Automatically Reducing Privilege for Access Control Policies

LORIS D'ANTONI, Amazon Web Services, USA
SHUO DING*, Georgia Institute of Technology, USA
AMIT GOEL, Amazon Web Services, USA
MATHANGI RAMESH, Amazon Web Services, USA
NEHA RUNGTA, Amazon Web Services, USA
CHUNGHA SUNG, Amazon Web Services, USA

Access control policies are programs used to secure cloud resources. These polices should only grant the necessary permissions that a given application needs. However, it is challenging to write and maintain policies as applications and their required permissions change over time.

In this paper, we focus on the Amazon Web Services (AWS) IAM policy language and present an approach that, given a policy, synthesizes a modified policy that is more restrictive and better abides to the principle of least privilege. Our approach looks at the actual access history (e.g., access logs) *used* by an application and computes the least permissive local modification of the user-given policy that still provides the same permissions that were observed in the access history. We treat the problem of computing the least permissive policy as a generalization problem in a lattice of possible policies (i.e., the set of local modifications). We show that our synthesis algorithm comes with correctness guarantees and is amendable to an efficient implementation that is easy to parallelize. We implement our algorithm in a tool `IAM-PolicyRefiner` and evaluate it on policies attached to AWS roles with access logs. For each role, `IAM-PolicyRefiner` can compute easy-to-inspect refined policies in less than 1 minute, and the refined policies do not overfit to the requests in the log—i.e., the policies also allow requests in a left-out test set of requests.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Security and privacy** → *Formal methods and theory of security*; • **Theory of computation** → *Semantics and reasoning*.

Additional Key Words and Phrases: Access Control, Authorization, Formal Methods

ACM Reference Format:
Loris D'Antoni, Shuo Ding, Amit Goel, Mathangi Ramesh, Neha Rungta, and Chungha Sung. 2024. Automatically Reducing Privilege for Access Control Policies. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 298 (October 2024), 28 pages. https://doi.org/10.1145/3689738

1 Introduction

Cloud computing provides on-demand access to IT resources via the Internet and access control policies that are the mechanism cloud users use to secure resources in the cloud. Theses policies are programs written in a domain-specific language and can describe who has access to what resources, and under what conditions. It is a known best practice that access policies to cloud

*Work carried out while at Amazon Web Services

Authors' Contact Information: Loris D'Antoni, Amazon Web Services, Seattle, USA, lorisd@amazon.com; Shuo Ding, Georgia Institute of Technology, Atlanta, USA, sding@gatech.edu; Amit Goel, Amazon Web Services, Seattle, USA, amgoel@amazon.com; Mathangi Ramesh, Amazon Web Services, Seattle, USA, mathanr@amazon.com; Neha Rungta, Amazon Web Services, Seattle, USA, rungta@amazon.com; Chungha Sung, Amazon Web Services, Seattle, USA, chunghs@amazon.com.

ACM 2475-1421/2024/10-ART298
https://doi.org/10.1145/3689738

resources should abide to the least-privilege principle [2, 8, 44]—i.e., one should only grant the smallest set of permissions that a given application needs.

While reducing privilege is the ultimate goal, in the early phases of development, writing lax policies can help developers move fast and focus on building the application. For example, users that are new to access control policy languages, often inquire about policies on Stack Overflow [53, 54], or use *managed policies* provided by cloud providers [36].

Even once an application is more mature, users might still write lax policies because they do not know the exact permissions their application needs, or they do not fully grasp the semantics of the policy language. Although users typically revisit their needs and modify their policies to remove unused privileges once applications are deployed, doing so can be time consuming. While there have been approaches that use formal methods to to help user verify access control policies or checking whether a modified policy is less permissive than the original one [3–5, 12], users still have to manually write and modify their policies, a task which requires specific expertise and time.

Refining AWS IAM Policies. This paper focuses on the Amazon Web Services (AWS) IAM policy language and presents a technique to automatically synthesize modifications of user policies that make them closer to least privilege.

Our approach is based on the assumption that a policy should not allow more permissions than those observed when running an application. Given a policy P and a set of concrete requests R granted by the application over a period of time (e.g., requests observed over 90 days on the user's AWS CloudTrail logs [33]), we want to compute the *least-permissive readable policy* P'—i.e., a modification of P such that the following three desiderata hold:

Readability The modified policy P' is syntactically similar to the original policy P. For example, the policy presented in Figure 1b can be modified to the syntactically similar policy shown in Figure 1c. In this paper we formalize the notion of readability by defining a search space of policies that are syntactically similar to the policy P; intuitively all policies in the search space should preserve the structure (i.e., what attributes were used to describe allowed/denied requests) of the original P and only modify attribute values in limited ways. This requirement prevents one from considering policies that are hard for the user to inspect/understand, as well as very large policies that simply enumerate all possible requests, thus overfitting to the observed requests.

Soundness The modified policy P' allows all requests in R and does not allow more requests than the original policy P does. This requirement guarantees that the refined policy is at most as permissive as the original one while still allowing all the observed events.

Tightness The modified policy P' is the least permissive of all sound and readable policies—i.e., P' is a solution to the problem we are trying to solve.

Our first key insight is that for AWS IAM one can formalize the set of readable modifications (for a well-defined natural notion of readability we define) of the original policy P as a lattice ordered by which policies are more permissive. This formalization allows us to use techniques inspired by abstract interpretation [28] and least-general generalizations [19] to compute what is *provably* the tightest policy in the lattice that accepts all requests in R. By carefully designing a lattice of readable modifications, we also obtain that the refined policies, although less permissive than the original ones, will not overfit to the requests in the log and generalize to unseen request (we empirically evaluate this claim).

A Well-Studied Paradigm. The idea of computing least general generalizations to "learn" from a set of examples is quite ubiquitous in computer science. Plotkin [24, 25] was the first to discuss least generalizations in the context of learning propositional logical formulas from examples,

where he introduced an algorithm called anti-unification to solve this problem. Anti-unification has also been used to compute least general tree patterns from tree structures, with applications to learning program transformations [52] and programming by examples [26]. An algorithm for computing least general generalizations over a lattice appears already in Mithell's book [19] on machine learning under the name of `find-s`. The `find-s` algorithm has been rephrased in abstract interpretation terms by Reps and Thakur [28] in their work on generating precise (i.e., least general) abstractions from data. Our presentation follows this last approach using the functions β (that maps one concrete element to an abstract element), and $\sqcup$ (that maps two abstract elements to one abstract element that contains both), as well as the idea of representing sets of possible policies via abstract domains and lattices.

Phrasing our problem into a well-established generalization framework, makes our approach grounded in a solid theory.

Formal Guarantees. The key contribution of our work is that it identifies properties of the AWS IAM language that enable a compositional and modular abstraction approach where the functions needed to compute least general generalizations are designed in terms of simpler building blocks. The formal properties of our approach make it reusable. For example, we propose an extension of our technique that takes advantage of the ability of cloud providers to detect vulnerabilities in existing policies in the form of descriptions of likely undesired requests (e.g., unused-access findings in AWS IAM Access Analyzer [50]). Given a set of undesired requests R that a policy P is allowing, we propose a synthesis approach based on most general specialization that computes the most permissive modification of P that rejects all requests in R.

The techniques presented in this paper take advantage of the structure of the AWS IAM policy language. Because some other policy languages (but not all!), such as Google Cloud IAM Policy [9], Azure IAM Policy [17], and Cedar [7] also use forms of allow/deny statements with a similar semantics to that of AWS IAM, the high-level structure of our technique could be in the future extended to these languages.

Contributions. In this paper, we make the following contributions:

- We formalize a notion of *least-permissive readable* access control policies for AWS cloud resources based on observed requests, together with the corresponding synthesis problem of automatically generating such policies (§ 2).
- We present an algorithm for automatically synthesizing least-permissive readable policies that is based on the concept of least-general generalization. The algorithm uses the monotonicity properties of IAM policies (§ 3) to break the problem of synthesizing least-permissive readable policies into simpler and generalization problems of synthesizing least-permissive predicates for attributes (§ 4). We provide concrete generalization approaches for predicates over all the basic datatypes used in the AWS IAM language policy (§ 5).
- We implement the approach in a tool, `IAM-PolicyRefiner`, and evaluate it on policies and request logs from AWS accounts (§ 7). `IAM-PolicyRefiner` is fast (<5s per policy), effective (modified policies allow up to 99% fewer actions and modify up to 100% resource attributes values), and does not overfit to the given requests (the refined policies allow all requests in a left-out test set of requests).

2 Motivating Example

Let us consider an example where an instructor named Emma plans to use AWS to manage a class where teaching assistants (TAs) need to download and grade student submissions. Also, TAs need to upload their graded versions of the submissions so that students can access them.

To make sure the TAs have all the needed permissions, Emma wants to run a mock version of the class before the semester starts.

An Initial Policy. Emma starts by creating the policy T in Fig. 1b that gives broad permissions needed for the IAM role *TA* involved in the application [41]. Informally, users who are assigned the role *TA* should be able to retrieve the objects of the S3 [32] bucket (the AWS cloud object storage) for accessing submissions or uploading graded submissions. To define the *TA* role, Emma creates four statements that respectively give permissions to list, download, and upload student submissions, and to encrypt/decrypt data to support server-side encryption when storing the data in S3 [43].

The first statement s_1 `allow`s TAs to list objects (i.e., submissions) in the S3 bucket. The statement *allows* any request that performs the `action` named `s3:ListBucket` on any `resource` with name `plclass`, such that the prefix of the requested S3 object path matches the wildcard `*` (i.e., any string). We simplify the resource for readability; AWS uses a more complex resource format called ARN [31]. While Emma eventually wants to restrict permissions for TAs to a specific path, i.e., the directory where submissions are stored, she does not know yet the final name of the directory. Therefore, for now she creates a field `s3prefix` but leaves it unconstrained (i.e., the `*`).

The second and the third statements, s_2 and s_3, respectively handle read and write permissions to the S3 bucket. These statements respectively `allow` requests that perform any `action` that starts with `s3:Get` (e.g., `s3:GetObject`, `s3:GetObjectTagging`), or that starts with `s3:Put` (e.g., `s3:PutObjectTagging`, `s3:PutObjectAcl`). Both statements require the `resource` paths to start with `plclass`. Emma does not know the exact path for the directory the TAs will use and thus leaves a `*` wildcard at the end of the path. Emma uses two separate statements for read and write permissions to distinguish permissions for the student-submission and graded-submission folders.

The fourth statement s_4 handles permissions for the AWS Key Management Service (KMS) [35], which is used for server-side encryption of AWS S3 bucket. The statement `allow`s requests that perform any valid AWS KMS `action` on any `resource` whose name starts with `instance645:`, from a `sourceIP` of the form `10.0.0.0/0` (CIDR [45] of the IP address that matches the first 0 bits of the IP address, meaning all IP addresses). Because Emma has not set up any key before the mock run of her application, she leaves the key name as a wildcard. Similarly, Emma expects TAs to work on VPN that has a specific range of IP addresses, but since she does not know yet what that range will be, she leaves a very permissive IP range to get started.

Collecting Access Logs. While the extra permissions Emma has provided are fine for the mock run, once the application is used during the actual class, Emma wants to tighten the permissions for each TA role to avoid problems such as TAs accidentally overwriting student submissions or checking student submissions assigned to another group the TA is not responsible for. The challenging part is to identify what requests the application *needs* to run. One possible approach would be to set up all detailed paths and conditions for different TAs, but such an approach is tedious, error-prone, and time consuming, especially if Emma is not an experienced AWS user.

Because Emma holds TA practice sessions through a mock run, we can treat the requests *observed* during the mock run are representative of the ones *needed* by the application.[1] For example, if an action that is currently granted by the policy T is never observed in the access log, providing permissions for such an action is probably not necessary. The above argument is the key premise of our work: we can learn what permissions are needed by the application by observing what permissions are used in practice.

[1]We assume that the logs do not contain "malicious" requests that a TA has issued to access more controls than needed. This assumption is reasonable because applications are often run in shadow mode before being openly deployed.

(a) Set of requests R (b) Initial policy T (c) Refined policy T^r

Fig. 1. Given a set of requests R (Figure 1a), and a policy T (Figure 1b), `IAM-PolicyRefiner` computes a policy T^r that is less permissive than T while still granting all requests in R (Figure 1c). Furthermore T^r is syntactically similar to T (the red text is replaced with the green text). Dashed lines separate which requests are granted by which statements.

AWS provides a service called CloudTrail [33] that logs requests received by an application and Figure 1a shows a list of the requests R observed (in a simplified version of the CloudTrail format due to space) for the TA role *TA* over a period of time. Each request contains action, resource, and context information. Figure 1 also aligns each request with the corresponding statement that allows it (denoted by the dashed lines).

Refining T using `IAM-PolicyRefiner`. The goal of `IAM-PolicyRefiner` is to automatically modify the policy T and obtain a refined policy T^r that provides enough permissions to grant all requests in R, but so that the refined policy also generalizes to similar requests outside of R. More formally, the refined policy T^r should match the following goals:

Readability T^r is syntactically similar to T; the difference can be explained to the user (i.e., Emma).
Soundness T^r allows all requests in R and does not allow more requests than T does.
Tightness T^r is the least permissive of all sound and readable policies.

