Quantifying Permissiveness of Access Control Policies[*]

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

Albert Li
University of California Santa Barbara
Santa Barbara, CA, USA
albert_li@cs.ucsb.edu

Emily O'Mahony
University of California Santa Barbara
Santa Barbara, CA, USA
emilyomahony@cs.ucsb.edu

Benjamin Prince
University of California Santa Barbara
Santa Barbara, CA, USA
benjaminprince@cs.ucsb.edu

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

ABSTRACT

Due to ubiquitous use of software services, protecting the confidentiality of private information stored in compute clouds is becoming an increasingly critical problem. Although access control specification languages and libraries provide mechanisms for protecting confidentiality of information, without verification and validation techniques that can assist developers in writing policies, complex policy specifications are likely to have errors that can lead to unintended and unauthorized access to data, possibly with disastrous consequences. In this paper, we present a quantitative and differential policy analysis framework that not only identifies if one policy is more permissive than another policy, but also quantifies the relative permissiveness of access control policies. We quantify permissiveness of policies using a model counting constraint solver. We present a heuristic that transforms constraints extracted from access control policies and significantly improves the model counting performance. We demonstrate the effectiveness of our approach by applying it to policies written in Amazon's AWS Identity and Access Management (IAM) policy language and Microsoft's Azure policy language.

CCS CONCEPTS

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

ACM Reference Format:
William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, and Tevfik Bultan. 2022. Quantifying Permissiveness of Access Control Policies. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510233

1 INTRODUCTION

Modern software services run on compute clouds. Among the most popular cloud service providers are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), each of which lets customers secure their services by writing *access control policies*. Access control policies specify rules that allow authorized access while denying unauthorized access to cloud data. Policies can be written using many access control specification languages, like the AWS Identity and Access Management (IAM) [40] language or the eXtensible Access Control Markup Language (XACML) [64]. In contrast, libraries such as CanCan [22] and Pundit [58] provide support for specification of policies at the implementation level. By themselves, these are useful languages and libraries; however, without verification and validation techniques that can assist in writing policies, policy specifications are likely to have errors that can lead to unintended and unauthorized access to data. In fact, incorrect specification of access control policies in cloud storage services has resulted in the exposure of millions of customers' data to the public. For example, it was reported that [28] data records for more than 2 million Dow Jones & Co. customers were exposed due to an access control error. Exposed data included names, addresses, account information, email addresses, and last four digits of credit card numbers of subscribers. The exposed data was in a publicly accessible AWS Simple Storage Service (s3) bucket. This is a disastrous error in the policy specification for cloud storage buckets. A similar error resulted data exposure of 50 thousand Australian employees that included full names, passwords, salaries, IDs, phone numbers, and credit card data [2]. Yet another error exposed the account records of 14 million Verizon customers [63]. A vulnerability in Microsoft's Azure Cosmos DB service [10] allowed public access to accounts and databases of thousands of customers.

These examples highlight the urgent need to develop techniques to protect cloud data. Automatically finding access control issues would prevent exposure of private data, protecting the privacy of millions of people. Hence, it is necessary to develop automated verification techniques that can analyze access control policies for compute clouds. In order to check for correctness of a policy, it is necessary to have a specification of correctness properties, but writing correctness properties manually can be challenging and time consuming. Moreover, writing expected properties of the policy is error-prone. Hence, when an inconsistency between a property specification and a policy is identified, it does not necessarily mean that the policy has an error; the property specification itself could be erroneous. A differential policy analysis approach removes the need to manually specify policy properties; instead, it compares different policies and identifies inconsistencies among them. Basic policies can be compared to a complex policy to verify that the

[*]This material is based on research supported by an Amazon Research Award and by NSF under Grants CCF-2008660, CCF-1901098, CCF-1817242

latter does not have unintended consequences. For example, we may want to verify that a complex policy specification is not more permissive than a simple policy that specifies common sense access rules. Moreover, differential policy analysis techniques can identify differences between different versions of a policy. When a policy specification is modified, it would be worthwhile to know how the permissiveness of the policy has changed. However, a binary answer to a question that compares two policies may be insufficient. For example, it may not suffice that we know *if* one policy is more permissive than another. We may want to know *how much* more permissive a policy is than another, i.e., we may want to *quantify* the relative permissiveness of different policies. Model counting constraint solvers find the number of satisfying assignments for a given constraint, within a given bound [8, 19, 24, 25, 47, 48]. They have been applied to several quantitative analysis problems such as probabilistic analysis, reliability analysis, and quantitative information flow analysis [12–14, 18, 31, 35, 55–57].

In this paper we propose a framework to quantify permissiveness of access control policies using model counting constraint solvers. Our contributions include the following:

- A formal model for access control policies
- A formalization of access control policy permissiveness
- An automated approach for quantifying permissiveness of access control policies by translating a policy to a SMT formula and using a model counting constraint solver to quantify its permissiveness
- An extension of the formal model and automated approach to quantify *relative* permissiveness between policies
- A heuristic that transforms formulas extracted from policies for improving model counting performance
- An open-source tool, QUACKY, that implements the automated approach to analyze policies written in AWS Identity and Access Management (IAM) and Azure policy languages
- A publicly available policy dataset consisting of dozens of real-world policies from AWS forums and Azure documentation, as well as hundreds of policies synthesized by applying mutation techniques to the real-world policies
- An experimental evaluation of QUACKY on the dataset

The rest of the paper is organized as follows. In section 2 we first introduce cloud policies and motivate the need for quantitative permissiveness analysis. In Section 3 we present our formal policy model, in Section 4 we discuss SMT-based policy analysis, in Section 5 we discuss quantitative permissiveness analysis, in Section 6 we present our constraint transformation heuristic, in Section 7 we discuss the implementation of our approach on AWS and Azure policies, in Section 8 we discuss our experiments, in Section 9 we survey related work, and in Section 10 we conclude the paper.

2 BACKGROUND AND MOTIVATION

In this section we first introduce access control policies for the popular cloud services Amazon Web Services (AWS) and Microsoft Azure. We then discuss several examples motivating the need for quantitative analysis of access control policies.

2.1 Access Control Policies for the Cloud

Amazon Web Services Policies. Amazon Web Services (AWS) uses a shared responsibility security model where AWS guarantees security *of the cloud*, but users are responsible for security *in the cloud.* AWS lets users control who has access to their resources with access control policies written in the AWS policy language. Access requests are evaluated against policies and a dynamic environment context within a policy evaluation engine that either allows or denies access.

AWS defines a policy language where policies either allow or deny access through declarative statements. A *statement* is a 5-tuple (*Principal*, *Effect*, *Action*, *Resource*, *Condition*) where

- *Principal* specifies a list of users, entities, or services
- *Effect* = $\{Allow, Deny\}$ specifies whether the statement allows or denies access
- *Action* specifies a list of actions
- *Resource* specifies a list of resources
- *Condition* is an optional list of conditions further constraining how access is allowed or denied

Each condition consists of a condition operator, condition key, and condition value on elements of the request context. Full details of the language can be found in [11]. Note that while most of the elements of a policy are strings, certain condition keys specify other types of constraints (e.g., S3:MAX-KEYS expects an integral number). Additionally, the AWS policy language allows the use of two special characters within strings: '*', or wildcard, represents any string, and '?' which represents any single character. Given an access request and associated policy, permission is granted if and only if, for the given principal, action, resource, and condition key values in the request context, a statement in the policy allows access and no statement in the policy explicitly denies access.

Microsoft Azure Policies. Like AWS, Azure uses a shared responsibility security model, where security *in the cloud* is achieved by role-based access control (RBAC). Azure RBAC defines a policy language consisting of role definitions and role assignments. A *role definition* is a set of allowed actions

$$(Actions \cup DataActions) \setminus (NotActions \cup NotDataActions)$$

where

- *Actions* is a list of allowed management actions
- *DataActions* is a list of allowed data actions
- *NotActions* $\subseteq Actions$ is a list of denied management actions
- *NotDataActions* $\subseteq NotActions$ is a list of denied data actions.

A *role assignment* is a tuple *(principalId, roleDefId, scope, condition)* where

- *principalId* identifies a *Principal* granted access
- *roleDefId* identifies the role definition
- *scope* identifies a set of *Resources* granted access
- *condition* is an optional expression for granting access

The scope is a path in Azure's resource hierarchy, rooted at *'/'*. Resources rooted at the path are granted access. Unlike in AWS, the Azure condition is an infix logical expression. Azure has logical operators and relational operators on strings and integral numbers, but it also supports cross product relational operators on sets, like FORANYOFALLVALUES:STRINGEQUALS. Like AWS, Azure allows wildcards in strings (except scope). Given an access request, role definition, and role assignment, permission is granted if and only if, both the role definition and role assignment explicitly allow the principal and action under the scope and condition.

