Quacky: Quantitative Access Control Permissiveness Analyzer*

William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, Tevfik Bultan
{weiers,ganesh,albert_li,emilyomahony,benjaminprince,bultan}@ucsb.edu
University of California Santa Barbara
Santa Barbara, USA

ABSTRACT

QUACKY is a tool for quantifying permissiveness of access control policies in the cloud. Given a policy, QUACKY translates it into a SMT formula and uses a model counting constraint solver to quantify permissiveness. When given multiple policies, QUACKY not only determines which policy is more permissive, but also quantifies the relative permissiveness between the policies. With QUACKY, policy authors can automatically analyze complex policies, helping them ensure that there is no unintended access to private data. QUACKY supports access control policies written in the Amazon Web Services (AWS) Identity and Access Management (IAM), Microsoft Azure, and Google Cloud Platform (GCP) policy languages. It has command-line and web interfaces. It is open-source and available at https://github.com/vlab-cs-ucsb/quacky.

Video URL: https://youtu.be/YsiGOI_SCtg.

CCS CONCEPTS

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

ACM Reference Format:
William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, Tevfik Bultan. 2022. Quacky: Quantitative Access Control Permissiveness Analyzer. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559530

1 INTRODUCTION

Software services are now ubiquitous; as a result, companies are storing their data in compute clouds. The most popular cloud service providers, namely Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), allow users to protect their data through *access control policies*, a set of rules specifying access to cloud data. However, such policy specifications can be error prone, leading to unintended and unauthorized access to private data. In fact, in recent years, there have been numerous instances in which millions of customers' confidential data was exposed due to incorrect policy specifications [5, 10]. Hence, validation and verification of policy specifications is of utmost importance.

In this paper, we introduce our open-source tool QUACKY for quantitatively assessing the permissiveness of access control policies in the cloud. QUACKY is based on the technical approach presented in [6]. We extend this prior work into a full-fledged open source tool, add support for GCP policies, and build a web interface to improve usability. QUACKY quantifies permissiveness of policies written in the AWS Identity and Access Management (IAM), Azure, and GCP policy languages. Using QUACKY, we analyze 41 real-world AWS policies, 5 Azure policies, and 5 GCP policies, showcasing its ability to analyze real-world policies.

The **envisioned users** of QUACKY include researchers, software engineers, cloud solutions architects, system administrators, and others who write or use access control policies in the cloud and want to ensure their policies do not allow unintended access to private data. The **challenge** we propose to address involves understanding the permissiveness of an access control policy. In Sections 3 and 4 we describe the **methodology** of how QUACKY aids users in understanding the permissiveness of policies. In Section 5 we describe the results of our experimental **validation** of QUACKY.

2 RELATED WORK

There are existing tools for analyzing access control policies, such as the tool from Hughes et al [7] and the Margrave tool for XACML policies [9], and the closed-source Zelkova tool for AWS policies [3]. The tool presented by Hughes et al. analyzes XACML policies by reducing the problem to a SAT formula and then using a SAT solver to obtain the result of the analysis. The Margrave tool also uses a SAT solver to answer queries about behaviors of XACML policies. The proprietary Zelkova tool analyzes properties of AWS policies using a reduction to SMT formulas. Unlike Margrave, the tool by Hughes et al. and Zelkova both use a differential analysis approach to compare policies and determine if one policy is more permissive than another. While these tools can reason over properties of policies (such as permissiveness), they cannot *quantify* such properties. Often, it is useful to know *how permissive* a policy is or *how much more permissive* one policy is than another, neither of which can be answered by existing tools. QUACKY not only reasons about properties of policies but it also quantifies them. Moreover, QUACKY supports multiple policy languages.

