Block Public Access: Trust Safety Verification of Access Control Policies

Malik Bouchet
Amazon, USA

Byron Cook
Amazon, USA

Bryant Cutler
Amazon, USA

Anna Druzkina
Amazon, USA

Andrew Gacek
Amazon, USA

Liana Hadarean
Amazon, USA

Ranjit Jhala
Amazon, USA

Brad Marshall
Amazon, USA

Dan Peebles
Amazon, USA

Neha Rungta
Amazon, USA

Cole Schlesinger
Amazon, USA

Chriss Stephens
Amazon, USA

Carsten Varming
Amazon, USA

Andy Warfield
Amazon, USA

ABSTRACT

Data stored in cloud services is highly sensitive and so access to it is controlled via policies written in domain-specific languages (DSLs). The expressiveness of these DSLs provides users flexibility to cover a wide variety of uses cases, however, unintended misconfigurations can lead to potential security issues. We introduce Block Public Access, a tool that formally verifies policies to ensure that they only allow access to *trusted* principals, *i.e.* that they prohibit access to the general public. To this end, we formalize the notion of Trust Safety that formally characterizes whether or not a policy allows unconstrained (public) access. Next, we present a method to compile the policy down to a logical formula whose unsatisfiability can be (1) checked by SMT and (2) ensures Trust Safety. The constructs of the policy DSLs render unsatisfiability checking PSPACE-complete, which precludes verifying the millions of requests per second seen at cloud scale. Hence, we present an approach that leverages the structure of the policy DSL to compute a much smaller residual policy that corresponds *only* to untrusted accesses. Our approach allows Block Public Access to, in the common case, syntactically verify Trust Safety without having to query the SMT solver. We have implemented Block Public Access and present an evaluation showing how the above optimization yields a low-latency policy verifier that the S3 team at AWS has integrated into their authorization system, where it is currently in production, analyzing millions of policies everyday to ensure that client buckets do not grant unintended public access.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409728

CCS CONCEPTS

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

KEYWORDS

access policies, cloud security, SMT, formal methods, cloud storage

ACM Reference Format:
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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3368089.3409728

1 INTRODUCTION

In the cloud computing era, clients ranging from banks to hospitals to commercial and government entities store and access their sensitive data via remote services managed by third-party providers. These clients elect to do so because centralized services are fast, inexpensive, reliable, available, and can scale up to serve the data to as many users as needed. For example, S3 is a high-performance, high-availability cloud object storage service offered by AWS that allows clients to create *buckets* within which they can store their data as S3 *objects*.

DSLs for Access Control Policies Much of the client data stored in such services is highly sensitive, and should *only be accessible* to the organizations to whom the data belongs. Even within those organizations, there may be strict *policies* governing which set of *principals* should be granted or denied access to different kinds of data. To this end, services like S3 also provide their clients a domain specific language (DSL) in which the clients can precisely specify access control policies as per their own requirements. As in many other settings, the policy DSL was originally designed to grant a single user access to a specific resource, but has, over time, evolved to support the myriad use cases that arise in large client corporations.

```
{(Allow,
  Principal   :  "arn:aws:iam::123456789012:role/dev",
  Action      :  "s3:*",
  Resource    :  "arn:aws:s3:::my-bucket/*"),
 (Allow,
  Principal   :  "*",
  NotAction   :  "s3:DeleteObject",
  Resource    :  "arn:aws:s3:::my-bucket/*")}
```

Figure 1: Misconfigured S3 access control policy.

For example, instead of specifying just a single user, the DSL has ways to describe sets of users and express hierarchical relationships between *organizations*, *groups*, *roles* and *users*. Similarly, instead of describing just buckets, there are primitives for describing the detailed structure of buckets, organized as *hierarchical* directory structures.

Who Polices the Policies? The flexibility provided by the DSL is crucial to support the diverse use-cases of different clients, however, unintentional misconfigurations can lead to potential security issues. For example, consider the policy in Figure 1 that is based on a real user policy adapted to S3 [20]. The policy consists of two statements. The first statement allows a "dev" role to execute any S3 action on any object in the "my-bucket" bucket. The second statement allows all actions *not-equal-to* (*i.e.* other than) s3:DeleteObject on any objects in the same bucket. The client intended their policy to allow the "dev" role to perform any action *except* s3:DeleteObject. The result, however, was a policy that allows *everyone* to access data in the bucket, as the second *allow* statement *does not* restrict access, but instead *gives* access to all principals. (In this case, the correct mechanism to restrict access as per the client's intent would be to change the *allow* in the second statement to a *deny* and the *not-equals-action* clause to an *equals-action*.) While the policy DSL allows for compact specifications of expressive policies, reasoning about the interaction between the semantics of different statements can be challenging to manually evaluate especially in large policies with multiple operators and conditions.