2.2 Motivating Examples

Capital One Data Breach. Capital One is one of many companies which use Amazon Web Services (AWS) for their cloud computing needs. AWS provides an access control mechanism for controlling access to resources through the Identity and Access Management (IAM) policy specification language. AWS IAM allows customers to create IAM roles and to give permissions to roles by attaching IAM policies to the role. Policies written in the IAM policy language allow AWS users to control access to resources and AWS services through fine-grained permissions. A role can then be assumed by a user or application. Recently, a server run by Capital One was breached by an outside attacker who was able to run user commands unrestricted [23, 52]. The attacker was then able to list the buckets (which store resources as objects) on the server and download the contents of each bucket. The attack itself involved two main components. First, the attacker was able to gain authenticated access to an AWS IAM role [23]. Secondly, the role had broad access to S3 buckets due to a misconfigured policy. For confidentiality reasons, the misconfigured policy is not publicly available.

The following represents a simplified model of the permissions allowed in the Capital One data breach.

```
Effect : Allow
Action : [s3:ListBucket,s3:GetObject]
Resource : *
```

When attached to an IAM role, the policy grants broad access to the s3 service, allowing the role to list and gather data within s3 buckets. However, if the attached role is compromised by a malicious user (as in the Capital One data breach) a great deal of data can be exposed. A less permissive policy might restrict resources to a single bucket

```
Resource = [firewall, firewall/*]
```

or restrict resources to only two objects within a bucket

```
Resource = [firewall, firewall/log10, firewall/log20]
```

Existing policy analysis techniques [11, 38] can verify if a policy is more or less permissive than another but they cannot quantify *the magnitude of permissiveness* in each case. Our work can quantify the differences in permissiveness between all three policies. If we assume valid resources are alphanumeric and ':', '-', '_', '/' characters with max length of 20, the initial policy allows the GetObject action on 2.50×10^{36} more resources than when access is restricted to a single bucket, which allows action on 1.05×10^{20} more resources than when access is restricted to two objects within a bucket.

Policies in the Wild. For the average user, the above policies are simple enough to manually analyze without sophisticated techniques. This is not always the case. Policies can be complex, especially to those unfamiliar with access control. AWS provides forums where users can post their policies and get feedback from other users (and AWS employees). We consider a set of policies taken from the forums to showcase the usefulness of our approach. For simplicity we assume values for fields in a policy contain alphanumeric and ':', '-', '_', '/' characters.

```
"Statement": [{
  "Effect": "Allow",
  "Action": [
    "s3:GetAccelerateConfiguration", ...,
    "s3:ListBucketMultipartUploads"],
  "Resource": "*"}]}
```

```
"Statement": [{
  "Effect": "Allow",
  "Action": [
    "s3:DescribeJob", ...,
    "s3:GetAccelerateConfiguration", ...,
    "s3:GetObject",
    "s3:GetObjectLegalHold", ...,
    "s3:ListBucketMultipartUploads"],
  "Resource": "*"}]}
```

```
"Statement": [{
  "Effect": "Allow",
  "Action": [
    "s3:DescribeJob", ...,
    "s3:GetAccelerateConfiguration", ...,
    "s3:GetObjectLegalHold", ...,
    "s3:ListBucketMultipartUploads"],
  "Resource": "*"}]}
```

Figure 1: Initial (topmost, (a)), modified (middle, (b)), and fixed (bottom, (c)), versions of a policy used by AWS Support

Quantifying Allowed Actions and Requests. In December 2021, the AWSSupportServiceRolePolicy policy used by AWS Support automated systems was modified to allow more actions. However, this modification inadvertently allowed the action s3:GETOBJECT [6], which greatly increased the number requests allowed by the policy (due to the nature of the GETOBJECT action). A bot detected the change and published it to GitHub, where several users raised concerns about GETOBJECT [7]. Without humans who had substantial AWS knowledge and who manually inspected the policy change, this vulnerability may not have been mitigated as quickly, thus necessitating the need for automated verification. Additionally, prior work (such as binary differential analysis) would be insufficient, as an additional action would undoubtedly increase permissiveness but the GETOBJECT action in particular increases permissiveness by almost an order of magnitude. AWS eventually fixed the policy, removing GETOBJECT. Simplified initial, modified, and fixed policies are shown in Fig 1. The fixed policy does not allow GETOBJECT

We can quantify the permissiveness of Policy 1(a) in terms of how many actions and requests are allowed by the policy. Assuming that resources are no more than 100 characters long, our tool reports that 24 actions and 4.09×10^{138} requests are allowed by the policy. This result is with respect to the set of valid AWS s3 actions and all possible resources, not the set of resources in the user's organization. If the set of resources is known, they can be added as a constraint and then our approach would count the requests allowed by the policy with respect to the set of known requests.

We can quantify the permissiveness of Policy 1(b) and 1(c). Our tool reports that 47 actions and 2.22×10^{205} requests are allowed by Policy 1(b). By removing GETOBJECT from Policy 1(b), our tool reports that 46 actions and 1.78×10^{205} requests are allowed by Policy 1(c). Note that both policies are identical except that Policy 1(c) does not contain the s3:GETOBJECT action. If instead of the s3:GETOBJECT action being removed from Policy 1(b), another action (such as s3:DESCRIBEJOB) were removed, then our tool reports that 46 actions and 2.22×10^{205} requests (as opposed to 1.78×10^{205}

```
"Statement": [{
  "Effect": "Allow",
  "Principal": "*",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::myexamplebucket/*"},
{
  "Effect": "Deny",
  "Principal": "*",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::myexamplebucket/*"}]
```

```
"Statement": [{
  "Effect": "Allow",
  "Principal": "*",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::myexamplebucket/*"},
{
  "Effect": "Deny",
  "Principal": "*",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::myexamplebucket/*",
  "Condition": {
    "StringNotLike": {
      "aws:userId": [
        "AROAEXAMPLEID:*", "AIDAEXAMPLEID", "111111111111"]}}}]
```

Figure 2: Initial (top, (a)) and fixed (bottom, (b)) versions of a policy for restricting access to certain users

requests) are allowed by the resulting policy. This demonstrates the need for quantitative analysis in the context of access control.

Quantifying Allowed Users. Depending on the scenario, most AWS users want a mix of public and private access to their data. This entails creating complex policies specifying access to their data, which often requires in-depth knowledge of the AWS policy language. In one scenario, a user posted on the forums seeking help in how to grant a specific set of users access to a bucket. The user was unable to craft a policy without allowing unintended access. Fig 2 shows the policies another user posted in response. Policy 2(a) denies all access to any data (eliminating all access). Policy 2(b) is a modification of the initial policy so that only a specific set of users can access data within the bucket (as well as denying anonymous access). Such policies that align with the user's intention can be difficult to craft, often due to complex access control logic needed or sheer complexity in how permissions should be governed.

We can quantify the permissiveness of Policy 2(b) in terms of how many aws:userIds are allowed by the policy. This lets the user verify that a change in policy semantics matches the original intention to only allow access for a certain set of users. Assuming that valid aws:userIds are no more than 20 characters long, our tool reports that 8.39×10^{10} aws:userIds are allowed by the policy out of 2.50×10^{36} possible aws:userIds. This result is with respect to the set of all possible userIds, and not the set of userIds in the user's organization. If the set of userIds is known, they can be added as a constraint and then our approach would count the userIds allowed by the policy with respect to the set of known userIds.

If we modify Policy 2(b) and remove the wildcard in the condition that defines allowed userIds (i.e., the line `"AROAEXAMPLEID:*"`), then our tool would report exactly the number of allowed userIds (2 in this case). So, in a scenario where an AWS user wants to specify a policy with a concrete number of permissions, our quantitative analysis can be used to verify the quantity of permissions.

Quantifying Trusted Values Inferred From A Policy. Trust Safety, introduced in [20], is the notion that a policy should not allow *untrusted* (i.e. public) access. Determining if a policy is *Trust Safe* requires inferring the set of *trusted values* from the policy by analyzing the values for *trusted keys* and making sure they do not match an "overly large" set of values. In [20] a syntactic check of the policy is performed to look for values containing a wildcard character ("*"), and if so, the policy is deemed not Trust Safe. However, in general, a syntactic check cannot determine the size of a set of values in an access control policy. Our approach can be used to precisely determine if the set of values is "overly large" by quantifying the size of the set of values. If the size surpasses a predetermined threshold, the policy would be deemed not Trust Safe.

3 POLICY MODEL

In this section, we introduce our policy model which forms the basis of our framework. Our model is designed to be expressive enough to model complex policy specifications that can be efficiently and precisely analyzed by modern verification and validation techniques.

We use an approach similar to [11] in defining our 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. An access request is a tuple $(\delta, a, r, e) \in \Delta \times A \times R \times E$ where 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

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

The policy grants access if the request is allowed by a rule in the policy and is not revoked by any other rule in the policy. Explicit denies overrule explicit allows (if a request is allowed by one rule and denied by another rule, the request is ultimately denied). 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

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

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

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

4 PERMISSIVENESS ANALYSIS

In this section we discuss how the permissiveness of our policy model is analyzed. Given a policy, the goal is to determine what requests are allowed by the policy, and if the policy is more or less permissive than another policy. This is done by reducing policies to logic formulas, similar to the approach used in [11, 38].