Given the policy T in Figure 1b and the set of requests R shown in Figure 1a, `IAM-PolicyRefiner` computes the refined policy T^r shown in Figure 1c. The policy T^r is syntactically similar to the policy T (the red parts in the original policy T have been changed to the corresponding green parts in the refined policy T^r), but allows fewer requests that are tied to each TA role.

The first refined statement s'_1 allows fewer requests than the original statement s_1 by restricting the `s3:prefix` value to a specific path named `fall/*`. `s3:prefix` limits the accesses on a specific prefix of the resources—i.e., it only allows the access to specific group of students.

The second refined statement s_2' now allows requests that perform the `s3:GetObject` `action` (policy T allowed all possible actions starting with `s3:Get`) on any `resource` whose name starts with the string `plclass/fall/` restricting specific path of the group assigned to the TA (whereas policy T allowed requests on all resources with names starting with `plclass/*`).

The third refined statement s_3' only allows the `s3:PutObject` `action` (i.e., TAs can only upload assignments) to the specific `resource` path `plclass/fall/graded`.

The fourth refined statement s_4' allows requests that perform `kms:Encrypt` or `kms:Decrypt` `action`s (whereas policy T allowed all possible `kms` actions) and *only* on the `resource` with name `instance645:key/5df8`. Furthermore, a request should come from a source `IPAddress` that matches the first 16 bits of the IP address `10.226.0.0` (a range of VPNs), whereas the original statement T allowed requests from all IP addresses. (We simplify the condition key for readability; AWS provides various condition keys as a global context and service context [47].)

The policy T^r is **sound** as it still allows all the requests observed in R and each field imposes more restrictions than T did. Furthermore, T^r is **readable**: it has the same structure as T and it only applies minimal changes to the given predicates that can be easily inspected from the red-to-green diff (e.g., it only changed the mask of the `sourceIP`). We will formalize this notion in Section 4.3.1.

Arguing that T^r is **tight** is a bit trickier as there are still requests that are not observed in R that T^r will allow—e.g., accessing an object at path `plclass/fall/sub/t1/` which might exist in the bucket, but does not appear explicitly in the set of requests.

While it is possible to write a policy P_R that *only* allows the requests in R, the policy P_R would not only be *unreadable*, but it would also *overfit* to the data in R. For example, the policy P_R will fail to allow requests when students submit assignments with different file names than those observed in the requests R. Furthermore, for the policy P_R to only allow events in R, the policy would also need to inspect fields that the original policy T did not contain—e.g., timestamps or destination IPs.

This last example surfaces a key design choice of `IAM-PolicyRefiner`: the refined policy T^r should be the tightest policy—i.e., the one that allows the fewest requests—in the space of "readable" modifications of T. In this paper, we choose to formalize readability as follows: a policy is *readable* if an only if it belongs to a search space $\mathcal{P}_T$ that is carefully designed to only contain policies that are small syntactic modifications of the original policy T (formally defined in Section 4.3). Because the search space $\mathcal{P}_T$ is defined to only contain small variations of the policies, it is trivial to prompt a user of `IAM-PolicyRefiner` with a clear diff between the refined policy and the original policy.

One way to think about our work is through the lens of "inductive bias". Our policy search space $\mathcal{P}_T$ imposes an inductive bias by only allowing readable variations of T. The set $\mathcal{P}_T$ contains an infinite space of possible policies (but not *all* IAM policies). The tightness definition makes sure that we pick the least permissive of all such policies (e.g., we do not generate P itself).

A contribution of this paper is that the set $\mathcal{P}_T$ of readable modifications of T is defined so that it forms a join-closed semi-lattice, a formalization that enables to compute *least-general generalizations* from requests efficiently and compositionally as shown in the rest of the paper.

3 The AWS Policy Language and its Properties

In this section, we first describe the AWS Policy language and its semantics, and then discuss properties of the semantics that will be useful for designing `IAM-PolicyRefiner`.

3.1 Semantics of IAM Policy Language

The AWS policy language is defined as serialized JSON[2], but in this paper, we describe a simplified abstract syntax of the core constructs of the policy language to simplify our exposition. Figure 2

[2] https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements.html

```
Policy → Statement*
Statement → (Effect, Principal, Action, Resource, Condition?)
Effect → allow | deny
Principal → principal: string*
Action → action: string*
Resource → resource: string*
Condition → condition: Operator*
Operator → (OperatorName, ConditionKeyName, Value*)
OperatorName → StringEquals | StringEqualsIfExists | StringLike |
               StringNotEquals | IpAddress | ...
ConditionKeyName → aws:sourceVpc | aws:sourceIp | s3:prefix | ...
Value → String | Num | Bool | IP
```

Fig. 2. Abstract syntax for the AWS IAM policy language (? denotes optional elements and * denotes lists).

shows the abstract syntax for the IAM policy language[3]. IAM policy is a set of statements that can either allow or deny a set of requests. Each statement defines an `Effect` to indicate whether to `allow` or `deny` a request, and a set of predicates over variables, (e.g., `principal`, `action`, `resource`) to describe when a request is matched by the statement . For example, the statement s_1' described in Figure 1c would be captured using the statement in the AWS IAM syntax shown in Figure 3.

To avoid dealing with complex syntax, we present the abstract view of predicates and statement used in Figure 1b. We assume a set of variables V, which represent the possible fields in a request—e.g., `action`, `resource`, and `s3prefix` are variables.

```
{effect   : allow,
 action   : "s3:ListBucket",
 resource : "plclass",
 condition: {
     StringLike: {
         "s3:prefix": "fall/*" }}}
```

Fig. 3. AWS IAM Syntax for statement s_1' in Figure 1c.

Statements. We formalize a statement s as a pair (e, Ψ) where e is either the value `allow` (we call these statements *allow-statements*) or the value `deny` (we call these statements *deny-statements*), and $\Psi : V \mapsto \texttt{pred}$ is a partial function that maps variables to predicates. For example, in the statement $s_2 = (\texttt{allow}, \Psi_2)$ in Figure 1b, Ψ_2 maps the variable `action` to the predicate stating that the action of a request should start with `s3:Get` (i.e., the predicate is represented by the pattern `s3:Get*`). Each variable $v \in V$ is associated with a predicate of a specific type in IAM and we use $\sigma(v)$ to denote it. Common predicate types are simplified regular expressions to restrict values of strings and integer comparisons, but in this section we will not go into these details. We use $V(s)$ to denote the set of variables in the domain of Ψ—i.e., all the values for which the partial function is defined. Every statement always contains the variables `action` and `resource`.

We often simplify the notation for predicates to avoid clutter. For example, to denote the predicate that describes the set of all strings denoted by the pattern `s3:Get*` we simply write the pattern `s3:Get*` itself instead of the more precise notation $\lambda v.v \in \mathcal{L}(\texttt{s3:Get*})$ (where $\mathcal{L}(\texttt{pat})$ denotes the set of strings accepted by a pattern `pat`). Similarly, we write IP masks to denote predicates denoting certain sets of IP addresses and drop lambda terms when the are clear from the context. To relate our formalization to Figure 2, all syntactic terms used to denote predicates in the IAM

[3] `principal` is optional for some policy types such as identity-based policies [34] and `condition` is optional for all policy types

syntax (whether via conditions or other forms) are represented as predicates. The translation from syntax to predicates is a simple syntax-directed transformation.

Requests. A request $r : V \to \texttt{val}$ maps a variable to its value. For example the first request in Figure 1a maps the variable `action` to the value `s3:ListBucket`, the variable `resource` to the value `plclass`, and variable `s3prefix` to the value `fall/sub/h1`. Each variable $v \in V$ is associated with a value of a specific type in IAM; we use $\tau(v)$ to denote the type of the values associated with v. Common value types are integers, floats, IP addresses, Booleans, and strings. Every request contains the variables `principal`, `action`, and `resource`, whereas others are optional. We omit `principal` and include only subset of variables in each request of Figure 1a to save the space. For example, variable `s3prefix` only appears in some of the request elements. In practice, requests might not contain values for all the variables, so we allow r to map variables to the special value `null`—e.g., in Figure 1a we assume the variable `resourcetag` does not appear in any request, and we would have $r_1(\texttt{resourcetag}) = \texttt{null}$. If a variable can be assigned a `null` value, its type can be thought of being a "nullable" value—i.e., its type contains null as an element.

Intuitively, a request matches a statement if, for every variable appearing in the statement, the request's values are models of the corresponding predicates in the statement.

Definition 3.1 (Statement-Matching Requests). Given a request r and a statement $s = (e, \Psi)$ we say that s *matches the request* r if and only if, for every $v \in V(s)$, the value $r(v)$ is a model of the predicate $\Psi[v]$. We write $M(s)$ to denote the set of all requests matched by s—i.e, $M(s) = \{r \in R \mid \bigwedge_{v \in V(s)} \Psi[v](r(v))\}$.

For example, the statement $s_4 = (\texttt{allow}, \Psi_4)$ in Figure 1b has three keys `action`, `resource`, and `sourceIP` and matches last three requests in Figure 1a. In particular, the value of each variable in the last request r_{10} of Figure 1a is a model of the corresponding predicate in the statement (missing keys in the statement are ignored), e.g., $r_{10}(\texttt{action}) = \texttt{kms:Decrypt}$ is a model of the predicate $\Psi_4[\texttt{action}] = \texttt{kms:*}$, and $r_{10}(\texttt{sourceIP}) = \texttt{10.226.104.212}$ is a model of the predicate $\Psi_4[\texttt{sourceIP}] = \texttt{10.0.0.0/0}$.

Note that in our formalization we simplify the syntax of IAM policies and simply assume that each variable is associated with its predicate. Our implementation maps the JSON representation of statements to this predicate format; this translation is straightforward and syntax-directed and we do not present it formally, though we provide some examples in Section 5.

Policies. A policy $P = \{s_1, s_2, ..., s_n\}$ is a set of statements and we use $AS(P)$ (resp. $DS(P)$) to denote all the allow (resp. deny) statements in P. We write $P = (AS(P), DS(P))$ to directly denote these two sets and a and d instead of s to denote an allow or deny statement respectively.

Definition 3.2 (Granted Requests). A policy P *grants a request* r, written as "$[\![P]\!](r) = \texttt{true}$", if and only if there exists an allow statement that matches the request, i.e., $\exists s_A \in AS(P).\ r \in M(s_A)$, and there does not exist a deny statement that matches the request, i.e., $\forall s_D \in DS(P).\ r \notin M(s_D)$. We write $Granted(P)$ to denote the set of all requests *granted* by the policy P, which can be defined as follows.

$$Granted(P) = \Big(\bigcup_{s_A \in AS(P)} M(s_A)\Big) \setminus \Big(\bigcup_{s_D \in DS(P)} M(s_D)\Big)$$

Therefore, we formally write $[\![P]\!](r) = r \in Granted(P)$.

3.2 Monotonicity of AWS Policies

Our policy-refinement algorithm (Sec. 4.2) uses two important properties of the AWS IAM language.

First, the set or requests $M(s)$ matched by statement s grows monotonically with respect to the set of values matched by the predicates appearing in s.

Theorem 3.3 (Monotonicity w.r.t. Predicates). *Given two statements $s_1 = (e, \Psi_1)$ and $s_2 = (e, \Psi_2)$, if $V(s_1) = V(s_2)$ and for every $v \in V(s_1)$ we have that $\Psi_1[v] \rightarrow \Psi_2[v]$, then $M(s_1) \subseteq M(s_2)$.*

Second, the set of requests allowed by a policy are monotonically increasing (resp. decreasing) with respect to the set of requests the allow statements (resp. deny statements) match.

Theorem 3.4 (Monotonicity w.r.t. Statements). *Let P_1 be a policy such that $AS(P_1) = \{a_1, \ldots, a_n\}$ and $DS(P_1) = \{d_1, \ldots, d_m\}$. For every policy P_2 the following is true:*

- *If $AS(P_2) = \{a'_1, \ldots, a'_n\}$ such that for every $1 \leq i \leq n$ we have $M(a_i) \subseteq M(a'_i)$ and $DS(P_1) = DS(P_2)$, then $Granted(P_1) \subseteq Granted(P_2)$.*
- *If $AS(P_1) = AS(P_2)$ and $DS(P_2) = \{d'_1, \ldots, d'_n\}$ such that for every $1 \leq i \leq m$ we have $M(d_i) \subseteq M(d'_i)$, then $Granted(P_1) \supseteq Granted(P_2)$.*

For what concerns our refinement problem, Theorem 3.4 guarantees that if we compute a policy in which all allow statements match a subset of the requests matched by the corresponding allow statements in the original policy, and all deny statements statements match a superset of the requests matched by the corresponding deny statements in the original policy, our policy is guaranteed to be sound and allow no more requests than the original policy. Theorem 3.3 tells us how we can refine individual statements to guarantee that they match fewer (or more) requests—i.e., by modifying individual predicates in isolation so that they imply the original predicates. The next section uses these two properties to build the refinement algorithm used by `IAM-PolicyRefiner`.