Block Public Access In this paper, we introduce Block Public Access, a tool that formally verifies policies to ensure that they only allow access to *trusted* principals, *i.e.* that they prohibit access to the general public. We develop Block Public Access via the following contributions.

1. Trust Safety Our first contribution is the notion of Trust Safety that formally characterizes whether or not a policy allows unconstrained (public) access § 3. The definition of Trust Safety varies on each use case which can have an implicit set of trusted entities outside of whom access should not be granted. Some policies may consider any access beyond a trusted part of the network to be public, while others may wish to restrict access to certain individuals regardless of location. At cloud scale, manually inspecting or annotating policies to specify trusted entities is infeasible. Likewise, asking clients to provide this set of trusted entities is not an option, as it imposes a significant burden on users by requiring them to annotate legacy policies. We address this problem by presenting an algorithm to *infer* the set of trusted entities from an access control policy (§ 4.3),

2. Semantic Verification Our second contribution is an algorithm to automatically verify whether a given policy is Trust Safe § 4. Our algorithm builds on work that encodes the semantics of policies as *logical constraints* that soundly and precisely characterize the sets of requests that are granted access by the policy [1]. We show how to compose this encoding with additional constraints that determine whether or not the policy is Trust Safe. This allows us to compile the policy down to a single logical formula, whose unsatisfiability, determined by an SMT solver [2, 6, 19], implies that the policy is Trust Safe. The rich policy language – which allows conditions with wildcards ranging over user-names, IP addresses, and sets of users, roles, and resources – yields SMT formulas that fall in the combined theories of strings, regular expressions, bit-vectors, and arithmetic. Checking the satisfiability of formulas over such theories is a PSPACE-complete problem [5]. The formulas encoding real-world policies are often large and complex, making satisfiability solving slow.

3. Syntactic Trust-Safety Verification Any solution that prevents public access must be part of the authorization system itself and therefore has to be extremely fast. (For example, S3 handles millions of requests per second, all of which need to be checked to ensure they are not gaining access based on a misconfigured policy.) Thus, our third contribution is an approach that efficiently rewrites the formulas to eliminate all the rules that allow *trusted* access, leaving behind a much smaller *residual* policy that corresponds only to untrusted accesses § 5. In the majority of cases, the residual policy is trivial, *i.e.* prohibits all (untrusted) accesses, allowing us to verify Trust Safety without even querying the SMT solver! In the remaining cases, the formula is drastically smaller, thus enabling efficient SMT-based verification.

4. Evaluation and Deployment Our final contribution is an implementation of Block Public Access that ensures that all created and updated policies are Trust Safe § 6. Our implementation ensures that untrusted public access to S3 buckets is not allowed *anytime* during the bucket's lifecycle. To this end, we introduce two additional checks in the authorization process: one when attaching policies and one when requesting bucket access. When a user tries to attach a new policy to a bucket, *e.g.* the policy from Figure 1, S3 invokes Block Public Access to check if the policy is Trust Safe. If this is not the case, S3 fails to attach the policy and returns a warning. Second, when a user requests access to a bucket, Block Public Access uses a cache to determine whether the policy attached to the bucket was Trust Safe, and if not, the S3 authorization mechanism denies the request based on Block Public Access's result. Thus, to be integrated as a blocking check in the request path of the S3 service, Block Public Access's checks must have very *low latency*. We present an evaluation that shows that the syntactic rewrite enables precisely such low-latency verification without compromising *soundness* § 7. Consequently, the S3 team at AWS has integrated Block Public Access into their authorization system, where it is currently in production enabling S3 to guard client buckets from unintended public access.

2 S3 ACCESS POLICIES

We begin with an overview of some of the features of the IAM policy language that highlights the features that make checking Trust Safety challenging. S3 is an object store where a logical unit of storage is called a *bucket*. S3 stores data as objects in these buckets. Each resource, *e.g.* the bucket and the objects in the bucket, is uniquely identified by an Amazon Resource Name (ARN). Access to S3 buckets and other AWS resources is granted via Identity & Access Management System (IAM) policies, which are attached to the bucket or resource, and are used to control access to the bucket and the objects in it.