4.1 SMT Encoding of a Policy

The permissiveness of a policy is determined by the number of requests that it allows: the more requests allowed by a policy, the higher its permissiveness. The policy allowing all possible requests is the most permissive policy, and the policy which denies all requests is the least permissive policy. It follows that, given a policy, reasoning over all possible requests allowed by the policy determines the permissiveness of the policy. We encode the set of possible requests by introducing variables $\{\delta_{smt} \in \Delta, r_{smt} \in R, a_{smt} \in A, e_{smt} \in E\}$ in the generated SMT formula.

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

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

The SMT encoding of a policy $\mathbb{P}$ is given by $[\![\mathbb{P}]\!]$ and represents the set of requests allowed by $\mathbb{P}$. 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.,

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

4.2 Relative Permissiveness of Policies

For a single policy, equations 8, 9 provide a way to model the semantics of a policy in isolation. Below, we provide a policy analysis framework that, given two policies, determines the relative permissiveness between the two.

Intuitively, given two policies $\mathbb{P}_1$ and $\mathbb{P}_2$ we can determine whether one is more permissive than the other by analyzing formulas $[\![\mathbb{P}_1]\!] \Rightarrow [\![\mathbb{P}_2]\!]$ and $[\![\mathbb{P}_2]\!] \Rightarrow [\![\mathbb{P}_1]\!]$. However, it is possible that both policies allow different sets of requests, or the set of requests overlap. In general, there are four possible outcomes:

(1) $\textsc{Allow}(\mathbb{P}_1) \subset \textsc{Allow}(\mathbb{P}_2)$
(2) $\textsc{Allow}(\mathbb{P}_1) \supset \textsc{Allow}(\mathbb{P}_2)$
(3) $\textsc{Allow}(\mathbb{P}_1) = \textsc{Allow}(\mathbb{P}_2)$
(4) $\mathbb{P}_1$ and $\mathbb{P}_2$ do not subsume each other

The relative permissiveness of $\mathbb{P}_1$ and $\mathbb{P}_2$ directly follows from each scenario: $\mathbb{P}_1$ is less permissive than $\mathbb{P}_2$, $\mathbb{P}_1$ is more permissive than $\mathbb{P}_2$, $\mathbb{P}_1$ and $\mathbb{P}_2$ are equally permissive, or $\mathbb{P}_1$ and $\mathbb{P}_2$ are incomparable. The calculation involves satisfiability checks of two formulas: $[\![\mathbb{P}_1]\!] \nRightarrow [\![\mathbb{P}_2]\!]$ and $[\![\mathbb{P}_2]\!] \nRightarrow [\![\mathbb{P}_1]\!]$

- If $[\![\mathbb{P}_1]\!] \nRightarrow [\![\mathbb{P}_2]\!]$ is not satisfiable, then $\mathbb{P}_1$ cannot be more permissive than $\mathbb{P}_2$ ($\mathbb{P}_2$ is at least as permissive as $\mathbb{P}_1$).
- If $[\![\mathbb{P}_2]\!] \nRightarrow [\![\mathbb{P}_1]\!]$ is not satisfiable, then $\mathbb{P}_2$ cannot be more permissive than $\mathbb{P}_1$ ($\mathbb{P}_1$ is at least as permissive as $\mathbb{P}_2$).
- If both $[\![\mathbb{P}_1]\!] \nRightarrow [\![\mathbb{P}_2]\!]$ and $[\![\mathbb{P}_2]\!] \nRightarrow [\![\mathbb{P}_1]\!]$ are not satisfiable, then $\mathbb{P}_1$ and $\mathbb{P}_2$ are *equivalent*.
- Otherwise, $\mathbb{P}_1$ and $\mathbb{P}_2$ do not subsume each other.

Note that the formula $[\![\mathbb{P}_1]\!] \nRightarrow [\![\mathbb{P}_2]\!]$ can be simplified as

$$[\![\mathbb{P}_1]\!] \nRightarrow [\![\mathbb{P}_2]\!] = [\![\mathbb{P}_1]\!] \wedge \neg [\![\mathbb{P}_2]\!] \tag{11}$$

which can be checked using an SMT solver.

5 QUANTIFYING PERMISSIVENESS

Translating an access control policy into an SMT formula for satisfiability checking allows some permissiveness analysis, but it does not give insight as to how permissive a policy is. In this section, we introduce a novel approach for more precise reasoning in determining the permissiveness of a single policy or the relative permissiveness of two policies.

Given $\mathbb{P}$, $\textsc{Allow}(\mathbb{P})$ is the set of all requests allowed by $\mathbb{P}$. Let $|\textsc{Allow}(\mathbb{P})|$ denote the number of such requests. The permissiveness of $\mathbb{P}$ is given by

$$|\textsc{Allow}(\mathbb{P})| = |[\![\mathbb{P}]\!]| \tag{12}$$

Where $|[\![\mathbb{P}]\!]|$ denotes the number of models for formula $[\![\mathbb{P}]\!]$. Using a model counting constraint solver, we can automatically compute the value of $|[\![\mathbb{P}]\!]|$. Larger values for $|[\![\mathbb{P}]\!]|$ indicate a more permissive policy; lower values indicate a less permissive policy. A metric for analyzing permissiveness of a policy is to consider the likelihood that a randomly generated request is allowed by the policy. Let D be the set of all possible requests, with $|D|$ being the number of all possible requests. If $|[\![\mathbb{P}]\!]| = 0$ all requests are denied by $\mathbb{P}$, if $|[\![\mathbb{P}]\!]| = |D|$ all requests are allowed by $\mathbb{P}$. Let $\sigma = (\delta, a, r, e)$ be a request chosen uniformly at random from the set all possible requests. The probability that σ is allowed by $\mathbb{P}$ is

$$p(\sigma \models [\![\mathbb{P}]\!]) = \frac{|[\![\mathbb{P}]\!]|}{|D|} \tag{13}$$

This effectively gives permissiveness of a policy with respect to its domain. Higher probabilities indicate more permissive policies, lower probabilities indicates less permissive policies. A probability of 0.5 indicates the policy allows half of all possible requests. Note that a probability of 0 indicates a policy which denies all requests while a probability of 1 indicates a policy allowing all requests.

This approach can be extended for quantifying relative permissiveness between policies. Given policies $\mathbb{P}_1$, $\mathbb{P}_2$, the number of requests allowed by $\mathbb{P}_1$ and not allowed by $\mathbb{P}_2$ is:

$$|[\![\mathbb{P}_1]\!] \nRightarrow [\![\mathbb{P}_2]\!]| = |\{(\delta, a, r, e) : (\delta, a, r, e) \models [\![\mathbb{P}_1]\!] \wedge \neg [\![\mathbb{P}_2]\!]\}| \tag{14}$$

The number of requests allowed by $\mathbb{P}_2$ and not allowed by $\mathbb{P}_1$ is:

$$|[\![\mathbb{P}_2]\!] \nRightarrow [\![\mathbb{P}_1]\!]| = |\{(\delta, a, r, e) : (\delta, a, r, e) \models [\![\mathbb{P}_2]\!] \wedge \neg [\![\mathbb{P}_1]\!]\}| \tag{15}$$

Recall that when calculating relative permissiveness there are four possible outcomes: $\mathbb{P}_1$ is equivalent to $\mathbb{P}_2$, $\mathbb{P}_1$ is more permissive

Algorithm 1 TransformActions(F, M):

Input: SMT formula F, map M
Output: SMT formula with mapping applied to actions

```
1:  if $F \equiv F_1 \vee F_2$ then
2:      return TransformActions($F_1, M$) $\vee$ TransformActions($F_2, M$)
3:  else if $F \equiv F_1 \wedge F_2$ then
4:      return TransformActions($F_1, M$) $\wedge$ TransformActions($F_2, M$)
5:  else if $F \equiv (a_{smt} = c)$ then return $(a_{smt} = M(c))$
6:  else if $F \equiv (a_{smt} \neq c)$ then return $(a_{smt} \neq M(c))$
7:  else if $F \equiv (a_{smt} \in regex)$ then
8:      $F' = $ False
9:      for $c_i \in$ GetActionsFromRegex($regex$) do
10:         $F' = F' \vee (a_{smt} = c_i)$
11:     end for
12:     return TransformActions($F', M$)
13: end if
14: return $F$
```

than $\mathbb{P}_2$, $\mathbb{P}_1$ is less permissive than $\mathbb{P}_2$, or $\mathbb{P}_1$ and $\mathbb{P}_2$ are incomparable. Using equations 14, 15:

- If $\mathbb{P}_1$ is more permissive than $\mathbb{P}_2$ then $|[\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!]|$ quantifies how much more permissive $\mathbb{P}_1$ is than $\mathbb{P}_2$
- If $\mathbb{P}_2$ is more permissive than $\mathbb{P}_1$ then $|[\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!]|$ quantifies how much more permissive $\mathbb{P}_2$ is than $\mathbb{P}_1$
- If $\mathbb{P}_1$ and $\mathbb{P}_2$ do not subsume each other, $|[\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!]|$ and $|[\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!]|$ can be used to determine which policy is *objectively more permissive* (total requests allowed)

6 CONSTRAINT TRANSFORMATION

In this section we present a heuristic that transforms a set of equality and inequality constraints for a string variable to a set of range constraints on an ordered set. We do this by mapping a set of string constants to an ordered set of values. As we discuss below, this enables us to compactly encode constraints on policy actions extracted from access control policies.

