Automatically Tightening Access Control Policies with RESTRICTER *

Ka Lok Wu[1][0000−0001−6315−9068], Christa Jenkins[2**][0000−0002−5434−5018], Scott D. Stoller[1][0000−0002−8824−6835], and Omar Chowdhury[1***][0000−0002−1356−6279]

[1] Stony Brook University, Stony Brook NY 11794, USA
{kalowu,stoller,omar}@cs.stonybrook.edu
[2] Galois, Inc. Portland, OR 97204, USA
christa.jenkins@galois.com

Abstract. Robust access control is a cornerstone of secure software, systems, and networks. An access control mechanism is as effective as the policy it enforces. However, authoring effective policies that satisfy desired properties such as the *principle of least privilege* is a challenging task even for experienced administrators. In this paper, we set out to address this pain point by proposing RESTRICTER, which *automatically* tightens each (permit) policy rule of a policy with respect to an access log, which captures some already exercised access requests and their corresponding access decisions (*i.e.*, allow or deny). RESTRICTER achieves policy tightening by reducing the number of access requests permitted by a policy rule without sacrificing the functionality of the underlying system it is regulating. We implement RESTRICTER for Amazon's Cedar policy language and demonstrate its effectiveness through two realistic case studies.

Keywords: Security Policy · Access Control · SyGuS.

1 Introduction

An access control mechanism is a critical defense for ensuring the resilient security posture of software, systems, and networks. In the 2024 CWE Top 25 Most Dangerous Software Weaknesses published by MITRE [25], three of them were related to access control and authorization management. In the similar vein, OWASP's Top 10 list of critical web application security risks published in 2021 ranks "*Broken Access Control*" at the top [27]. One reason for such access control-related vulnerabilities can be attributed to access control checks being intertwined with application logic in source code, making it challenging for maintaining and understanding the access control being enforced. It is well understood that decoupling the access control checks from the application business logic ensures better management and

* This material is based on work supported in part by an Amazon Research Award and NSF award CCF-1954837.
** Work done while the author was at Stony Brook University.
*** Corresponding author.

maintainability. To promote such decoupling, many access control systems have been developed (*e.g.*, Casbin [13], Open Policy Agent [15], Amazon's Cedar [7]).

At the heart of an access control system is an *access control policy*, which aims to precisely capture the conditions under which users may access critical resources. Designing such policies to prevent unauthorized access and allow legitimate users to carry out their business obligations is a challenging task. Misconfiguring a policy can lead to the following pitfalls: ❶ overly permissive policies violate the *principle of least privilege* (*i.e.*, granting users only necessary privileges) and allow unauthorized users to access critical resources, opening the door to misuse; ❷ overly restrictive policies prevent legitimate users from accessing resources necessary for performing their business tasks. Although policy misconfigurations of type ❷ often get rectified due to user complaining due to failure to carry out business operations, type ❶ misconfiguration can remain undetected, waiting to be exploited. Examples of type ❶ policy misconfigurations abound in practice [25]. For instance, exploitable misconfigurations of type ❶ were identified in mandatory access control policies written by experts for Android's SELinux [14]. *This paper designs, develops, and evaluates* Restricter*, an automated technique, based on Satisfiability Modulo Theory (SMT) [10] and Syntax-guided Synthesis (SyGuS) [6], for reducing over-permissiveness in policies.*

Designing a policy that follows the principle of least privilege, which we call the *tightest policy*, is a challenging and error-prone task. Over-privileges (*i.e.*, requests permitted by the policy but not needed for any current business use case) can easily creep in as policies evolve. Over-privileges can also result from imprecision in identifying the permissions required for a use case and imprecision in formulating policy rules that grant those permissions. Restricter addresses over-privileges by ① inferring such unintended over-privileges permitted by the current policy and then ② refining (*tightening*) the policy by removing them.

Besides scalability, the main technical challenge in realizing Restricter's approach is identifying over-privileges (step ①). Unfortunately, a precise characterization of over-privileges in the current policy is often unavailable in practice. Restricter's over-privilege inference is based on analysis of the current policy together with an *access log* in which all attempted access requests and their current policy decisions are logged. Restricter's over-privilege inference is based on the insight that *the over-privileges are a subset of the permissions granted by the current policy and not exercised in the log.*

Restricter focuses on Attribute-based Access Control (ABAC) [22]. It formulates ABAC policy tightening as a *program synthesis problem* and solves it using a Syntax-Guided Synthesis (SyGuS) solver. Unfortunately, tightening an entire policy at once with SyGuS does not scale even for moderate-sized policies. Restricter addresses the scalability challenges by limiting the size and number of terms that are enumerated and by adopting a *rule-level analysis*, tightening permit rules individually. Rule-level analysis is also amenable to parallelization.

Restricter is instantiated for a large subset of Cedar [18], an ABAC system developed by Amazon Web Services. It is then evaluated on two case studies. It is able to scalably infer the potential over-privileges and refine most (6 out of 8)

```
entity User { isPCChair: Bool, isPcMember: Bool};
entity Paper { authors: Set<User>, reviewers: Set<User>};
entity Review { ofPaper: Paper, author: User, isMetaReview: Bool };
action Read appliesTo {
principal: User, resource: [Review, Paper], context: {isReleased?: Bool}};
```

(a) Conference management system schema (snippet)

```
permit (principal, action == Action::"Read", resource is Paper)
when { principal.isPcMember };

permit (principal, action, resource is Review)
when { principal in resource.ofPaper.authors };
```

(b) Conference management system policies (snippet)

Fig. 1: Example Cedar schemas and policies

of the deliberately loose permit rules by adding appropriate general conditions instead of point-solutions, while also removing some over-privileges in the other cases. In summary, this paper makes the following contributions:

1. We formulate and formally define the access control policy tightening problem with respect to a given policy, access log, and policy state.
2. We present an approach that can incrementally tighten each policy rule using automated reasoning (*i.e.*, SyGuS, SMT).
3. We empirically demonstrate RESTRICTER's effectiveness and scalability on two realistic case studies.

2 Background on Cedar

This section presents a primer on Cedar. Due to space restrictions, we assume readers have some familiarity with ABAC, SyGuS, and SMT solvers. Further details on Cedar and these other topics are available from other sources [18,29,5,4,23,9,31]. Cedar [7] is an open-source authorization policy language developed (and used at scale, especially, to protect customer API end-points) by Amazon Web Services (AWS). We chose Cedar as the target of RESTRICTER to show that our approach is applicable to an authorization policy language that is practical, expressive, and actively used in industry, but RESTRICTER can also be extended to work on other ABAC policy languages.

2.1 Cedar Primer

We introduce some of Cedar's salient language features, using a conference management system inspired by HotCRP as a motivating example. This example comes from one of our case studies, which is described further in Section 7.1.