Request Contexts When an access request is made to an AWS service, a *request context* is generated which includes the *principal* making the request, the *resource* being requested, and the *action* being invoked. The request context also contains a set of service-specific key-value pairs that can be referenced in policy conditions. The IAM policy evaluation engine compares the request context with the policies for the user and the resource to determine if access is granted or denied. S3 uses this evaluation engine to make sure only authorized requests are serviced. The IAM policy evaluation engine is called on all requests so it must only add a very small overhead. For example, the IAM engine would allow the following request context under the policy in Figure 1:

Principal	:	"arn:aws:iam::999999999999:user/malicious"
Action	:	"s3:PutObject"
Resource	:	"arn:aws:s3:::my-bucket/dir/filename"
SourceIp	:	"192.0.2.3"

(1)

The AWS policy language is expressed as serialized JSON, and hence, the request context can contain extra keys not present in the policy.

Policies Figure 2 summarizes the core constructs of the AWS policy language in a simplified abstract syntax. A *policy* comprises a set of **Allow** and **Deny** statements that respectively grant or block access to sets of users. A request is only granted access if the policy contains an **Allow** statement that matches the request, and does not contain any **Deny** statements that match it (regardless of the order in which the statements appear in the policy.) When writing example policies we will elide set brackets for singleton sets, and elide the **Condition** section when it is empty. The policy language has grown to be flexible enough to support all current AWS services as well as future ones and extensions. This flexibility stems from two crucial features.

Wildcards First, allowing *wildcards* in the *Principal*, *Resource* and *Action* constructs. Wildcards can range over (possibly unboundedly) many values and hence, let users concisely specify permissions at a fine level of granularity. For example the user can refer to all S3 actions as "s3:*" or use "arn:aws:s3:::my-bucket/*.html" to refer to all HTML files in a given bucket.

Conditions Second, the *Condition* construct over key-value pairs, allows specifying a set of logical conditions under which access is granted or denied. Conditional expressions are constructed using *Operators* on key-value pairs. These operators can, for example, compare dates (DateLessThan), check IP ranges (IpAddress), do case-insensitive string matching (StringEqualsIgnoreCase) and also match regular expressions (StringLike) over a set of candidate values. If any of these values match the key, the condition evaluates to true. Additionally, the negation (Not), quantification (ForAllValues) and optional check (IfExists) qualifiers can suitably extend the operator's semantics.

Reasoning about Policies Wildcards and conditionals – which are the key to the flexibility of the policy language – combine to make manually reasoning about policies challenging. For example, consider:

(**Allow**,

Principal	:	"*",
Action	:	"s3:GetObject",
Resource	:	"arn:aws:s3:::my-bucket/*",
Condition	:	{(IpAddress, **SourceIp**, "192.0.2.0/24"), (StringNotLike, **SourceIp**, "192.?.*.*")})

This policy will allow "s3:GetObject" requests if the source IP address matches both conditions. The source IP must be in the "192.0.2.0/24" range but at the same time not match the "192.?.*.*" regular expression, where "?" stands for any one character, and "*" for one or more arbitrary characters. All IP addresses in this range will match the regular expression, so this allow statement will deny all requests. Automatically inferring this fact, requires combining bit-vector constraint solving for the "IPAddress", a NP-complete problem, with string constraints for "StringNotLike", an even harder PSPACE-complete problem [17]. While this is obviously a crafted example, large organizations often write complex policies hundreds of lines long that are similarly challenging to reason about.

Service Specific Constraints To precisely analyze policies, we need to also take into account constraints that are specific to different *services* using AWS. Each service publishes the custom set of actions, resource and condition key combinations they support. Certain condition keys are defined globally across all services, *e.g.*, **SourceIp**, while other condition keys are service specific, *e.g.* **s3:prefix**. There are currently 24 global context keys and S3 has 26 additional service-specific context keys. Some keys are not always present, some keys are only compatible with certain actions and some key combinations are not compatible. For example the **s3:prefix** key only exists for s3:ListBucket action requests, and it limits the response of s3:ListBucket to object key names with a specific prefix.

3 TRUST SAFETY

Next, let us formalize what it means for a policy to be Trust Safe, *i.e.* whether the policy blocks all requests coming from *untrusted* entities. Specifically, we delineate exactly what aspects of policies and requests are trusted and hence what constitutes an untrusted public access.