In practice, there are a finite number of valid actions in an access control policy. For example, s3:GetObject is a valid action, but the fictitious action s3:FooBar is not. For our analysis to be precise, constraints specifying valid actions must be specified. Recall that $[\![\mathbb{P}]\!]$ is the constraint formula extracted from policy $\mathbb{P}$. I.e., $[\![\mathbb{P}]\!] \equiv F$ where F is an SMT formula. In a formula F extracted from an access control policy, we observe three types of terms that involve actions

$$a_{smt} = c \qquad a_{smt} \neq c \qquad a_{smt} \in regex \tag{16}$$

where c is a string constant and $regex$ is a regular expression. We first consider cases where only the first two types of terms are present in a formula, and then discuss how the transformation handles regular expression constraints. Consider the formula:

$$\begin{aligned} F \equiv\ & (a_{smt} = \text{s3:ListBucket}) \\ & \vee (a_{smt} = \text{s3:ListBucketVersions}) \\ & \vee (a_{smt} = \text{s3:ListBucketMultipartUploads}) \end{aligned} \tag{17}$$

By mapping s3:ListBucket $\mapsto$ 0, s3:ListBucketVersions $\mapsto$ 1, s3:ListBucketMultipartUploads $\mapsto$ 2, F can be rewritten as

$$F \equiv (a_{smt} \geq 0 \wedge a_{smt} \leq 2) \tag{18}$$

The use of range constraints gives a more compact encoding for constraints on policy actions, particularly when there is a large number of constraints on policy actions (such as the constraints specifying the set of all valid actions). We introduce a constraint

Algorithm 2 DisjunctionToRange(F):

Input: SMT formula F with mapped actions
Output: Transformed SMT formula with disjunctions collapsed into range constraints when possible

```
1:  if $F \equiv F_1 \vee ... \vee F_n$ then
2:      $F_R = $ False
3:      $F' = $ False
4:      $S = \{\}$
5:      for $F_i \in \{F_1, ..., F_n\}$ do
6:          if $F_i \equiv (a_{smt} = c)$ then
7:              $F_R = F_R \vee F_i$
8:              $S = S \cup \{c\}$
9:          else
10:             $F' = F' \vee$ DisjunctionToRange($F_i$)
11:         end if
12:     end for
13:     if size($S$) $\geq 2$ and size($S$) $- 1 = $ max($S$) $-$ min($S$) then
14:         return $F' \vee (a_{smt} \geq$ min($S$) $\wedge a_{smt} \leq$ max($S$))
15:     else
16:         return $F' \vee F_R$
17:     end if
18: else if $F \equiv F_1 \wedge ... \wedge F_n$ then
19:     $F' = $ True
20:     for $F_i \in \{F_1, ..., F_n\}$ do
21:         $F' = F' \wedge$ DisjunctionToRange($F_i$)
22:     end for
23:     return $F'$
24: end if
25: return $F$
```

transformation which transforms the constraints on valid actions into a much smaller set of range constraints. Let $V(a)$ be the set of all valid actions. The key insight is that the set $V(a)$ can be mapped to a totally ordered set $V'(a)$ which can be compactly represented using a combination of equality and inequality constraints. The mapping and $V'(a)$ are straightforward to construct: each valid action $a \in V(a)$ is mapped to a unique integer $i \in [0, |V(a)| - 1]$, and $V'(a)$ is the set of all such integers.

The constraint transformation heuristic consists of two phases: the first applies the mapping to constraints on actions, the second transforms disjunction constraints into range constraints. Given a constraint formula in negation normal form and the action mapping, Algorithm 1 first transforms constraints containing action variable a_{smt} so it is consistent with the mapping. For constraints $a_{smt} = c$ or $a_{smt} \neq c$ where c is some string constant, c is replaced by the integer according to the mapping. For regular expression constraints on action $a_{smt} \in regex$, the function GetActionsFromRegex($regex$) returns all valid actions satisfied by the regex (the number of valid actions is finite) and a disjunction on all possibilities is returned: e.g., if the constraint is ($a_{smt} \in$ s3:ListB$*$) (where $*$ corresponds to a wildcard) then GetActionsFromRegex returns the only valid actions matching the regex, s3:ListBucket, s3:ListBucketVersions, s3:ListBucketMultipartUploads. After the action constraints have been mapped, Algorithm 2 attempts to transform equality constraints on actions under a single disjunction into range constraints (such as in equation 18). If the transformation is not possible (e.g., the constants are not contiguous) the input formula is returned.

7 ANALYZING AWS AND AZURE POLICIES

Based on our proposed notion of policy permissiveness and our approach for quantifying permissiveness, we have developed a differential policy analysis framework for permissiveness analysis of access control policies. Our framework is general enough to be applied to a variety of policies written in multiple policy languages. To demonstrate the effectiveness of our approach, we show that it can be applied to existing real world access control models: policies for AWS IAM and Microsoft Azure.

7.1 Translation and Implementation

Scope and Translation of the AWS Policy Language. The AWS policy language is enormous, with each service having its own rules on actions and resources. We consider three of the most popular AWS services: Elastic Compute Cloud (EC2), Identity and Access Management (IAM), and Simple Storage Service (S3). We consider two levels of constraints for each service. First, actions are constrained to the set of actions defined by the service. S3:LISTBUCKET or S3:PUTOBJECT are valid S3 actions but S3:FOOBAR is not. Second, actions and resource types are constrained by each other: certain actions can act only on certain resource types; e.g., action S3:LISTBUCKET operates on resource ARN:AWS:S3:::BUCKET. Additionally, resource types are constrained by naming requirements; e.g., length of bucket names is between 3 and 63 characters

An AWS policy is a list of statements, each statement allowing or denying access for a given set of principals, actions and resources. For each statement, we create a rule ρ capturing its semantics. Principals, actions, and resources within a statement map to Δ, A, R in ρ. Modeling conditions into environment attributes of E is more complex. Each condition key together with a condition operator specifies values for which access is allowed or denied. The environment attributes are thus a set of tuples specifying the condition key and their respective values, where the number of tuples depends on the condition operator. For wildcard or anychar ('*','?') symbols, we use regular expressions to capture the set of allowed strings. For example, *resource* = BUCKET* translates to (MATCH *resource* /bucket.*/) where '.' corresponds to anychar, '/' denotes the start and end of a regular expression, '*' represents Kleene star. We handle condition operators such as STRINGLIKE similarly.

Scope and Translation of the Azure Policy Language. Like AWS, each Azure service has its respective set of rules on actions and resources. We consider Azure VMs and Blob Storage, which are analogous to EC2 and S3. We consider the same two levels of constraints as we do for AWS.

An Azure "policy" is given by a list of role definitions and a list of role assignments. We join them together on the *roleDefId* into rules ρ. For each ρ, we map *principalId* to Δ, $(Actions \cup DataActions) \setminus (NotActions \cup NotDataActions)$ to A, and *scope* to R. The condition is parsed into a tree whose leaves specify condition keys and their respective values; these are the environment attributes. Like for AWS, we use regex for wildcards.