Beyond AWS IAM. The techniques presented in this paper take advantage of the monotonicity properties of the AWS IAM policy language. Because other policy languages, such as Google Cloud IAM Policy [9], Azure IAM Policy [17], and Cedar [7] also use forms of allow/deny statements with a similar semantics to that of AWS IAM, these languages enjoy similar monotonicity properties—i.e., making individual allow (resp. deny) statements match fewer (resp. more) requests results in the overall policy granting fewer requests. Therefore, the high-level structure of the modular algorithm we present in Section 4 can be adapted to these other authorization languages. However, the design of the policy search space (Definition 4.9) will have to be modified to fit the structure of the statements used in other languages, and new predicate abstractions will need to be designed to fit the predicates of other languages (see Section 5.7).

Some authorization languages, e.g., Rego [27] (which targets structured documents and not IAM requests), do not follow the same paradigm as AWS IAM and are instead based on ideas from logic programming. These languages do not fit our algorithm (at least not directly).

4 Refining AWS Policies from Data Using Least General Generalization

In this section, we formalize the problem of finding a least general policy that is consistent with a given set of events. We draw a connection to the principle of least general generalization [24, 25] and use the properties of AWS IAM stated in Theorems 3.3 and 3.4 to show how least general generalizations can be computed in a modular way for AWS IAM policies.

4.1 The Policy Refinement Problem and its Connection to Least General Generalization

Given a policy P and set of requests R that are granted by P, our goal is to identify a new policy P' that grants all requests in R, but no more requests than those granted by P. As discussed in Section 2, this definition admits two trivial solutions: the most general solution is the policy P itself (which does not inch toward least privilege), and the least general solution is the policy P_R that grants exactly all requests in R (thus overfitting). Neither of these policies is a desirable solution.

The problem at hand requires to find ways to "slightly" generalize from the given examples (and not overfit), but also to restrict access to follow the least-privilege principle. To achieve this goal,

we can impose a bias over what set of policies we are interested in (e.g., we want to disallow the policy P_R). We can thus introduce the notion of a policy search space $\mathcal{P}_P$, which is a set of policies we are interested in, and our goal is to then find the most restrictive policy in $\mathcal{P}_P$.

Definition 4.1 (Policy Refinement). Given a policy P, a set of requests R, and a policy search space $\mathcal{P}_P$, we say that $P^* \in \mathcal{P}_P$ is ***a** least-privilege policy consistent with the requests R* if:

Soundness $R \subseteq Granted(P^*) \subseteq Granted(P)$.
Tightness there does not exists a policy $P' \in \mathcal{P}_P$ such that $R \subseteq Granted(P') \subset Granted(P^*)$.

The *policy refinement problem* is to find *a least-privilege policy* consistent with the requests R.

In Section 2, we had presented a third requirement that the above definition does not capture:

Readability P^* is syntactically similar to P and the difference can be explained to the user.

To capture this requirement we can carefully define the set $\mathcal{P}_P$ to only contain policies that are syntactically similar to P. We formally define the search spaces used in our implementation and their properties in Section 4.3, but here we provide some examples of what they look like.

Example 4.2 (Policy-Refinement). Consider the policy T in Figure 1b and the requests R in Figure 1a. We can define the set $\mathcal{P}_T$ so that it only contain variations of T that have the same set of statements and with the same variables—e.g., the refined policy T^r in Figure 1c satisfies this requirement. Furthermore, the policies in $\mathcal{P}_T$ use predicates that are syntactically similar to the ones in the policy T—e.g., the red-to-green changes in Figure 1.

4.2 Computing Unique Refined Policies

We next present the algorithm `IAM-PolicyRefiner` uses to solve the policy-refinement problem. Because the search space $\mathcal{P}_P$ might admit multiple incomparable solutions, Definition 4.1 requires P^* to be **a** least-privilege policy consistent with the requests instead of **the** least-privilege policy.

Example 4.3 (Ambiguous Search Spaces). Consider a policy P_U that allows all requests where the value of the variable `resourcetag` is unconstrained by `*`. Assume we are refining the policy with respect to two requests in R_U where the respective values for the variable `resourcetag` are `aaa` and `aa`. Now assume that our search space $\mathcal{P}_{P_U}$ allows predicates where at most one wildcard `*` can appear in each string predicate. Given these requests, we can compute at least two possible refined policies that are both sound and tight with respect to such a policy search space: one constraining the value of `resourcetag` to `*aa` (i.e., all values that end in `aa`) and one to `aa*` (i.e., all values that start with `aa`). However, neither of these two policies is tigheter than the other.

Example 4.3 showed that the structure of the search space influences whether one can compute a unique refined policy. To address this problem, we provide sufficient conditions that restrict the search space of possible policies and guarantee that the problem admits a unique solution. The following definitions use concepts from abstract interpretation to capture when a set of elements can be mapped to a *unique representation* of the set of those elements.

Definition 4.4 (Concrete and Abstract Domain). A *concrete domain* C is a set of values, in our setting, a set of requests. An *abstract $\sqcup$-closed (join-closed) semilattice* is a partially ordered set $(A, \sqsubseteq)$ where: *(i)* A is the abstract domain: in our case, policies matching sets of requests; *(ii)* $\sqsubseteq$ is a partial order on elements in A: in our case, whether a policy allows a subset of the requests allowed by another policy; and *(iii)* for any two elements $x, y \in A$, the join (or supremum) of the two elements $m = x \sqcup y$ is **the unique** element of A such that $x \sqsubseteq m$, $y \sqsubseteq m$, and for every $w \in A$, if $x \sqsubseteq w$ and $y \sqsubseteq w$, then $m \sqsubseteq w$.

To relate C to A, we use a concretization function $\gamma : A \mapsto 2^C$ maps an abstract element to the set of concrete values that it represents, and an abstraction function $\beta : C \mapsto A$ such that $\beta(c)$

maps a concrete element c to **the** least element in A with respect to $\sqsubseteq$—i.e., for every $c \in C$, the function $\beta(c)$ guarantees that there is no element $a \in A$ such that $a \sqsubset \beta(c)$ and $c \in \gamma(a)$. It is also convenient for A to contain an element $\bot$ that is smaller than every other element in A with respect to $\sqsubseteq$.

Because the join $\sqcup$ computes the least element, the operation is by definition associative and commutative. In our setting, A will represent a set of policies and our goal is to find the least policy in A that is consistent with a set of concrete requests (i.e., the set C).

Definition 4.5 (Well-behaved Abstraction). We say that a tuple $\mathcal{A} = (C, A, \sqsubseteq, \beta, \gamma, \sqcup, \bot)$ is a *well-behaved* abstraction if all the components respect the above definitions.

Example 4.6 (Well-behaved Abstraction for Integers). We illustrate the above definitions with well-behaved abstraction $\mathcal{A}_{\leq 5} = (\{0, 1, 2, 3, 4, 5\}, \{false, v \leq 0, v \leq 1, v \leq 2, v \leq 3, v \leq 4, v \leq 5\}, \rightarrow, \beta_{\leq 5}, \gamma_{\leq 5}, \sqcup_{\leq 5}, false)$ that maps numbers in the set $\{0, 1, 2, 3, 4, 5\}$ to predicates of the form $v \leq c$. Intuitively, given a set of numbers $\{4, 2, 1\}$, we want their abstraction to yield the predicate $v \leq 4$—i.e., the least predicate that is consistent with those numbers. Hence, we can define $\beta_{\leq 5}(c) = v \leq c$ and $\sqcup_{\leq 5}(v \leq a, v \leq b) = v \leq max(a, b)$. The order over abstract elements is logical implication $\rightarrow$, and $\gamma_{\leq 5}$ simply returns all the numbers that satisfy the inequality.

Theorem 4.7 (Uniqueness of Least Element). *Given a well-behaved abstraction $\mathcal{A} = (C, A, \sqsubseteq, \beta, \gamma, \sqcup, \bot)$, for every set of concrete elements $C' \subseteq C$, there exists one unique least element a^* in A such that (i) $C' \subseteq \gamma(a^*)$, and (ii) for every $a \in A$, if $S \subseteq \gamma(a)$ then $a^* \subseteq a$. Furthermore, a^* can be defined (and computed) as $a^* = \bigsqcup_{c \in C'} \beta(c)$.*

Theorem 4.7 gives us a perfect recipe for solving the policy refinement problem. Given a policy P, if we can define a well-behaved abstraction $\mathcal{A}_P = (Granted(P), \mathcal{P}_P, \sqsubseteq, \beta_P, \gamma_P, \sqcup_P, \bot_P)$ from the concrete domain $Granted(P)$ to an abstract domain of IAM policies (i.e., a policy search space) that are less permissive than P, we can compute **the unique** least permissive policy P^* for a set of requests R as $\bigsqcup_{c \in R} \beta(c)$. Furthermore, because the join operator is associative and commutative, we can efficiently compute P^*_r in a map-reduce style as $\texttt{refine}(P, R) = \texttt{reduce}(\sqcup, \texttt{map}(\beta, R), \bot)$.

4.3 From Computing Refined Policies to Computing Refined Predicates

In the previous section, we presented an approach for computing refined policies from a set of requests. However, we have not provided an actual algorithm as we have not defined a policy search space $\mathcal{P}_P$ and the corresponding well-behaved abstraction operations β_P, γ_P, and $\sqcup_P$.

Doing so for AWS IAM policies is challenging. First, there are multiple ways to modify the multiple statements appearing in a policy to match the provided requests. Second, individual statements can contain predicates of different kinds—e.g., numerical constraints, regular expressions—and with multiple restrictions—e.g., regular expressions can only contain wildcards and no Kleene stars.

In this section, we define a parameterized search space for policies and show how we can reduce the problem of computing refined policies for such a search space to a simpler problem of computing refined statements first and then to computing refined predicates appearing in the statements.

Most important, our definitions of policy search space (Definition 4.9), statement search space (Definition 4.8), and predicate abstractions (Section 5) formalize what we consider to be the set of *readable* modifications of a given policy—i.e., the modified policies must preserve the set of statements and their variables, and can only "slightly" modify the predicates appearing in each statement.

4.3.1 The Modular Policy Search Space $\mathcal{P}_P$. We define the search spaces we will consider in this paper when refining policies—i.e., what is the set $\mathcal{P}_P$. In section 5, we will define specific ways in

which each predicate in a statement can be modified. For now, we assume that for every predicate $\Psi[v]$ corresponding to a variable v, there exists a *predicate search space* $\Phi_{\Psi[v]}$, i.e., a set of predicates each of which describes a subset of the models of $\Psi[v]$ and are syntactically similar to $\Psi[v]$. Furthermore, the set $\Phi_{\Psi[v]}$ contains a bottom element $\bot_{\Psi[v]}$ that has no satisfying values/models.

We first define statement search spaces and then policy search spaces.

Definition 4.8 (Statement Search Space). Given a statement $s = (e, \Psi_s)$, the statement search space $\mathcal{S}_s$ contains statements that maintain the sets of variables and predicate structures of s. Formally, $(e, \Psi'_s) \in \mathcal{S}_{(e,\Psi_s)}$ iff $V(\Psi_s) = V(\Psi'_s)$ and $\forall v \in V(\Psi_s),\ \Psi'_s[v] \in \Phi_{\Psi_s[v]}$ (the predicate search space).

All statements in $\mathcal{S}_s$ grant subsets of the requests granted by s

Note that, by definition, all requests we use to refine a policy are already allowed by some `allow` statement in the policy and are not denied by any `deny` statement. In general, there are two ways to make a policy less permissive (Theorem 3.4): 1) we can make the `allow` statement that allowed the requests under consideration less permissive, or 2) we can make a `deny` statement more restrictive. Because `allow` statements are necessary to allow any permission and `deny` statements are not always used in all policies and cannot directly be associated with allowed requests, it is more natural to focus on search spaces that make `allow` statements less permissive.

Definition 4.9 (Policy Search Space). Given a policy P such that $AS(P) = \{a_1, \ldots, a_n\}$ and $DS(P) = \{d_1, \ldots, d_m\}$, the *policy search space* $\mathcal{P}_P$ is comprised of policies that can modify only allow statements. Formally, $\{a'_1, \ldots, a'_n, d_1, \ldots, d_m\} \in \mathcal{P}_P$ iff $\forall 1 \leq i \leq n.\ a'_i \in \mathcal{S}_{a_i}$ (see Def. 4.8).

Whenever we write $\mathcal{S}_{(e,\Psi)}$ and $\mathcal{P}_P$ we assume they are defined as in Definitions 4.8 and 4.9.

Example 4.10 (Policy Search Space). Consider the input policy T in Fig. 1b. The output policy T^r in Fig. 1c is an example of what a policy in $\mathcal{P}_T$ looks like. In particular, each statement s'_i in Fig. 1c belongs to the statement search space $\mathcal{S}_{s_i}$ of the corresponding statement s_i in Fig. 1b. Note that even if we had a `deny` statement in the input policy, $\mathcal{P}_T$ would only contain policies that modify the `allow` statements and leave the `deny` statement unaffected.

All policies in $\mathcal{P}_P$ grant subsets of the requests granted by P

4.3.2 Modular Policy Refinement for $\mathcal{P}_P$. The design of the policy search space is such that, as long as the elements of each predicate search space $\Phi_{\Psi[v]}$ are similar to the predicate $\Psi[v]$, then every policy in $\mathcal{P}_P$ is similar to P and therefore satisfies our readability requirement—i.e., the policies in $\mathcal{P}_P$ are modifications of the allow statements in P and preserve their variables.

Moreover, our definition has another nice property that is formalized in the following theorems.