3.1 Motivating Examples

Recall that S3 is an object store where a logical unit of storage is called a *bucket*. S3 stores data as objects in these buckets. Each resource, *e.g.* the bucket and the objects in the bucket, is uniquely identified by an Amazon Resource Name (ARN). The *bucket policy*, which is attached to the bucket, controls access to the bucket and the

$$
\begin{aligned}
\mathit{Policy} &\rightarrow \{\mathit{Statement}\} \\
\mathit{Statement} &\rightarrow (\mathit{Effect}, \mathit{Principal}, \mathit{Action}, \mathit{Resource}, \{\mathit{Condition}\}) \\
\mathit{Effect} &\rightarrow \textbf{Allow} \mid \textbf{Deny} \\
\mathit{Principal} &\rightarrow (\textbf{Principal} \mid \textbf{NotPrincipal}) : \{\mathit{string}\} \\
\mathit{Action} &\rightarrow (\textbf{Action} \mid \textbf{NotAction}) : \{\mathit{string}\} \\
\mathit{Resource} &\rightarrow (\textbf{Resource} \mid \textbf{NotResource}) : \{\mathit{string}\} \\
\mathit{Condition} &\rightarrow \textbf{Condition} : \{\mathit{Operator}\} \\
\mathit{Operator} &\rightarrow (\mathit{OpName}, \mathit{KeyName}, \{\mathit{Value}\}) \\
\mathit{OpName} &\rightarrow \text{Null} \mid \text{StringEquals} \mid \text{StringEqualsIfExists} \mid \\
&\quad \text{ForAllValues:StringEquals} \mid \\
&\quad \text{ForAnyValue:StringEquals} \mid \\
&\quad \text{StringLike} \mid \text{StringNotEquals} \mid \text{IpAddress} \mid \ldots \\
\mathit{KeyName} &\rightarrow \textbf{SourceVpc} \mid \textbf{SourceIp} \mid \textbf{s3:prefix} \mid \ldots \\
\mathit{Value} &\rightarrow \mathit{string} \mid \mathit{num} \mid \mathit{bool}
\end{aligned}
$$

Figure 2: Core Syntax of the AWS Policy Language

```
{(Allow,
  Principal   : {"arn:aws:iam::123456789012:role/dev",
                 "arn:aws:iam::123456789012:role/support"},
  Action      : "s3:GetObject",
  Resource    : "arn:aws:s3:::my-bucket/*"),
 (Allow,
  Principal   : *,
  Action      : {"s3:GetObject", "s3:PutObject"},
  Resource    : "arn:aws:s3:::my-bucket/*",
  Condition   : (StringEquals, Username, "admin")),
(Deny,
  Principal   : "*",
  Action      : "s3:*",
  Resource    : "arn:aws:iam::my-bucket/accounts/*",
  Condition   : (StringNotEquals, SourceVpc, "vpc-abcdef"))}
```

(a) Policy $\mathcal{P}_1$ is not Trust Safe: allows untrusted access

```
{(Allow,
  Principal   : *,
  Action      : "s3:*",
  Resource    : "arn:aws:s3:::my-bucket/*")
(Deny,
  Principal   : "*",
  Action      : "s3:*",
  Resource    : "arn:aws:s3:::my-bucket/*",
  Condition   : (StringNotEquals, PrincipalOrgID, "o-1234"))}
```

(b) Policy $\mathcal{P}_2$ is Trust Safe: prohibits untrusted access

Figure 3: Example policies.

objects in it. Figure 3 shows two example policies in the simplified abstract syntax for S3, that we will use to motivate our definition of Trust Safety.

Example 1: Not Trust Safe Consider policy $\mathcal{P}_1$ from Figure 3(a). The first statement (**Allow**) gives bucket read access to the "dev" and "support" roles. The second statement (**Allow**) gives read and write access to all principals if the username is "admin". The final statement (**Deny**) restricts access to "my-bucket/accounts" only to requests originating from the "vpc-abcdef" virtual private cloud[1].

This policy allows untrusted entities access to certain resources. Consider the following request context:

```
Principal   :  "arn:aws:iam::999999999999:user/admin"
Action      :  "s3:GetObject"
Resource    :  "arn:aws:s3:::my-bucket/secret/filename"
Username    :  "admin"
```
(2)

This request context will match the second allow statement. While there is a restriction on the **Username**, there is no restriction on the account containing the username. An adversary could create a user called "admin" in their own account and thus gain access to the bucket. While the **Username** key in the request is populated by the service from the user part of the **Principal** identifier, the value is controlled by the issuer of the request. The **Username** is not by itself sufficient to uniquely identify a request issuer. Note that the **Deny** statement does not apply to this request context, because the deny only applies to resources that match "arn:aws:s3:::my-bucket/accounts/*".