Schemas. Cedar schemas define the data model for the entities and actions to which policies apply. Declarations of entity types specify the names and types of

the entities' attributes. Valid attribute types include Booleans, entity types, and sets. Attributes can be *mandatory* or *optional*, indicated resp. by the absence or presence of the suffix `?` in their name. As an example, in Figure 1a, the entity type `Paper` declares that each paper has two attributes: its sets of authors and reviewers.

Schemas also list the set of *actions* and the types of entities to which each action applies. Optionally, action declarations may specify the type of contexts associated with the action. For example, Figure 1a declares the action `Read`, which a `User` may take on a `Paper` or `Review`, and which comes with a context containing an optional Boolean attribute indicating whether the resource has been released.

Policies. A Cedar policy is composed of *rules*[3]. Each rule is a Boolean-valued function of four parameters — `principal`, `action`, `resource`, and `context` (implicit) — comprising:

- an ***effect***, `permit` or `deny`, indicating whether to allow or prohibit requests to which the rule applies;
- a ***scope*** constraining the principals, actions, and resources to which the rule applies; and
- a ***body*** (optional), which places further constraints on the parameters, usually involving their attributes.

Example: In Figure 1b, the second rule has ***effect*** `permit`, a ***scope*** applying to any `principal` and `action` but only `resource`s of type `Review`, and a ***body*** further constraining the rule to only those `principal`s which are authors of the paper the review concerns.

Composing rule effects. When Cedar's authorization engine is given a request and policy, conceptually it applies each rule in the policy to the request. To assemble these results into a final decision, it resolves ambiguous cases as follows. ❶ *Default Deny:* If no `permit` rule applies, the request is denied. ❷ *Deny Overrides Permit:* If a `forbid` rule applies, the request is always denied.

Entity Hierarchies. Entity type declarations can also specify the types of *parent* entities. Each entity has zero or more parents. The reflexive transitive closure of the *parent of* relation comprises Cedar's *ancestor hierarchy*. This hierarchy provides support for role-based access control: the ancestor-descendant relationship can express membership of users in roles as well as role hierarchy.

3 Problem Definition

This section presents a motivating example and then formally defines the policy tightening problem that Restricter aims to solve.

3.1 Motivating Example

Suppose the program committee (PC) chairs of a popular computer security conference are setting up the HotCRP conference review system instance for this year.

[3] In Cedar, these are called "policy set" and "policy," resp. For our presentation, we choose the terms more commonly used in access control literature.

The current PC chairs obtained the policy from the previous year's chairs; this policy is the one from which the listing in Figure 1 samples. However, the chairs anticipate receiving a substantial number of paper submissions this year, which induced the decision of partitioning the papers into mutually disjoint areas (*e.g.*, usable security, software security, network security, *etc.*). In addition, they decided to have a decentralized administration by assigning an area to each reviewer. It is analogous to having a set of small conferences under a bigger conference.

In preparation, they extended every `Paper` to have an `area` attribute of type `Area`, denoting the area assigned to a paper. The `User` type is updated to include a new Boolean attribute `isAreaChair` and to replace the `isPcMember` attribute with an optional attribute `pcMember` of type `Area`, which when set indicates the user is a PC member for that area. The first rule in the example is modified to reflect the change. These modifications are shown in Figure 2.

```
// modified schema (snippet)
entity User { isPCChair: Bool, isAreaChair: Bool, pcMember?: Area};
entity Area;
entity Paper { authors: Set<User>, reviewers: Set<User>, area: Area};
// modified policy (snippet)
permit (principal, action == Action::"Read", resource is Paper)
when { principal has pcMember };
```

Fig. 2: Modified Cedar policy

We may observe in the log that each reviewer accesses only papers in their area. After simplification, RESTRICTER is able to tighten the modified rule in Figure 2 to:

```
permit(principal, action == Action::"Read", resource is Paper)
when { principal has pcMember && principal.pcMember == resource.area};}
```

which exactly captures the semantics of our observation.

3.2 Notations and Problem Definition

Notations. Let Σ be the schema that specifies the entity types, the attributes associated with them, the allowed hierarchies, and the allowed principals and resources for each action. Let $\mathcal{E}$ be the environment, also known as the entity store, that consists of the principals, resources, and their attributes. Let $\texttt{req} = \langle prin, act, obj \rangle$ be an access request that consists of the principal, action, and resource, respectively. Let $\mathcal{R}$ be the universe of all requests consistent with the schema. Let $\mathsf{PolicyEval}(\texttt{req}, \mathcal{P}, \mathcal{E})$ be the function that evaluates a request `req` with respect to a policy $\mathcal{P}$ and environment $\mathcal{E}$ according to Cedar's semantics, returning an access decision $d \in \{\mathsf{allowed}, \mathsf{denied}\}$. Let $[\![\mathcal{P}]\!]$ be the set of requests allowed by $\mathcal{P}$, *i.e.*, $[\![\mathcal{P}]\!] \triangleq \{\texttt{req} \mid \texttt{req} \in \mathcal{R} \wedge \mathsf{PolicyEval}(\texttt{req}, \mathcal{P}, \mathcal{E}) = \mathsf{allowed}\}$. If R^+

is one of the permit rules in $\mathcal{P}$, then we can similarly define the set of requests allowed by R^+ as $[\![R^+]\!] \triangleq \{\texttt{req} \mid \texttt{req} \in \mathcal{R} \wedge \mathsf{PolicyEval}(\texttt{req}, \{R^+\}, \mathcal{E}) = \mathsf{allowed}\}$.

An access log $\mathcal{L} \subset \mathcal{R} \times \{\mathsf{allowed}, \mathsf{denied}\}$ is a set of requests labeled with the decision for that request from the initial policy $\mathcal{P}_{\text{init}}$. Policy $\mathcal{P}$ is *consistent* with $\mathcal{L}$, denoted $\mathcal{P} \sim \mathcal{L}$, *iff* $\forall \langle \texttt{req}, d \rangle \in \mathcal{L}.\mathsf{PolicyEval}(\texttt{req}, \mathcal{P}, \mathcal{E}) = d$. We assume $\mathcal{P}_{\text{init}} \sim \mathcal{L}$.

The Policy Tightening Problem. Given a type-safe Cedar policy $\mathcal{P}_{\mathsf{init}}$, an access log $\mathcal{L}$, a universe of access requests $\mathcal{R}$, an entity schema Σ, and an environment $\mathcal{E}$, the *policy tightening problem* is to synthesize a type-safe Cedar policy $\mathcal{P}_\star$ such that: (I) $\mathcal{P}_\star \sim \mathcal{L}$, and (II) $[\![\mathcal{P}_\star]\!] \subseteq [\![\mathcal{P}_{\mathsf{init}}]\!]$. The rationale for (II) is to avoid violating the least privilege principle. In what follows, "*policy*" refers to a type-safe Cedar policy unless mentioned otherwise. When not explicitly identified, we consider a fixed schema Σ and an environment $\mathcal{E}$. We use $\mathcal{P}_\star$ to denote the tightened policy throughout the paper.