We use $pred(s)$ (resp. $pred(P)$) to denote all predicates appearing in the statement s (resp. policy P), and given a predicate ψ, we use $val(\psi)$ to denote all models of the predicate ψ. Definition 4.11 states that as long as we have abstractions for the predicates appearing in the a statement—i.e., we have a way to turn concrete values into least general predicates—we can use these abstractions to build abstractions for such a statement. The idea is that one can map the values appearing in a request matched by a statement to the predicates of the corresponding variables and use the well-behaved abstractions of each predicate to compute least predicates.

Definition 4.11 (Statement Abstractions). Let $s = (e, \Psi)$ be a statement, and for every predicate $\psi \in pred(s)$ assume that we have an abstraction $\mathcal{A}_\psi = (val(\psi), \Phi_\psi, \rightarrow, \beta_\psi, \gamma_\psi, \sqcup_\psi)$. The statement abstraction $\mathcal{A}_s = (M(s), \mathcal{S}_s, \sqsubseteq_s, \beta_s, \gamma_s, \sqcup_s, \bot_s)$ is defined as follows:

- If $e =$ `allow` then $s' \sqsubseteq_s s''$ iff $M(s') \subseteq M(s'')$. If $e =$ `deny` then $s' \sqsubseteq_s s''$ iff $M(s'') \subseteq M(s')$.
- $\beta_s(r) = (e, \Psi')$ such that $V(\Psi) = V(\Psi')$ and $\forall v \in V(\Psi),\ \Psi'[v] = \beta_{\Psi[v]}(r(v))$.
- $\gamma_s(e, \Psi') = M(\Psi')$.

- $(e_1, \Psi_1) \sqcup_{(e,\Psi)} (e_2, \Psi_2) = (e, \Psi')$ if for every variable $v \in V(\Psi)$ we have $\Psi'[v] = \Psi_1[v] \sqcup_{\Psi[v]} \Psi_2[v]$. (Note that due to Definition 4.8 we have $V(\Psi) = V(\Psi_1) = V(\Psi_2) = V(\Psi')$.)
- $\bot_s = (e, \Psi_\bot)$ where for every variable $v \in V(\Psi)$ we have $\Psi_\bot[v] = \bot_{\Psi[v]}$.

Example 4.12. Consider the first two requests r_1 and r_2 in Fig 1a, and the statement s_1 in Fig 1b. We can define $\beta_{s_1}(r_1) = (\texttt{allow}, \Psi_1)$ as follows (we will clarify in Section 4.3 how the individual predicates are computed): $\Psi_1[\texttt{action}] = \lambda v.v = \texttt{s3:ListBucket}$, $\Psi_1[\texttt{resource}] = \lambda v.v = \texttt{plclass}$, $\Psi_1[\texttt{s3prefix}] = \lambda v.v \in L(\texttt{fall/sub/h1})$—i.e., a regular expression that only accepts one string. Similarly, we can define $\beta_{s_1}(r_2) = (\texttt{allow}, \Psi_2)$ as follows: $\Psi_2[\texttt{action}] = \lambda v.v = \texttt{s3:ListBucket}$, $\Psi_2[\texttt{resource}] = \lambda v.v = \texttt{plclass}$, $\Psi_2[\texttt{s3prefix}] = \lambda v.v \in L(\texttt{fall/grade/t2})$. When we apply the join operator $\sqcup_{s_1}$ to the two statements $\beta_{s_1}(r_1)$ and $\beta_{s_1}(r_2)$ we obtain a statement $(\texttt{allow}, \Psi_{1\sqcup 2})$ with the following predicates: *(i)* $\Psi_{1\sqcup 2}[\texttt{action}] = \lambda v.v = \texttt{s3:ListBucket}$, *(ii)* $\Psi_{1\sqcup 2}[\texttt{resource}] = \lambda v.v = \texttt{plclass}$, and *(iii)* $\Psi_{1\sqcup 2}[\texttt{s3prefix}] = \lambda v.v \in L(\texttt{fall/*})$.

We will show a well-behaved abstraction for string predicates where $\Psi_{1\sqcup 2}[\texttt{s3prefix}]$ is the the least element obtained by joining $\Psi_1[\texttt{s3prefix}]$ and $\Psi_2[\texttt{s3prefix}]$ in Section 5.3.3.

A statement abstraction is well-behaved when its predicate abstractions are well-behaved.

Theorem 4.13 (Statement Abstractions are Well-behaved). *Given a statement $s = (e, \Psi)$, if for every predicate $\psi \in pred(s)$ the abstraction $\mathcal{A}_\psi$ is well-behaved, then the abstraction $\mathcal{A}_s$ is well-behaved and $\mathcal{S}_s$ only contains statements with the same effect e.*

Theorem 4.13 guarantees that if s is an allow-statement, the least general statement in $\mathcal{S}_s$ consistent with a set of requests R is the one that allows as few requests as possible while allowing all requests in R. Interestingly, if s is a deny-statement the least general statement in $\mathcal{S}_s$ consistent with a set of requests R is the one that allows as many requests as possible while denying all requests in R. This last property will be useful in a different problem setting we consider in Section 6.1.

Definition 4.14 shows how to build policy abstractions by mapping requests to the corresponding statements and then using statement abstractions.

Definition 4.14 (Policy Abstractions). Let P be a policy with allows statements $AS(P) = \{a_1, \ldots, a_n\}$ (Φ_ψ is a set of predicates and $\rightarrow$ denotes logical implication) and deny statements $DS(P) = \{d_1, \ldots, d_m\}$. Assume that we have an abstraction $\mathcal{A}_\psi = (val(\psi), \Phi_\psi, \rightarrow, \beta_\psi, \gamma_\psi, \sqcup_\psi)$ for every predicate $\psi \in \bigcup_i pred(a_i)$. The policy abstraction $\mathcal{A}_P = (Granted(P), \mathcal{P}_P, \sqsubseteq, \beta_P, \gamma_P, \sqcup_P, \bot_P)$ is defined as follows:

- $P' \sqsubseteq_P P''$ iff $Granted(a') \subseteq Granted(a'')$.
- $\beta_P(r) = \{a'_1, \ldots, a'_n, d_1, \ldots, d_m\}$ such that for every $1 \le i \le n$,
  - $a'_i = \bot_{a_i}$ if $r \in \cup_{j<i} M(a_j)$ or $r \notin M(a_i)$, and
  - $a'_i = \beta_{a_i}(r)$, otherwise (i.e., apply the β-function of the first statement a_i matching r).
- $\gamma_P(P_1) = Granted(P_1)$.
- $\{a^1_1, \ldots, a^1_n, d_1, \ldots, d_m\} \sqcup_P \{a^2_1, \ldots, a^2_n, d_1, \ldots, d_m\} = \{a^1_1 \sqcup_{a_1} a^2_1, \ldots, a^1_n \sqcup_{a_n} a^2_n, d_1, \ldots, d_m\}$.
- $\bot_P = \{\bot_{a_1}, \ldots, \bot_{a_n}, d_1, \ldots, d_m\}$.

The policy abstraction presented in Definition 4.14 is not, in general, well-behaved.

Example 4.15 (Overlapping Statements). Consider a policy with two statements that allow every possible action, but restrict the set of possible `resource` to ones that match the predicates `a*` (starts with an `a`) and `*b` (ends with an `b`), respectively. Assume a request where the value of the variable `resource` is `ab`, which is matched by both the first and second statements. The abstraction presented in Definition 4.14 will refine the first statement by, let's say, changing the regular expression `a*` to `ab*`, and replace the predicate in the second statement with $\bot$ (i.e., the statement will not match

any request). The resulting policy is *not the* least permissive one, but instead *a* least permissive one as one could have instead replaced `*b` in the second statement with `*ab` (and made the first statement $\bot$). This problem is well-known in abstract interpretation over disjunctive domains [23], where one cannot have a least element due to the ambiguity of disjunction.

If the two statements had matched disjoint sets of requests (e.g., they had predicates `a*` and `b*`), this problem would not appear and the abstraction given in Definition 4.14 would be well-behaved.

If all `allow` statements match disjoint set of requests and all the predicate abstractions are well-behaved, the resulting policy abstraction is also well-behaved.

Theorem 4.16 (Policy Abstractions are Well-behaved). *Let P be a policy with a set of allows statements $AS(P) = \{a_1, \ldots, a_n\}$. If all allow statements match disjoint sets of requests—i.e., for every $1 \leq i < j \leq n$ we have $M(a_i) \cap M(a_j) = \emptyset$ and for every predicate $\psi \in \bigcup_i pred(a_i)$ the abstraction $\mathcal{A}_\psi$ is well-behaved, then the abstraction $\mathcal{A}_P$ is well-behaved.*

It is decidable to check if two statements match disjoint sets of requests (this check can be done using the tool Zelkova [4]). It is important to note that even if statements match overlapping sets of requests, the abstraction given Definition 4.14 is not well-behaved, but it will still produce policies that are *sound* and *readable*, though they might not be guaranteed to be the tightest as the join operator is not guaranteed to return a least element.

Corollary 4.17 (Uniqueness of Least Permissive Policy). *Let P be a policy with allow statements that match disjoint sets of requests. Let $\mathcal{A}_P = (Granted(P), \mathcal{P}_P, \sqsubseteq, \beta_P, \gamma_P, \sqcup_P, \bot_P)$ be the corresponding well-behaved abstraction. Given a set of requests $R \subseteq Granted(P)$, there exists exactly one least-permissive policy P^* in $\mathcal{P}_P$ consistent with R, which can be computed as $P^* = \bigsqcup_{r \in R} \beta(r)$.*

The above theorems give us a way to make the policy refinement problem modular and easy to customize. Instead of designing complex abstractions for arbitrary policies, we can focus on defining abstractions for individual predicate types, which will be the main goal of Section 5.

5 Policy Abstractions for Least General Generalization

The previous section showed how we can reduce the problem of refining policies to the problem of refining predicates appearing within statements. Theorems 4.13 and 4.16 show that we simply need to create a well-behaved abstraction $\mathcal{A}_\psi = (val(\psi), \Phi_\psi, \rightarrow, \beta_\psi, \gamma_\psi, \sqcup_\psi)$ for every predicate $\psi \in pred(P)$. In this section, we show what types of predicates we need to model the syntactic features supported by AWS IAM and define abstractions for each type of predicate.

The specific abstractions we present are one possible choice of well-behaved abstractions and are the ones used in our evaluation. However, our implementation allows one to swap specific abstractions for different ones (some of which we discuss in this section).

Notation. Whenever we write a predicate with a free variable v, we assume that v is the input to the predicate—e.g., instead of writing $\Psi[\texttt{action}] = (\lambda v.v = \texttt{s3:ListBucket})$, we simply write $v = \texttt{s3:ListBucket}$. We use c to denote constants—e.g., a specific action name or an integer. For every well behaved abstraction $\mathcal{A}_t$, we include a bottom element $\bot_t$ that describes an empty set of values—i.e., $\gamma_t(\bot_t) = \emptyset$—and therefore often omit the description of γ_t for bottom values.

5.1 Equal and Not-Equal

Equal. An equality predicate ψ is of the form $v = c$, where c is a constant. For example, this type of predicate can be used to specify that a certain action is allowed—e.g., `action = s3:ListBucket`—on a specific resource—e.g., `resource = plclass`. Because equality predicates only allow a single value c, our abstraction will only contain the equality predicate $v = c$ itself and the bottom predicate

$\bot_{=c}$ that is not satisfied by any values/model. Intuitively, the abstraction stays at $\bot_{=c}$ until a value c is observed, in which case it generalizes to $v = c$.

Definition 5.1 (Equal Abstraction). For a predicate $v = c$, we define the well-behaved abstraction $\mathcal{A}_{=c} = (\{c\}, \Phi_{=c}, \rightarrow, \beta_{=c}, \gamma_{=c}, \sqcup_{=c}, \bot_{=c})$ as follows:

- $\Phi_{=c} = \{v = c, \bot_{=c}\}$.
- $\beta_{=c}(a) = (v = a)$.
- $\gamma_{=c}(v = c) = \{c\}$.
- $\psi_1 \sqcup_{=c} \psi_2$ is ψ_1 if $\psi_2 \rightarrow \psi_1$, and ψ_2 otherwise.

Theorem 5.2 (Equality Abstractions are Well-behaved). *Given a predicate $v = c$, the abstraction $\mathcal{A}_{=c}$ is well-behaved.*

Proof. The abstract domain only contains the elements $\Phi_{=c} = \{v = c, \bot_{=c}\}$, where $\bot_{=c} \rightarrow v = c$. The abstraction is well-behaved since $\beta_{=c}$ and $\sqcup_{=c}$ return the least elements in all possible cases. In particular, if $\psi_2 \rightarrow \psi_1$, then $\psi_1 \sqcup_{=c} \psi_2 = \psi_1$, otherwise, it means that $\psi_1 \rightarrow \psi_2$ (due to the limited structure of the lattice), and thus $\psi_1 \sqcup_{=c} \psi_2 = \psi_2$. Commutativity follows from the fact that the least element is correctly computed.

□

Not-Equal. A not-equal predicate ψ is of the form $v \neq c$, where c is a constant. For example, this type of predicate can be used to specify that a certain resource should be accessible to everyone but one specific user—e.g., `aws:username` ≠ `darth` [48].