Example 2: Trust Safe The policy $\mathcal{P}_2$ in Figure 3(b) consists of one statement that allows any principal to execute any action on the bucket, and another that denies all requests not originating from the "o-1234" organization. This policy does not allow public access, because it only allows requests coming from the "o-1234" organization. The **PrincipalOrgID** is a unique identifier that cannot be duplicated or modified by the issuer of the request.

3.2 Requests and Policies

We start by defining requests and access control policies.

Keys and Values Let $\mathcal{K}$ denote the set of all request context keys, including global keys and S3-specific keys. Each key $k \in \mathcal{K}$ has a type Type(k) associated with it, representing the set of the values that can be associated with the key. For example, Type(SourceIP) = "IpAddress" and Type(s3:prefix) = "string".

Request Contexts A *request context* r is a set of key-value pairs mapping keys to their values, *i.e.*

$$r = \{(k, val) \mid k \in \mathcal{K} \text{ and } val \in \text{Type(k)}\}$$

where **principal**, **action** and **resource** are always present. Other request keys may be optionally present.

Allowed Requests Let $\mathcal{P}$ be a policy. We write Allow($\mathcal{P}$) to denote all the request contexts that are *allowed* by a given policy:

$$\text{Allow}(\mathcal{P}) = \{r \mid \text{policy } \mathcal{P} \text{ allows request context } r\}$$

3.3 Trusted Requests

As the above examples illustrate, in order to formalize Trust Safety, we must answer the question: given a request context comprising a set of key-value pairs, how do we know it is coming from a *trusted*

[1]Virtual private clouds are a feature in most public cloud offerings that provides isolation between different parts of the cloud through a private IP subnet or VLAN.

entity?[2] Not all request keys are created equal: some of the key values can be controlled by the issuer of the request while others are populated by the service. Recall the request context from (1). The values of the **Action** and the **s3:prefix** keys are controlled by the issuer of the request: they choose what action to call and how to configure it. On the other hand, the values of **Principal** and **SourceIP** are set by the *service* and are always the unique principal identifier of the request issuer and the request source IP, respectively. Similarly, in the above examples, we assumed that the virtual private cloud "vpc-abcdef" and the organization "o-1234" are trusted and that requests that can be proven to come from these sources are safe. Thus, intuitively, the keys that *cannot be altered* by the user are considered trusted. We will use this notion of trusted keys (and their values) to define Trust Safety.

Trusted Keys The *trusted key set* $\mathsf{Tr}^{\mathcal{K}}$ is a subset of $\mathcal{K}$ containing the set of keys in a request context that cannot be altered by the issuer of the request. Examples of trusted keys in $\mathsf{Tr}^{\mathcal{K}}$ include **SourceVpc**, **SourceIp** and **UserId**. The **Username** key is not a trusted key because, as we have seen in Example 1 above, one can create a user with any name in a different account. The set of trusted keys for a given service requires domain expertise, but only needs to be determined once.

Trusted Values Let $\mathsf{Tr}^{\mathcal{V}}(k)$ be a given set of trusted values for key k, where a *trusted value* corresponds to an entity the author of the policy trusts. For policy $\mathcal{P}_1$ in Example 1, the set of trusted values is $\mathsf{Tr}^{\mathcal{V}}$(**SourceVpc**) = {"vpc-abcdef"}. For policy $\mathcal{P}_2$ in Example 2, the trusted value set is $\mathsf{Tr}^{\mathcal{V}}$(**PrincipalOrgID**) = {"o-1234"}. We will use these sets of trusted values to define a trusted context.

Definition 1 (Trusted Request Context)**.** A request context r is *trusted* for a given set of trusted values $\mathsf{Tr}^{\mathcal{V}}(k)$ if there exists $(k, val) \in r$ such that $k \in \mathsf{Tr}^{\mathcal{K}}$ and $val \in \mathsf{Tr}^{\mathcal{V}}(k)$, and r is *untrusted* otherwise.

In other words, if the request context contains at least one trusted key—*i.e.* one that cannot be modified by the issuer of the request—and the value associated with this key is trusted, then the request is considered trusted.

Definition 2 (Trust Safety)**.** A policy $\mathcal{P}$ is Trust Safe if there *does not exist* an untrusted request context $r \in \mathsf{Allow}(\mathcal{P})$.

That is, a policy is Trust Safe if it *only* allows trusted requests, or dually, if it blocks all untrusted (public) accesses.