Translating Action and Resource Type Constraints. Let T be the set of constraints representing action and resource type restrictions. Equation 12 now becomes

$$|[\![\mathbb{P}]\!]| = |[\![\mathbb{P}]\!] \wedge \mathrm{T}| \tag{19}$$

For comparing multiple policies, equations 14, 15 become

$$|[\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!]| = |([\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!]) \wedge \mathrm{T}| \tag{20}$$

$$|[\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!]| = |([\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!]) \wedge \mathrm{T}| \tag{21}$$

Algorithm 3 TRANSLATEPOLICY($\mathbb{P}$):

```
Input: policy $\mathbb{P}$
Output: SMT formula $[\![\mathbb{P}]\!]$ encoding $\mathbb{P}$
1: $[\![\mathbb{P}_{Allow}]\!]$ = FALSE
2: $[\![\mathbb{P}_{Deny}]\!]$ = FALSE
3: for rule $\rho$ in $\mathbb{P}$ do
4:     $[\![\delta]\!]$ = ENCODE($\rho(\delta)$)
5:     $[\![a]\!]$ = ENCODE($\rho(a)$)
6:     $[\![r]\!]$ = ENCODE($\rho(r)$)
7:     $[\![e]\!]$ = ENCODE($\rho(e)$)
8:     $[\![\rho]\!] = [\![\delta]\!] \wedge [\![a]\!] \wedge [\![r]\!] \wedge [\![e]\!]$
9:     if $\rho \in [\![\mathbb{P}_{Allow}]\!]$ then $[\![\mathbb{P}_{Allow}]\!] = [\![\mathbb{P}_{Allow}]\!] \vee [\![\rho]\!]$
10:    else $[\![\mathbb{P}_{Deny}]\!] = [\![\mathbb{P}_{Deny}]\!] \vee [\![\rho]\!]$
11:    end if
12: end for
13: return $[\![\mathbb{P}]\!] = [\![\mathbb{P}_{Allow}]\!] \wedge \neg[\![\mathbb{P}_{Deny}]\!]$
```

Algorithm 4 PERMISSIVENESS($\mathbb{P}, b$):

```
Input: policy $\mathbb{P}$, bound $b$
Output: permissiveness of $\mathbb{P}$
1: $[\![\mathbb{P}]\!]$ = TRANSLATEPOLICY($\mathbb{P}$)
2: T = GETTYPECONSTRAINTS()
3: if IsSAT($[\![\mathbb{P}]\!] \wedge$ T) then return COUNTMODELS($[\![\mathbb{P}]\!] \wedge$ T, $b$)
4: else return 0
5: end if
```

Algorithm 5 RELATIVEPERMISSIVENESS($\mathbb{P}_1, \mathbb{P}_2, b$):

```
Input: policies $\mathbb{P}_1, \mathbb{P}_2$; bound $b$
Output: relative permissiveness of $\mathbb{P}_1, \mathbb{P}_2$
1: $[\![\mathbb{P}_1]\!]$ = TRANSLATEPOLICY($\mathbb{P}_1$)
2: $[\![\mathbb{P}_2]\!]$ = TRANSLATEPOLICY($\mathbb{P}_2$)
3: T = GETTYPECONSTRAINTS()
4: $F_1 = [\![\mathbb{P}_1]\!] \wedge \neg[\![\mathbb{P}_2]\!] \wedge$ T
5: $F_2 = [\![\mathbb{P}_2]\!] \wedge \neg[\![\mathbb{P}_1]\!] \wedge$ T
6: if IsSAT($F_1$) and not IsSAT($F_2$) then
7:     return "$\mathbb{P}_1$ is more permissive", COUNTMODELS($F_1, b$)
8: else if not IsSAT($F_1$) and IsSAT($F_2$) then
9:     return "$\mathbb{P}_2$ is more permissive", COUNTMODELS($F_2, b$)
10: else if not IsSAT($F_1$) and not IsSAT($F_2$) then
11:    return "$\mathbb{P}_1$ and $\mathbb{P}_2$ are equivalent"
12: else if IsSAT($F_1$) and IsSAT($F_2$) then
13:    return "$\mathbb{P}_1$ and $\mathbb{P}_2$ do not subsume each other",
14:            COUNTMODELS($F_1, b$), COUNTMODELS($F_2, b$)
15: end if
```

We implement translation for T for AWS by scraping the AWS resource and property types reference webpages to identify the resource types each action can operate on. For Azure, we generate constraints by reading a CSV file from the Azure Portal that relates actions to resource types. Note that prior work [11, 20] does not consider type constraints in their analysis of access control policies.

Policy Translator. Based on our approach, we implemented an open-source tool called QUACKY that quantifies permissiveness or relative permissiveness by translating policies into SMT formulas and passing the formulas to a model counting constraint solver. Our implementation uses the popular Automata-based Model Counter (ABC) [8, 9] which uses automata-theoretic to model count string and numeric constraints. ABC counts satisfying solutions to the formula by constructing automata for an SMT formula and performing path counting on the automata. SMT formulas from QUACKY can also be fed into other SMT-LIB-conformant constraint solvers, such as Microsoft Z3.

QUACKY translates a policy $\mathbb{P}$ into a SMT formula $[\![\mathbb{P}]\!]$ by translating each rule ρ, as shown in Algorithm 3. To quantify the permissiveness of a policy $\mathbb{P}$, QUACKY translates $\mathbb{P}$, appends the type constraints T, and calls ABC to count the solutions satisfying $[\![\mathbb{P}]\!] \wedge \mathrm{T}$, as shown in Algorithm 4. To analyze the relative permissiveness between two policies $\mathbb{P}_1$ and $\mathbb{P}_2$, QUACKY produces two SMT formulas $[\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!]$ and $[\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!]$ and calls ABC to check their satisfiability and to count models, as shown in Algorithm 5.

8 EXPERIMENTAL EVALUATION

Below, we first describe our methodology for gathering policies; then we discuss the four experiments we conducted to evaluate our approach and its implementation in QUACKY [1]. The first experiment benchmarks QUACKY, and it evaluates QUACKY's performance and identifies which factors influence the analysis. The second experiment evaluates how effective QUACKY is at reasoning about the relative permissiveness of access control policies. The third experiment compares the performance of QUACKY with an enumerative model counting approach based on SMT solvers. The fourth experiment demonstrates that our approach can be applied to Azure policies. Unless otherwise noted, all experiments use the constraint transformation heuristic, and include type constraints.

In the experiments reported below we assume string variables (principal, action, resource, condition keys) contain any of the 256 ASCII characters and at most 100 characters long, unless otherwise specified. We report permissiveness as number of requests allowed (a request is a tuple (δ, a, r, e)). Results are reported in log-scale. For all experiments, we use a desktop machine with an Intel i5 3.5GHz X4 processor, 128GB DDR3 RAM, with a Linux 4.4.0-198 64-bit kernel, Z3 v4.8.11, and the latest build of ABC [2].

8.1 Policy Datasets

Due to security implications of making access control policies that are used in an organization public, policies that are both publicly available and representative of real-world policies are practically non-existent. We are unaware of any such dataset for neither AWS (those in [11, 21] were not released to the public) nor Azure policies. To evaluate our approach, a comprehensive dataset is required. We use two AWS policy datasets collected from users and argue these datasets are representative of real-world policies and comprehensive enough to show that our approach is effective. We also compile a dataset of Azure role definitions from Microsoft Docs.

Obtaining AWS Policies from Users. The lack of publicly available policy datasets for AWS means that finding quality policies is a cumbersome task. AWS users tend not to share policies possibly containing sensitive data (policies can leak organization structure). However, we found this to *not* be the case when users needed assistance designing and debugging their policies. AWS policies can be complex and unwieldy, especially to those unfamiliar with access control. Consequently, AWS provides forums where users needing assistance often post their policies and other users (and AWS employees) can provide assistance. Such policies are usually sanitized and vary in complexity, making the AWS forums a good source for compiling a dataset.

[1]Tool and benchmarks available at https://github.com/vlab-cs-ucsb/quacky
[2]https://github.com/vlab-cs-ucsb/ABC

AWS Policy Selection Criteria and Breakdown. As of 2021, AWS offers more than 200 services, many of which use access control policies and all of which have dedicated forums. We searched for policies based on several criteria. We focused on IAM, S3, and EC2 as they are among the most popular services and are more likely to yield the best sample of policies. Our goal was to have a good balance of simple and complex policies as well as policy sets, and we only included policies that are semantically valid.

Out of several hundred forum posts dating back several years, we identified 30 posts containing a total of 41 well-formed policies (the vast majority of posts either contained no policies or fragmented/invalid policies): from EC2 9 posts with single policies and 2 posts with multiple policies (4 policies), from IAM 2 posts with single policies and 3 posts with multiple policies (6 policies), from S3 9 posts with single policies and 5 posts with multiple policies (11 policies). From our observations, we found that when users sought assistance via the forums, they often only posted a single policy in isolation. Only 10 posts contained either multiple versions of the same policy or multiple policies combined together in a policy set (multiple AWS policies can be combined into a single policy).

Synthesizing AWS Policies Through Mutations. We synthesize AWS policies through mutations for two reasons. First, we want a larger dataset on which to evaluate our policy analysis framework and tool. Second, we want to mimic realistic scenarios where the semantic meaning of a policy is slightly modified by an employee within some organization. Modifications to a policy can alter the permissiveness of a policy in ways indiscernible without intensive manual inspection. A simple modification could allow one more user access to a resource or it could allow one thousand more users access to a resource; in either case, the modified policy is more permissive but clearly differs in magnitude. Synthesizing policies through mutation is one approach for modeling such scenarios.

We use ideas from mutation testing to synthesize policies [53, 65]. Mutation testing is a widely used software testing technique for measuring test suite strength. The technique applies mutations to a program under test to generate variations of the program, and evaluates them against a test suite. A faulty program, or mutant, is *killed* if at least one test in the suite fails. The more mutants killed, the higher the confidence in the test suite.