It is easy to see that the policy-tightening problem is under-constrained, since any subset of the unexercised permissions can be removed. A trivial solution is to return $\mathcal{P}_{\mathsf{init}}$; our algorithm is designed to remove unexercised permissions without changing the size of the policy too much and modifying the structure of the policy.

4 Design Dimensions

We discuss three design dimensions and their trade-offs that one may consider in developing an approach like RESTRICTER.

Global vs. Local Tightening. The first dimension is to consider whether to tighten the whole policy at once (*global*) or each policy rule individually (*local*).

- While costly, global tightening has the benefit of simplifying and minimizing the *entire* policy during tightening.
- Despite having a narrower view, local tightening can be more scalable: one can focus on the portions of $\mathcal{L}$ and entity store that are relevant to individual rules.

The local approach is further facilitated by Cedar's *default deny, deny overrides permit* semantics (Section 2.1): *as permit rules are the only ones allowing accesses, it is sufficient to tighten only permit rules locally.* We can refine the policy tightening problem as follows.

Let the set of allowed log requests be $\mathcal{L}^+ \triangleq \{\texttt{req} \mid \langle \texttt{req}, d \rangle \in \mathcal{L} \wedge d = \mathsf{allowed}\}$, and the log slice with respect to permit rule R^+ be $\mathcal{L}{\downarrow_{R^+}} \triangleq \mathcal{L}^+ \cap [\![R^+]\!]$. For policy tightening, it suffices to find a permit rule R^+_* for each permit rule R^+ in the input policy such that:

(I′): $R^+_\star \sim \mathcal{L}{\downarrow_{R^+}} \times \{\mathsf{allowed}\}$, and

(II′): $\forall \texttt{req} \in \mathcal{R}, \mathsf{PolicyEval}(\texttt{req}, \{R^+\}, \mathcal{E}) = \mathsf{denied} \implies \mathsf{PolicyEval}(\texttt{req}, \{R^+_\star\}, \mathcal{E}) = \mathsf{denied}$.

Here, we need to focus only on the permitted part of the log in (I′) as the denied part is guaranteed to be consistent by (II′).

Synthesizing vs. Incremental Strengthening. The next design dimension is whether to generate a new policy or rule, replacing the previous one, or strengthen the current policy or rule incrementally.

– Synthesizing new policies or rules is *prima facie* a more general approach, and has a larger search space.
– Incremental strengthening has a much more tractable search space.

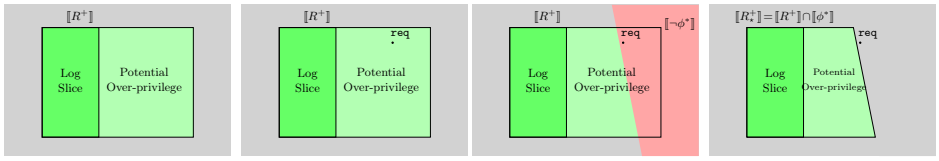
Permit vs. Deny. Even when one chooses the latter approach, there are two alternatives: (i) adding or broadening deny rules to remove over-privileges, or (ii) strengthening permit rules to reduce over-privilege. From the perspective of scalability, the search space for (ii) can be further reduced as it suffices to identify additional conditions to be imposed on the permit rules. The additional conditions can be in the form of restricting the ***scope*** or ***body*** of the permit rule. The conditions one can place in the ***scope*** of a Cedar rule are restricted to certain forms — too restrictive for productively tightening. For the ***body***, one can always add a conjunctive formula (any Boolean-valued Cedar expression) to a rule to tighten it. This has the benefit of being amenable to incremental tightening of a rule by adding one conjunct at a time.

Encoding Approaches. Irrespective of global/local, and synthesis/refinement approaches to tightening, a primary challenge is to ensure that any automated reasoning approaches for policy tightening not only scale with the size of the input but also generate a tightened Cedar policy that is both type-safe and satisfies the restrictions (I) and (II) (or their primed versions) discussed above.
– We can take a *syntactic approach* to policy encoding, modeling the abstract syntax tree (AST) of our supported subset of Cedar as an algebraic datatype (ADT); or
– We can take a *semantic approach*, translating the policy as a quantifier-free first-order logic (QF-FOL) formula.

The *syntactic approach* (a.k.a. *deep embedding*) is natural: the policy to be synthesized is a constant of the ADT type, and PolicyEval is a recursive function over values of the ADT expressing Cedar's operational semantics. One can then invoke the SMT solver's finite model finding capability for policy tightening. In our evaluation, such an approach suffered from severe scalability issues, as *user-defined ADTs are not amenable to optimizations enjoyed by native SMT theories.*

In the *semantic approach* (a.k.a a *shallow embedding*) a Cedar policy $\mathcal{P}$ with a set of permit rules $\{R_1^+,R_2^+,...,R_m^+\}$ and a set of forbid rules $\{R_1^-,R_2^-,...,R_n^-\}$, can be viewed as a QF-FOL formula: $\bigwedge_{i=1}^{n}\boxminus(R_i^-)\wedge\bigvee_{j=1}^{m}\boxplus(R_j^+)$. Here, $\boxplus(\cdot)$ and $\boxminus(\cdot)$ are functions that take a permit and deny policy rule, respectively, and return their QF-FOL representation. A rule has the form $\langle p,a,o,\phi\rangle$, where p,a,o represent conditions on principals, actions, and resources respectively, and ϕ represents the body. A permit rule $R^+=\langle p,a,o,\phi\rangle$ is translated to $\Box p\wedge\Box a\wedge\Box o\wedge\Box\phi$, where $\Box c$ is the translation (as a conjunctive formula in the QF-FOL fragment) of a Cedar condition c. Similarly, a forbid policy rule $R^-=\langle p,a,o,\phi\rangle$ is translated to $\Box p\wedge\Box a\wedge\Box o\rightarrow\neg\Box\phi$ (notice the negation).

Therefore, we can represent a Cedar policy as an SMT term mimicking the above QF-FOL form and restricting ourselves mostly to native theories when possible. PolicyEval conceptually reduces to the evaluation of an SMT term. Finding a policy/rule becomes an instance of a SyGuS problem where the function to be synthesized is a QF-FOL formula that captures the evaluation of all the



(a) $[\![R^+]\!]$ is partitioned into the log slice and potential over-privilege.

(b) Choose a request `req` in the set of potential over-privileges.

(c) Use SyGuS to synthesize a predicate ϕ^* that denies `req`.