3 ANALYZING ACCESS CONTROL POLICIES

An access control policy is a mechanism for preventing unintended access to data. Generally, access control policies consist of rules specifying which *principals* can perform which *actions* on which *resources* under which *conditions*. When a principal (such as a user) makes an access request to perform an action on some resource, the request is evaluated against the relevant policy. If the policy allows the request, then it is granted access. Otherwise, the request is denied access. A properly configured policy should allow only

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

ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3559530

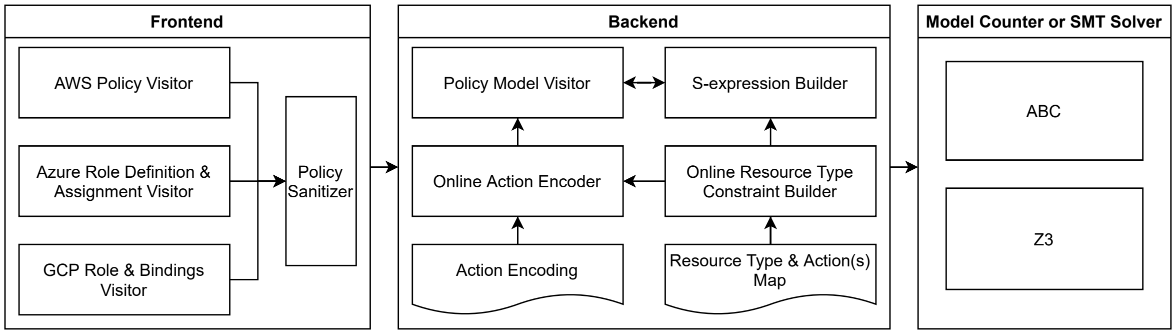

Frontend
AWS Policy Visitor
Azure Role Definition & Assignment Visitor
GCP Role & Bindings Visitor
Policy Sanitizer
Backend
Policy Model Visitor
S-expression Builder
Online Action Encoder
Online Resource Type Constraint Builder
Action Encoding
Resource Type & Action(s) Map
Model Counter or SMT Solver
ABC
Z3

Figure 1: Architecture of Quacky (online)

Table 1: The four relative permissiveness outcomes based on the model counts of formulas $[\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!]$ and $[\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!]$

	$\lvert [\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!] \rvert = 0$	$\lvert [\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!] \rvert > 0$
$\lvert [\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!] \rvert = 0$	$\mathbb{P}1, \mathbb{P}2$ equivalent	$\mathbb{P}1$ less permissive
$\lvert [\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!] \rvert > 0$	$\mathbb{P}1$ more permissive	$\mathbb{P}1, \mathbb{P}2$ incomparable

the requests intended by the user who wrote it. If it allows more requests than intended, it is said to be an *overly permissive* policy.

3.1 Quantitative Permissiveness Analysis

The goal in permissiveness analysis is to determine what requests are allowed by a policy, or, in the case of multiple policies, to check if one policy is more permissive than the other. This can be done by encoding policies as logic formulas [3, 6, 7, 9] whose satisfying solutions represent the requests it allows. Put more concretely, the SMT encoding of $\mathbb{P}$, or $[\![\mathbb{P}]\!]$, represents the set of requests $\mathbb{P}$ allows, where a satisfying solution to $[\![\mathbb{P}]\!]$ represents a request $\mathbb{P}$ allows. The permissiveness of $\mathbb{P}$, given by $\lvert [\![\mathbb{P}]\!] \rvert$, is the number of solutions to $[\![\mathbb{P}]\!]$, which equals the number of distinct requests allowed by $\mathbb{P}$. In other words, to quantify the permissiveness of $\mathbb{P}$, it suffices to count the number of solutions to $[\![\mathbb{P}]\!]$.

