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.


CCS CONCEPTS
- Security and privacy → Logic and verification: Access control.

ACM Reference Format:

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 [??]. 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
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 \([P_1] \not\Rightarrow [P_2]\) whose satisfying solutions represent the requests it allows. Put more concretely, the SMT encoding of \([P_1]\) or \([P_2]\), represents the set of requests \(P\) allows, where a satisfying solution to \([P_1]\) represents a request \(P\) allows. The permissiveness of \(P\), given by \([P_1] \not\Rightarrow [P_2]\), is the number of solutions to \([P_2]\), which equals the number of distinct requests allowed by \(P\). In other words, to quantify the permissiveness of \(P\), it suffices to count the number of solutions to \([P_2]\).

The relative permissiveness between a pair of policies \(P_1\) and \(P_2\) 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 \([P_1] \not\Rightarrow [P_2]\) and \([P_2] \not\Rightarrow [P_1]\). The formula \([P_1] \not\Rightarrow [P_2]\), logically equivalent to \([P_1] \not\Rightarrow [P_2]\), represents the set of requests allowed by \(P_1\) but not \(P_2\) (vice versa for \([P_2] \not\Rightarrow [P_1]\)). 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, \(P_1\) is less permissive than \(P_2\) if the set of requests allowed by \(P_1\) is a proper subset of the set of requests allowed by \(P_2\); in this case, the number of solutions \([P_1] \not\Rightarrow [P_2]\) quantifies how much more permissive \(P_2\) is than \(P_1\), or equivalently, how many more requests \(P_2\) allows than \(P_1\). Note that if \(P_1\) and \(P_2\) are incomparable, then the permissiveness of each policy by itself can still be compared using \([P_1]\) and \([P_2]\).

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).

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

Table 1: The four relative permissiveness outcomes based on the model counts of formulas \([P_1] \not\Rightarrow [P_2]\) and \([P_2] \not\Rightarrow [P_1]\)

<table>
<thead>
<tr>
<th>Outcome</th>
<th>([P_1] \not\Rightarrow [P_2])</th>
<th>([P_2] \not\Rightarrow [P_1])</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1 more permissive</td>
<td>(1)</td>
<td>(0)</td>
</tr>
<tr>
<td>F1 less permissive</td>
<td>(0)</td>
<td>(1)</td>
</tr>
<tr>
<td>F1 incomparable</td>
<td>(0)</td>
<td>(1)</td>
</tr>
<tr>
<td>F1 comparable, F2 incomparable</td>
<td>(1)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

...
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-0a-f(17,f))
 (or (= action "ec2: associateaddress")
 (action "ec2: attachclassiclinkvpc") ... ))}

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

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

Quacky: Quantitative Access Control Permissiveness Analyzer

Quacky Backend. The backend 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] 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.
We evaluated 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 2188 distinct requests that the latter does not. Moreover, object admin is more permissive than object creator by 21203 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 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 other does not, according to the GCP documentation.

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 $2^{188}$ distinct requests that the latter does not. Moreover, object admin is more permissive than object creator by $2^{1203}$ 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 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 other does not, according to the GCP documentation.

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 $2^{188}$ distinct requests that the latter does not. Moreover, object admin is more permissive than object creator by $2^{1203}$ 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 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.

Table 6 CONCLUSION

We presented the QUACKY tool for quantifying permissiveness of access control policies in the cloud. We showed that QUACKY can

---

3The tool’s source code, policy datasets, experimental results, and documentation are publicly available at https://github.com/vlab-cs-ucsb/ABC

4https://github.com/vlab-cs-ucsb/ABC

5https://github.com/vlab-cs-ucsb/ABC
Figure 6: The number of actions allowed by each mutant that are not allowed by the original policy

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