A predicate $v \neq c$ is very permissive as it matches any value different than c. We define the abstraction $\mathcal{A}_{\neq c}$ so that a predicate stays at $\bot_{\neq c}$ until a value d is observed, in which case it generalizes to $v = d$ (this predicate captures the case in which the application only observes one possible value d and not multiple values that are different than c). If two different values are observed, let's say d_1 and d_2, the abstraction generalizes to $v \neq c$.

Definition 5.3 (Not-Equal Abstraction). For a predicate $v = c$, we define the well-behaved abstraction $\mathcal{A}_{\neq c} = (val(\neq c), \Phi_{\neq c}, \rightarrow, \beta_{\neq c}, \gamma_{\neq c}, \sqcup_{\neq c}, \bot_{\neq c})$ as follows:

- $val(\neq c) = \{d \mid d \neq c \wedge \tau(c) = \tau(d)\}$—i.e., values of type $\tau(c)$ that are different from c.
- $\Phi_{\neq c} = \{v = d \mid d \neq c\} \cup \{v \neq c\} \cup \{\bot_{\neq c}\}$.
- $\beta_{\neq c}(d) = v = d$.
- $\gamma_{\neq c}(v = d) = \{d\}$ and $\gamma_{\neq c}(v \neq c) = \{a \mid a \neq c \wedge d \in val(\neq c)\}$.
- We define the join as follows:

$$\psi_1 \sqcup_{\neq c} \psi_2 = \begin{cases} \psi_1 & \psi_2 \rightarrow \psi_1 \\ \psi_2 & \psi_1 \rightarrow \psi_2 \\ v \neq c & \text{otherwise} \end{cases}$$

Theorem 5.4 (Not-equal Abstractions are Well-behaved). *Given a predicate $v \neq c$, the abstraction $\mathcal{A}_{\neq c}$ is well-behaved.*

Another option for this abstraction is to allow a finite disjunction of equalities up to a bound k and only generalize to $v \neq c$ when the bound k is exceeded. For example, we could have that if three values d_1, d_2, d_3 are observed, then the resulting predicate is $v \in \{d_1, d_2, d_3\}$; but once a fourth value d_4 is observed the predicate is generalized to $v \neq c$.

Our implementation also provides abstractions for case-insensitive equal and not-equal operators (e.g., to represent `StringEqualIgnoreCase`) that can be defined analogously to the ones we just defined by just ignoring the cases of the inputs.

5.2 Order Comparisons

Certain types such as numbers and dates admit order comparisons through operations such as $\leq$, $\geq$, $<$, and $>$. We present the construction for the case in which the predicate ψ is of the form $v \leq c$, where c is a constant (the other cases are similar). For example, this type of predicate can be used to specify that some attribute should be bounded—e.g., `s3:max-keys` $\leq$ `5000` [38].

Intuitively, the abstraction stays at $\bot_{\neq c}$ until a value d is observed, in which case it generalizes to $v \leq d$. If different values are observed, let's say $d_1, \ldots, d_n$, we then generalize to $v \leq max(d_1, \ldots, d_n)$.

Definition 5.5 (Comparison Abstraction). For a predicate $v \leq c$, we define the well-behaved abstraction $\mathcal{A}_{\leq c} = (val(\leq c), \Phi_{\leq c}, \rightarrow, \beta_{\leq c}, \gamma_{\leq c}, \sqcup_{\leq c}, \bot_{\leq c})$ as follows:

- $val(\leq c) = \{d \mid d \leq c \wedge \tau(c) = \tau(d)\}$—i.e., values of type $\tau(c)$ that are smaller than c.
- $\Phi_{\leq c} = \{v \leq d \mid d \leq c\} \cup \{\bot_{\leq c}\}$.
- $\beta_{\leq c}(d) = v \leq d$.
- $\gamma_{\leq c}(v \leq d) = \{d' \mid d' \leq d\}$.
- Because our abstract domain is a total order, $\psi_1 \sqcup_{\leq c} \psi_2$ is ψ_1 if $\psi_2 \rightarrow \psi_1$, and ψ_2 otherwise.

Theorem 5.6 (Order Abstractions are Well-behaved). *Given a predicate $v \leq c$, the abstraction $\mathcal{A}_{\leq c}$ is well-behaved.*

The abstraction $\mathcal{A}_{\geq c}$ for a predicate of the form $v \geq c$ is defined similarly by replacing the $\leq$ operator with $\geq$ in the elements of the abstract domain. We can also define similar abstractions for $\mathcal{A}_{<c}$ and $\mathcal{A}_{>c}$ by compiling them into equivalent formats with $v \leq c - 1$ and $v \geq c + 1$, respectively, for appropriate definitions of -1 and $+1$. Note that in this last case we assume a non-dense numerical domain where -1 and $+1$ can be naturally defined (e.g., natural numbers). Similarly to what we mentioned for inequalities, one can also define finer abstractions where one keeps finite disjunctions of equalities before generalizing to comparison operators.

5.3 String Predicates

String values are perhaps the most used values in the AWS IAM policy language. It is common to use simplified regular expressions with wildcards—which we call patterns—to describe sets of such values. We assume an alphabet of characters Σ and use Σ^* to denote all strings over this alphabet. We use ε to denote the empty string of length 0. A pattern `pat` is a sequence $\sigma_0 W_1 \sigma_1 \cdots W_n \sigma_n$ (such that $n \geq 0$) where each $\sigma_i \in \Sigma^*$ is a string and each $W_i \in \{?, *\}$ is a either a character wildcard ? (i.e., any possible character) or a string wildcard $*$ (i.e., any possible string). A string is accepted by a pattern whenever there is a way to replace the wildcards with parts of the input string so that the result is the string itself. Given a string $\sigma = c_1 \cdots c_m$ and a pattern $\texttt{pat} = \sigma_0 W_1 \sigma_1 \cdots W_n \sigma_n$, we say that a mapping $\omega : \{W_1, \ldots, W_n\} \mapsto \Sigma^*$ is a valid match for σ on `pat` iff $\sigma = \sigma_0 \omega(W_1) \sigma_1 \cdots \omega(W_n) \sigma_n$ and for every $1 \leq i \leq n$, if $W_i = ?$ then $\omega(W_i) \in \Sigma$, and if $W_i = *$ then $\omega(W_i) \in \Sigma^*$. We use $\Omega(\sigma, \texttt{pat})$ to denote the set of all matches and assume a total order over them. We use $\Omega(\sigma, \texttt{pat})[0]$ to denote the first match in the order. We say a string σ is accepted by a pattern `pat` iff $\Omega(\sigma, \texttt{pat}) \neq \emptyset$. We write $\mathcal{L}(\texttt{pat})$ to denote the set of strings accepted by `pat` and say that a pattern `pat` implies another pattern `pat'`, i.e., $\texttt{pat} \rightarrow \texttt{pat}'$, iff $\mathcal{L}(\texttt{pat}) \subseteq \mathcal{L}(\texttt{pat}')$. We also assume a special pattern $\texttt{pat}_\bot$ that defines the empty set of strings, i.e., $\mathcal{L}(\texttt{pat}_\bot) = \emptyset$.

We will often simply write `pat` in place of the more verbose predicate $v \in \mathcal{L}(\texttt{pat})$. Our search space will consist of variations of the pattern `pat` where wildcards can be replaced with patterns that narrow the set of possible strings. First we define ways to create abstractions for wildcards—i.e., ? and $*$—and then show how to combine such abstractions for full patterns.

5.3.1 Wild-Character Abstraction. The following abstraction $\mathcal{A}_?$ captures the case in which we are trying abstracts sets of strings of length 1 (i.e., characters). Initially, the abstraction is just $\bot_?$ and when one character d is observed it generalizes to d itself—i.e., only the character d should be accepted. If different values are observed, we then generalize to ?.

Definition 5.7 (?-Abstraction). The abstraction $\mathcal{A}_? = (\Sigma, \Phi_?, \subseteq_{\mathcal{L}}, \beta_?, \gamma_?, \sqcup_?, \bot_?)$ is defined as:

- $\Phi_? = \Sigma \cup \{?\} \cup \{\bot_?\}$.
- $\beta_?(d) = d$.
- $\gamma_?(d) = \{d\}$ and $\gamma_?(?) = \Sigma$.
- The join is defined as follows:

$$x \sqcup_? y = \begin{cases} x & y \to x \\ y & x \to y \\ ? & x, y \in \Sigma \text{ and } x \neq y \end{cases}$$

Theorem 5.8 (?-Abstraction is Well-behaved). *The abstraction $\mathcal{A}_?$ is well-behaved.*

5.3.2 Prefix and Suffix Abstractions for $\star$. When considering strings accepted by a wild string $\star$, there are many ways to perform an abstraction from a finite set of strings, but not all of them are well-behaved. We present well-behaved abstractions and describe versions that are not.

The following abstraction $\mathcal{A}_{\texttt{pref}}$ captures the case in which we are trying abstracts sets of strings of variable length by taking their longest common prefix. We write $\texttt{lcp}(\sigma_1, \sigma_2)$ (resp. $\texttt{lcs}(\sigma_1, \sigma_2)$) to denote the longest common prefix (resp. suffix) of two strings σ_1 and σ_2. We use $|\sigma|$ to denote the length of a string even when the string denotes a pattern—e.g., $|ab| = |a?| = |\star b| = 2$.

Definition 5.9 (Prefix Abstraction). The abstraction $\mathcal{A}_{\texttt{pref}} = (\Sigma^*, \Phi_{\texttt{pref}}, \subseteq_{\mathcal{L}}, \beta_{\texttt{pref}}, \gamma_{\texttt{pref}}, \sqcup_{\texttt{pref}}, \bot_{\texttt{pref}})$ is defined as follows:

- $\Phi_{\texttt{pref}} = \Sigma^* \cup \{\sigma W \mid \sigma \in \Sigma^* \text{ and } W \in \{?, \star\} \cup \{\bot_{\texttt{pref}}\}\}$—i.e., constant strings or strings followed by a wildcard. Note that every element of the set is also a pattern.
- $\beta_{\texttt{pref}}(\sigma) = \sigma$.
- $\gamma_{\texttt{pref}}(\texttt{pat}) = \mathcal{L}(\texttt{pat})$.
- The following cases are evaluated in order

$$x \sqcup_{\texttt{pref}} y = \begin{cases} x & y \to x \\ y & x \to y \\ z? & z = \texttt{lcp}(x, y) \text{ and } |z| + 1 = |x| = |y| \text{ and } x \neq z? \text{ and } y \neq z? \\ z\star & z = \texttt{lcp}(x, y) \end{cases}$$

Lemma 5.10 (Order Abstractions are Well-behaved). *The abstraction $\mathcal{A}_{\texttt{pref}}$ is well-behaved.*

It is easy to modify the above abstraction to one that finds longest common suffixes instead of prefixes: $\mathcal{A}_{\texttt{suf}} = (\Sigma^*, \Phi_{\texttt{suf}}, \subseteq_{\mathcal{L}}, \beta_{\texttt{suf}}, \gamma_{\texttt{suf}}, \sqcup_{\texttt{suf}}, \bot_{\texttt{suf}})$.

Ideally, one would want to combine these two abstractions into one that abstracts both in terms of prefixes and suffixes—e.g., the abstraction of *aa*, *aaa*, *aca* should be the conjunction of $a\star$ and $\star a$. In this particular case, we can see that one can express such a conjunction can be expressed as $a\star a$, but in general this is not always possible. For example, after abstracting the first two strings *aa* and *aaa*, the abstraction should be the conjunction of $aa\star$ and $\star aa$, but the combination $aa\star aa$ would be incorrect in this case as it would only accept strings with at least 4 *a*s! However, the combination is always sound when the strings used to build it have length greater or equal to the length of the prefix plus the length of the suffix. In our implementation, we support all such abstractions (`pref`, `suf`, and `pref + suf` with appropriate length checks); the default one is `pref`.

5.3.3 Patterns. We now define abstractions for string patterns. We assume $\mathcal{A}_* = (\Sigma^*, \Phi_*, \subseteq_{\mathcal{L}}, \beta_*, \gamma_*, \sqcup_*, \bot_*)$ is a well-behaved abstraction for $*$ such as $\mathcal{A}_{\texttt{pref}}$ or $\mathcal{A}_{\texttt{suf}}$. Given a pattern $\texttt{pat} = \sigma_0 W_1 \sigma_1 \cdots W_n \sigma_n$, the abstraction $\mathcal{A}_{\texttt{pat}}$ works as follows. Given a set of strings $\sigma_1, \ldots, \sigma_n$, it finds the first possible match for each string in the pattern and then applies the wildcard abstractions described in the previous sections for the substrings matched by each wildcard.