The relative permissiveness between a pair of policies $\mathbb{P}_1$ and $\mathbb{P}_2$ can be determined by comparing the sets of requests allowed by each policy. This can be done by checking the satisfiability of two SMT formulas, namely $[\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!]$ and $[\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!]$. The formula $[\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!]$, logically equivalent to $[\![\mathbb{P}_1]\!] \wedge \neg [\![\mathbb{P}_2]\!]$, represents the set of requests allowed by $\mathbb{P}1$ but not $\mathbb{P}2$ (vice versa for $[\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!]$). Relative permissiveness can be quantified by counting the number of solutions to both formulas, which yields one of four outcomes, as shown in Table 1. For example, $\mathbb{P}_1$ is *less permissive* than $\mathbb{P}_2$ if the set of requests allowed by $\mathbb{P}_1$ is a proper subset of the set of requests allowed by $\mathbb{P}_2$; in this case, the number of solutions $\lvert [\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!] \rvert$ quantifies how much more permissive $\mathbb{P}_2$ is than $\mathbb{P}_1$, or equivalently, how many more requests $\mathbb{P}_2$ allows than $\mathbb{P}_1$. Note that if $\mathbb{P}_1$ and $\mathbb{P}_2$ are incomparable, then the permissiveness of each policy by itself can still be compared using $\lvert [\![\mathbb{P}_1]\!] \rvert$ and $\lvert [\![\mathbb{P}_2]\!] \rvert$.

4 QUACKY

Figure 1 shows the core framework of QUACKY. QUACKY takes in policies written in the AWS IAM, Microsoft Azure, or GCP policy languages into its *frontend*, which encodes policies into an intermediate policy model. The *backend* translates the policy model into one or more SMT formulas, depending on whether the analysis is on a single policy or on multiple policies. The *solver* component analyzes the SMT formulas through queries to a constraint solver or model counter and outputs the desired permissiveness result. The analysis is supplemented by an *offline resource type constraint generator*, shown in Figure 3, which prepares resource type constraints for the SMT formulas (discussed in more detail below).

```
{"Statement": [{
  "Effect": "Allow",
  "Principal": "*",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::myexamplebucket/*",
  "Condition": {
    "StringLike": {
      "aws:userId": ["AWSUSERID:*", "JOHNDOE1111"]}}}]}
```

```
; Resource: p0.s0.r
(declare-const p0.s0.r Bool)
(assert (= p0.s0.r (or
    (in resource /arn:aws:s3:::myexamplebucket\/.*/))))

; Condition: p0.s0.cStringLikeaws.userId
(declare-const p0.s0.cStringLikeaws.userId Bool)
(assert (= p0.s0.cStringLikeaws.userId (and aws.userId.exists (or
    (in aws.userId /AWSUSERID:.*/) (= aws.userId "JOHNDOE1111")))))
```

Figure 2: Sample policy (top) and a snippet from its SMT encoding (bottom)

QUACKY Frontend. The frontend takes access control policies as input and outputs instances of our formal policy model, implemented as tree data structures. The input depends on the cloud provider. For AWS, the input is 1 or 2 policies, saved as serialized JSON. Figure 2(a) shows an example policy written in the AWS IAM policy language. The AWS Policy Visitor checks and makes sure the JSON files are well-formed. For Azure, the input is 1 or 2 pairs of *role definitions* and *role assignments*, which are also JSON. The Role Definition and Assignment Visitor opens the files and checks if they are well-formed. If a role definition and a role assignment both refer to the same `RoleId`, the visitor joins them on that role ID, producing an AWS-like policy. GCP's input is similar to that of Azure, except its versions of role definitions and role assignments are called *roles* and *role bindings*, respectively. The Role and Bindings Visitor joins the role(s) with the role binding(s) on any common role name(s),

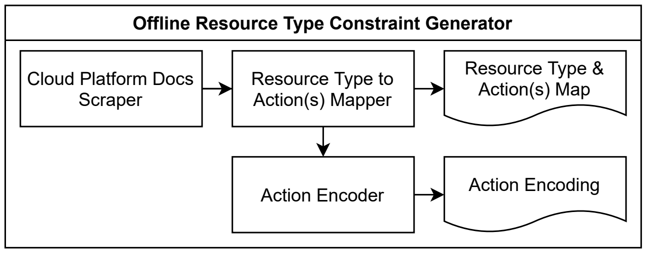

Offline Resource Type Constraint Generator
Cloud Platform Docs Scraper
Resource Type to Action(s) Mapper
Resource Type & Action(s) Map
Action Encoder
Action Encoding

