Quantitative Policy Repair for Access Control on the Cloud*

William Eiers
University of California Santa Barbara
Santa Barbara, CA, USA
weiers@cs.ucsb.edu

Ganesh Sankaran
University of California Santa Barbara
Santa Barbara, CA, USA
ganesh@cs.ucsb.edu

Tevfik Bultan
University of California Santa Barbara
Santa Barbara, CA, USA
bultan@cs.ucsb.edu

ABSTRACT

With the growing prevalence of cloud computing, providing secure access to information stored in the cloud has become a critical problem. Due to the complexity of access control policies, administrators may inadvertently allow unintended access to private information, and this is a common source of data breaches in cloud based services. In this paper, we present a quantitative symbolic analysis approach for automated policy repair in order to fix overly permissive policies. We encode the semantics of the access control policies using SMT formulas and assess their permissiveness using model counting. Given a policy, a permissiveness bound, and a set of requests that should be allowed, we iteratively repair the policy through permissiveness reduction and refinement, so that the permissiveness bound is reached while the given set of requests are still allowed. We demonstrate the effectiveness of our automated policy repair technique by applying it to policies written in Amazon's AWS Identity and Access Management (IAM) policy language.

CCS CONCEPTS

• **Security and privacy → Logic and verification**; **Access control**.

KEYWORDS

access control, policy analysis, policy repair, quantitative analysis

ACM Reference Format:
William Eiers, Ganesh Sankaran, and Tevfik Bultan. 2023. Quantitative Policy Repair for Access Control on the Cloud. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598078

1 INTRODUCTION

It is critical to protect privacy of the data stored in software services that run on compute clouds like Amazon Web Services (AWS) since data breaches in the cloud can have significant negative impact on millions of users. AWS Identity and Access Management (IAM) service [15] allows administrators to write policies that specify authorization and access control rules for the resources available in a software service. Although IAM provides a convenient language for writing policies, specification errors in manually written complex policies are bound to happen, leading to unintended and unauthorized access to data. Indeed, errors in access control policies in cloud storage services have already resulted in the exposure of millions of customers' data to the public, such as the exposure of the data records (including names, addresses, account information, email addresses, and last four digits of credit card numbers) of more than 2 million Dow Jones & Co. customers [7], and exposure of account records of 14 million Verizon customers [26].

Given the ubiquity of software services running on compute clouds, automated access control policy analysis techniques that can help administrators are critically important. There has been prior work on automatically analyzing access control policies for finding permissiveness errors [4, 5, 9, 10]. In this paper, we go one step further, and investigate the problem of automatically repairing overly permissive policies.

The access control policy analysis and repair problems are inherently quantitative since the issue is not whether an access control policy allows access to data resources, but *how much* access it allows. We call this the *permissiveness* of an access control policy. In particular, we investigate the problem of quantitative policy repair where the goal is not to completely eliminate all access to data (which would not be a feasible fix) but to reduce permissiveness to an acceptable level specified by the service administrator.

In this paper, we present a quantitative symbolic analysis approach for automated policy repair in order to reduce the permissiveness of access control policies. Our repair approach is sound, i.e., it guarantees that the repaired policy meets the given permissiveness constraints. Our contributions are the following:

- A formalization of the access control policy repair problem.
- A quantitative and symbolic policy repair algorithm for automatically reducing the permissiveness of a given access control policy.
- Access control policy permissiveness localization and reduction techniques, including a regular expression generalization technique for characterizing the set of resources based on a given set of access control requests.
- An experimental evaluation of our quantitative policy repair approach.

The rest of the paper is organized as follows. In Section 2 we formalize the policy repair problem and demonstrate its applicability through motivating examples, in Section 3 we discuss the policy model for analyzing the semantics of access control policies, in Section 4 we introduce our novel approach for policy repair through a quantitative policy repair algorithm, in Section 5 we discuss how

*This material is based on research sponsored by NSF under grants CCF-2008660, CCF-1901098 and CCF-1817242, and by DARPA under the agreement number N66001-22-2-4037. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598078

our approach can be applied to the AWS IAM policy language, in Section 6 we experimentally evaluate our approach, in Section 7 we discuss related work, and in Section 8 we conclude the paper.

2 MOTIVATION AND OVERVIEW

In this section we give an overview of the access control policy repair problem and provide motivating examples. From a security perspective, access control policies should grant only the permissions required to perform a task. Overly permissive policies, which grant more permissions than necessary, can allow attackers unfettered access to secure data if the associated role or users are compromised. Thus, overly permissive policies should be modified, or repaired, to allow only those requests which are necessary.

2.1 Policy Repair Problem

Access control policies grant permissions to users or services by allowing requests. The more permissions granted by a policy, the more requests it allows. The fewer permissions granted by a policy, the lower the number of requests it allows. In this sense, we can think of the number of permissions granted, or requests allowed, by the policy as defining the *permissiveness* of the policy. A natural question then is, given an overly permissive policy, is it possible to modify, or *repair* the policy so that it is no longer overly permissive? This gives rise to the *Policy Repair Problem*: Given an access control policy, ensure that it only allows the requests necessary to achieve its intended purpose. However, this is difficult to ensure as complex policy specifications are difficult to craft and the set of requests to be allowed or denied may not be explicitly defined or known.

In this paper, we introduce and formalize the following variation of the access control policy repair problem: Given a policy, a permissiveness bound, and a set of must-allow requests, check that the policy meets the permissiveness bound while allowing all the requests in the must-allow set, or repair it such that it meets the permissiveness bound and allows all the requests in the must-allow set. Note that permissiveness bound puts an *upper bound* on the desired level of permissiveness while the must-allow request set puts a *lower bound* on the desired level of permissiveness.

Permissiveness Bound. The permissiveness bound is a restriction on the permissiveness of the policy. That is, it is a restriction on the maximum number of requests allowed by the policy. If the permissiveness of the policy (number of requests allowed by the policy) is greater than the permissiveness bound, then we call this policy an *overly permissive* policy. Our approach aims to repair overly permissive policies by reducing the permissiveness of the policy so that the permissiveness of the policy is less than or equal to the permissiveness bound. While in this paper we assume that such a permissiveness bound is given a priori, we also discuss methods for automatically finding permissiveness bounds later in the paper.

Must-Allow Request Sets. The set of must-allow requests are requests which must be allowed by the policy. Without a must-allow request set, a policy that does not allow any requests would meet any permissiveness bound and would be a viable (but meaningless) solution to the policy repair problem. The must-allow request set is used to guide the algorithm towards a less permissive but still

```
"Statement": [{
 "Effect": "Allow",
 "Action": [
  "s3:ListBucket",
  "s3:GetObject",
  "s3:PutObject",
  "s3:DeleteObject"],
 "Resource": [
  "backend",
  "backend/logs"]},
{
 "Effect": "Allow",
 "Action": "s3:GetObject",
 "Resource": "backend/*"}]
```

```
"Statement": [{
 "Effect": "Allow",
 "Action": [
  "s3:ListBucket",
  "s3:GetObject",
  "s3:PutObject",
  "s3:DeleteObject"],
 "Resource": [
  "backend",
  "backend/logs"]},
{
 "Effect": "Allow",
 "Action": "s3:GetObject",
 "Resource": [
  "backend/user44012/status.log",
  "backend/user00000/status.log",
  "backend/user12345/status.log",
  "backend/user91232/status.log",
  "backend/admin12/status.log",
  "backend/admin02/status.log",
  "backend/admin443/status.log",
  "backend/admin3/status.log"]}]
```

Figure 1: Original (left, (a)), first repaired policy (right, (b))

```
"Statement": [{
 "Effect": "Allow",
 "Action": [
  "s3:ListBucket",
  "s3:GetObject",
  "s3:PutObject",
  "s3:DeleteObject"],
 "Resource": [
  "backend",
  "backend/logs"]},
{
 "Effect": "Allow",
 "Action": "s3:GetObject",
 "Resource": [
  "backend/user?????/status.log"
  "backend/admin*/status.log"]}]
```

```
"Statement": [{
 "Effect": "Allow",
 "Action": [
  "s3:ListBucket",
  "s3:GetObject",
  "s3:PutObject",
  "s3:DeleteObject"],
 "Resource": [
  "backend",
  "backend/logs"]},
{
 "Effect": "Allow",
 "Action": "s3:GetObject",
 "Resource": [
  "backend/user*/status.log",
  "backend/admin*/status.log"]}]
```

Figure 2: Second repaired policy (left, (a)), third repaired policy (right, (b))

useful policy. In our approach, we assume that the set of must-allow requests is given as input to the policy repair algorithm.