Definition 5.11 (Pattern Abstraction). Given a pattern $\texttt{pat} = \sigma_0 W_1 \sigma_1 \cdots W_n \sigma_n$, we can define the abstraction $\mathcal{A}_{\texttt{pat}}$ with the following components:

- $\Phi_{\texttt{pat}} = \{\sigma_0 x_1 \sigma_1 \cdots x_n \sigma_n \mid x_i \in \Phi_{W_i}\} \cup \{\texttt{pat}_\bot\}$.
- $\beta_{\texttt{pat}}(\sigma) = \sigma_0 x_1 \sigma_1 \cdots x_n \sigma_n$ such that $x_i = \Omega(\sigma, \texttt{pat})[0](W_i)$.
- $\gamma_{\texttt{pat}}(\texttt{pat}') = \mathcal{L}(\texttt{pat}')$.
- The join is defined as:

$$\texttt{pat}_1 \sqcup_{\texttt{pat}} \texttt{pat}_2 = \begin{cases} \texttt{pat}_1 & \texttt{pat}_2 \longrightarrow \texttt{pat}_1 \\ \texttt{pat}_2 & \texttt{pat}_1 \longrightarrow \texttt{pat}_2 \\ \sigma_0 z_1 \sigma_1 \cdots z_n \sigma_n & \texttt{pat}_j = \sigma_0 x_1^j \sigma_1 \cdots x_n^j \sigma_n \text{ and for every } i, z_i = x_i^1 \sqcup_{W_i} x_i^2 \end{cases}$$

The above abstraction is not always well-behaved as shown by the following example.

Example 5.12 (Ambiguous Pattern). Consider the pattern $\texttt{pat}_{amb}$ =`a*a*` and the following strings:

- $\sigma_1 = aaaa$ with set of matches $\Omega(\sigma_1, \texttt{pat}_{amb}) = [(aa, \varepsilon), (a, a), (\varepsilon, aa)]$;
- $\sigma_2 = aaaaa$ with set of matches $\Omega(\sigma_2, \texttt{pat}_{amb}) = [(aaa, \varepsilon), (aa, a), (a, aa), (\varepsilon, aaa)]$;
- $\sigma_3 = abaa$ with set of matches $\Omega(\sigma_3, \texttt{pat}_{amb}) = [(b, a)]$.

The function $\beta_{\texttt{pat}_{amb}}$ only considers the first match and thus maps the first and second wildcards to the two sets of values $\{aa, aaa, b\}$ and $\{\varepsilon, \varepsilon, a\}$, respectively. Thus, the abstraction computed by $\mathcal{A}_{\texttt{pat}_{amb}}$ on these input strings is `a*a*` itself. However, there exists a tighter abstraction `a(?)a(a*)` (we highlight the updated wildcard abstractions using parenthesis) that can be computed using the matches $(a, a), (a, aa), (b, a)$ for the three strings σ_1, σ_2, and σ_3, respectively.

Patterns like the one in Example 5.12 are uncommon and undesirable in AWS IAM, as they are ambiguous and can lead to behaviors that are hard to understand. We say that `pat` is unambiguous if for every string σ, the set $\Omega(\sigma, \texttt{pat})$ contains at most one match (it is decidable in polynomial time to check if a pattern is unambiguous [1]).

Theorem 5.13 (Order Abstractions are Well-behaved). *Given an unambiguous pattern* `pat` *the abstraction* $\mathcal{A}_{\texttt{pat}}$ *is well-behaved.*

If a pattern is ambiguous, the abstraction given Definition 5.11 is not well-behaved, but it will still produce refined patterns that are *sound* and `readable`, though they might not be guaranteed to be *tight* as the join operator is not guaranteed to return a least element.

Similarly to what we mentioned for other domains, one can also define a fine abstractions for strings that keeps a finite disjunction of string equalities before generalizing to prefix- or suffix-abstractions. We discuss this choice in one of our case studies in Section 7.

5.4 Finite Disjunctions

The AWS IAM Policy syntax provides conjunction and disjunction operators [37]. In this section, we consider disjunctions of other predicates—i.e., $\psi_\vee = \psi_1 \vee \cdots \vee \psi_n$. For example, this type of predicate can be used to specify that actions should be allowed if they start with one of two possible prefixes—e.g., `action` ∈ `s3:List*` ∨ `action` ∈ `s3:Get*`.

In the following, we assume $\mathcal{A}_{\psi_i} = (val(\psi_i), \Phi_{\psi_i}, \subseteq_{\mathcal{L}}, \beta_{\psi_i}, \gamma_{\psi_i}, \sqcup_{\psi_i}, \bot_{\psi_i})$ is a well-behaved abstraction for every ψ_i. Intuitively, the abstraction $\psi_\vee$ starts with the predicate $\bot_{\psi_1} \vee \cdots \vee \bot_{\psi_n}$ and

iteratively updates each disjunct. At any point, when the current abstract value is $\psi'_1 \vee \cdots \vee \psi'_n$, given an input value d the abstraction finds the first disjunct ψ'_i such that $val(\psi_i)$ contains d and updates the predicate ψ'_1 by computing the i-th join with the i-th beta function applied to d—i.e., the new i-th disjunct is $\psi'_i \sqcup_{\psi_i} \beta_{\psi_i}(d)$. All other disjuncts are untouched. This abstraction is similar to the one for multiple allow statements in Theorem 4.16.

Definition 5.14 (Disjunction Abstraction). The abstraction $\mathcal{A}\psi_\vee = (val(\psi_\vee), \Phi_{\psi_\vee}, \rightarrow, \beta_{\psi_\vee}, \gamma_{\psi_\vee}, \sqcup_{\psi_\vee})$ is defined as follows:

- $\Phi_{\psi_\vee} = \{\psi'_1 \vee \cdots \vee \psi'_n \mid \forall i. \psi'_i \in \Phi_{\psi_i}\}$.
- $\beta_{\psi_\vee}(c) = \psi'_1 \vee \cdots \vee \psi'_n$ such that for every i, if $c \in val(\vee_{j<i}\psi_j)$ or $c \notin val(\psi)_i$ then $\psi'_i = \bot_{\psi_i}$ else $\psi'_i = \beta_{\psi_i}(c)$ (i.e., only put it in the first i that matches).
- $\gamma_{\psi_\vee}(\psi'_1 \vee \cdots \vee \psi'_n) = \bigcup_i \gamma_{\psi_i}(\psi'_i)$.
- $\psi'_1 \vee \cdots \vee \psi'_n \sqcup_{\psi_\vee} \psi''_1 \vee \cdots \vee \psi''_n = \psi'_1 \sqcup_{\psi_1} \psi''_1 \vee \cdots \vee \psi'_n \sqcup_{\psi_n} \psi''_n$.
- $\bot_{\psi_\vee} = \bot_{\psi_1} \vee \cdots \vee \bot_{\psi_n}$.

The abstraction $\mathcal{A}_{\psi_\vee}$ is very similar to the one we defined for policies (see Example 4.15) and therefore requires similar restrictions—i.e., each disjunct should match disjoint sets of values.

Theorem 5.15 (Disjunction Abstractions are Well-behaved). *Given a predicate $\psi_\vee = \psi_1 \vee \cdots \vee \psi_n$, if the predicates $\psi_1, \ldots, \psi_n$ describe disjoint sets of values—i.e., for every distinct i and j, $val(\psi_i) \cap val(\psi_j) = \emptyset$—the abstraction $\mathcal{A}\psi_\vee$ is well-behaved.*

If the disjuncts are not disjoint, the abstraction given in Definition 5.14 is not well-behaved, but it will still produce refined predicates that are *sound* and `readable`, though they might not be guaranteed to be *tight*.

5.5 Finite Conjunctions

In this section, we consider conjunctions of other predicates—i.e., $\psi_\wedge = \psi_1 \wedge \cdots \wedge \psi_n$. For example, this type of predicate can be used to specify that a request value should be in a specific range—e.g., `s3:max-keys` $\leq$ `5000` $\wedge$ `s3:max-keys` $\geq$ `4000`.

In the following, we assume $\mathcal{A}_{\psi_i} = (val(\psi_i), \Phi_{\psi_i}, \subseteq_{\mathcal{L}}, \beta_{\psi_i}, \gamma_{\psi_i}, \sqcup_{\psi_i}, \bot_{\psi_i})$ is a well-behaved abstraction for every ψ_i. Intuitively, the abstraction $\psi_\wedge$ starts with the predicate $\bot_{\psi_1} \wedge \cdots \wedge \bot_{\psi_n}$ and iteratively updates each conjunct for every value being considered (note that every value must match every conjunct since we are taking the conjunction).

Definition 5.16 (Conjunction Abstraction). The abstraction $\mathcal{A}\psi_\wedge = (val(\psi_\wedge), \Phi_{\psi_\wedge}, \rightarrow, \beta_{\psi_\wedge}, \gamma_{\psi_\wedge}, \sqcup_{\psi_\wedge})$ is defined as follows:

- $\Phi_{\psi_\wedge} = \{\psi'_1 \wedge \cdots \wedge \psi'_n \mid \forall i. \psi'_i \in \Phi_{\psi_i}\}$.
- $\beta_{\psi_\wedge}(c) = \beta_{\psi_1}(c) \wedge \cdots \wedge \beta_{\psi_n}(c)$ (note that c must satisfy every ψ_i by definition).
- $\gamma_{\psi_\wedge}(\psi'_1 \wedge \cdots \wedge \psi'_n) = \bigcap_i \gamma_{\psi_i}(\psi'_i)$.
- $\psi'_1 \wedge \cdots \wedge \psi'_n \sqcup_{\psi_\wedge} \psi''_1 \wedge \cdots \wedge \psi'_n = \psi'_1 \sqcup_{\psi_1} \psi''_1 \wedge \cdots \wedge \psi'_n \sqcup_{\psi_n} \psi''_n$.
- $\bot_{\psi_\wedge} = \bot_{\psi_1} \wedge \cdots \wedge \bot_{\psi_n}$.

Theorem 5.17 (Conjunction Abstractions are Well-behaved). *Given a predicate $\psi_\wedge = \psi_1 \wedge \cdots \wedge \psi_n$, the abstraction $\mathcal{A}\psi_\wedge$ is well-behaved.*

5.6 Other AWS IAM Predicates

In this section, we describe special predicate types not discussed in the previous sections.

Null and IfExists Operators. AWS IAM contains a `Null` operator that is used to check that a certain variable *does not* appear in a request. We can define `Null` as an equality constraint since we have defined our requests to allow the special value `null` to denote the absence of a value.

Similarly, the `IfExists` operator is used to deal with requests that may or may not contain values for a specific variable [42]. For example, one can write `StringLikeIfExists: { ec2:InstanceType: "t1.*" }` to express that a request does not need to contain the variable `ec2:InstanceType`, but if such a variable is present in the request, it should match the pattern `t1.*`. We can represent this operator as a disjunction of a `Null` and a `StringLike` predicates—i.e., the value of `ec2:InstanceType` is `null` or it matches the pattern in the `StringLike` operation.

ARN operator. ARN (AWS Resource Name) [31] is a special string format provided by AWS where strings are comprised of six fields divided by `:`—e.g., `plclass`. AWS IAM provides special ARN string-comparison operators which restrict the semantics of matched strings to be valid ARNs. Our string abstractions can handle ARN operators.

Negation. AWS IAM allows a few negated operators—e.g., `StringNotLike`—which are error-prone and uncommon in practice. In general, it is hard to define abstractions in the presence of negation as this operator breaks monotonicity. For negated predicates, we define a trivial abstraction that contains only the original predicate—i.e., the refined predicate is always the original one.

Quantifiers. AWS IAM also provides "quantified" predicates. For example, the predicate `ForAllValues:StringLike: { "aws:TagKeys": [ "a*", "b*" ] }` matches if the variable `aws:TagKeys` does not appear in the request or its value matches one of the patterns `a*` or `b*`. Quantified predicates can be desugared into predicates for which we have defined abstractions. For example, the above predicate can be rewritten as a predicate of the form `IfExists(a* ∨ b*)`.

Policy Variables. AWS supports policy variables that can be used to dynamically instantiate elements from the request context in a condition—e.g., `StringLike: { s3:prefix: "home/${aws:username}/*"}`. Because of this dynamic nature, in our implementation we do not modify attributes containing policy variables.

5.7 Predicates in other Policy Languages

As we discussed at the end of Section 3, other authorization languages such as Google Cloud IAM Policy [9], Azure IAM Policy [17], and Cedar [7] also use forms of allow/deny statements that enable modular refinements assuming one has abstractions for the predicates appearing in each statement. We describe what predicates used in these languages can and cannot be captured using the abstractions presented so far.

The above languages can contain complex custom functions involving API calls, which cannot be modeled with the abstractions we presented because the values computed by such API calls cannot be resolved statically. Cedar allows conditions to be negated using the `unless` keyword, a feature that breaks monotonicity. Cedar also allows policy templates, which are families of policies that can be instantiated with multiple variable values. Our refinement approach would need to be lifted to support this convenient "compression" feature of Cedar.

Despite these limitations, many actual policies written in practice in these languages, and particularly in Cedar, often use limited predicates—e.g., disjunctions of equalities, conjunctions of inequalities, IP addresses—that fall in the setting considered so far (see https://www.cedarpolicy.com/en/tutorial/context). These policies could be refined using the approach we presented.

We thus believe that our approach could be generalized to other access control languages. However, due to the aforementioned differences and because for other languages we do not have

access to example policies and access logs, our presentation, implementation, and evaluation focus on refining AWS IAM policies.

6 Generalizing from Negative Requests

We have shown how to compute a least-general version of a policy P that still allows all the requests in a set R. In some cases, it can be useful to solve a dual problem where we have identified a set of "negative" requests F that we *do not* want the policy P to allow anymore—e.g., AWS provides tools that identify permissions that are "suspicious" [51] or have *not* been used in a certain period of time [49]. We call these "negative" requests *findings*. The version of our tool that solves this dual problem is called `IAM-PolicyRefinerNeg`.