Figure 3: Architecture of Quacky (offline)

```
(assert (and
    (in resource /arn:aws:ec2:...:instance\/i-[0-9a-f]{17,17}/)
    (or (= action "ec2:associateaddress")
        (= action "ec2:associateiaminstanceprofile")
        (= action "ec2:attachclassiclinkvpc") ... )))
```

```
(assert (and
    (in resource /arn:aws:ec2:...:instance\/i-[0-9a-f]{17,17}/)
    (and (>= action 4066) (<= action 4106))))
```

Figure 4: Snippet from the resource type constraint for EC2 instances without (top) and with (bottom) action encoding

producing an AWS-like policy. Visitors' outputs are passed to the Policy Sanitizer, which rewrites keys and action values in lowercase and replaces scalar values with lists. Then, the frontend transforms each policy into a tree by doing a post-order, depth-first traversal of the JSON policy and constructing nodes at each key, such as `Policy` (root), `Statement`, `Principal`, etc.

QUACKY Backend. The backend takes in 1 or 2 trees representing policies. It outputs SMT formulas that encode the semantics of each policy. The Policy Model Visitor builds the SMT formula incrementally. It visits each node in the tree in a post-order, depth-first traversal. For each node, it appends a set of constraints to the SMT formula. These constraints are built by the S-expression Builder, which takes in operands and an operator and returns a constraint conforming to the SMT-LIB standard. Figure 2 shows a policy and its SMT encoding.

For a more precise analysis, the backend can add *resource type constraints*, which capture a cloud service provider's valid resource types, actions, and pairings thereof, to the formula. The Online Resource Type Constraint Builder builds constraints on valid resource type and action pairs. A map of each resource type to the actions operating on it is pre-built offline (discussed below). The Constraint Builder takes a set of actions from an `Action` node, reads the map, identifies the relevant types, and builds constraints on those types and their actions. An example is shown in Figure 4. Note that this process is *online*; that is, the constraints are built during translation, based on actions in the policy. Irrelevant constraints are not built, reducing the size and complexity of the SMT formula.

Adding resource type constraints may significantly slow down model counting. To mitigate it, the backend does *action encoding*. The Action Encoder replaces action names, which are strings, with a numeric encoding. The encoding is specified by a JSON map that is pre-built offline. Action encoding replaces constraints with disjunctions of action names with more compact constraints with ranges of numbers. An example is shown in Figure 4.

Model Counter. QUACKY uses the Automata Based model Counter (ABC) [1, 2], which can model count string and numeric constraints. ABC takes a SMT formula F as input, and it returns the number of models satisfying F, up to a bound k. It implements model counting by constructing automata for F and counting paths to accepting states of the automata. The SMT formulas produced by the QUACKY backend can be sent to other SMT-LIB-conformant constraint solvers. For example, Microsoft's Z3 [4, 8] can be used to get a model (i.e. an allowed request).

Offline Resource Type Constraint Generator. Figure 3 shows the *offline resource type constraint generator*. In the backend, the Online Resource Type Constraint Generator and Action Encoder depend on pre-built maps, as we discussed earlier. These are pre-built *offline* to avoid repeating work every time QUACKY is run. The valid resource type and action pairings are specified in the cloud service provider's documentation, which are scraped and processed into a JSON map. The Action Encoder assigns numbers to actions, where a set of actions for a given resource type is assigned to a contiguous set of numbers. This enables the online action encoder to build more compact range constraints.