We synthesize mutants of a policy with mutations intended to alter the permissiveness of a policy, which we use to evaluate the effectiveness of our approach. We implement three types of mutations which mimic realistic scenarios and generally yield more permissive mutants:

(1) If a statement's *Effect* is *Deny*, change it to *Allow* and negate the statement's *Action* and *Resource* keys to *NotAction* and *NotResource* or vice versa.
(2) If a statement's *Action* or *Resource* values are lists, change them to a single string containing a wildcard. For example, an *Action* list containing s3:LISTBUCKET and s3:GETOBJECT is changed to a single string s3:*.
(3) If a statement contains any *Condition*s, remove them.

For each statement of a given policy, we create a set of applicable mutation types. For example, consider a statement with an *Allow* effect, a list of *Action* values, and a *Condition*. The set of applicable mutations is {*type 2, type 3*} because the type 1 mutation does not

Table 1: Times for each AWS service, with and without the constraint transformation heuristic. Times are in seconds.

	Without Transformation			With Transformation		
	Min	Max	Avg	Min	Max	Avg
EC2	2.08	880.18	128.98	0.50	33.41	10.11
IAM	0.26	8.65	1.50	0.16	0.71	0.27
S3	0.06	29.60	3.64	0.05	7.37	0.77

Table 2: Results for each AWS service, with and without type constraints. Permissiveness is the number of requests allowed. AM is Arithmetic Mean, GM is Geometric Mean.

	Avg exec time (s)		log_2(AM)		log_2(GM)	
	No Type	Type	No Type	Type	No Type	Type
EC2	0.65	10.11	1,705.65	1,579.70	1,308.86	918.49
IAM	0.05	0.27	1,598.60	1,321.92	827.41	669.75
S3	0.52	0.77	2,494.85	2,344.58	1,499.67	1,432.77

apply to the *Allow* effect. The power set of applicable mutation types represents combinations of mutations that can be applied to that statement. Thus, we create such a powerset for each statement. By choosing one set from each powerset and applying the mutation types in that set to its respective statement, we output a mutated policy. From 9 original EC2 policies, we generated 240 mutants. From 6 original IAM policies, we generated 26 mutants. From 14 original S3 policies, we generated 280 mutants. In total, from 29 original policies, we generated 546 mutants.

Obtaining Azure Policies from Microsoft Docs. As of 2021, Azure comprises more than 200 services and 120 built-in roles. We are unaware of any forums where users post custom role definitions, so we searched Microsoft Docs for built-in role definitions. We focused on Azure VMs and Blob Storage because they are analogous to EC2 and S3. We obtained 2 policies from VMs and 3 from Blob Storage for our proof of concept.

8.2 QUACKY Benchmarking

The goal of these experiments is to evaluate QUACKY's performance and identify which factors influence the effect of the analysis (in terms of counts and time taken). We evaluate the performance and effectiveness of QUACKY on 41 policies taken from AWS forums. First we evaluate the effectiveness of the constraint transformation heuristic from Section 5 by analyzing each policy, with type constraints, twice, both without the heuristic and with the heuristic enabled. Then, we analyze each policy twice, once without type constraints and once with type constraints.

Effectiveness of Constraint Transformation. The results, separated by AWS service, are shown in Table 1. The decrease in minimum times with the constraint transformation heuristic was between 16% for S3 to 76% for EC2. The maximum times decreased between 75% for S3 to 96% for EC2. The results for average times were similar, with a decrease of between 78% for S3 to 92% for EC2. The heuristic reduced the minimum, maximum, and average times by about an order of magnitude for EC2, but not as much for IAM and S3. This may be because EC2 has more actions (311 as of writing) than both IAM (183) and S3 (223), and thus it may reap more benefits from range constraints as opposed to equality constraints.

Impact of Type Constraints. The results for each AWS service are shown in Table 2. Out of the 41 policies, 1 policy allowed no request both with and without type constraints; 1 policy allowed requests without type constraints but allowed none when type constraints were present. Without type constraints, QUACKY analyzed each policy in under a second. Type constraints slow the analysis considerably but drastically effect permissiveness, decreasing the number of allowed requests by hundreds of orders of magnitude. This is due to type constraints restricting the set of possible actions and constraining actions to only act on specific resource types. Type constraints represent all possible action and resource type restrictions and must be explicitly enumerated within the constraint, slowing down the analysis. For every policy, the presence of type constraints resulted in a more precise analysis. Without type constraints to model the semantics of the policy language, QUACKY gives an *overapproximation* of the permissiveness for a policy.

8.3 Relative Permissiveness Quantification

The goal of this experiment is to evaluate how effective QUACKY is at reasoning about the relative permissiveness of access control policies, and to showcase the effectiveness of quantifying relative permissiveness in general. We evaluate the effectiveness of QUACKY in quantifying relative permissiveness between a policy and its synthesized mutants. We record the average times and differences in permissiveness between the mutants and the original policy.

Each policy $\mathbb{P}$ is compared against every one of its mutants $\mathbb{P}_m$ twice: once to quantify the number of requests allowed by $\mathbb{P}$ but not $\mathbb{P}_m$ and once to quantify the number of requests allowed by $\mathbb{P}_m$ but not $\mathbb{P}$. We used type constraints, constraint transformation, and a timeout of 10 minutes for each pair of comparisons. The results are shown in Table 3. The third column shows the average time across all pairs of comparisons.

Columns 3-6 of Table 3 show the distribution of permissiveness between each policy and its mutants. The majority of mutants were either less permissive, more permissive, or equivalent to the original policy. Columns 7-10 show the results of quantifying the difference in permissiveness whenever a policy and its mutant were not equivalent and did not subsume each other. For each policy and its set of mutants, columns 7 and 8 report the arithmetic and geometric means for the number of requests allowed by $\mathbb{P}$ but not by $\mathbb{P}_m$. Conversely, columns 9 and 10 report the means for the number of requests allowed by $\mathbb{P}_m$ but not by $\mathbb{P}$.

8.4 Comparison with Enumerative Model Counting

SAT/SMT solvers have been used in prior access control policy analysis techniques to resolve queries about policy behavior (e.g., Zelkova, Margrave [11, 33, 51]). This often involves enumerating the set of solutions to the query, through repeated calls to a constraint solver. In each call, the constraints are revised by appending the negation of all prior solutions. Our approach differs fundamentally

Table 3: Results for AWS policies compared with their mutants. Arithmetic/Geometric Mean (AM/GM) for number of requests allowed (log-scale, $\perp$ used when the count is 0) are reported when the mutant is less or more permissive than its original policy.