While when refining policies from positive requests R it is desirable to compute the least-permissive policy that still allows all the requests in R, doing the same for findings F would not make any sense; a policy that rejects all requests would always be a solution. In this case, the goal is to obtain a *most-permissive* variation of the policy P that does not grant any request in F.

Definition 6.1 (Most General Policies and Policy Refinement from Findings). Given a policy P, a set of findings F, and a policy search space $\mathcal{D}_P$, we say that $P^* \in \mathcal{D}_P$ is ***a** most general policy that does not grant the requests in F*: **Soundness:** $Granted(P^*) \subseteq Granted(P) \setminus F$; **Tightness:** there is no policy $P' \in \mathcal{D}_P$ such that $Granted(P^*) \subset Granted(P') \subseteq Granted(P) \setminus F$. The *policy-refinement-from-findings problem* is to find *a most general policy* that does not grant the requests in F.

The algorithm for computing least-general policies relies on computing joins over an abstract domain. To adapt the algorithm to compute most-general policies one would require to first find the most general policies that reject each of the requests in F and then taking their meet (instead of join). These operations cannot be defined uniquely for allow-statements—e.g., to remove a value `aaa` from the string predicate `*` using the prefix abstraction, one could output any of `b*`, `c*`, etc.

To avoid this problem, we exploit the duality of `allow` and `deny` statements. Instead of searching for a most permissive modification of the given allow statements, we search for *a new least-denying deny statement* to add to our policy P, something we are already equipped to do thanks to our algorithm for computing least permissive allow statements in Definition 4.11!

Definition 6.2 (Policy Search Space for Refinement from Negative Requests). Given a policy $P = \{s_1, \ldots, s_n\}$ and an initial `deny` statement d, the policy search space $\mathcal{D}_{P,d}$ is comprised of policies that can add a modified version of d to P—i.e., $\{s_1, \ldots, s_n, d'\} \in \mathcal{D}_P$ iff $d' \in \mathcal{S}_d$ (Definition 4.8).

The search space $\mathcal{D}_{P,d}$ only contains policies that are at most as permissive as P.

Theorem 6.3. *Every policy $P' \in \mathcal{D}_{P,d}$ is at most as permissive as P-, written as $Granted(P') \subseteq Granted(P)$.*

The definition of the search space $\mathcal{D}_{P,d}$ does not specify what the initial `deny` statement d should be. Our policy refinement algorithm works for any initial statement that already rejects all the findings in F. By default, our implementation uses the "universal" deny-statement (`deny, action : *, resource: *`), which rejects all possible requests. This search space allows one to refine what actions and what resource paths the denied requests can match.

When defining statement abstractions in Definition 4.11, we intentionally did not specify whether the effect e had to be an `allow` or a `deny` and, as discussed in Theorem 4.13, if s is a deny-statement, the least general statement in $\mathcal{S}_s$ consistent with a set of findings F is the one that allows as many requests as possible while denying all requests in F, which is exactly what we want! Thus the algorithm we presented in Section 4.2 can be reused with very little intervention. The idea is that one can simply compute the most general deny-statement d' in $\mathcal{S}_d$ that denies all findings in F

Table 1. Summary of results for running `IAM-PolicyRefiner`.

Name	# of requests	Time (s)		# policies		# statements		# actions		# resources refined	# conditions refined
		1-CPU	96-vCPUs	before	after	before	after	before	after		
mot	100,000	6.73	1.29	1	1	4	4	148	3	4/4	2/2
role1	78,568	5.36	1.00	1	1	3	1	4	2	3/3	0/0
role2	76,571	11.02	1.12	2	2	6	4	83	11	6/6	0/1
role3	197,100	75.33	5.46	5	3	76	12	726	20	75/131	27/28
role4	86,003	9.96	1.02	3	1	7	3	10	3	7/7	0/0
role5	229,580	14.98	3.01	1	1	4	3	6	3	4/4	0/0
role6	151,844	17.99	2.09	1	1	1	1	7	2	1/1	0/0
role7	76,748	10.35	1.14	2	2	4	3	7	4	4/4	0/0
role8	135,049	22.06	2.54	4	2	9	2	39	2	8/12	0/0
role9	65,456	9.62	1.09	1	1	9	6	78	5	5/12	1/2
role10	92,521	13.37	1.31	2	2	6	5	83	13	6/6	0/1
role11	322,100	53.92	5.45	3	2	3	2	235	2	3/3	0/0

using the abstraction $\mathcal{A}_d$ (as per Theorem 4.13). Because of how the search space $\mathcal{D}_{P,d}$ is defined, the policy $\{s_1, \ldots, s_n, d'\}$ will be the *most* general policy that denies the requests in F and thus a solution to the policy-refinement-from-findings problem. Furthermore, from the definition of d, the abstraction $\mathcal{A}_d$ is guaranteed to be well-behaved as long as the requests match the restrictions discussed in the previous sections.

7 Evaluation

`IAM-PolicyRefiner` is implemented as approximately 3K lines of Scala. `IAM-PolicyRefiner` takes as input IAM policies [39] and JSON representations of the requests compatible with CloudTrail. We use `IAM-PolicyRefinerNeg` to denote the version of `IAM-PolicyRefiner` that refines from negative requests (as discussed in Section 6). On top of the formats supported by `IAM-PolicyRefiner`, `IAM-PolicyRefinerNeg` also supports a "partial" representation of events where only some variables are present with the semantics that any request that can be obtained by adding missing variables counts as a negative requests—e.g., if a partial event contains only a value for the variable `action`, every request with that action value counts as negative.

At the high level, our implementation of `IAM-PolicyRefiner` operates as follows (the one for `IAM-PolicyRefinerNeg` is similar). First, each request is mapped to the first matched allow statement. Second, we collect a map from each statement to matched requests and compute the corresponding abstractions (by mapping variables to the corresponding predicates) and take their joins. In our implementation, parallelization is achieved by adopting parallel versions of the `map` and `reduce` functions in Scala (i.e., the implementation is the same with and without parallelization) for the above operations.

Our evaluations is performed on m5.24xlarge Amazon EC2 instance [46]. The machine is powered by Intel Xeon Platinum 8175M or 8259CL processors, and the machine we chose offers 96 vCPUs and 3894 GiB of memory. To avoid performance variations from JVM cold start and vCPU allocations on actual CPUs, we run each experiment 6 times for each task, ignore the first run, and report the average time of the remaining runs.

7.1 Effectiveness of `IAM-PolicyRefiner`

On top of the motivating example (mot) discussed in Section 2 (for which we manually generated 100K requests similar to the ones used in Section 2), we collected AWS Identity-based policies [34] attached to 11 AWS IAM roles [41]—i.e., identities used to delegate access to users, applications or services for repetitive or temporary tasks. The 11 roles (and the attached policies) are actual roles in 4 AWS accounts volunteered by colleagues in our organization. The number of policies attached to

each role ranges from 1 to 5, and the number of statements in each policy ranges from 1 to 19 (in our setting, we simply take the union of all the statements even when they are across multiple policies). We collected AWS CloudTrail logs [33] from the same 11 roles for three contiguous months in 2023 (70K to 320K requests per role).

Table 1 shows how `IAM-PolicyRefiner` (with parallelization) terminates within seconds on every benchmark. To better understand performance, we note that the time for mapping requests to policies is proportional to the number of requests and the number of statements (the slowest policy has 76 statements). The time taken to compute joins is affected by the number of variables and conditions in each statement (the slowest policy has 131 resources and 28 conditions).

To strengthen the confidence of our implementation, we used the policy checker from AWS IAM Access Analyzer [40] and verified that all refined policies are less permissive than the original ones—i.e., `IAM-PolicyRefiner` is sound.

Q1: How are policies changed by `IAM-PolicyRefiner`*?* Columns 5-10 in Table 1 show the number of policies/statements/actions before and after refining the policies. For actions, we reported actual numbers of AWS actions allowed; we use the tool Zelkova to compute the full set of allowed actions [4]. Columns 11-12 show what fractions of resources-values/condition-values were modified—e.g., for `role3`, 75 out of 131 resource values are modified when producing the refined policy. Note that each statement can have multiple resource values due to disjunction.

`IAM-PolicyRefiner` always removes at least one action and modifies at least one resource. On average, `IAM-PolicyRefiner` removed 76.46% of the actions (ranging from 50% to 99.15%) and refined or removed 87.79% of the resources (41.67% to 100%) present in the original policies. Furthermore, even some policies and statements are entirely removed when they are never required for the collected requests. For example, the number of policies attached to `role3` is reduced from 5 to 3, and overall 64 statements are removed from the policies.

Across all roles, we encountered 12 AWS and customer managed policies, which provide permissions for common user use cases. Managed policies typically allow many actions (e.g., one managed policy in `role2` allows 80 actions) on many resources. In these cases, the refined policies can drastically reduce permissions.

Only 5 roles contained policies with `condition` values. For these roles, `IAM-PolicyRefiner` refined between 0% to 96.43% of the condition values. Conditions are usually set by users to specific values, thus resulting in fewer opportunities for refinements. Despite this, 27 out of 28 condition values for `role3` were refined by `IAM-PolicyRefiner`.

Some policies or statements were not matched by any request and were removed altogether.

Q1 summary: `IAM-PolicyRefiner` is *effective* at reducing permissions and often modifies several components of the given policies.

Q2: Do refined policies overfit to the given requests? To assess whether the refined policies are not overfitting to the given requests, we divide the requests into training and test set and then refined policies using only requests in the training set and assess generalization on the test set.

We vary the size of the "training set" between 5 and 100% of the CloudTrail requests at hand (X% means we select the first X% requests in the log), and Figure 4 shows what percentage of *all* requests (y axis) is allowed when only a percentage of the requests is used for training/refining (x axis). Any point below 100 denotes that some requests

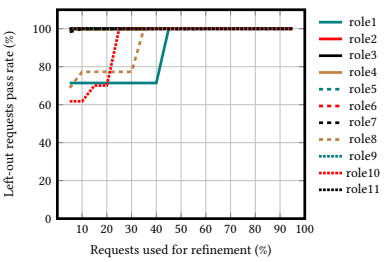

Fig. 4. Accuracy on left-out requests

were incorrectly denied and the refined policy partly overfit—e.g., when providing only 10% of the requests in the log for role 10, only 62% of all requests are allowed.

When refining using only 5% of the requests, 6 roles already allow all requests. When refining using 50% of the requests, all roles allow all requests.

We analyzed the policies that required more than 5% of the requests to generalize. For these benchmarks, the beginning of the log contains specific requests that create resources or write data on resources, and the resources are deleted or data is transferred to other resources later in the log.

Q2 summary: When considering logs spanning longer than 2 months, the policies computed by `IAM-PolicyRefiner` *generalize well* to unseen requests.

Q3: How effective is parallelization? Parallelization using 96 vCPUs improves the execution time by 8.7x (geometric mean) and brings the total execution times from tens of seconds to seconds, thus making the `IAM-PolicyRefiner` practical. We attribute the limited gain of parallelization to parallelization overhead over low cost of computing the least general generalization.

We verified that the policies computed with and without parellization are the same.

Q3 summary: Parallelization is *effective* and makes `IAM-PolicyRefiner` practical.

Q4: Do policies match the restrictions imposed by our theorems? We found that for 2 roles *some requests are allowed by more than one statement*, thus violating the "disjoint sets" condition of Theorem 4.16. In such cases, the policies might not admit a unique minimal policy, but the policy computed by `IAM-PolicyRefiner` is still sound (which we validated using IAM Access Analyzer). Furthermore, as shown in Q3, the policies computed with and without parallelization are identical, as predicted by our theory.

Q4 summary: 2/11 roles did not meet our assumption, but the refined policies are still sound and `IAM-PolicyRefiner` is effective at reducing permissions for such cases.

7.2 Effectiveness of `IAM-PolicyRefinerNeg`

To evaluate `IAM-PolicyRefinerNeg` (i.e., the approach presented in Section 6), we collected policies for 2 accounts and collected 6 roles with corresponding unused findings produced using AWS IAM Access Analyzer over 180 days [49]. Since the AWS unused finding service became available in the late of 2023, we could not use the roles described in Section 7.1.

The current version of AWS IAM Access Analyzer [49] only produces findings in the form of sets of unused actions and does not provide complete requests. Therefore, in this experiment we will only generalize the action component of the given deny statement (`deny`, `action`: *, `resource`: *).

Q5: Quality of `IAM-PolicyRefinerNeg` *refinements?* When computing refined policies using the same action abstraction considered so far (i.e., the prefix abstraction), `IAM-PolicyRefinerNeg` returned the universal deny statement (`deny`, `action`: *, `resource`: *) for all benchmarks—such a statement would result in no request being granted! This undesired outcome happens whenever two findings contain action names from different AWS services—e.g., `lambda:DeleteFunction` and `logs:CreateGroup`—and thus do not share any prefix or suffix.

To address this problem, we took advantage of the modularity of our approach and considered the more "precise" abstraction for strings that maintains a finite disjunction of string equalities (last paragraph of Section 5.3). On average, the number of actions allowed by each role is reduced by 56.58% ranging from 18.18% to 97.7%.