4.1 Support for GCP Policies

We handle policies written in GCP's policy language by extending QUACKY's frontend, backend, and offline resource type constraint generator (see Figures 1 and 3). In the frontend, we implemented the GCP Role and Bindings Visitor, which specifies how roles and role bindings are transformed into the formal policy model. In the backend, we added routines to translate GCP-specific conditions to SMT-LIB. In the offline resource type constraint generator, we wrote a new scraper to get the GCP resource type constraints from GCP's online documentation, and we generated a new resource type and actions map and a new action encoding.

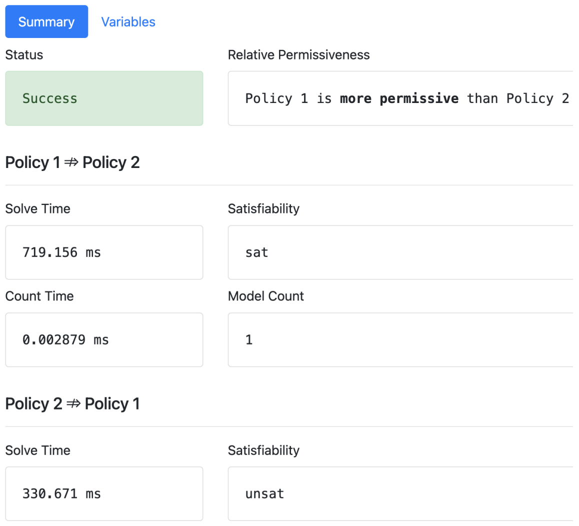
QUACKY can support other policy languages by further extending the aforementioned components. Note that the formal policy model need not be extended as long as the input(s) for that language can be transformed into the model.

4.2 Usage

QUACKY[1] has a command-line interface and a web-based interface[2]. QUACKY's command-line arguments include the input file name(s), the output file name(s), the model counting bound, and the timeout. There are flags to use resource type constraints, action encoding, and the PCRE regular expression syntax (which ABC can parse). For AWS, the input files are AWS policies; for Azure, the input files are role definitions and role assignments; for GCP, the input files are roles and role bindings. For all, the input files are in JSON, and the output files (the SMT formulas) are in SMT-LIB.

The QUACKY web app takes a subset of these arguments as input. The input form on the web app has textareas for policies, a number

[1]The tool's source code, policy datasets, experimental results, and documentation are publicly available at https://github.com/vlab-cs-ucsb/quacky
[2]The web app's source code and documentation are publicly available at https://github.com/vlab-cs-ucsb/quacky-web-app



Status: Success

Relative Permissiveness: Policy 1 is **more permissive** than Policy 2

Policy 1 ⇏ Policy 2

Solve Time: 719.156 ms

Satisfiability: sat

Count Time: 0.002879 ms

Model Count: 1

Policy 2 ⇏ Policy 1

Solve Time: 330.671 ms

Satisfiability: unsat

Figure 5: Results summary tab on the QUACKY web app

Table 2: Real AWS, Azure, and GCP policy analysis results. The average permissiveness and time, grouped by service, are reported. Permissiveness is in log scale.

Provider	Service	Avg. Perm.	Avg. Time (s)
AWS	EC2	3879.71	30.8
AWS	IAM	3721.92	0.96
AWS	S3	5787.7	2.3
Azure	Blob Storage	809.07	1.07
Azure	Virtual Machines	1047.15	5.4
GCP	Cloud Storage	1202.81	1.75
GCP	Compute Engine	1190.67	2.18

for bound, and checkboxes for the resource type constraints and action encoding flags. To reduce CPU, memory, and disk usage, the web app has a fixed timeout and does not store SMT formulas; consequently, there are no timeout or regex syntax arguments.

Both the command-line and web interfaces output satisfiability, solve time, model count, and count time for each SMT formula. Figure 5 is a screenshot of a results summary tab on the web app. In addition to the aforementioned outputs, it shows a status (success) and relative permissiveness. The variables tab (not shown) outputs the model counts for individual string and numeric variables, like `action`, `resource`, and `aws:userId`.