In our approach we assume that the policy developer has access to a set of must-allow requests. We assume that the policy developer has knowledge of, and access to, what kinds of requests should be definitely allowed by the policy. The concept of a must-allow request set is analogous to the concept of whitelists from the security domain which explicitly enumerate what should be allowed (e.g., a firewall only allowing requests from a certain domain). Typically, policy developers have access to such a whitelist, and we make the same assumption for the set of must-allow requests [19, 20, 28].

2.2 Motivating Examples

The goal of the repair algorithm is to find a policy repair that satisfies both of the above constraints (permissiveness bound and must-allow requests). To illustrate the policy repair problem concretely, we discuss a couple of motivating examples below.

Consider the role of an automated log consolidator in the Amazon Web Services (AWS) cloud, hereafter referred to as simply *logger*, which routinely gathers logs and consolidates them into a single log file for further analysis. The permissions granted to the *logger* role are given by the policy attached to the role. The initial policy attached to the logger role is given in Figure 1(a). This policy gives varied access to the "backend" AWS S3 bucket: The first statement allows the logger role to list objects within the bucket and gives read and write access to the "logs" object, while the second statement allows the logger role to read all objects within the "backend" bucket. Note that broad access is achieved through the use of the wildcard symbol '*' (representing any string) within the resource description "backend/*". Though not present in this first policy, the '?' symbol is used similarly to represent any character.

Essentially, this second statement allows the logger role to gather all logs in the bucket, while the first statement allows the logger role to consolidate those logs into a single logs file. This policy allows the logger role to accomplish its tasks. However, the policy gives the logger role read access to all objects in the "backend" bucket using the S3:GetObject action, regardless of whether or not the object is a log file. Ideally, the policy should be repaired so that it only allows access to log files within the "backend" bucket.

Repairing the permissiveness of the policy in 1(a) requires some information to be known regarding the requests fielded (allowed or denied) by the policy. Without such domain specific knowledge, the best repair would be to modify the policy to allow no requests.

Suppose that the following requests, which specify action and resource pairs, should be allowed by the policy:

```
("s3:ListBucket", "backend"), ("s3:PutObject", "backend/logs")
("s3:DeleteObject", "backend/logs")
("s3:GetObject", "backend/logs")
("s3:GetObject", "backend/user44012/status.log")
("s3:GetObject", "backend/user00000/status.log")
("s3:GetObject", "backend/user12345/status.log")
("s3:GetObject", "backend/user91232/status.log")
("s3:GetObject", "backend/admin12/status.log")
("s3:GetObject", "backend/admin02/status.log")
("s3:GetObject", "backend/admin443/status.log")
("s3:GetObject", "backend/admin3/status.log")
```

These requests represent what kind of actions and resources should be allowed by the original policy, which we refer to as the *must-allow request set*. Any repaired policy *must allow* these requests.

The simplest way to repair the policy is to explicitly enumerate the allowed requests within a statement in the policy, as shown in Figure 1(b). Instead of specifying "bucket/*" in the second statement (which specified all objects within the bucket), the list of known resources is explicitly specified by explicitly enumerating them. While this is a valid repair and does in fact reduce permissiveness, it does not handle other log files which may exist but were not captured in the must-allow request set. It simply makes the must-allow set the policy. In our approach, we remedy this by generalizing the allowed requests using resource characterization techniques.

The policies in Figure 2 show two repairs which our quantitative repair approach generates. Both policies reduce the permissiveness of the original policy. However, the second and third repaired policies *generalize* the resources from the must-allow request set. The second repaired policy (Figure 2(a)) generalizes requests containing the "user" and "admin" strings, but is more restrictive for resources containing the "user" string: It allows resources such as `bucket/user44012/status.log` which is in the must-allow request set, but does not allow `bucket/user1234567/status.log` which is not in the must-allow request set. The third repaired policy (Figure 2(b)) also generalizes requests containing the "user" and "admin" strings, but is equally as restrictive in both cases. Based on the input permissiveness constraints and parameters, our approach can generate repairs with different levels of permissiveness while meeting the permissiveness constraints. We discuss this further in Section 4.

```
"Statement": [{
 "Effect": "Allow",
 "Action": "s3:GetObject"
 "Resource": "backend/*"}]
```

```
"Statement": [{
 "Effect": "Allow",
 "Action": "s3:GetObject"
 "Resource":"backend/logs/user*"}]
```

```
"Statement": [{
 "Effect": "Allow",
 "Action": "s3:GetObject"
 "Resource": "backend/logs/user?????"}]
```

Figure 3: Original policy (top left, (a)), partially repaired policy (top right, (b)), fully repaired policy (bottom, (c))

Permissiveness Bound Example. In this example we discuss the importance of the permissiveness bound in the repair process. Recall that the permissiveness of a policy is the number of requests allowed by the policy. Given a permissiveness bound, a policy is determined to be overly permissive if the permissiveness of the policy is greater than the permissiveness bound. For example, if the desired permissiveness bound is 1,000 (maximum of 1,000 distinct requests allowed), and the permissiveness of a given policy is 10,000, then the permissiveness of the policy exceeds the permissiveness bound and is in need of repair. While the permissiveness bound is a bound on the maximum number of requests allowed by the policy, it can also be used to interpret the maximum number of wild characters allowed within the policy; that is, the number of characters which are allowed to be unspecified in the policy.

Consider the policies in Figure 3 together with the following set of must-allow requests:

```
("s3:GetObject", "backend/logs/user00102")
("s3:GetObject", "backend/logs/user94319")
("s3:GetObject", "backend/logs/user22212")
("s3:GetObject", "backend/logs/user30100")
("s3:GetObject", "backend/logs/user49763")
```

Let us assume that the desired permissiveness bound is 5 wild characters, which corresponds to a maximum of $256^5 = 1.1 \times 10^{12}$ distinct requests which can be allowed by the policy. Note that the number of wild characters can be obtained by taking the log_{256} of the desired permissiveness (since each wild character corresponds to 256 possible characters). Additionally, assume only ASCII characters are allowed in the resource field, and the length of resources can be at most 30 characters long. The first policy (Figure 3(a)) has a permissiveness of 9.6×10^{52}, or 22 wild characters, which far exceeds the permissiveness bound. The second policy (Figure 3(b)) is a partially repaired version of the first policy, which further restricts the requests allowed by the policy. The permissiveness of this second policy is 2.0×10^{31}, or 13 wild characters which still exceeds the permissiveness bound. The third policy (Figure 3(c)) shows a fully repaired policy with a permissiveness of 1.1×10^{12}, or 5 wild characters, which does not exceed the permissiveness bound, and is thus repaired. In this case, note that the resource field in the policy `"Resource": "backend/logs/user?????"` limits the number of wildcard characters to 5, which meets the permissiveness bound.

3 MODELING ACCESS CONTROL POLICIES

In this section we present a semantic model for access control policies and the encoding of this semantic model as SMT formulas. The model and its encoding are expressive enough to capture complex policy specifications from cloud services; E.g, policies written in the AWS Identity and Access Management (IAM) policy language.

3.1 Policy Model

An access control policy specifies *who* can do *what* under *which* conditions. We define an access control model in which *declarative* policies field access requests from a dynamic environment, and all requests are initially denied.

We use the policy model from [10] where an access request is a tuple $(\delta, a, r, e) \in \Delta \times A \times R \times E$, Δ is the set of all possible principals making a request, R is the set of all possible resources which access is allowed or denied, A is the set of all possible actions, and E is the environment attributes involved in an access request.

An access control policy $\mathbb{P} = \{\rho_0, \rho_1, ...\rho_n\}$ consists of a set of rules ρ_i where each rule is defined as a partial function $\rho : \Delta \times A \times R \times E \hookrightarrow \{Allow, Deny\}$. The set of principals specified by a rule ρ is

$$\rho(\delta) = \{\delta \in \Delta : \exists a, r, e : (\delta, a, r, e) \in \rho\} \tag{1}$$

$\rho(a)$ for $a \in A$, $\rho(r)$ for $r \in R$, $\rho(e)$ for $e \in E$ are similarly defined.