(d) The rule $R^+_\star$ that includes ϕ^* as a conjunct denies `req`.

Fig. 3: An illustration of RESTRICTER's main idea for Step ❷. RESTRICTER chooses requests in the set of potential over-privileges to be denied by a conjunct ϕ (to be synthesized with SyGuS) while keeping the log slices permitted.

policies. We can also encode the original policy rules as SMT terms and use them as semantic constraints. A challenge still remains: *capturing Cedar type-safety as semantic constraints does not scale.*

5 RESTRICTER's Approach

RESTRICTER takes as input the current policy $\mathcal{P}_{\mathsf{init}}$, an environment $\mathcal{E}$, and an access log $\mathcal{L}$, and generates a tightened policy $\mathcal{P}_\star$. Out of the design choices mentioned in Section 4, RESTRICTER's approach embraces the following choices: (a) local, rule-based tightening; (b) incremental strengthening of each permit rule of the input policy through the introduction of conjunctive conditions; and (c) SyGuS-based approach for identifying the conjunctive conditions for rule strengthening. *By adding conjunctive formulas to permit rules, RESTRICTER syntactically guarantees that the tightened policy $\mathcal{P}_\star$ satisfies $[\![\mathcal{P}_\star]\!] \subseteq [\![\mathcal{P}_{\mathsf{init}}]\!]$.*

Sketch of RESTRICTER's Approach. RESTRICTER's algorithm is presented in Algorithm 1. At a high level, RESTRICTER's approach is as follows: ❶ collect each permit rule R^+_i from $\mathcal{P}_{\mathsf{init}}$; ❷ iteratively strengthen each permit rule R^+_i from $\mathcal{P}_{\mathsf{init}}$ *in parallel* to obtain a tighter permit rule $R^+_{i\star}$; and ❸ collect all the forbid rules of $\mathcal{P}_{\mathsf{init}}$ and combine them with the newly strengthened permit rules $R^+_{i\star}$ to obtain the tightened policy $\mathcal{P}_\star$. We mainly discuss Step ❷ of the overall approach.

High-level View of Tightening a Permit Rule (Step ❷). Before delving into the details, we first present a high-level view of RESTRICTER's approach to Step ❷; an illustration of the ideas is also presented in Figure 3. The rest of this section expands upon each of the following steps. RESTRICTER iteratively performs the following steps, while using the obtained strengthened policy of an iteration as input to the next, until some termination condition is met.

i. Calculate Relevant Log Slice: We first calculate $\mathcal{L}{\downarrow_{R^+_i}}$, the set of permitted requests in $\mathcal{L}$ that are also permitted by R^+_i.

ii. Calculate Approximate Over-Privileges in R^+_i: We then calculate an over-approximation of the set of over-privilege requests $\mathbb{R}$, such that all $\texttt{req} \in \mathbb{R}$ satisfies $\texttt{req} \notin \mathsf{dom}(\mathcal{L})$ (`req` is not in the log) and `req` is permitted by R^+_i.

Algorithm 1 Main algorithm of RESTRICTER

```
1: procedure RESTRICT(P_init, L, R, Σ, E, t)
2:     P_* ← ∅
3:     for all positive rule R+ of P_init do
4:         R*+ ← R+
5:         repeat
6:             if [[R*+]] \ L↓_{R+} = ∅ then break
7:             POP ← [[R*+]] \ L↓_{R+}                          ▷ Steps i. and ii.
8:             Pick requests {req_i}_i ⊂ POP
9:             R*+ ← RESTRICT_ONE(R*+, L↓_{R+}, Σ, E, {req_i}_i)   ▷ Steps iii. and iv.
10:            if R*+ = ⊥ then restore the previous R*+
11:        until R*+ = ⊥ for t times
12:        Insert the previous value of R*+ into P_*
13:    Insert all negative rules R− into P_*
14:    return P_*
```

iii. Calculate Type-safe Cedar Predicate List: We precalculate the candidate predicates that SyGuS enumerates over. This approach avoids needing to encode Cedar's type checking rules into SyGuS, and instead delegate it to the meta-program generating the SyGuS problem instance.

iv. Invoke SyGuS: We finally invoke SyGuS to generate a type-safe Cedar predicate p such that $\boxplus(R_i^+) \wedge \square(p)$ (*i.e.*, the tighter rule $R_{i\star}^+$) allows every request in $\mathcal{L}{\downarrow_{R_i^+}}$, and denies at least one request in $\mathbb{R}$.

5.1 Synthesizing a tighter permit rule (Step ❷).

We now present descriptions of Steps (i) - (iii). Details of Step (iv) are in Section 6.

Finding a Separating Predicate for Tightening. Figure 3 shows the main idea of tightening a permit rule. We over-approximate the over-privileges as the accesses that do not appear in the log. Given the log and the permit rule, we compute the slice $\mathcal{L}{\downarrow_{R^+}}$. Then, the set of *potential unexercised over-privileges* $\mathsf{POP} \triangleq [\![R^+]\!] \setminus \mathcal{L}{\downarrow_{R^+}}$ consists of the requests permitted by the rule but not in $\mathcal{L}{\downarrow_{R^+}}$ (Figure 3a).

Given a permit rule $R^+ = \langle p,a,o,\phi\rangle$, we use SyGuS to find a *separating* predicate ϕ^* such that $R_\star^+ = \langle p,a,o,\phi \wedge \phi^*\rangle$ is consistent with the log slice, and denies some requests in POP (Figure 3c), Here, the restrictiveness requirement (II') can be dropped because $R_\star^+$ is syntactically guaranteed to be more restrictive. Denying some requests in POP makes $R_\star^+$ *strictly* more restrictive than R^+. We curate a list of candidate predicates that are type-safe, and the syntactic constraint becomes choosing one of the candidate predicates. If there is a predicate that satisfies the constraints, then $R_\star^+$ is the solution. Otherwise, $\bot$ is returned.

Avoiding Existential Quantification. To formulate that the separating predicate denies *some* (unspecified) request, we would need existential quantification. To avoid this, we explicitly choose some requests $\{\texttt{req}_i\}_i$ in POP, and assert that at least one of them is denied in $R_\star^+$ (Figure 3b). The new rule $R_\star^+$ (if SyGuS

Algorithm 2 Restrict a rule by adding one conjunct

procedure RESTRICT_ONE(R^+, Log_Slice, Σ, $\mathcal{E}$, D)
  $\langle p,a,o,\phi\rangle \leftarrow$ symbolic encoding of R^+ under Σ
  $P \leftarrow$ Set of type-safe candidate predicates ▷ *Step iii.*
  Ask SyGuS to construct a function $f : \mathcal{R} \to \{\mathsf{allowed},\mathsf{denied}\}$ that takes the form of a rule $\langle p,a,o,\phi\wedge\phi^*\rangle$ where $\phi^* \in P$, such that

    1. $\forall \mathtt{req} \in \mathsf{Log\_Slice}, f(\mathtt{req}) = \mathsf{allowed}$, and
    2. $\bigvee_{\mathtt{req}^* \in D} (f(\mathtt{req}^*) = \mathsf{denied})$ holds ▷ *Step iv.*

  if SyGuS fails **then return** $\bot$
  $R^+_\star \leftarrow$ encoding of $\langle p,a,o,\phi\wedge\phi^*\rangle$ in Cedar
  return $R^+_\star$

returns one) is strictly more restrictive than R^+ (Figure 3d). We can repeat this process until either POP is empty, or we fail to generate a predicate for a specified number of iterations (with different chosen requests $\mathtt{req}_i$). The former means that we have obtained the tightest rule that only permits everything in the $\mathcal{L}{\downarrow_{R^+}}$. Section 9 discusses other termination conditions.