Policy	Avg exec time (s)	#$\mathbb{P}_m$ Less permissive	#$\mathbb{P}_m$ More permissive	# Equivalent	# Neither subsumes	$\mathbb{P}_m$ Less permissive $\log_2$(AM)	$\mathbb{P}_m$ Less permissive $\log_2$(GM)	$\mathbb{P}_m$ More permissive $\log_2$(AM)	$\mathbb{P}_m$ More permissive $\log_2$(GM)
[EC2] P1	30.23	0 (0%)	60 (93.8%)	4 (6.3%)	0 (0%)	$\perp$	$\perp$	1823.2	1614.7
[EC2] P2	85.18	0 (0%)	28 (87.5%)	4 (12.5%)	0 (0%)	$\perp$	$\perp$	1361.5	1162.8
[EC2] P3	57.79	0 (0%)	6 (75%)	2 (25%)	0 (0%)	$\perp$	$\perp$	1331.8	993.2
[EC2] P4	71.64	0 (0%)	12 (75%)	4 (25%)	0 (0%)	$\perp$	$\perp$	461.8	431.6
[EC2] P5	24.48	0 (0%)	12 (37.5%)	4 (12.5%)	16 (50%)	$\perp$	$\perp$	1197.7	788.9
[EC2] P6	45.68	4 (25%)	8 (50%)	0 (0%)	4 (25%)	461.4	292.7	123.1	123.1
[EC2] P7	47.29	0 (0%)	28 (87.5%)	4 (12.5%)	0 (0%)	$\perp$	$\perp$	1361.5	1008.3
[EC2] P8	170.28	8 (25%)	0 (0%)	24 (75%)	0 (0%)	154.1	154.1	$\perp$	$\perp$
[EC2] P9	3.11	0 (0%)	0 (0%)	8 (100%)	0 (0%)	$\perp$	$\perp$	$\perp$	$\perp$
[IAM] P10	1.38	0 (0%)	2 (50%)	2 (50%)	0 (0%)	$\perp$	$\perp$	486.7	486.7
[IAM] P11	4.71	0 (0%)	6 (75%)	2 (25%)	0 (0%)	$\perp$	$\perp$	1385.0	1288.9
[IAM] P12	1.05	0 (0%)	2 (50%)	2 (50%)	0 (0%)	$\perp$	$\perp$	486.6	486.6
[IAM] P13	6.13	0 (0%)	0 (0%)	2 (100%)	0 (0%)	$\perp$	$\perp$	$\perp$	$\perp$
[IAM] P14	0.92	0 (0%)	2 (50%)	2 (50%)	0 (0%)	$\perp$	$\perp$	5.6	5.6
[IAM] P15	3.60	0 (0%)	2 (50%)	2 (50%)	0 (0%)	$\perp$	$\perp$	2124.9	2124.9
[s3] P16	2.28	6 (37.5%)	4 (25%)	4 (25%)	2 (12.5%)	628.9	628.9	684.7	684.7
[s3] P17	1.37	0 (0%)	6 (75%)	2 (25%)	0 (0%)	$\perp$	$\perp$	2287.3	1953.9
[s3] P18	1.02	0 (0%)	6 (75%)	2 (25%)	0 (0%)	$\perp$	$\perp$	800.4	536.2
[s3] P19	10.22	2 (25%)	4 (50%)	2 (25%)	0 (0%)	1484.7	1484.7	1276.9	1276.9
[s3] P20	3.46	0 (0%)	0 (0%)	16 (100%)	0 (0%)	$\perp$	$\perp$	$\perp$	$\perp$
[s3] P21	1.38	0 (0%)	12 (75%)	4 (25%)	0 (0%)	$\perp$	$\perp$	684.7	684.7
[s3] P22	10.56	16 (25%)	40 (62.5%)	8 (12.5%)	0 (0%)	2192.0	2192.0	2294.7	2268.8
[s3] P23	2.51	0 (0%)	8 (50%)	8 (50%)	0 (0%)	$\perp$	$\perp$	5.6	5.6
[s3] P24	2.83	0 (0%)	0 (0%)	4 (100%)	0 (0%)	$\perp$	$\perp$	$\perp$	$\perp$
[s3] P25	2.06	0 (0%)	4 (50%)	4 (50%)	0 (0%)	$\perp$	$\perp$	2144.0	2144.0
[s3] P26	0.67	0 (0%)	10 (62.5%)	6 (37.5%)	0 (0%)	$\perp$	$\perp$	1479.1	1435.1
[s3] P27	5.06	6 (18.8%)	20 (62.5%)	4 (12.5%)	2 (6.3%)	2056.0	2056.0	2378.8	2273.9
[s3] P28	2.57	0 (0%)	2 (50%)	2 (50%)	0 (0%)	$\perp$	$\perp$	684.7	684.7
[s3] P29	76.08	8 (12.5%)	24 (37.5%)	24 (37.5%)	8 (12.5%)	2076.9	2076.9	2268.9	2268.9

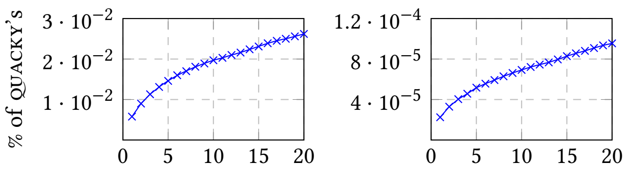

Figure 3: Counts for the enumerative approach as percentage of the count from QUACKY on a simple policy over a 20 minute period, for bounds 18 (left) and 19 (right).

Table 4: Average model counting rates for the enumerative approach and QUACKY, with type constraints. The former's average model counting rates in the first half (0-5 min.) and second half (5-10 min.) of the 10 minute timeout interval are reported.

	Average models counted per second		
	Enum. (0-5 min.)	Enum. (5-10 min.)	QUACKY
EC2	2.33	1.32	$10^{474.53}$
IAM	4.02	2.30	$10^{398.54}$
S3	0.94	0.76	$10^{705.92}$

as we do not rely on enumerating solutions by repeatedly calling a constraint solver, but rather we use a model counting constraint solver (ABC) that can count all solutions in a single call.

In these experiments we compare our approach to an enumerative approach using the Z3 SMT constraint solver [27, 49]. First, we analyze a simple policy allowing 2 s3 actions on 2 resources: ARN:AWS:S3:::FOO* and ARN:AWS:S3:::BAR. We varied the string bound from 16 to 21 to let the wildcard match 0 to 5 characters (resp.), and we set a 20 minute timeout. For bounds 16 and 17, both approaches finished counting 4 and 516 models in 0.15 and 16.77 seconds (resp.) for the enumerative approach and in 0.03 and 0.03 seconds (resp.) for QUACKY. For bounds 18 to 21, the enumerative approach timed out after counting 3446, 3217, 3340, 3125 models (resp.), whereas QUACKY finished counting 1.3×10^5, 3.4×10^7, 8.6×10^9, 2.2×10^{12} models (resp.) within one second. The results for bounds 18 and 19 are shown in Fig. 3.

We also analyze the 41 AWS policies using both approaches. The results are shown in Table 4. For each namespace, QUACKY yielded

Table 5: Results for Azure VM and Blob Storage policies, with and without type constraints. Permissiveness is the number of requests allowed and is reported in log-scale (base 2)

	Time (s)		Permissiveness	
	No Type	Type	No Type	Type
[VM] LoginUser	0.73	8.73	3096.01	1046.57
[VM] LoginAdmin	0.79	8.79	3096.01	1047.57
[BS] DataReader	0.36	1.43	1409.59	806.57
[BS] DataContributor	0.63	2.04	1411.18	808.89
[BS] DataOwner	0.37	3.76	2944.01	810.03

an astronomically greater average model counting rate than the enumerative approach. Moreover, the average rate of the enumerative approach decreased between the first and second halves of the 10 minute timeout interval. These results show that quantifying permissiveness using an enumerative approach for policy analysis (such as [51]) based on an off-the-shelf SMT or SAT solver is not a viable option for quantitative permissiveness analysis.

8.5 Microsoft Azure Policies

The goal of this experiment is to demonstrate that our approach can be used to analyze Azure policies. Like we did for AWS, we evaluate the performance and effectiveness of QUACKY on the 5 policies taken from Microsoft Docs. We analyze each policy twice, once without type constraints and once with type constraints. Because many string variables in Azure policies are more than 100 characters long, we assume that they are at most 250 characters long.

The results are shown in Table 5. Like previous experiments, there is a tradeoff between time and permissiveness. Without type constraints, the two VM policies seem to have the same permissiveness in log scale (base 2), but with type constraints, it is clear that more distinct requests are allowed by LoginAdmin than by LoginUser. The Blob Storage DataReader, DataContributor, and DataOwner policies are increasingly permissive. Without type constraints, DataOwner seems much more permissive than DataReader and DataContributor. With type constraints, we see that $2^{810.03}$ distinct requests are allowed by DataOwner, whereas $2^{806.57}$, $2^{808.89}$ distinct requests are allowed by DataReader, DataContributor (resp.).

8.6 Threats to Validity

The policies extracted from AWS forums/Azure documentation are based on a small sample and may not be representative enough. We mitigate this threat by expanding the dataset through mutations, creating a larger benchmark, and publicly releasing this benchmark. Our current experimental evaluation focuses on a subset of AWS/Azure services (S3, EC2, IAM for AWS and VM, BS for Azure). Although our techniques are extensible, extensions of our approach to more services requires further experimental evaluation.

9 RELATED WORK

Access control has been the subject of extensive research [59–61], many access policy languages have been proposed [1, 41–43], and the problem with access policies becoming large and difficult to reason about has been noted in the past [36].

There has been earlier work on verification of access control policies [29, 39], as well as on assisting policy creation [30, 32]. Some earlier work analyze role based access control schemas using the Alloy analyzer [62, 66].

The work most closely related to our work is that of Zelkova [11]. Zelkova is a closed-source tool for analyzing properties of AWS policies which can automatically compare two AWS policies and determine whether one is more permissive than the other. The two crucial distinctions between Zelkova and our work is that (1) we provide a general policy framework for analyzing access control policies which can be applied to other policy languages, and (2) we introduce a novel approach for quantifying the permissiveness of access control policies (rather than a binary yes/no answer in Zelkova). Both our work and Zelkova build from ideas from the SAT-based checking of XACML [37]. In their approach, Hughes et al use a bounded approach to analyze properties of XACML policies with SAT solvers. Recent work has built upon Zelkova [21] but does not provide quantitative assessments of permissiveness. Margrave [34] is a tool that analyzes XACML policies using a multi-terminal decision diagrams. Margrave goes beyond binary/ternary differential analysis, allowing a user to write general-purpose queries over changes to a policy. In a later work [51], Margrave uses a SAT solver to enumeratively produce sets of solutions to queries. Our experiments show that this type of enumerative analysis approach is not nearly scalable enough for meaningful quantitative analysis.

Verification techniques for analyzing access control policies embedded in programs have been studied [17, 26, 50]. Derailer is interactive tool that let the developer traverse the tree of all data exposed by an application and interactively generate a desired policy [50]. RubyX [26] is a tool for symbolic execution for Rails that can be used to find access control bugs. CanCheck [17] is an automated verification tool that uses first order logic encoding and theorem proving for finding access control bugs in Rails applications.