Given a policy $\mathbb{P} = \{\rho_0, \rho_1, ...\rho_n\}$, a request (δ, a, r, e) is granted access if and only if

$$\exists \rho_i \in \mathbb{P} : \rho_i(\delta, a, r, e) = Allow \wedge \nexists \rho_j \in \mathbb{P} : \rho_j(\delta, a, r, e) = Deny$$

The policy grants access if and only if the request is allowed by a rule in the policy and is not revoked by any other rule in the policy. If a request is allowed by one rule and denied by another rule, the request is denied, i.e., the explicit denies overrules explicit allows. The set of allow rules and deny rules for $\mathbb{P}$ are defined as:

$$\mathbb{P}_{Allow} = \{\rho_i \in \mathbb{P} : (\delta_i, a_i, r_i, e_i) \in \rho_i \wedge \rho_i(\delta_i, a_i, r_i, e_i) = Allow\} \tag{2}$$

$$\mathbb{P}_{Deny} = \{\rho_j \in \mathbb{P} : (\delta_j, a_j, r_j, e_j) \in \rho_j \wedge \rho_j(\delta_j, a_j, r_j, e_j) = Deny\} \tag{3}$$

Given a policy $\mathbb{P}$, the requests allowed by the policy are those in which a policy rule grants the access through an *Allow* effect and is not revoked by any policy rule with a *Deny* effect:

$$\begin{aligned} \textsc{Allow}(\mathbb{P}) = \{(\delta, a, r, e) \in \Delta \times A \times R \times E \\ : \exists \rho_i \in \mathbb{P} : (\delta, a, r, e) \in \rho_i \wedge \rho_i(\delta, a, r, e) = Allow \\ \wedge \nexists \rho_j \in \mathbb{P} : (\delta, a, r, e) \in \rho_j \wedge \rho_j(\delta, a, r, e) = Deny\} \end{aligned} \tag{4}$$

The set of principals, resources, or actions allowed by a policy is

$$\textsc{Allow}(\mathbb{P}, \Delta) = \{\delta \in \Delta : (\delta, a, r, e) \in \textsc{Allow}(\mathbb{P})\} \tag{5}$$

$$\textsc{Allow}(\mathbb{P}, A) = \{a \in A : (\delta, a, r, e) \in \textsc{Allow}(\mathbb{P})\} \tag{6}$$

$$\textsc{Allow}(\mathbb{P}, R) = \{r \in R : (\delta, a, r, e) \in \textsc{Allow}(\mathbb{P})\} \tag{7}$$

Combining Policies. Recall that a policy $\mathbb{P}$ consists of a set of rules $\{\rho_0, ..., \rho_n\}$. Two policies $\mathbb{P}_1$ and $\mathbb{P}_2$ can be combined into a single policy $\mathbb{P}_3$ by combining the set of rules in $\mathbb{P}_1$ with the set of rules in $\mathbb{P}_2$ as $\mathbb{P}_3 = \mathbb{P}_1 \cup \mathbb{P}_2$. Based on the policy semantics we defined above, the allowed requests of $\mathbb{P}_3$ is the set of requests allowed by either $\mathbb{P}_1$ or $\mathbb{P}_2$ that are not denied by $\mathbb{P}_1$ and not denied by $\mathbb{P}_2$.

3.2 Symbolic Encoding of Policies

Access control policies can be translated to SMT formulas in order to enable symbolic analysis using constraint solvers [10]. The set of possible requests are encoded by introducing variables $\{\delta_{smt} \in \Delta, r_{smt} \in R, a_{smt} \in A, e_{smt} \in E\}$ in the generated SMT formula. The SMT encoding of a policy $\mathbb{P}$ is given by $[\![\mathbb{P}]\!]$ and represents the set of requests allowed by $\mathbb{P}$:

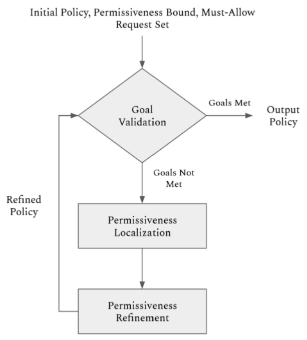
Initial Policy, Permissiveness Bound, Must-Allow Request Set
Goal Validation
Goals Met
Output Policy
Goals Not Met
Refined Policy
Permissiveness Localization
Permissiveness Refinement

Figure 4: Flow of repair algorithm. The inputs are Initial Policy, Permissiveness Bound, Must-Allow Request Set

$$[\![\mathbb{P}]\!] = \Big(\bigvee_{\rho \in \mathbb{P}_{Allow}} [\![\rho]\!]\Big) \bigwedge \neg\Big(\bigvee_{\rho \in \mathbb{P}_{Deny}} [\![\rho]\!]\Big) \tag{8}$$

$$\begin{aligned} [\![\rho]\!] = \Big(\bigvee_{\delta \in \rho(\delta)} \delta_{smt} = \delta\Big) \bigwedge \Big(\bigvee_{a \in \rho(a)} a_{smt} = a\Big) \bigwedge \\ \Big(\bigvee_{r \in \rho(r)} r_{smt} = r\Big) \bigwedge \Big(\bigvee_{e \in \rho(e)} e_{smt} = e\Big) \end{aligned} \tag{9}$$

Policy rules are encoded as values for sets of (δ, a, r, e), where each value set potentially grants or revokes permissions. Satisfying solutions to $[\![\mathbb{P}]\!]$ correspond to requests allowed by the policy, i.e.,

$$\textsc{Allow}(\mathbb{P}) = \{(\delta, a, r, e) : (\delta, a, r, e) \models [\![\mathbb{P}]\!]\} \tag{10}$$

4 QUANTITATIVE POLICY REPAIR

Recall that policy repair has three inputs: 1) a permissiveness bound, 2) a set of must-allow requests, and 3) a policy to be repaired. The goal is to create a revised (repaired) version of the input policy in which all must-allow requests are allowed *and* the permissiveness bound is not exceeded.

Our policy repair algorithm consists of three main stages: (1) Goal Validation, (2) Permissiveness Localization, and (3) Permissiveness Refinement. Figure 4 shows the overall flow of the repair algorithm. Algorithm 1 is the core repair algorithm corresponding to the flowchart shown in Figure 4. Given a policy (consisting of one or more rules), a permissiveness bound, and set of requests, the repair algorithm first checks if the permissiveness goals are met using Goal Validation. If they are met, then the algorithm stops and returns the policy. Otherwise, it finds the most permissive elements of the policy through Permissiveness Localization, then reduces permissiveness and refines the policy elements through Permissiveness Refinement. The algorithm then goes back to Goal Validation and repeats the process until the policy is successfully repaired meeting the permissiveness constraints. In the following sections, we will discuss the algorithms corresponding to each of the stages.

Since our repair approach uses a greedy strategy to quantitatively repair overly permissive policies, it is not guaranteed to produce an

Algorithm 1 PolicyRepair

Input: Policy $\mathbb{P}$, Permissiveness bound η, must-allow requests Q, length threshold α, depth threshold ω, refinement threshold ϵ, map M
Output: Repaired Policy

```
1: $\mathbb{P}' = \mathbb{P}$
2: $\eta_r$ = GetPermissiveness($\mathbb{P}'$)
3: while $(\eta_r > \eta) \wedge$ HasUnrefinedResources($M$) do
4:     $(\rho, a_\rho, r_\rho)$ = Localize($\mathbb{P}', M$)
5:     $\rho^*$ = ReduceRule($\rho, a_\rho, r_\rho$)
6:     $\mathbb{P}^* = (\mathbb{P}' \setminus \{\rho\}) \cup \{\rho^*\}$
7:     $Q^*$ = ValidateRequests($\mathbb{P}^*, Q$)
8:     if $Q^* \neq \emptyset$ then
9:         $R^*$ = GenerateResourceCharacterization($Q^*, \alpha, \omega$)
10:        $\rho_{\text{refined}}$ = GenerateRefinedRule($\rho, a_\rho, R^*$)
11:        $\mathbb{P}_{\text{refined}} = (\mathbb{P}^* \setminus \{\rho^*\}) \cup \{\rho_{\text{refined}}\}$
12:        if GetPermissiveness($\mathbb{P}_{\text{refined}}$) $\geq \eta - \epsilon$ then
13:            MarkRuleResourceAsRefined($M, \rho, r_\rho$)
14:        else $\mathbb{P}' = \mathbb{P}_{\text{refined}}$
15:        end if
16:    end if
17:    $\eta_r$ = GetPermissiveness($\mathbb{P}'$)
18: end while
19: if $\eta_r > \eta$ then $\mathbb{P}'$ = EnumerateRequests($\mathbb{P}', Q$)
20: end if
21: return $\mathbb{P}'$
```

Algorithm 2 ValidateRequests

Input: Policy $\mathbb{P}$, Request set $Q \subseteq \Delta \times A \times R \times E$
Output: Requests not allowed by policy $\mathbb{P}$

```
1: $Q_{\text{allowed}} = \emptyset$
2: $[\![\mathbb{P}]\!]$ = Encode($\mathbb{P}$)
3: for $(\delta, a, r, e) \in Q$ do
4:     if $(\delta, a, r, e) \models [\![\mathbb{P}]\!]$ then $Q_{\text{allowed}} = Q_{\text{allowed}} \cup \{(\delta, a, r, e)\}$
5:     end if
6: end for
7: return $Q \setminus Q_{\text{allowed}}$
```