Generating Type-safe Predicates. To generate type-safe predicates, we analyze the Cedar schema and policy in the meta-program that generates candidate predicates. Then, we enumerate relevant type-safe predicates (Section 6.1) in our own symbolic encoding to be fed to SyGuS as syntactic constraints.

5.2 Choosing Concrete Requests to Deny

Explicit Enumeration. We can choose a random point in POP, or try to restrict every point in POP in parallel (see Figure 3b). Both would require explicit computation of the set POP, which entails enumerating over the set of requests $\mathcal{R}$ (with types restricted when possible) and checking whether R^+_* applies. As the size of $\mathcal{R}$ grows with the number of entities, this would introduce a large overhead.

Request Generation with SMT Solver. Alternatively, we can take the SMT encoding of the permit rule R^+, and ask the SMT solver to generate a concrete request req that satisfies R^+ and is not in $\mathcal{L}{\downarrow_{R^+}}$. If SyGuS fails to produce a tightened rule, then we add the picked request to the set of requests to be blocked and repeat. To reduce the likelihood of the SMT solver returning similar requests in different iterations, we randomize the entity encoding in our SMT representation, specifically, the mapping from entities to integers (discussed in Section 6.1).

6 Implementation

RESTRICTER is implemented in ∼2100 lines of Python and ∼400 lines of Rust. The main challenge is to implement the function RESTRICT_ONE (Algorithm 2). We first encode the input Cedar policy using our symbolic compiler as an SMT

term. Along with the encoded policy, entity store and problem constraints, we invoke CVC5's [8] SyGuS engine [28], which takes semantic constraints and syntactic constraints in the form of SMT terms, to solve the tightening problem. The resulting rule, encoded as an SMT term, is translated back to a Cedar rule as output. The request generation routine also uses a fragment of the symbolic compiler to encode the policy rules.

6.1 Symbolic Encoding of Cedar

We now discuss our encoding of entity stores and policies as SMT terms to make them amenable to constraint solving (See Section 5.2) and synthesis (Finding separating predicate as discussed in Section 5.1). We also discuss how our approach differs from that of Cutler *et al.*'s symbolic compiler [18, Section 4].

There are three main ways that our encoding differs from the one in Cutler *et al. First*, Cutler *et al.*'s encoding does not consider specific entity stores, so operators like hierarchy relations are left as uninterpreted functions. We are given a concrete entity store, and can encode the hierarchy with concrete values.

Second, Cutler *et al.*'s *encoder* treat different entities as different types. The encoder meta-program instantiates the concrete types and pre-evaluate some expressions into the SMT solver. In contrast, the policy evaluation function in the *SMT solver* should take all possible entity types that are permitted by the schema as possible inputs. Therefore, we define one general entity type in the encoder and allow entities of any type to act as input to our function to synthesize, then make sure we only produce cedar-type-safe predicates in the encoder. As an example, consider the expression "`resource has ofPaper`" using the schema in 1a that is true when the type of `resource` is `Review`, and false otherwise. It is not possible to reason on the types of SMT terms using SMT terms. The approach in Cutler *et al.* tries each instantiation of types for `resource`. For type `Review`, for example, the expression is compiled directly to the SMT term `true`. In contrast, we have to evaluate this expression inside the SMT solver, since we cannot determine the concrete type of `resource` that goes into the policy evaluation beforehand. Therefore, our encoding of `has` would be a function that takes the entity type and consists of `ite`'s that test on the `type` (which is an element in the constructor of the one general entity type) and the name of the attribute.

Finally, we try to use simpler theories when possible in our encoding. As an example, we encode the name of the entities as integers instead of strings as the entity store only contains a small number of entities, and we can avoid invoking the string solver that may be expensive.

General entity type. In Cedar, an entity has a type and a name string. In our approach, all entities are encoded using a single datatype E that has a single constructor with `type` and `name` as integer arguments instead of the string type used in Cedar. We map each type in the schema and entity name in the entity store to an integer. In the semantic constraint, we state that the range of type and id of principals and resources is bounded by the image of these mappings. Equalities

Feature Support Examples		
Equality	●	`principal.role == Role::"Teacher"`
Inequality	●	`User::"Alice" != User::"Bob"`
Attribute presence	●	`principal has pcMember`
Entity type test	●	`principal is Student`
Boolean negation	●	`!(principal in resource.authors)`
Integer comparison	◐	`principal.balance >= resource.cost`
Hierarchy membership	◐	`principal in resource.course`
Set membership	◐	`principal in resources.authors`
Conjunction, Disjunction	○	`true && true  true || true`
Integer operators	○	`4 * 10 + 5 - 3`

●: Full predicate generation support ◐: Limited predicate generation support
○: No predicate generation support

Table 1: Summary of RESTRICTER's support for Cedar.

over entities are simply equalities over the type and id in the constructors. Representing these as integers rather than finite strings makes reasoning more efficient.

Entity attributes. The attributes of each entity type are defined in the schema with the name f and type t. For each attribute name-type pair $f:t$ in the entity store, we define a function $get_{f,t}: E \to \text{Option } t$ that takes an entity of any type that are in the entity store and returns either the value of the attribute of the entity or `None` when either (1) the attribute does not exist for the type, or (2) the attribute is marked optional for the entity in the store, and the given entity does not have the attribute. We also have for every attribute a predicate $has_f: E \to \text{Bool}$ that returns *true* if $f:t$ is an attribute for e in the entity store for some type t.

Entity Hierarchy. We encode a given entity store's hierarchy relation as a concrete predicate by pre-computing the relation's reflexive, transitive closure.