Differential analysis techniques have also been investigated in the past [3–5, 15, 16, 44–46, 54]. For example, in [54] differential symbolic execution is used to find differences between original and refactored code by summarizing procedures into symbolic constraints and then comparing different summaries using an SMT solver. SYMDIFF [44] computes the semantic difference between two functions using the Z3 SMT solver [27, 49]. However, we are not aware of any prior work on quantitative differential analysis.

10 CONCLUSIONS

Errors in access control policies used for controlling access to data sources available on cloud servers can have disastrous consequences. In this paper we presented a new approach for modeling and quantifying permissiveness of access control policies. Our approach relies on model counting constraint solvers to assess the permissiveness of a given policy. We implemented this approach for AWS policies and experimentally evaluated its effectiveness on AWS policies we collected from discussion forums. Our results demonstrate that our quantitative permissiveness analysis approach is applicable in practice. In future work, we aim to investigate how quantitative analysis techniques can be applied to other policy analysis problems, such as policy repair.

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] aleak [n.d.]. Another misconfigured Amazon S3 server leaks data of 50,000 Australians. https://www.scmagazineuk.com/another-misconfigured-amazon\-s3-server-leaks-data-of-50000-australians/article/705125/.

[3] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. 2014. Semantic Differential Repair for Input Validation and Sanitization. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'14).*

[4] Muath Alkhalaf, Tevfik Bultan, and Jose L. Gallegos. 2012. Verifying Client-side Input Validation Functions Using String Analysis. In *Proceedings of the 34th International Conference on Software Engineering (ICSE).* 947–957.

[5] Muath Alkhalaf, Shauvik Roy Choudhary, Mattia Fazzini, Tevfik Bultan, Alessandro Orso, and Christopher Kruegel. 2012. ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'12).*

[6] awssupportleak [n.d.]. AWSSupportServiceRolePolicy Informational Update. https://aws.amazon.com/security/security-bulletins/AWS-2021-007/.

[7] awssupportleakgh [n.d.]. Update detected. https://github.com/z0ph/MAMIP/commit/9d72709.

[8] 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

[9] 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.

[10] azureflaw [n.d.]. Microsoft Azure cloud vulnerability is the 'worst you can imagine'. https://www.theverge.com/2021/8/27/22644161/microsoft-azure-database-vulnerabilty-chaosdb?fbclid=IwAR2nKV8uslH4EGDslnogYT4ulQRGz7NsD0xuIb3lgK2sP1-WG_O1tJbR-eE.

[11] 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.

[12] Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic Discovery and Quantification of Information Leaks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA.* 141–153.

[13] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. 2016. String Analysis for Side Channels with Segmented Oracles. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering.*

[14] Lucas Bang, Nicolás Rosner, and Tevfik Bultan. 2018. Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018.* 307–322.

[15] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V. N. Venkatakrishnan. 2010. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10).* ACM, New York, NY, USA, 607–618. https://doi.org/10.1145/1866307.1866375

[16] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, and V. N. Venkatakrishnan. 2011. WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11).* ACM, New York, NY, USA, 575–586. https://doi.org/10.1145/2046707.2046774

[17] Ivan Bocic and Tevfik Bultan. 2016. Finding Access Control Bugs in Web Applications with CanCheck. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016).*

[18] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, and Corina S. Pasareanu. 2015. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015.* 866–877.

[19] Mateus Borges, Quoc-Sang Phan, Antonio Filieri, and Corina S. Pasareanu. 2017. Model-Counting Approaches for Nonlinear Numerical Constraints. In *Proceedings of the 9th International NASA Formal Methods Symposium.* 131–138.

[20] 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

[21] 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 *ESEC/SIGSOFT FSE 2020, Sacramento, California, United States of America, November 8-13, 2020.*

[22] cancan 2015. ryanb/cancan • GitHub. https://github.com/ryanb/cancan.

[23] Capital One Data Breach Analysis 2019. A Technical Analysis of the Capital One Hack. https://blog.cloudsploit.com/a-technical-analysis-of-the-capital-one-hack-a9b43d7c8aea.

[24] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2014. Distribution-Aware Sampling and Weighted Model Counting for SAT. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence.* 1722–1730.

[25] Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. 2016. Approximate Probabilistic Inference via Word-Level Counting. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence.* 3218–3224.

[26] Avik Chaudhuri and Jeffrey S. Foster. 2010. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010.* 585–594. https://doi.org/10.1145/1866307.1866373

[27] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

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

[29] 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

[30] Serge Egelman, Andrew Oates, and Shriram Krishnamurthi. 2011. Oops, I Did It Again: Mitigating Repeated Access Control Errors on Facebook. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI '11).* ACM, New York, NY, USA, 2295–2304. https://doi.org/10.1145/1978942.1979280

[31] Antonio Filieri, Corina S. Pasareanu, and Willem Visser. 2013. Reliability analysis in symbolic pathfinder. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013.* 622–631.

[32] Kathi Fisler and Shriram Krishnamurthi. 2010. A model of triangulating environments for policy authoring. In *SACMAT 2010, 15th ACM Symposium on Access Control Models and Technologies, Pittsburgh, Pennsylvania, USA, June 9-11, 2010, Proceedings*, James B. D. Joshi and Barbara Carminati (Eds.). ACM, 3–12. https://doi.org/10.1145/1809842.1809847

[33] 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 (ICSE 05).* 196–205.

[34] 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.

[35] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012.* 166–176.

[36] Mark Heckman and Karl N. Levitt. 1998. Applying the Composition Principle to Verify a Hierarchy of Security Servers. In *HICSS (3).* 338–347. http://citeseer.nj.nec.com/134822.html

[37] Graham Hughes and Tevfik Bultan. 2007. Interface Grammars for Modular Software Model Checking. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07).* 39–49. x-yojimbo-item://CC3D1214-37E5-4BFA-9D7F-D22F43161CEC

[38] Graham Hughes and Tevfik Bultan. 2008. Automated Verification of Access Control Policies Using a SAT Solver. , 18 pages. https://doi.org/10.1007/s10009-008-0087-9

[39] 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

[40] iam [n.d.]. AWS IAM Policy Language. http://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html.

[41] 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

[42] 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

[43] 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

[44] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs. In *Proceedings of the 24th International Conference on Computer Aided Verification* (Berkeley, CA) *(CAV'12)*. Springer-Verlag, Berlin, Heidelberg, 712–717. https://doi.org/10.1007/978-3-642-31424-7_54

[45] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential Assertion Checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 345–355. https://doi.org/10.1145/2491411.2491452

[46] Shuvendu K. Lahiri, Kapil Vaswani, and C A. R. Hoare. 2010. Differential Static Analysis: Opportunities, Applications, and Challenges. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (Santa Fe, New Mexico, USA) *(FoSER '10)*. ACM, New York, NY, USA, 201–204. https://doi.org/10.1145/1882362.1882405

[47] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. 2004. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation* 38, 4 (2004), 1273 – 1302. https://doi.org/10.1016/j.jsc.2003.04.003

[48] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. 2014. A model counter for constraints over unbounded strings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 57.

[49] Microsoft Inc. [n.d.]. Z3 SMT Solver. https://github.com/Z3Prover/z3.

[50] Joseph P. Near and Daniel Jackson. 2014. Derailer: interactive security analysis for web applications. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 587–598. https://doi.org/10.1145/2642937.2643012

[51] 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.

[52] Paige Thompson Indictment 2019. United States of America vs Paige A. Thompson. https://www.justice.gov/usao-wdwa/press-release/file/1188626/download.

[53] Mike Papadakis, Marinos Kintis, J. Zhang, Yue Jia, Y. L. Traon, and M. Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.* 112 (2019), 275–378.

[54] Suzette J. Person. 2009. *Differential Symbolic Execution*. Ph.D. Dissertation. University of Nebraska at Lincoln, Lincoln, NB, USA. Advisor(s) Dwyer, Matthew B. AAI3365729.

[55] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. 328–342.

[56] Quoc-Sang Phan, Pasquale Malacaria, Corina S. Pasareanu, and Marcelo d'Amorim. 2014. Quantifying information leaks using reliability analysis. In *Proceedings of the International Symposium on Model Checking of Software, SPIN 2014, San Jose, CA, USA*. 105–108.

[57] Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Pasareanu. 2012. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[58] pundit 2016. GitHub - elabs/pundit: Minimal authorization throught OO design and pure Ruby classes. https://github.com/elabs/pundit.

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

[60] 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

[61] 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

[62] 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)*.

[63] 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/.

[64] XACML 2003. eXtensible Access Control Markup Language (XACML) Version 1.0. OASIS Standard. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[65] Dianxiang Xu, Roshan Shrestha, and Ning Shen. 2020. Automated Strong Mutation Testing of XACML Policies *(SACMAT '20)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/3381991.3395599

[66] 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*.