Algorithm 3 Localize

Input: Policy $\mathbb{P}$, map M
Output: Most permissive rule and elements in policy

```
1: $\rho_{\max} = \rho_{\text{empty}}$
2: $(a_{\max}, r_{\max}) = ()$
3: $\eta_{\max} = 0$
4: for $\rho \in \mathbb{P}_{Allow}$ do
5:     if IsRuleRefined($M, \rho$) then continue
6:     end if
7:     $\eta$ = GetPermissiveness($\{\rho\}$)
8:     if $\eta > \eta_{max}$ then
9:         $\eta_{\max} = \eta$
10:        $\rho_{\max} = \rho$
11:    end if
12: end for
13: $\eta_{max} = 0$
14: for $(a_i, r_i) \in \rho_{max}(a) \times \rho_{max}(r)$ do
15:    if IsResourceRefined($M, r_i$) then continue
16:    end if
17:    $\rho$ = CreateRule($(\rho_{\max}(\delta), a_i, r_i, \rho_{\max}(e)), Allow$)
18:    $\eta$ = GetPermissiveness($\{\rho\}$)
19:    if $\eta > \eta_{max}$ then
20:        $\eta_{\max} = \eta$
21:        $(a_{max}, r_{max}) = (a_i, r_i)$
22:    end if
23: end for
24: return $(\rho_{\max}, a_{\max}, r_{max})$
```

optimum repair. However, we believe that a greedy repair strategy like ours that focuses on most permissive elements of the policy first is a reasonable and practical approach.

Algorithm 4 GenerateResourceCharacterization

Input: Must-allow requests $Q^* \subseteq Q$, length threshold α, depth threshold ω
Output: List of resources characterizing set resources from Q^*

```
1: $A_R = \emptyset$
2: $R_{Q^*}$ = GetResourcesFromRequests($Q^*$)
3: for $r \in R_{Q^*}$ do
4:     $A_r$ = ConstructDFA($r$)
5:     $A_R = A_R \cup A_r$
6: end for
7: $reg$ = GetRegexFromDFA($A_R$)
8: $reg^*$ = GeneralizeRegex($reg, \alpha, \omega, 0$)
9: $R_{reg^*}$ = EnumerateRegex($reg^*$)
10: return $R_{reg^*}$
```

Algorithm 5 GeneralizeRegex

Input: Regular expression reg, length threshold α, depth threshold ω, current depth d
Output: Generalization of regular expression reg

```
1: if $reg \equiv (reg_1 \mid reg_2)$ then
2:     $reg'_1$ = GeneralizeRegex($reg_1, \alpha, \omega, d+1$)
3:     $reg'_2$ = GeneralizeRegex($reg_2, \alpha, \omega, d+1$)
4:     if $(reg'_1 \in \Sigma^*) \wedge (reg'_2 \in \Sigma^*)$ then
5:         $l_{reg'_1}$ = Length($reg'_1$)
6:         $l_{reg'_2}$ = Length($reg'_2$)
7:         if $(l_{reg'_1} = l_{reg'_2}) \wedge (l_{reg'_1} <= \alpha)$ then
8:             return MakeRegex($?, l_{reg'_1}$)          ▷ '?' is regex for any character
9:         end if
10:    end if
11:    if $(d \geq \omega) \vee (reg'_1 \equiv \Sigma^*) \vee (reg'_2 \equiv \Sigma^*)$ then return $\Sigma^*$
12:    else return $(reg'_1 \mid reg'_2)$
13:    end if
14: else if $reg \equiv (reg_1 \cdot reg_2)$ then
15:    $reg'_1$ = GeneralizeRegex($reg_1, \alpha, \omega, d$)
16:    $reg'_2$ = GeneralizeRegex($reg_2, \alpha, \omega, d$)
17:    return $reg'_1 \cdot reg'_2$          ▷ '·' is regex concatenation
18: else return $reg$
19: end if
```

4.1 Repair Goal Validation

Recall that the main goal of policy repair is to reduce the permissiveness of the given policy to meet the given permissiveness bound while preserving the set of must-allow requests. Validating that the repair goal is reached requires two steps: (1) quantitatively assessing that the permissiveness of the repaired policy is within the given permissiveness bound, and (2) verifying that the given set of must-allow requests are allowed by the repaired policy. When both of these goal validation steps are achieved, the repair algorithm stops and we return the repaired policy. Note that it may not be possible to achieve the permissiveness bound without changing the policy to only allow the requests that are in the must-allow set. In such a scenario we generate a policy that corresponds to explicit enumeration of the requests in the must-allow set.

In cases where permissiveness bound cannot be reached without enumeration of the must-allow set, our approach uses a stopping condition where only rules that have not been previously refined (from the permissiveness refinement stage) are eligible for refinement; the repair algorithm stops if there are no rules left to refine, regardless of whether the permissiveness goal has been reached.

To simplify the presentation of our policy repair algorithm, we assume that the permissiveness level required by the must-allow set is not more than the input permissiveness bound (which would correspond to an unsatisfiable set of permissiveness constraints), and furthermore, we assume that the initial policy does allow all the requests in the must-allow set. We can easily get rid of these assumptions with extra checks.

The permissiveness goal is checked on lines 2 and 3 in the repair Algorithm 1 through the GetPermissiveness function. A policy $\mathbb{P}$ is first encoded into an SMT formula $[\![\mathbb{P}]\!]$ then sent to a model counter which returns number of satisfying solutions to $[\![\mathbb{P}]\!]$, which corresponds to the number of requests allowed by policy $\mathbb{P}$. Recall that the number of requests allowed by $\mathbb{P}$ corresponds to the permissiveness of $\mathbb{P}$. If the permissiveness is less than the bound, then the permissiveness goal has been reached and the algorithm returns the current policy. Otherwise, it gets in the while loop starting in line 3 in order to modify the current policy to reduce its permissiveness.

Algorithm 2 shows how the set of must-allow requests Q are checked against a policy $\mathbb{P}$. For each request (δ, a, r, e) in the must-allow set, we have to determine if $(\delta, a, r, e) \models [\![\mathbb{P}]\!]$, i.e., does $\mathbb{P}$ allow (δ, a, r, e)? This is done by generating the SMT formula $[\![(\delta, a, r, e)]\!] \wedge [\![\mathbb{P}]\!]$ and checking if it is satisfiable using an SMT solver. Note that $[\![(\delta, a, r, e)]\!]$ corresponds to SMT encoding of the request (δ, a, r, e) and $[\![\mathbb{P}]\!]$ is the SMT encoding of all the requests allowed by $\mathbb{P}$. So, if SMT solver reports that $[\![(\delta, a, r, e)]\!] \wedge [\![\mathbb{P}]\!]$ is satisfiable, then we know that the request (δ, a, r, e) is among the requests allowed by $\mathbb{P}$. If the SMT solver reports that it is not satisfiable, then we know that the request (δ, a, r, e) is not allowed by $\mathbb{P}$. By encoding requests and policies as SMT formulas, we can implement the goal validation step using an SMT solver, and without requiring access to an access control policy evaluation engine.

Algorithm 2 accumulates the requests in Q that are allowed by $\mathbb{P}$ in the set Q_{allowed}. At the end it returns the set difference $Q \setminus Q_{\text{allowed}}$, i.e., the set of requests in Q that are *not* allowed by $\mathbb{P}$. These requests are used in the permissiveness refinement step.

4.2 Permissiveness Localization

We use a greedy strategy in repairing the permissiveness of a policy. We quantitatively assess permissiveness by first finding the most permissive rule in the policy, then finding the most permissive elements within the rule. This is done using calls to a model counter.

Permissiveness Analysis. Recall from Section 3 that Allow($\mathbb{P}$) is the set of all requests allowed by $\mathbb{P}$. It follows then that $|$Allow$(\mathbb{P})|$ is the number of such requests. Following the work from [10], the permissiveness of $\mathbb{P}$ is the number of requests allowed by $\mathbb{P}$, which corresponds to the number of solutions to the formula encoding $\mathbb{P}$, which is $[\![\mathbb{P}]\!]$. I.e., $|$Allow$(\mathbb{P})| = |[\![\mathbb{P}]\!]|$. Thus, a lower permissiveness corresponds to a lower number of allowed requests, while a higher permissiveness corresponds to a higher number of allowed requests. Note that, although satisfiability of $[\![\mathbb{P}]\!]$ can be computed using a constraint solver, computing cardinality of $[\![\mathbb{P}]\!]$, i.e, computing $|[\![\mathbb{P}]\!]|$, requires a model counting constraint solver.