6.2 Syntactic Constraints

We now describe the syntactic constraints that are posed to SyGuS to restrict candidate rules. If the original rule is $\langle p,a,o,\phi \rangle$, then the output rule has the form $\langle p,a,o,\phi \wedge \phi^* \rangle$, where ϕ^* is a type-safe Cedar predicate. We pre-compute the list of candidate type-safe Cedar predicates. Table 1 shows the subset of Cedar that RESTRICTER can parse, as well as the degree of RESTRICTER's ability to generate predicates utilizing the listed features. The features with only partial support for predicate generation, such as set membership tests, are intentionally limited to avoid enumerating arbitrary constants and instead only search applicable combinations of parameters, their attributes, and constants that appear in the entity store. Features not listed in Table 1, such as Cedar's various extension types, cannot be consumed by the current version of RESTRICTER.

Type-safety. For scalability, our encoding ensures type-safety outside SyGuS. As an example, the rule permit(principal,action,resource) when {principal.isAdmin} is not type-safe in Cedar unless all entity types that can be principals have

the Boolean attribute `isAdmin`. In contrast, the following permit rule in Cedar permit(principal,action,resource) when {**principal is User**&&principal.isAdmin}; is type-safe if `User` has the Boolean attribute `isAdmin`. To ensure our predicates are type-safe in Cedar, we syntactically restrict the types appearing in the expressions.

Equality. Since equality requires both sides to have the same type, we can enumerate possible equalities by enumerating all constants of each type, identifying entities with attributes of that type, and generating equalities between them. Enumerating the constants is straightforward, since the environment is known, and we can collect the values that appear in attributes of known entities. The main challenge is identifying when an entity has an attribute of a given type. To ensure an entity e has a given attribute f of type t, we can add a guard $has_f(e)$ at the front. Alternatively, for an entity e of type t_e, if t_e has a (required) attribute $f : t$, we can add the guard e `is` t_e. Now, for every entity $e_1 : t_1$ and entity $e_2 : t_2$ that have attributes $f_1 : t$ and $f_2 : t$, respectively, we can write the following type-safe predicate as a candidate predicate for Restricter: e_1 `is` $t_1 \wedge e_2$ `is` $t_2 \longrightarrow e_1.f_1 == e_2.f_2$. We also consider equality where one side is a constant like e `is` $t \longrightarrow e.f == const$ where f and $const$ have the same type.

Similar constructions work for set membership and hierarchies. For example, for set membership, we can write the type-safe predicate for Restricter: e_1 `is` $t_1 \wedge e_2$ `is` $t_2 \longrightarrow e_1.f_1 \in e_2.f_2$ if the type of f_1 is t and type of f_2 is `Set` t.

Attribute Chains. In Cedar, we can chain attributes together when the intermediate attributes have Entity or Record types. Using the schema in Figure 1a, if `resource` has type `Review`, we can write `resource.author.isPCChair`. It may be possible, depending on the schema, to have arbitrary nested or even cyclic attribute chains. For example, if `Paper` has a `metareview` attribute of type `Review`, then we can nest an arbitrary number of `.ofPaper.metareview` in `resource.ofPaper.metareview`.

While generating candidate expressions, instead of leaving the attribute accesses recursive on the grammar level, we enumerate all type-safe attribute chains up to a specified depth, which is a parameter of our algorithm, to limit our search space. We can ensure that the attribute chains are well-typed in Cedar, because we know the exact type of each attribute from the schema.

7 Evaluation

This section details our evaluation of Restricter, beginning with our guiding research questions.

Q1: How frequently does Restricter produce desired tightenings? When it fails to do so, how much over-privilege does it eliminate, and how much legitimate privilege does it preserve?

Q2: How well does the performance of Restricter scale with respect to entity store size and log size (*i.e.*, the numbers of entities in the entity store and access requests in the access log)?

Some challenges arise in answering these questions which further influence our case study design and evaluations.

```
// P_tight
permit (principal, action in Action::"Read",resource is Paper)
when { principal has isPcMember && principal.pcMember == resource.area };
// P_init
permit (principal, action in Action::"Read",resource is Paper)
when { principal has isPcMember };
```

Fig. 4: Example corresponding rule pair (HotCRP)

Challenge 0: Unfortunately, none of the previous works on policy tightening [19,20,24] open source the tools or the test cases they used for evaluation. As such, we are not able to benchmark against the previous works.
Challenge 1: Due to their sensitive nature, it is difficult to obtain real-world examples of access control policies or logs. *Solution:* Our case studies are based on real-world management systems or examples found in the access control literature [1,2].
Challenge 2: Quantifying the answer to **Q1** requires a precise characterization of a policy's over-privilege. In practice, this is usually not available. *Solution:* Our case studies are designed with a loose initial policy and an intended tight policy. This intent is reflected indirectly in the access log and used to evaluate the policy rules produced by RESTRICTER, but is never given to it directly.
Challenge 3: The sizes of the entity store and log are not the only factors impacting the performance. The shape of the entity store (*i.e.*, the number and structure of relationships between entities) also affects performance. *Solution:* Our input generation approach is highly flexible, allowing control over the shape as well as the size of the entity store.

7.1 Design of Case Studies

We now overview the high-level design of our case studies, while the details can be found in the full version of the paper [32]. The two case studies used to evaluate RESTRICTER are a classroom management system inspired by Google Classroom [1] and a conference management system inspired by HotCRP [2]. Figure 4 shows an example for the HotCRP case study. For each case study, we crafted a schema and two policies: $\mathcal{P}_{\mathsf{tight}}$, the desired tight policy; and $\mathcal{P}_{\mathsf{init}}$, which we derived by deliberately introducing over-privilege in some rules of $\mathcal{P}_{\mathsf{tight}}$ by dropping a conjunct. We generate the random entity stores and access logs for our case studies with a bespoke program (∼3K LoC in Haskell). The program can generate sets—a.k.a. "families"—of related entity stores and logs of varying size, where entities and log entries in smaller members also appear in larger members. The members are indexed by a `size` parameter. The size of the entity store and logs are roughly linear to the `size` parameter.

7.2 Evaluation Results

We ran our experiments on a server with two Xeon Gold 5418Y CPUs totaling 96 threads and 256GB of RAM. We used CVC5 version 1.2.0 and the Cedar

authorization engine version 3.2.1. For each test case, we invoke Restricter to tighten every permit rule sequentially to obtain accurate measurements, even though Restricter is parallelizable. We set the termination condition to 2 total failures, and generate three requests as a potential over-privilege to be denied in each iteration. For the datasets, we generated 100 dataset families using distinct seeds and varying log density from 30 to 100 percent with 10 percent increments. We repeat each experiment three times, taking the median running time (CPU time), among other statistics, as measurements. The full experimental results are in the full version of the paper [32].