5 EVALUATION

We evaluated QUACKY using a dataset of 41 real AWS policies from forums, 5 Azure policies from Microsoft Docs, and 5 GCP policies from GCP documentation. We selected well-formed policies that varied from simple to complex. For all experiments, we used 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 [3].

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

Table 3: GCP policy analysis results. Each policy's permissiveness and each pair's relative permissiveness are reported. All numbers are in log scale (⊥ means the result was zero).

$\mathbb{P}_1$	$\mathbb{P}_2$	$\lvert[\![\mathbb{P}_1]\!]\rvert$	$\lvert[\![\mathbb{P}_2]\!]\rvert$	$\lvert[\![\mathbb{P}_1]\!] \not\Rightarrow [\![\mathbb{P}_2]\!]\rvert$	$\lvert[\![\mathbb{P}_2]\!] \not\Rightarrow [\![\mathbb{P}_1]\!]\rvert$
User Login	Admin Login	1190.44	1190.86	⊥	1188.86
Obj Creator	Obj Viewer	1201.07	1202.07	1201.07	1202.07
Obj Creator	Obj Admin	1201.07	1203.88	⊥	1203.65

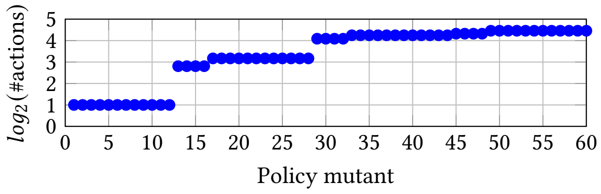


Figure 6: The number of actions allowed by each mutant that are not allowed by the original policy

To evaluate QUACKY's performance, we quantified the permissiveness of our original AWS, Azure, and GCP policies. We used a model counting bound of 250. The average permissiveness and analysis time, grouped by cloud service, are shown in Table 2. For most services, the average time was on the order of a few seconds. The exception was AWS Elastic Compute Cloud (EC2), which generally has the most complex real-world policies and resource type constraints.

Table 3 shows a closer look at QUACKY's results for GCP's Storage and Compute services. We can see that the OS admin login policy is more permissive than the OS user login policy, where the former allows $2^{1188.86}$ distinct requests that the latter does not. Moreover, object admin is more permissive than object creator by $2^{1203.65}$ distinct requests. Object viewer is incomparable to object creator, but individually, the former allows more requests than the latter. These results make sense intuitively; we expect admins to have absolutely more permissions than regular users, whereas we expect object creators and object viewers to each have permissions that the other does not, according to the GCP documentation.

To demonstrate the usefulness of quantitative permissiveness analysis, we mutated an original AWS policy to make 64 mutants. Figure 6 shows the number of actions 60 mutants allowed that the original policy denied (4 mutants are not shown because they were equivalent to the original). By quantifying relative permissiveness, we see that the mutants shown allow anywhere between 2 and 22 more actions than the original. Without quantitative analysis, all mutants shown would simply be classified as "more permissive" than the original, which is less insightful to policy authors.

6 CONCLUSION

We presented the QUACKY tool for quantifying permissiveness of access control policies in the cloud. We showed that QUACKY can handle a variety of policies written in the most popular cloud policy languages. In the future, we aim to investigate how QUACKY can be used to quantify properties of access control policies other than permissiveness.

REFERENCES

[1] 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

[2] 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.

[3] 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.

[4] 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

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

[6] 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)*.

[7] Graham Hughes and Tevfik Bultan. 2008. Automated Verification of Access Control Policies Using a SAT Solver. *Int. J. Softw. Tools Technol. Transf.* 10, 6 (Dec. 2008), 503–520. https://doi.org/10.1007/s10009-008-0087-9

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

[9] 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.

[10] 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/.