Permissiveness Localization. Similar to *fault localization* techniques in traditional repair algorithms, we introduce the notion of *permissiveness localization* for policy repair to find the most permissive sections of a policy. Our permissiveness localization technique consists of a two-step process: (1) a *course-grained approach* which first finds the most permissive rules in a policy, and (2) a *fine-grained approach* is used to find the most permissive elements within each rule. In the *course-grained approach* each rule is analyzed independently of other rules within the policy: each rule $\rho_i \in \mathbb{P}$ is treated as an independent policy $\mathbb{P}_i = \{\rho_i\}$. The permissiveness of each $\mathbb{P}_i$ is assessed using a model counter, where the most permissive rule in $\mathbb{P}$ corresponds to the most permissive policy $\mathbb{P}_i$. A rule contains principals, actions, resources, and environment conditions. In order to better analyze the permissive elements of the most permissive rule, we use a *fine-grained* approach to determine the greatest source of permissiveness. More specifically, we analyze the actions and resources within the rule, as in our observations these tend to be the most permissive elements. Once this is done, the repair algorithm refines the permissiveness of the rule and its elements.

Algorithm 3 shows the how the repair is localized. First, in lines 4-12 the most permissive rule is found by iterating through the allow rules (those that allow requests) in the policy. Only rules which contain unrefined resources are considered; additionally, we do not consider deny rules (those that deny requests) as by definition deny rules cannot increase permissiveness. We keep track of which parts of a policy is already refined by using a map M.

The GetPermissiveness function encodes the given policy as a SMT formula using the techniques in Section 3 and calls the model counter on the formula. The GetPermissiveness function is called on a policy consisting only of the given rule. Next, on lines 14-23 the most permissive action, resource pairs are located within the rule. This is done by iterating over all action, resource pairs and creating a new rule where the action, resource pair is allowed with any combination of the principals and environment attributes specified in the most permissive rule. Note that, as before, only unrefined resources are considered. The permissiveness of the newly created rule is calculated using the GetPermissiveness function. Once found, the most permissive rule and its respective action, resource pair is returned. Note that Algorithm 3 involves numerous calls to a model counter through the GetPermissiveness function, and calls to a model counter can be expensive. This is a concern that we later discuss while presenting our implementation and experiments.

4.3 Permissiveness Refinement

Once the most permissive rule and elements in the rule are found using permissiveness localization, the rule is modified to reduce permissiveness. However reducing permissiveness has the possible effect that some requests in the set of must-allow requests are now not allowed. In this situation, the denied requests are analyzed and the rule is then refined using resource characterization and generalization techniques so that all must-allow requests are allowed. Algorithm 1 shows how a rule is reduced and refined, while Algorithm 4 and Algorithm 5 show how the resource characterization is generated from the denied requests.

Permissiveness Reduction. Within Algorithm 1, once the most permissive rule and its permissive elements are located using Algorithm 3 on line 4, on line 5 the ReduceRule function modifies the rule so that permissiveness is reduced. Our approach for reducing permissiveness greedily removes the most permissive element of the most permissive rule. The rule is only modified so that the permissive action and resource pair is removed. On line 6, a new policy is created by removing the permissive rule from the original policy and replacing it with the reduced rule from line 5.

While the permissiveness of the rule is clearly reduced using this approach, a clear consequence is that some requests (possibly from the set of must-allow requests) that were previously allowed are now denied. This is an intentional consequence of our approach. It allows us to remove redundant elements of a policy while refining the rule (as we discuss below). The goal is to generate a possibly less permissive characterization of resources while keeping the must-allow requests still valid.

Permissiveness Refinement. Lines 7-17 in Algorithm 1 details how permissiveness is refined through the construction of a new policy. In the case that the permissiveness reduction results in the set of must-allow requests being invalidated, we must refine the permissiveness in order to fix the set of must-allow requests. On line 7, using Algorithm 2 we determine which requests from the set of must-allow requests are denied in the new policy. If the set of must-allow requests are still valid, the current repair iteration ends and the next iteration starts with the modified policy as the policy to be further repaired. Otherwise, the modified policy must first be repaired so that the set of must-allow requests are valid. Lines 9-11 show how this is done. We first generate a characterization of resources from the invalid requests in the must-allow request set. This is done by extracting a regular expression from the finite-state automaton by state elimination [12]. Once the characterization is obtained, the new resources are added into the rule through the GenerateRefinedRule function which generates a new rule using the newly refined resource and the other existing elements within the rule. However, it can be the case that the newly refined rule does not decrease permissiveness, either at all or by an appreciable amount. If the permissiveness of the refined policy does not appreciably decrease (lines 12-15), the current repair is rolled back and the resource within the rule is marked as not eligible for refinement.

Resource Characterization from Invalid Requests. To finish the permissiveness reduction and refinement step, the modified policy must be further refined so that the set of must-allow requests is valid. Trivially, this can be done by enumerating the invalid requests and adding a new rule to the policy which allows only that specific requests. However this does not generalize for requests not in the must-allow set but were intended to be allowed, and can make the policy more complicated in the case that the must-allow set is large. Thus, we aim to generate a characterization of the invalid requests, but more specifically the resources in the requests, which can be added to the modified rule. Ideally, this characterization will increase permissiveness to fix the invalid requests, but still remain less permissive than previously. To do so, we generate a regular expression characterizing the set of requests.

Algorithm 4 shows our regular expression and automata-based approach for resource characterization. The algorithm works by constructing a deterministic finite-state automaton (DFA) for each resource and then taking the automata union of all such DFAs (lines 3-6). Each DFA constructed for a resource (line 4) is a DFA that accepts only that resource, which is a constant. Thus, the union of all such DFAs is a DFA with no loops. We then use the state elimination algorithm [12] to obtain a regular expression characterizing the set of resources. It is well known that this regular extraction algorithm can produce arbitrarily complex regular expressions which are often not useful in practice. This is mainly due to the presence of loops within the DFA, and since our DFAs contain no loops, the resulting regular expression contains only concatenation and unions.

Consider the resources from an example must-allow request set: `bucket/users/client155`, `bucket/users/client115`, `bucket/users/client055`, `bucket/users/client200`, `bucket/logs/client544`, `bucket/logs/client333`, `bucket/logs/client12`, `bucket/logs/client411`

Figure 5 shows the DFA constructed from the union of these requests, and the initial regular expression extracted from the DFA. The extracted regular expression however is an enumeration of the input resources using disjunctions, and must be generalized.

The recursive GeneralizeRegex algorithm (Algorithm 5) takes the extracted regular expression and transforms the regular expression to a more general regular expression which specifies a broader list of resources. The algorithm works to eliminate some disjunctions in a depth-first manner by replacing them with anychars ('?') and wildcards ('*') when possible. The length threshold controls when strings of the same length should be collapsed into anychar symbols: e.g., if the length threshold is 3, then "(123|456)" will be simplified to "???", while "(1234|5678)" will remain the same. The depth threshold controls when nested disjunctions get simplified into the wildcard (anystring) character: the greater the threshold, the deeper the nesting of disjunctions is allowed. Once the generalized regular expression is obtained, the refined resources are gathered by enumerating the leftover disjunctions within the general regular expression. Using a length threshold of 3, depth threshold of 2, we obtain a more general, more permissive regular expression:
`bucket/(logs/client*|users/client???)`
Note that different values for the thresholds yield different regular expressions. For example, length threshold of 3, depth threshold of 4 yields the less general, less permissive regular expression:
`bucket/(logs/client(((12|333)|411)|544)|users/client???)`

5 POLICY REPAIR FOR THE CLOUD

Currently, our policy repair approach works on the policy model we introduced in Section 3. This policy model abstracts away the implementation and intricacies of modern policy languages used in the cloud. In this section, we show how our policy model can be applied to one of the most popular policy languages for the cloud, that of Amazon Web Services (AWS), and we demonstrate that our approach repairs such policies.

5.1 AWS Policy Language

In the AWS policy language, a policy consists of a list of statements which either allow or deny access. A statement consists of *Principal*, an *Effect*, *Action*, *Resource*, and *Condition*, where:
• *Principal* is a list of users or other entities specifying who or what is requesting access.
• *Effect* ∈ {*Allow*, *Deny*} specifying allows or denies access.
• *Action* is a list of actions specifying what operations on the resources are being requested.
• *Resource* is a list of resources specifying what is being accessed.
• *Condition* is a list of conditions specifying additional constraints on how access is governed.