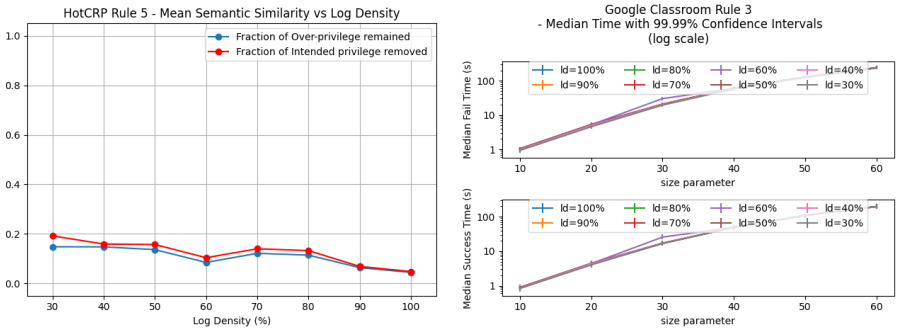
Q1: Effectiveness. We run Restricter on the input policy and compare its output to $\mathcal{P}_{\mathsf{tight}}$. For cases where we do not get the intended tightening, we can still compute semantic similarities between the generated tightening and the intended tightening by enumerating all the over-privileges and intended privileges for each test case, and computing, for the generated tightening, the percentage of over-privileges remaining and intended (but unexercised) privileges removed. In the Google Classroom case study, all loosened rules are tightened with the expected conjunct (except with added type guards). For the HotCRP case study, two out of the five loosened rules were tightened to the ideal tight rules, while the other three still achieved good semantic similarity. Figure 5a shows the semantic similarity from one rule (with `size`=30) of the HotCRP case study. For this rule, both semantic similarity metrics improve (lower is better) as the log density increases, most likely because the real over-privileges got chosen instead of intended privileges in the request selection stage. Our results also confirm that our approach of adding one conjunct at a time discourages point solutions where we end up with an expression that denies one specific point.

In short, in our case studies, Restricter successfully tightens most of the loose rules, and preserves most rules that were not loosened. On the few loosened rules where it was not tightened ideally, it still removes some over-privileges while keeping most intended privileges, even when the log density is low.

Q2: Scalability. To evaluate scalability, we measure Restricter's running time. We vary the problem size in two dimensions: the `size` parameter and the log density. Figure 5b shows the time for each successful and failed tightening from SyGuS when varying the `size` parameter and the log density (ld). Figure 5b is representative, showing that in general, Restricter runs in sub-exponential time when increasing the number of entities while being generally unaffected by the size of the log, which is proportional to the log density. This shows that Restricter can handle problems of larger sizes efficiently.

8 Related Work

Manually developing access control policies is well-known to be challenging. There is a sizable body of literature on algorithms for mining (*a.k.a.* learning) access control policies. Work on learning ABAC or ReBAC policies starts with Xu et al. [34,33] and now spans many papers (*e.g.*, [26,21,30,11,17,12,16]). The



(a) Semantic Similarity of the synthesized rule as a function of the log density, in terms of the fraction of the over-privilege remained and the intended privilege removed (lower is better) in the HotCRP conference management case study.

(b) Running time (log scale) as a function of the size parameter and the log density (ld) with the 99.99% confidence intervals (stemming from the sign test for the median) of the median shown as error bars when SyGuS succeeds and fails to produce a tightening one of the rules in the Google classroom case study.

Fig. 5: Excerpt of the evaluation results in terms of semantic similarity of the generated rules versus the ideal ones and the running time.

general problem, like the policy tightening problem and for similar reasons, is under-constrained. None of these algorithms are based on SyGuS.

There are only a few efforts on algorithmically refining a given ABAC policy. IAM-PolicyRefiner [19] and Eiers *et al.* [20] both tighten AWS IAM policies whereas Mitani et al. [24] use machine learning to refine ABAC policies. IAM-PolicyRefiner and Eiers *et al.*, like Restricter, tighten one rule at a time. However, both restrict the changes to a rule to *local* changes to predicates that appear in the original rule. Restricter is able to introduce predicates absent in the original policy. This makes tightening possible in more cases. They mainly focus on generating regular expressions that restricts the names of the principal and resource, which is the main feature of IAM policies. We target Cedar, which is more expressive. Mitani *et al.* relies on "qualitative intention" as extra input, which Restricter does not. This makes Mitani *et al.* less suitable for cloud access control providers, where user intent is not known a priori.

9 Discussion

Assumptions and Limitations. Restricter assumes the input policy is type-safe, and that the schema and environment are fixed. A limitation of Restricter is that it only considers tightening by adding a conjunction of atomic predicates to a rule. Extending Restricter to consider more complex changes, such as adding arbitrary Boolean combinations of atomic predicates, is future work. We also assume that all permitted requests in the log should be permitted in the tightened policy. However, this may not be always true in practice. This is

because there might be accidental or malicious accesses in the log that exercise permissions that are not intended. It is feasible to extend RESTRICTER to consider this case by loosening our semantic constraints to allow some of the entries in the log to not be permitted in the tightened rule.

Termination Criterion. By default, RESTRICTER stops tightening a permit rule when SyGuS fails a specified number of times. RESTRICTER supports other termination criteria, such as a preset timeout or termination when a certain percentage of tightening has been achieved (*i.e.*, a certain percentage of POP has been removed). Exploring such termination conditions is future work.

Rule overlap. If there is overlap between rules (*i.e.*, a request is permitted because of two or more permit rules), then RESTRICTER's rule-level tightening may not produce a policy that is as tight as expected. This is because, in order to remove this overlap, RESTRICTER would need to successfully tighten all such rules, and each such tightening must exclude the overlap. This difficulty gives rise to an interesting, orthogonal direction of future work: *can the techniques employed by RESTRICTER be effectively adapted to reducing overlap between rules, as an aid for policy maintainability*?

Manual Vetting. Our intention with RESTRICTER is for it to propose tightened rules that need to be vetted by administrators to ensure that tightening does not remove intended privileges. Manual vetting is practical, as RESTRICTER preserves the original rule's readability by limiting changes to each rule by adding conjuncts.

Using SyGuS. Using a SyGuS solver to choose from a set of pre-computed predicates, as RESTRICTER currently does, is one of many methods to consider for generating and evaluating potential tightenings. Although we do not currently use SyGuS for predicate generation, we wish to keep this open for further extension, since a bespoke term enumerator will likely be more difficult to extend.

Concerning the evaluation stage, one can encode the predicate directly into SMT constraints, but this would also require encoding *type safety* constraints. RESTRICTER avoids this difficulty by providing the SyGuS solver with *type-safe predicates*, effectively enforcing type safety through the grammar.

Missing features of Cedar. Currently, there are features in Cedar for which RESTRICTER does generate predicates during tightening. RESTRICTER can be modularly extended, under some assumptions, to support some of them. For example, predicates of the form $x \in S$ where S is a constant set can be generated by limiting the cardinality of S and considering only constants that appear in the log and entity store. Among the missing Cedar features, a notably challenging feature to support is linear integer arithmetic (LIA) predicates of the form x + y $<$ c, where c is a symbolic constant to be synthesized. One can consider existing approaches to this problem to support this feature [3].

