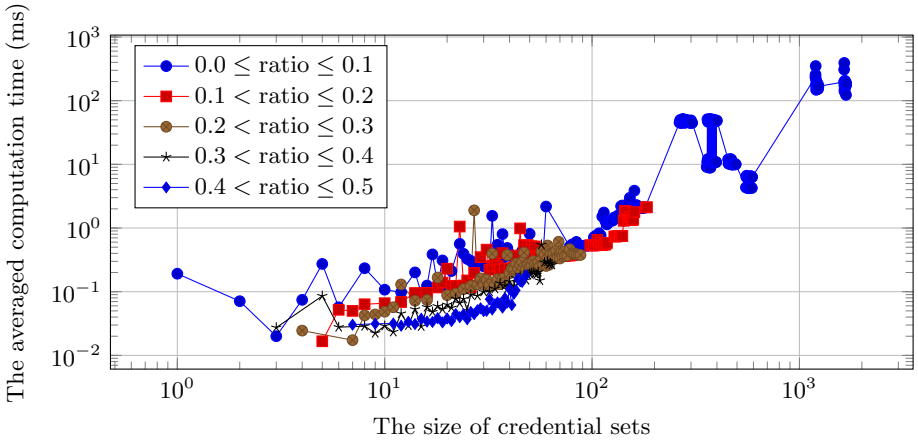


Figure 10: The averaged computation time with different ratios of noisy credentials

This variant of the type-3 problem is different in that we construct all constraint-compliant proofs of a policy. With respect to usage constraints on credentials, this work shares the same problem with [3]. While Hu et al. propose a solution by encoding credentials in answer set programming, we develop a novel algorithm and implement it for web applications. Unlike [4] and [3], this work did not consider searching only *minimal* proofs of a policy. Concerning usage constraints on credentials, Bauer et al. [2] propose an approach for proof-carrying authorization. It is not clear how to generate all constraint-compliant proofs with their approach. Approaches to CBAC for web applications include, for ex-

ample, [6, 1]. However, neither of them addresses enforcing usage constraints on credentials.

We have presented a new algorithm for constructing constraint-compliant policy proofs. We have proved the correctness of the algorithm and shown its efficiency by experiments. The algorithm is implemented as a JavaScript library, used in a browser extension, and integrated for an example Ruby on Rails application.

A The helper functions

This appendix lists the helper functions that are called by the Sage algorithm in Figure 3. More details of the algorihtm can be found on authors' website.

HANDLE-SC($c, p, nodes, rolepath, cons$)

```
1  return CONSTRUCT-PROOFS(BODY(c), p, nodes, rolepath, cons)
```

HANDLE-IC($c, p, nodes, rolepath, cons$)

```
1  sets = ∅
2  ip = ∅
3  for each role r_i ∈ BODY(c)
4        ip = ip ∪ CONSTRUCT-PROOFS(r_i, p, nodes, rolepath, cons)
5  let a_1, ..., a_n denote all sets in ip, with a_i = a_j ⇔ i = j
6  for each combination e = {e_1, ..., e_n} with e_i ∈ a_i
7        sets = sets ∪ e
8  return sets
```

HANDLE-LC($c, p, nodes, rolepath, cons$)

```
 1  sets = ∅
 2  defining-role = defining role of BODY(c)
 3  linked-role-term = linked role of BODY(c)
 4  for each principal p who defines a role r with the role term linked-role-term
 5        dp = CONSTRUCT-PROOFS(defining-role, p, nodes, (), ∅)
 6        if dp is not empty
 7              lp = CONSTRUCT-PROOFS(r, p, nodes, rolepath, cons)
 8              for each e ∈ dp × lp
 9                    sets = sets ∪ e
10  return sets
```

If the input credential c is a simple containment credential, HANDLE-SC is called. It simply calls CONSTRUCT-PROOFS with a new role parameter: the roles in the body of c. Every such proof is then later used in CONSTRUCT-PROOFS in combination with c to add proofs to the set *proofs*.

If c is an intersection containment credential, HANDLE-IC is called. In lines 3 - 4, for every role in the intersection of the body of c proofs are created by evaluating CONSTRUCT-PROOFS and then assigned to the variable *ip*. Finally, in lines 5 - 7, the Cartesian product of all proofs in *ip* is calculated. Note that the Cartesian product is used to model that an intersection containment credential

allows a principal to obtain the head role if the principal is assigned to all roles in the body.

If c is a linking containment credential, HANDLE-LC is called. First, in line 4, all principals who define a role using the linked role term are traversed. For each such principal, CONSTRUCT-PROOFS is called in line 5 to calculate proofs showing the principal is assigned to the defining role of c, which are assigned to the variable dp. If there exist proofs of this form, i.e., if dp is non-empty, CONSTRUCT-PROOFS is called again for the actual principal and the linked role, and the result is assigned to the variable lp. Then, in line 8, the Cartesian product of dp and lp is calculated and later returned.

Acknowledgments This work was supported by CASED (www.cased.de).

References

[1] Vimercati, S.D.C.D., Foresti, S., Jajodia, S., Paraboschi, S., Psaila, G., Samarati, P.: Integrating trust management and access control in data-intensive web applications. ACM Trans. Web **6**(2) (June 2012) 6:1–6:43

[2] Bauer, L., Jia, L., Sharma, D.: Constraining credential usage in logic-based access control. In: CSF. (2010) 154–168

[3] Hu, J., Khan, K.M., Bai, Y., Zhang, Y.: Compliance checking for usage-constrained credentials in trust negotiation systems. In: ISC. (2012) 290–305

[4] Lee, A.J., Winslett, M.: Towards an efficient and language-agnostic compliance checker for trust negotiation systems. In: Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2008. (2008) 228–239

[5] Oster, Z.J., Santhanam, G.R., Basu, S., Honavar, V.: Model checking of qualitative sensitivity preferences to minimize credential disclosure. In: FACS. (2012) 205–223

[6] Bauer, L.: Access Control for the Web via Proof-carrying Authorization. PhD thesis, Princeton University (2003)

[7] Li, N., Winsborough, W.H., Mitchell, J.C.: Distributed credential chain discovery in trust management. Journal of Computer Security **11**(1) (February 2003) 35–86

[8] Mazurek, M.L., Liang, Y., Melicher, W., Sleeper, M., Bauer, L., Ganger, G.R., Gupta, N., Reiter, M.K.: Toward strong, usable access control for shared distributed data. In: Proceedings of the 12th USENIX Conference on File and Storage Technologies. (2014)

[9] Seamons, K.E., Winslett, M., Yu, T., Smith, B., Child, E., Jacobson, J., Mills, H., Yu, L.: Requirements for policy languages for trust negotiation. In: 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002). (2002) 68–79