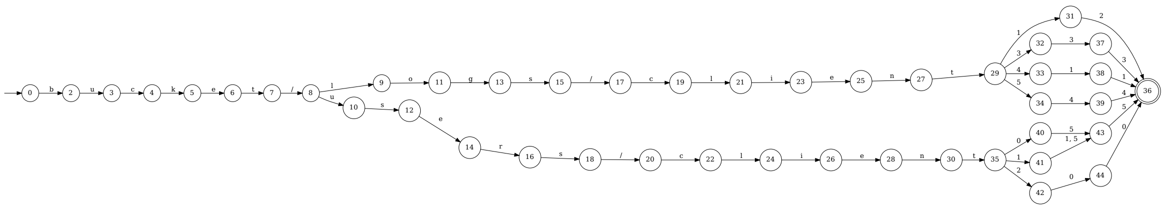

Figure 5: DFA example. Resulting regex: `bucket/(logs/client(((12|333)|411)|544)|users/client((05|1(5|1))5|200))`

```
{"Statement": [
    {"Effect": "Allow",
    "Principal": "foo",
    "NotAction": "bar",
    "Resource": "baz",
    "Condition": {"StringNotEquals": {"key": "value"}}}]}
```

```
{"Statement": [
    {"Effect": "Allow",
    "Principal": "foo",
    "Action": "*",
    "Resource": "baz",
    "Condition": {"StringLike": {"key": "*"}}},
    {"Effect": "Deny",
    "Principal": "foo",
    "Action": "*",
    "Resource": "baz",
    "Condition": {"StringEquals": {"key": "value"}}},
    {"Effect": "Deny",
    "Principal": "foo",
    "Action": "bar",
    "Resource": "baz",
    "Condition": {"StringNotEquals": {"key": "value"}}}]}
```

Figure 6: Original AWS IAM policy with one statement with *NotAction* and *StringNotEquals* condition operator (top, (a)); Transformed policy with three statements (bottom, (b)).

While *Principal, Action, Resource, Condition* are all lists, *Condition* also contains a condition key and condition value corresponding to elements of the access request. For more information on the AWS policy language, we refer the reader to [15]. Each field or element in the policy are ASCII strings (aside from some condition keys and condition values), with two special characters: the wildcard '*' character, and the wild/anychar '?'character. The wildcard character represents any string; the wild/anychar character represents any character. This allows policy developers to specify sets of strings within elements using these two special characters. Given an access request and a policy, the policy allows the access request if and only if there is a statement in the policy which allows the request and there is no statement in the policy which denies the request.

Modeling AWS Policies. For each statement an AWS policy, we create a rule that captures the semantics of the statement. The principals, resources, actions, and effects map one-to-one from statements to rules, while the environment attributes captures the condition keys and values within a statement. Once the rules for the statements in the policy are created, we can encode the policy into an SMT formula using the techniques from Section 3. Thus, we can model AWS policies within our policy model framework.

5.2 Policy Transformations for Repair

Recall that our approach localizes permissiveness to the most permissive action, resource pair and then mutates it when possible. We cannot directly apply the approach to AWS IAM policies that may contain *NotPrincipal*s, *NotAction*s, *NotResource*s, and/or negative condition operators like *StringNotEquals* because such elements let policy developers specify the *complements* of allowed values. If we directly applied our approach, then removing policy elements would increase permissiveness, straying away from the repair goal. To avoid this and to avoid complicating the repair approach, we *transform* original policies, removing "negative" policy elements.

Algorithm 6 shows how an AWS statement ρ is transformed. In the algorithm, $\rho.keys$ refers to the *Principal*, *Action*, and *Resource* (or their negations) which exist in the statement. This is done in two passes. In the first pass on lines 4-8, the *NotPrincipal*s, *NotAction*s, and/or *NotResource*s are removed (there is no *NotCondition* in the AWS IAM policy language). This is not enough, because condition operators like *StringNotLike* may be used to specify complements of allowed condition values. In the second pass on lines 10-20, these negative condition operators are removed similarly. Figure 6 shows the transformation applied to an AWS IAM policy. Figure 6(a) shows the original policy, which has one statement with a *NotAction* element and a *StringNotEquals* condition operator. Figure 6(b) shows the transformed policy after both passes are done.

The transformation has three limitations: (1) We do not transform deny statements in the original policy. (2) We assume that the original policy does not have statements allowing requests that the newly added statements deny. Otherwise, the transformed policy is less permissive than the original policy because a statement denying a request overrules one allowing it. (3) We currently support the case-sensitive string condition operators only.

5.3 Determining Permissiveness Bounds

Our approach can be used to automatically reduce the permissiveness of policies while making sure that they allow what is necessary (based on the must-allow request set). Even without a desired permissiveness bound, our approach can be used to find a less permissive policy by using permissiveness of other policies as a bound or by giving a permissiveness bound that is less than the current permissiveness of a policy as we discuss below.

Inferring a Permissiveness Bound from Other Policies. When the permissiveness bound is not known, the permissiveness value of another policy can be used as the permissiveness bound. Assume that a policy $\mathbb{P}$ is given and the policy developer wants to determine if it is overly permissive, but the permissiveness bound is not

Algorithm 6 TransformStmtPrincipalActionResource

```
Input: Statement $\rho$
Output: Transformed statement(s) $\mathbb{P}$
1: $\mathbb{P} = \emptyset$
2: $K^- = \{k : k \in \rho.keys \cap$ "Not" $in\ k\}$
3: $\rho_{Allow} = \{$"Effect" : "Allow"$\}$
4: for $k \in \rho.keys$ do
5:     if $k \in K^-$ then $\rho_{Allow}[\text{Negation}(k)] =$ " $*$ "
6:     else $\rho_{Allow}[k] = \rho[k]$
7:     end if
8: end for
9: $\mathbb{P} = \mathbb{P} \cup \{\rho_{Allow}\}$
10: for $k' \in K^-$ do
11:     $\rho_{Deny} = \{$"Effect" : "Deny"$\}$
12:     for $k \in \rho.keys$ do
13:         if $k = k'$ then $\rho_{Deny}[\text{Negation}(k)] = \rho[k]$
14:         else if $k \in K^-$ then $\rho_{Deny}[\text{Negation}(k)] =$ " $*$ "
15:         else $\rho_{Deny}[k] = \rho[k]$
16:         end if
17:     end for
18:     $\mathbb{P} = \mathbb{P} \cup \{\rho_{Deny}\}$
19: end for
20: return $\mathbb{P}$
```

known or is difficult to determine. In this instance, assume that the policy developer has another policy $\mathbb{P}'$ which is known to be not overly permissive. Let $\eta_{\mathbb{P}'}$ be the permissiveness of $\mathbb{P}'$. Then, to determine if $\mathbb{P}$ is overly permissive, $\eta_{\mathbb{P}'}$ can be used as the desired permissiveness bound. If the permissiveness of $\mathbb{P}$ is greater than $\eta_{\mathbb{P}'}$ then $\mathbb{P}$ is overly permissive and should be repaired using our approach with the permissiveness bound being $\eta_{\mathbb{P}'}$. Note that this approach assumes that the policy developer has access to another policy $\mathbb{P}'$ whose permissiveness can be used as the permissiveness bound when repairing $\mathbb{P}$; for example, for a new role, a policy should be attached to the new role which has similar permissiveness to policies attached to other roles. If such a policy $\mathbb{P}'$ does not exist, then an iterative approach for reducing the permissiveness of a policy can be used as we discuss next.

Iteratively Decreasing Permissiveness. Consider when the permissiveness bound for a given policy $\mathbb{P}$ is not known but the policy developer wants to ensure that $\mathbb{P}$ is not overly permissive. That is, the policy developer wants to ensure that $\mathbb{P}$ does not allow more requests (permissions) than what is required. In this case, our repair algorithm can be used to iteratively reduce the permissiveness of $\mathbb{P}$

(1) Let $\eta_{\mathbb{P}}$ be the permissiveness of $\mathbb{P}$
(2) Set the permissiveness bound as $\eta_{\mathbb{P}'} = \eta_{\mathbb{P}} - \delta$
(3) Repair $\mathbb{P}$ using permissiveness bound $\eta_{\mathbb{P}'}$ to obtain a repaired policy $\mathbb{P}_r$
(4) If $\mathbb{P}_r$ has the desired permissiveness level, halt; otherwise go back to step (1) with $\mathbb{P} = \mathbb{P}_r$

where δ is a positive integer defining the step size which determines how much the permissiveness bound should decrease in each step. This can be continued until a repaired policy with a desired level of permissiveness is produced, or the approach cannot further repair the policy. In each step the permissiveness of $\mathbb{P}$ is decreased by δ.

6 EXPERIMENTS

In order to evaluate our repair algorithm, we consider the following research questions:
RQ1: Does the policy repair algorithm successfully find repairs for policies collected from user forums?
RQ2: How does the effectiveness of the algorithm change for varying permissiveness bounds?
RQ3: What factors contribute to the overall performance (execution time/iterations/calls) of the repair algorithm?
We discuss below the policy dataset we use in our approach, how we set up our experiments to answer the research questions, and the results of our experiments. For quantifying the permissiveness, we use the model counter ABC [2, 3]; for validating requests in the must-allow request set we use the SMT solver Z3 from Microsoft, and the quacky tool for translating policies into SMT formulas. [1]

6.1 Experimental Setup

AWS Policy Dataset. AWS offers over 200 services. Each service has its own actions and resource types that can be allowed or denied in access control policies. For our repair experiments, we used the policy dataset published in [10], which includes 43 real-world policies collected from using forums from the most popular AWS services, namely iam, s3, and ec2.