We verified refined policies still allow actions that are allowed by original policies but do not appear in the set of findings [4]. To avoid producing a deny statement that simply lists all denied actions, we used a trie to create a smaller list of action values for the policy with longest prefix that

avoids actions that are allowed by original policies. For example, the trie creates `s3:List*` when unused findings contain all possible actions starting with `s3:List`.

Q5 summary: When using a more precise abstraction, `IAM-PolicyRefinerNeg` can refine policies from negative findings.

8 Related Work

Formal Methods for Identity Management. Policy languages have been used in a variety of domains, e.g., trust management, distributed authorization, role-based access, and formal methods have been used to reason about them in a variety of ways [11, 13, 15]. The work most related to ours in this space is Zelkova [4], which targets the same policy language we target, AWS IAM. `IAM-PolicyRefiner` and Zelkova are two different automated reasoning tools for AWS IAM policies and serve different purposes. While `IAM-PolicyRefiner` is a tool for synthesizing modifications of AWS IAM policies, Zelkova provides an intermediate SMT-based representation for reasoning about AWS IAM policies and perform various forms of automated analysis—e.g., checking whether two policies are equivalent.

Program Repair. Our work can be thought of a form of program repair [14], where the goal is to find a correct program, starting from an incorrect program and a violated specification (typically a set of test cases). What differentiates our work from the existing literature on program repair is that we only have access to an initial (potentially incorrect) policy, but no "failing" test cases with respect to which we want to modify our policy. Because of this reason, our work defines the problem of repairing the initial policy in terms of finding the most precise policy in a search space that is still consistent with the set of accepted requests (i.e., the test cases). The idea of having a fixed search space is common in most forms of program repair and program synthesis and allows one to avoid overfitting to the given test cases.

The approach in Section 5.3.3 for identifying what substrings are relevant when modifying wildcards is similar to how RFixer [22], a tool for repairing regular expressions from examples, executes a regular expression symbolically to identify what characters can appear in a character class. Despite this minor similarity, our approaches solve different problems as RFixer's goal is to find *the smallest* modification of an initial regular expression that is consistent with a set of positive an negative examples, whereas our goal is to find *the least general* modification of the wildcards in a regular expression that is *still* consistent with a set of positive examples.

Generating Access Control from Logs. Cotrini et al. [10] syntactically mine access control rules from logs, and other approaches use machine learning (ML) to achieve the same task [6, 16, 18, 20, 21, 29, 30]. Our work differs from such approaches in three key ways. First, most existing approaches generate an entirely new policy from scratch, whereas `IAM-PolicyRefiner` refines the policy the user has already provided (i.e., the goal of readability). Second, ML-based approaches are limited in what syntactic features they can use (typically they only involve few variables and equality/inequality predicates) and provide no guarantees on the tightness and readability of the generated policy. Instead, `IAM-PolicyRefiner` provably solves a formally defined refinement problem where the goals are (formal definitions of) tightness, soundness and readability. The only prior refinement work [18] also advocates for the importance of readability. However, their approach is heuristic (i.e., it does not define readability), targets an artificial refinement language with only equality predicates and does not support AWS IAM predicates (e.g., strings, which are ubiquitous in our evaluation and in AWS IAM), and is based on machine learning (thus it does not have guarantees). Instead, `IAM-PolicyRefiner` takes a language-based formal approach based on least general generalization to efficiently find a provably sound, readable, and tight policy in AWS IAM. Due to the these limitations, we cannot compare to existing tools in our evaluation.

9 Conclusions

We introduced `IAM-PolicyRefiner`, a tool that automatically synthesizes refined AWS IAM access control policies from access logs. `IAM-PolicyRefiner` frames the problem of computing refined policies as a least-general generalization problem, an approach that enables modular language design and efficient parallelization. Our evaluation on policies and logs from development accounts shows that `IAM-PolicyRefiner` is fast (<5s per policy), effective (modified policies allow up to 99% fewer actions and modify up to 100% resource attributes values), and does not overfit to the requests in the given access log.

Our methodology opens opportunities for automatically refining policies in other languages and frameworks (e.g., Cedar [7], Google GCP [9], Azure [17]).

References

[1] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. 2011. General Algorithms for Testing the Ambiguity of Finite Automata and the Double-Tape Ambiguity of Finite-State Transducers. *Int. J. Found. Comput. Sci.* 22, 4 (2011), 883–904. https://doi.org/10.1142/S0129054111008477

[2] Azure. 2023. Best practices for Azure AD roles. https://learn.microsoft.com/en-us/azure/active-directory/roles/best-practices

[3] John Backes, Ulises Berrueco, Tyler Bray, Daniel Brim, Byron Cook, Andrew Gacek, Ranjit Jhala, Kasper Luckow, Sean McLaughlin, Madhav Menon, Daniel Peebles, Ujjwal Pugalia, Neha Rungta, Cole Schlesinger, Adam Schodde, Anvesh Tanuku, Carsten Varming, and Deepa Viswanathan. 2020. Stratified Abstraction of Access Control Policies. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 165–176. https://doi.org/10.1007/978-3-030-53288-8_9

[4] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. Institute of Electrical and Electronics Engineers (IEEE), 1–9. https://doi.org/10.23919/FMCAD.2018.8602994

[5] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Dan Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. 2020. Block public access: trust safety verification of access control policies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 281–291. https://doi.org/10.1145/3368089.3409728

[6] Luca Cappelletti, Stefano Valtolina, Giorgio Valentini, Marco Mesiti, and Elisa Bertino. 2019. On the Quality of Classification Models for Inferring ABAC Policies from Access Logs. In *2019 IEEE International Conference on Big Data (Big Data)*. Institute of Electrical and Electronics Engineers (IEEE), 4000–4007. https://doi.org/10.1109/BigData47090.2019.9005959

[7] Cedar. 2024. Cedar. https://www.cedarpolicy.com/en

[8] Google Cloud. 2023. Least privilege for Cloud Functions using Cloud IAM. https://cloud.google.com/blog/products/application-development/least-privilege-for-cloud-functions-using-cloud-iam

[9] Google Cloud. 2024. GCP Policy. https://cloud.google.com/iam/docs/reference/rest/v1/Policy

[10] Carlos Cotrini, Thilo Weghorn, and David Basin. 2018. Mining ABAC Rules from Sparse Logs. In *2018 IEEE European Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers (IEEE), 31–46. https://doi.org/10.1109/EuroSP.2018.00011

[11] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2006. Specifying and Reasoning About Dynamic Access-Control Policies. In *Automated Reasoning*, Ulrich Furbach and Natarajan Shankar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 632–646.

[12] William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, and Tevfik Bultan. 2022. Quantifying permissiveness of access control policies. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1805–1817. https://doi.org/10.1145/3510003.3510233

[13] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. 2005. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 196–205. https://doi.org/10.1145/1062455.1062502

[14] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013
[15] Graham Hughes and Tevfik Bultan. 2008. Automated verification of access control policies using a SAT solver. *STTT* 10 (12 2008), 503–520. https://doi.org/10.1007/s10009-008-0087-9
[16] Leila Karimi and James Joshi. 2018. An Unsupervised Learning Based Approach for Mining Attribute Based Access Control Policies. In *2018 IEEE International Conference on Big Data (Big Data)*. Institute of Electrical and Electronics Engineers (IEEE), 1427–1436. https://doi.org/10.1109/BigData.2018.8622037
[17] Microsoft. 2024. Azure Policy. https://learn.microsoft.com/en-us/azure/governance/policy/overview
[18] Shohei Mitani, Jonghoon Kwon, Nakul Ghate, Taniya Singh, Hirofumi Ueda, and Adrian Perrig. 2023. Qualitative Intention-aware Attribute-based Access Control Policy Refinement. In *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies* (Trento, Italy) *(SACMAT '23)*. Association for Computing Machinery, New York, NY, USA, 201–208. https://doi.org/10.1145/3589608.3593841
[19] Tom M Mitchell. 1997. *Machine learning*. Vol. 1. McGraw-hill New York. Issue 9.
[20] Ian Molloy, Youngja Park, and Suresh Chari. 2012. Generative models for access control policies: applications to role mining over logs with attribution. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies* (Newark, New Jersey, USA) *(SACMAT '12)*. Association for Computing Machinery, New York, NY, USA, 45–56. https://doi.org/10.1145/2295136.2295145
[21] Mohammad Nur Nobi, Ram Krishnan, Yufei Huang, Mehrnoosh Shakarami, and Ravi Sandhu. 2022. Toward Deep Learning Based Access Control. In *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy* (Baltimore, MD, USA) *(CODASPY '22)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/3508398.3511497
[22] Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic Repair of Regular Expressions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 139 (oct 2019), 29 pages. https://doi.org/10.1145/3360565
[23] Hila Peleg, Sharon Shoham, and Eran Yahav. 2016. D3: Data-Driven Disjunctive Abstraction. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 185–205.
[24] G. Plotkin. 1971. A further note on inductive generalization. In *Machine Intelligence*. Vol. 6. Edinburgh University Press.
[25] Gordon D. Plotkin. 1970. A Note on Inductive Generalization. *Machine Intelligence* 5 (1970), 153–163.
[26] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2014. Programming by Example Using Least General Generalizations. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence* (Québec City, Québec, Canada) *(AAAI'14)*. AAAI Press, 283–290.
[27] Rego. 2024. Rego. https://www.openpolicyagent.org/docs/latest/policy-language/
[28] Thomas Reps and Aditya Thakur. 2016. Automating Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–40.
[29] Matthew W. Sanders and Chuan Yue. 2018. Minimizing Privilege Assignment Errors in Cloud Services. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (Tempe, AZ, USA) *(CODASPY '18)*. Association for Computing Machinery, New York, NY, USA, 2–12. https://doi.org/10.1145/3176258.3176307
[30] Matthew W Sanders and Chuan Yue. 2019. Mining least privilege attribute based access control policies. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 404–416. https://doi.org/10.1145/3359789.3359805
[31] Amazon Web Services. 2023. Amazon Resource Names (ARNs). https://docs.aws.amazon.com/IAM/latest/UserGuide/reference-arns.html
[32] Amazon Web Services. 2023. Amazon S3. https://aws.amazon.com/s3/
[33] Amazon Web Services. 2023. AWS CloudTrail. https://aws.amazon.com/cloudtrail/
[34] Amazon Web Services. 2023. AWS Identity-based policies. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html#policies_id-based
[35] Amazon Web Services. 2023. AWS Key Management Service. https://aws.amazon.com/kms/
[36] Amazon Web Services. 2023. AWS managed policies. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_managed-vs-inline.html#aws-managed-policies
[37] Amazon Web Services. 2023. Evaluation logic for multiple context keys or values. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_condition-logic-multiple-context-keys-or-values.html
[38] Amazon Web Services. 2023. Example 3: Setting the maximum number of keys. https://docs.aws.amazon.com/AmazonS3/latest/userguide/amazon-s3-policy-keys.html#example-numeric-condition-operators
[39] Amazon Web Services. 2023. Grammar of the IAM JSON policy. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_grammar.html

[40] Amazon Web Services. 2023. IAM Access Analyzer custom policy checks. https://docs.aws.amazon.com/IAM/latest/UserGuide/access-analyzer-custom-policy-checks.html
[41] Amazon Web Services. 2023. IAM roles. https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html
[42] Amazon Web Services. 2023. ...IfExists condition operators. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_condition_operators.html#Conditions_IfExists
[43] Amazon Web Services. 2023. Protecting data with server-side encryption. https://docs.aws.amazon.com/AmazonS3/latest/userguide/serv-side-encryption.html
[44] Amazon Web Services. 2023. Security best practices in IAM. https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html#bp-use-aws-defined-policies
[45] Amazon Web Services. 2023. What is CIDR? https://aws.amazon.com/what-is/cidr/
[46] Amazon Web Services. 2024. Amazon EC2. https://aws.amazon.com/ec2
[47] Amazon Web Services. 2024. AWS global condition context keys. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_condition-keys.html
[48] Amazon Web Services. 2024. AWS username key. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_condition-keys.html#condition-keys-username
[49] Amazon Web Services. 2024. Find unused access, check policies before deployment. https://aws.amazon.com/blogs/aws/iam-access-analyzer-updates-find-unused-access-check-policies-before-deployment/
[50] Amazon Web Services. 2024. Findings for external and unused access. https://docs.aws.amazon.com/IAM/latest/UserGuide/access-analyzer-findings.html
[51] Amazon Web Services. 2024. Findings for public and cross-account access. https://docs.aws.amazon.com/IAM/latest/UserGuide/access-analyzer-findings.html
[52] Reudismam Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2021. Learning Quick Fixes from Code Repositories. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering* (Joinville, Brazil) *(SBES '21)*. Association for Computing Machinery, New York, NY, USA, 74–83. https://doi.org/10.1145/3474624.3474650
[53] Stackoverflow. 2022. How to write an S3 bucket policy to *only* allow specific IAM role and Cloudfront Origin Access Identity? https://stackoverflow.com/questions/46141593/how-to-write-an-s3-bucket-policy-to-only-allow-specific-iam-role-and-cloudfron
[54] Stackoverflow. 2022. Is there an S3 policy for limiting access to only see/access one bucket? https://stackoverflow.com/questions/6615168/is-there-an-s3-policy-for-limiting-access-to-only-see-access-one-bucket/6955864#6955864

Received 2024-03-26; accepted 2024-08-18