Data-Availability Statement

We have open-sourced RESTRICTER. RESTRICTER, the case studies, and the test case generation code can be found at https://github.com/REASONERSLab/Restricter.

References

1. Google Classroom. https://edu.google.com/intl/ALL_us/workspace-for-education/products/classroom/, accessed: 2025-05-08
2. HotCRP. https://hotcrp.com/, accessed: 2025-05-08
3. Abate, A., Barbosa, H., Barrett, C., David, C., Kesseli, P., Kroening, D., Polgreen, E., Reynolds, A., Tinelli, C.: Synthesising programs with non-trivial constants. Journal of automated reasoning **67**(2), 19 (2023)
4. Abate, A., David, C., Kesseli, P., Kroening, D., Polgreen, E.: Counterexample guided inductive synthesis modulo theories. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 270–288. Springer International Publishing, Cham (2018)
5. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. IEEE (2013)
6. Alur, R., Singh, R., Fisman, D., Solar-Lezama, A.: Search-based program synthesis. Communications of the ACM **61**(12), 84–93 (Nov 2018). https://doi.org/10.1145/3208071, https://doi.org/10.1145/3208071
7. Amazon Web Services, Inc.: Cedar Language. https://www.cedarpolicy.com/en (2025), Accessed: Jan 2025
8. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24
9. Barrett, C.W., de Moura, L.M., Ranise, S., Stump, A., Tinelli, C.: The SMT-LIB initiative and the rise of SMT - (HVC 2010 award talk). In: Barner, S., Harris, I.G., Kroening, D., Raz, O. (eds.) Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers. Lecture Notes in Computer Science, vol. 6504, p. 3. Springer (2010). https://doi.org/10.1007/978-3-642-19583-9_2, https://doi.org/10.1007/978-3-642-19583-9_2
10. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1267–1329. IOS Press (2021). https://doi.org/10.3233/FAIA201017, https://doi.org/10.3233/FAIA201017
11. Bui, T., Stoller, S.D., Li, J.: Mining relationship-based access control policies. In: Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies. pp. 239–246 (2017)

12. Bui, T., Stoller, S.D., Li, J.: Greedy and evolutionary algorithms for mining relationship-based access control policies. Computers & Security **80**, 317–333 (jan 2019)
13. Casbin Organization: Casbin. https://casbin.org/ (2025), Accessed: Jul 2025
14. Chen, H., Li, N., Enck, W., Aafer, Y., Zhang, X.: Analysis of SEAndroid Policies: Combining MAC and DAC in Android. In: Proceedings of the 33rd Annual Computer Security Applications Conference. pp. 553–565 (2017)
15. Cloud Native Computing Foundation: Open Policy Agent. https://www.openpolicyagent.org/ (2025), Accessed: Jul 2025
16. Cotrini, C., Corinzia, L., Weghorn, T., Basin, D.: The next 700 policy miners: A universal method for building policy miners. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 95–112 (2019)
17. Cotrini, C., Weghorn, T., Basin, D.: Mining ABAC rules from sparse logs. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 31–46. IEEE (2018)
18. Cutler, J.W., Disselkoen, C., Eline, A., He, S., Headley, K., Hicks, M., Hietala, K., Ioannidis, E., Kastner, J., Mamat, A., McAdams, D., McCutchen, M., Rungta, N., Torlak, E., Wells, A.M.: Cedar: A new language for expressive, fast, safe, and analyzable authorization. Proc. ACM Program. Lang. **8**(OOPSLA1) (apr 2024). https://doi.org/10.1145/3649835, https://doi.org/10.1145/3649835
19. D'Antoni, L., Ding, S., Goel, A., Ramesh, M., Rungta, N., Sung, C.: Automatically Reducing Privilege for Access Control Policies. Proceedings of the ACM on Programming Languages **8**(OOPSLA2), 763–790 (2024)
20. Eiers, W., Sankaran, G., Bultan, T.: Quantitative policy repair for access control on the cloud. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 564–575 (2023)
21. Gautam, M., Jha, S., Sural, S., Vaidya, J., Atluri, V.: Poster: Constrained policy mining in attribute based access control. In: Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies. p. 121–123. SACMAT '17 Abstracts, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3078861.3084163, https://doi.org/10.1145/3078861.3084163
22. Hu, V.C., Ferraiolo, D., Kuhn, R., Friedman, A.R., Lang, A.J., Cogdell, M.M., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K., et al.: Guide to Attribute Based Access Control (ABAC) Definition and Considerations (Aug 2019). https://doi.org/https://doi.org/10.6028/NIST.SP.800-162
23. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. pp. 215–224 (2010)
24. Mitani, S., Kwon, J., Ghate, N., Singh, T., Ueda, H., Perrig, A.: Qualitative intention-aware attribute-based access control policy refinement. In: Proceedings of the 28th ACM Symposium on Access Control Models and Technologies. pp. 201–208 (2023)
25. MITRE: CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html (2024)
26. Mocanu, D., Turkmen, F., Liotta, A.: Towards ABAC policy mining from logs with deep learning. In: The 18th International Multiconference, IS2015, Intelligent Systems, Ljubljana, Slovenia. (2015)
27. OWASP: Top 10 web application security risks. https://owasp.org/Top10/ (2021)
28. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City,

NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 74–83. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_5, https://doi.org/10.1007/978-3-030-25543-5_5
29. Standard, O.: eXtensible Access Control Markup Language (XACML) Version 3.0. A:(22 January 2013). URl: http://docs.oasis-open. org/xacml/3.0/xacml-3.0-core-spec-os-en. html (2013)
30. Talukdar, T., Batra, G., Vaidya, J., Atluri, V., Sural, S.: Efficient bottom-up mining of attribute based access control policies. In: 2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC). pp. 339–348. IEEE (2017)
31. The SMT-Lib Initiative: SMT-Lib The Satisfiability Modulo Theories Library. https://smt-lib.org (2025), Accessed: Jan 2025
32. Wu, K.L., Jenkins, C., Stoller, S.D., Chowdhury, O.: Automatically tightening access control policies with restricter (2026), https://arxiv.org/abs/2601.14582
33. Xu, Z., Stoller, S.D.: Mining attribute-based access control policies. IEEE Transactions on Dependable and Secure Computing (2014). https://doi.org/http://dx.doi.org/10.1109/TDSC.2014.2369048, http://dx.doi.org/10.1109/TDSC.2014.2369048
34. Xu, Z., Stoller, S.D.: Mining attribute-based access control policies from logs. In: Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec 2014). Lecture Notes in Computer Science, vol. 8566, pp. 276–291. Springer (2014)