Permissiveness Bounds. Recall that the policy repair problem specifies a permissiveness bound. In general, this permissiveness bound relates to the number of requests allowed by the policy. In our repair algorithm, and in our experiments, we consider a more restrictive permissiveness bound definition in which the permissiveness is determined by the number of actions and requests allowed by a policy. The reason for this is that in the policies we have observed, the most permissive element is the resource element, and since the action and resource elements are tied very closely in the policy semantics (e.g., only S3 actions work on S3 resources) it makes sense to consider them together.

Because resources are strings, and strings can be infinitely long, we must bound the maximum length of allowed resources (otherwise the permissiveness of a policy is infinite due to wildcard characters). In our analysis, we bound the maximum length of any resource to be 100. Note that actions are also strings, but there are a finite number of actions (e.g., S3:GetObject is a valid action, S3:FooBar is not). Thus, the maximum number of actions allowed by a policy is the number of possible actions, which in practice is relatively small (a few hundred for the AWS services we consider). In our experiments, we use the action constraint encoding from [10] which maps constraints on actions into numeric range constraints to simplify the constraint formulas generated in our approach.

In our experiments, the permissiveness bound is given in terms of $\log_{256}$. Intuitively, since resources are strings where each character in the string can be one of 256 ASCII characters, this gives a measure of uncertainty regarding the number of unknown characters in the resource. For example, the resource "foo12" has a $\log_{256}$ permissiveness of 0 (all characters in the string are known) while the resource "foo??" has a $\log_{256}$ permissiveness of 2 (2 characters in the string are unknown) since '?' is a special character denoting any possible character. We bound the maximum length of strings at 100 so giving permissiveness bounds in terms of $\log_{256}$ gives a restriction on how many of characters of the resource be unknown. Note that while this is just an approximate measure (strings can be

[1]Our policy repair tool and policy and request datasets are publicly available at https://github.com/vlab-cs-ucsb/policy-repair

Table 1: Results for 43 total policies with length threshold of 2 and depth threshold of 3. Policies are repaired using varying permissiveness bounds (given as $\log_{256}$, interpreted as number of unknown characters allowed in a resource) .

	Permissiveness Bound						
	30	40	50	60	70	80	90
without enumeration	29	29	31	33	39	43	43
with enumeration	14	14	12	10	4	0	0
% without enumeration	67%	67%	72%	77%	91%	100%	100%

less than 100 characters) it nevertheless gives a useful measure for bounding the permissiveness of a policy.

Allowed Requests. We augmented the policy dataset we used with sets of allowed requests. We created requests containing only the action and resource field, as our repair approach is currently tailored for reducing permissiveness based on action and resources. Our methodlogy for sythensizing requests was to create requests which are likely to resemble requests created by actual users. For actions, we focus on the most common actions for the AWS services in our policies (such as S3:GetObject and EC2:RunInstances). For resources, we observed from the policy dataset and AWS online documentation that resources generally have the following structure:

$$resource = service\ .\ prefix\ .\ middle\ .\ suffix$$

The *service* section consists of AWS service, region, and account number. The *prefix* section corresponds to the resource type and is generally dependent on the action in question: e.g., the prefix for s3 resources generally corresponds to the bucket name. The *middle* consists of the intermediate directory structure (usually delimited using '/'). The *suffix* consists of the object, filename, or instance in question. Consider the following resource

arn:aws:s3:::mybucket/folder1/folder2/clients.txt

where the *service* is "*arn:aws:s3:::*", the *prefix* is "*mybucket/*", the *middle* is "*folder1/folder2*", and the *suffix* is "*clients.txt*". When synthesizing the requests, we observed that the *service* and *prefix* parts of the resource were specific to services for the particular policy, while the *middle* and *suffix* parts of the resource depended on the actions and service being used. For each policy, we constructed 10-20 requests using the base policy as reference, varying the relevant parts for each. An example request for S3 would be:

```
(S3:GetObject)
(arn:aws:s3:::bucket/production/user00000/status.log)
```

We ran all experiments on a machine with an Intel i5 3.5GHz X4 processor, 32GB DDR3 RAM, a Linux 4.4.0-198 64-bit kernel, Z3 v4.11.1, the latest build of ABC [2], and the latest release of QUACKY[3].

6.2 Results

To answer our research questions, we conducted a wide variety of experiments on 43 policies collected from user forums using our quantitative repair algorithm. We now discuss the results and how they answer the aforementioned research questions.

[2]https://github.com/vlab-cs-ucsb/ABC
[3]https://github.com/vlab-cs-ucsb/quacky

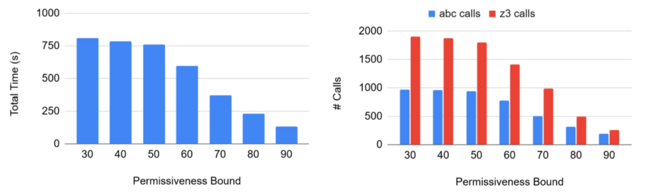

Figure 7: For all 43 policies and each permissiveness bound: total time taken (left (a)); total calls to ABC and Z3 (right (b)).

RQ1*: Does the policy repair algorithm successfully find repairs for policies collected from user forums?* Recall that the policy repair algorithm tries to find a repair meeting the permissiveness bound through goal validation, permissiveness localization, and permissiveness refinement, and if it cannot will begin enumerating requests and replacing elements of the policy with these requests. Our repair algorithm will always successfully find repairs (so long as the initial assumptions are met, see Section 4). Some of these repairs may require request enumeration, which is not ideal.

We ran the repair algorithm on the dataset of 43 policies with varying permissiveness bounds to determine if the repair algorithm could generate a repair without request enumeration and how often our repair was generated with request enumeration. Table 1 shows the results. The permissiveness bounds ranged from 30 to 90, meaning that the repair algorithm must generate a repaired policy with permissiveness less than the given bound. For each permissiveness bound, we used a length threshold (α) of 2, depth threshold (ω) of 3, and refinement threshold (ϵ) of 0.01.

For lower bounds, request enumeration was required to generate successful repairs, with 14 of the 43 (67%) repairs requiring request enumeration for bounds 30 and 40. As the permissiveness bound increases, the number of repairs generated which required request enumeration decreases. For permissiveness bounds of 80 and 90, 100% of the repairs generated by algorithm were generated without enumerating requests. Intuitively, this makes sense as a lower permissiveness bound requires the policy to more concretely specify the requests allowed by the policy; a higher permissiveness bound means that the policy can be more generalized in what is allowed.

RQ2*: How does the effectiveness of the algorithm change for varying permissiveness bounds?* As the results in Table 1 show, while the repair algorithm generates repairs for all policies for all given permissiveness bounds, lower permissiveness bounds required the repair algorithm to resort to enumerating requests. This means that while the permissiveness localization algorithm from Section 4 (Algorithm 3) was able to localize where the most permissive elements were, the permissiveness refinement algorithms (Algorithms 4,5) could not generate a resource characterization to reduce the permissiveness enough to meet the permissiveness bound. This could be due to the length (α) and depth (ω) threshold values used in Algorithm 5. Thus, we ran the repair algorithm again on the 43 policies, but this time for a single permissiveness bound but with different threshold values. Table 2 shows the results. We observed that, in general, the length and depth threshold values did not have

Table 2: Results for varying length (α) and depth (ω) thresholds for a single permissivness bound of 60 (i.e., $\log_{256}(perm) \leq 60$)

α, ω thresholds	Repaired without enum	Repaired with enum	Avg $\log_{256}$ Permissiveness	Total time (s)	# ABC calls	# Z3 calls
2,3	33 (77%)	10 (23%)	17	597.9	780	1415
2,5	43 (100%)	0 (0%)	11.7	452.4	550	1001
0,3	33 (77%)	10 (23%)	20.2	602.3	786	1423
2,3	33 (77%)	10 (23%)	17	597.9	780	1415
5,3	37 (86%)	6 (14%)	17.7	518	679	1310
10,3	37 (86%)	6 (14%)	16.6	525	682	1310
15,3	37 (86%)	6 (14%)	16.5	515	686	1292

an appreciable impact on the total time taken by the repair algorithm. However, we did observe that a higher depth threshold corresponded to more repairs not requiring explicit request enumeration. We believe this is because a higher depth threshold results in a less generalized, more enumerative regular expression characterization. Recall from Algorithm 5 that the depth threshold corresponds to the maximum level of nested disjunctions within a regular expression. When the level of nested disjunctions reaches the depth threshold, it gets "squashed", or generalized into a wildcard '*' (anystring) regular expression. Thus, while the repair algorithm with length threshold of 2 and depth threshold of 5 repaired all 43 policies without explicit enumeration, it is likely that regular expression characterizations generated in this case allowed more disjunctions, and thus were a more enumerative generalization.

***RQ3**: What factors contribute to the overall performance (execution time/iterations/calls) of the repair algorithm?* The repair algorithm utilizes a constraint solver (Z3) and model counter (ABC) for verifying the requests in the must-allow request set and for quantifying permissiveness. Both tools incur significant overhead in the process. Figure 7(a) shows the time taken for various permissiveness bounds, while Figure 7(b) shows the number of calls to Z3 and ABC for each permissiveness bounds. As the permissiveness bound is increased, the total time taken for repairing the 43 policies significantly decreases. Looking at Figure 7(b), the number of calls to both Z3 and ABC decrease in a similar fashion. Both the number of calls and total time were similar for the lowest few bounds. This may be due to the fact that those policies which required enumeration during the repair process for the bounds of 30, 40, and 50 are the ones which took more time to repair and more calls to Z3/ABC. For the depth and length thresholds, we did not notice a significant increase or decrease in time taken or calls to Z3/ABC when the thresholds were varied against a constant permissiveness threshold.

6.3 Threats to Validity

Requests in the must-allow request set may not be representative of the what should be allowed by the policy. We mitigate this threat as much as possible by synthesizing requests not randomly but instead based on the common structure of actions and resources we observed from both the policy dataset and AWS documention. In this way, the requests were not randomly generated but were generated such that they aligned with the user's intention regarding the kinds of requests that should be allowed by the policy. As the 43 policies did not have associated requests which should be allowed by the policy, this was the most straightforward approach for generating a must-allow request set.

7 RELATED WORK

There has been much research on access control policies [22–24] and access control policy languages [1, 16–18]. Early work in verification of access control policies exist [8, 14] and there has been some work using the Alloy Analyzer [25, 29].

Recently, there has been interest in the verification of access control policies using SAT/SMT solvers. In [4], the authors present Zelkova, a closed-source, proprietary tool that can compare AWS policies and tell if one is more permissive than the other. In [10] the authors introduce an approach for quantifying permissiveness of access control policies for AWS and Microsoft Azure and implement it in a tool called QUACKY. Our work uses the authors' notion of permissiveness for quantitative repair. Zelkova cannot quantitatively compare policies like [10] can, and Zelkova does not use policy comparisons to guide policy repair. In [5] the authors use Zelkova to determine if a policy is Trust Safe (i.e., blocks public access and does not allow untrusted requests). Both [10] and [4] draw on the approach in [13], which uses a SAT solver to check XACML policies; recent work has built on this but does not quantitatively analyze nor repair access control policies. Another tool is Margrave [11] which analyzes XACML policies using a multi-terminal decision diagrams. In later work [21], Margrave incorporates a SAT solver in the analysis of XACML policies to produce solutions to queries and enumerate the possible solutions. While quantitative in nature [10] showed that this type of enumerative approach does not scale for quantitave analysis of access control policies. In [6], the authors present Qlose, which uses a program repair approach based on quantitative objectives. In [27], the authors present an approach for repairing XACML policies by fault localization and mutation-based repair. We focus on policies and not programs, and our use of symbolic quantitative permissiveness analysis and our iterative repair generation approach differ from both of these prior approaches.

8 CONCLUSION

In this work we present a novel quantitative policy repair algorithm for repairing the permissiveness of access control policies for the cloud. Given a permissiveness bound and must-allow request set, our approach works by iteratively localizing the most permissive elements of the policy using quantitative analysis techniques and reducing and refining these elements using regular expression generalization techniques. Our experiments on 43 AWS IAM policies show that our repair algorithm successfully generates repairs for the given policies and does so in a reasonable amount of time. As future work, we plan to automate techniques we discussed for determining permissiveness bounds.

REFERENCES

[1] Jose L. Abad-Peiro, Hervé Debar, Thomas Schweinberger, and Peter Trommler. 1999. *PLAS — Policy Language for Authorizations.* Technical Report RZ 3126. IBM Research Division. http://citeseer.nj.nec.com/abad-peiro99plas.html

[2] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, Proceedings, Part I.* 255–272. https://doi.org/10.1007/978-3-319-21690-4_15

[3] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. 2018. Parameterized model counting for string and numeric constraints. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018.* 400–410.

[4] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachu, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design (FMCAD 2018), Austin, Texas, USA, October 30 - November 2, 2018.* 1–9.

[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] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 383–401.

[7] djleak [n.d.]. Cloud Leak: WSJ Parent Company Dow Jones Exposed Customer Data. https://www.upguard.com/breaches/cloud-leak-dow-jones.

[8] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2006. Specifying and Reasoning About Dynamic Access-Control Policies. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4130)*, Ulrich Furbach and Natarajan Shankar (Eds.). Springer, 632–646. https://doi.org/10.1007/11814771_51

[9] William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, and Tevfik Bultan. 2022. Quacky: Quantitative Access Control Permissiveness Analyzer. In *ASE Tool Paper.*

[10] 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 (ICSE 2022).*

[11] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. 2005. Verification and Change-Impact Analysis of Access-Control Policies. In *Proceedings of the 27th International Conference on Software Engineering.* St. Louis, Missouri, 196–205.

[12] J.E. Hopcroft and J.D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley.

[13] Graham Hughes and Tevfik Bultan. 2007. Automated Verification of XACML Policies Using a SAT Solver. In *Proc. Workshop on Web Quality, Verification and Validation (WQVV).* 378–392.

[14] Graham Hughes and Tevfik Bultan. 2008. Automated verification of access control policies using a SAT solver. *STTT* 10, 6 (2008), 503–520. https://doi.org/10.1007/s10009-008-0087-9

[15] iamreference 2022. IAM JSON policy reference. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies.html.

[16] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. 2001. Flexible support for multiple access control policies. *ACM Transactions on Database Systems* 26, 2 (2001), 214–260. http://doi.acm.org/10.1145/383891.383894

[17] S. Jajodia, P. Samarati, and V. S. Subrahmanian. 1997. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy.* IEEE Press, Oakland, CA, USA, 31–42. http://citeseer.nj.nec.com/jajodia97logical.html

[18] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Eliza Bertino. 1997. A unified framework for enforcing multiple access control policies. In *SIGMOD'97.* Tucson, AZ, 474–485. http://citeseer.nj.nec.com/jajodia97unified.html

[19] Kotaro Kataoka, Saurabh Gangwar, and Prashanth Podili. 2018. Trust list: Internet-wide and distributed IoT traffic management using blockchain and SDN. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT).* 296–301. https://doi.org/10.1109/WF-IoT.2018.8355139

[20] Leo A. Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *2010 IEEE Symposium on Security and Privacy.* 481–496. https://doi.org/10.1109/SP.2010.36

[21] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis. In *Proceedings of the 24th International Conference on Large Installation System Administration* (San Jose, CA) *(LISA'10).* USENIX Association, USA, 1–8.

[22] Pierangela Samarati and Sabrina De Capitani di Vimercati. 2001. *Foundations of Security Analysis and Design.* Springer Verlag, Chapter 3, 137–196.

[23] Ravi Sandhu and Pierangela Samarati. 1996. Authentication, access control, and audit. *Comput. Surveys* 28, 1 (1996), 241–243. http://doi.acm.org/10.1145/234313.234412

[24] Ravi S. Sandhu and Pierangela Samarati. 1994. Access Control: Principles and Practice. *IEEE Communications Magazine* 32, 9 (1994 1994), 40–48. http://citeseer.nj.nec.com/article/sandhu94access.html

[25] Andreas Schaad and Jonathan Moffet. 2002. A Lightweight Approach to Specification and Analysis of Role-based Access Control Extensions. In *7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002).*

[26] verizonleak [n.d.]. 14 MEEELLION Verizon subscribers' details leak from crappily configured AWS S3 data store. https://www.theregister.co.uk/2017/07/12/14m_verizon_customers_details_out/.

[27] Dianxiang Xu and Shuai Peng. 2014. Towards automatic repair of access control policies. In *Proceedings of the 14th Annual Conference on Privacy, Security and Trust, PST (PST 2014).*

[28] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and P. Mohapatra. 2006. FIREMAN: a toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S&P'06).* 15 pp.–213. https://doi.org/10.1109/SP.2006.16

[29] John Zao, Hoetech Wee, Jonathan Chu, and Daniel Jackson. 2003. RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis. In *Proceedings of the eighth ACM symposium on Access Control Models and Technologies.*

Received 2023-02-16; accepted 2023-05-03