4 VERIFYING TRUST SAFETY

Next, we will use the definition of Trust Safety to construct an SMT formula that is unsatisfiable iff the policy is Trust Safe, thereby yielding an algorithm to automatically verify whether a policy is Trust Safe. We develop our algorithm in three steps. First, we build up a semantic representation of a policy, *i.e.* a logical formula for which each satisfiable assignment represents an allowed request context. Second, we *assume* that we are provided with a set of trusted entities, and we show how to combine this set with the policy to obtain a logical formula that is unsatisfiable if and only if the policy is Trust Safe (§ 4.2). Third, we present our technique

$$
\begin{aligned}
S_1 \;\doteq\; & (p = \text{"arn:aws:iam::123456789012:role/dev"} \vee \\
& \quad p = \text{"arn:aws:iam::123456789012:role/support"}) \wedge \\
& \quad a = \text{"s3:GetObject"} \wedge r = \text{"arn:aws:s3:::my-bucket/*"} \\
S_2 \;\doteq\; & (a = \text{"s3:PutObject"} \vee a = \text{"s3:GetObject"}) \wedge \\
& \quad r = \text{"arn:aws:s3:::my-bucket/*"} \wedge \\
& \quad \mathit{username}^{\exists} \wedge \mathit{username} = \text{"admin"} \\
S_3 \;\doteq\; & a = \text{"s3:*"} \wedge \\
& \quad r = \text{"arn:aws:s3:::my-bucket/accounts/*"} \wedge \\
& \quad (\neg \mathit{sourceVpc}^{\exists} \vee \mathit{sourceVpc} \neq \text{"vpc-abcdef"})
\end{aligned}
$$

$$\mathsf{Allow}(\mathcal{P}_1) \;\doteq\; (S_1 \vee S_2) \wedge \neg S_3$$

Figure 4: SMT encoding of $\mathcal{P}_1$

for *inferring* a set of trusted entities automatically from the policy description, which together with the logical encoding lets us use SMT solvers to automatically determine whether a policy is Trust Safe.

4.1 A Logical Encoding of Policy

Zelkova [1] is an SMT-based checker for IAM access control policies that can answer questions about policies by (1) converting the policies into a logical representation and (2) using SMT solvers to answer queries about interesting properties of the policies. For example, Zelkova can check whether one policy is more permissive than another. Block Public Access uses Zelkova to generate an SMT formula corresponding to $\mathsf{Allow}(\mathcal{P})$, where satisfiable assignments are in one-to-one correspondence with request contexts $r \in \mathsf{Allow}(\mathcal{P})$.

Figure 4 (resp. figure 5) shows the encoding corresponding to $\mathsf{Allow}(\mathcal{P}_1)$ (resp. $\mathsf{Allow}(\mathcal{P}_2)$). The encoding for each policy is a formula over the variables p, a, and r that correspond to the principal, action, and resource in the request context. In addition to these variables, the encoding also introduces a variable for every condition key referenced in the policy. In the above example these are *sourceVpc*, *username* and *principalOrg*. Because some condition keys are optional, we introduce an additional boolean variable $key^{\exists}$ which is true if the key is present in the context. The set of all satisfying assignments of the formula corresponds to the set of all authorization contexts that will be allowed by the policy.

For $\mathcal{P}_1$, we have two allow statements, S_1 and S_2, and one deny statement S_3. The policy allows a request if either of the two allow statements match, *i.e.* $S_1 \vee S_2$ is satisfied, and the deny statement does not match ($\neg S_3$). The policy is represented by the conjunction of the encoding of the allow statements and the negation of the deny statement. Thus the policy allows access for requests matching any of the allow statements but that do not match any of deny statements.[3] The encoding of $\mathcal{P}_2$ follows a similar pattern.

While we use Zelkova for policy translation, we do not use the public check mentioned in [1]. That check is not formally defined,

[2]Note that we assume the request is *authenticated*: our goal is to ensure it is *authorized* to perform the requested action.

[3]Note that we are abusing notation to say a = "s3:*" since this, in fact, will correspond to a form of string matching rather than equality.

$$S_1 \doteq a = \text{"s3:*"} \wedge r = \text{"arn:aws:s3:::my-bucket/*"}$$
$$S_2 \doteq a = \text{"s3:*"} \wedge r = \text{"arn:aws:s3:::my-bucket/*"} \wedge (\neg principalOrg^{\exists} \vee principalOrg \neq \text{"o-1234"})$$

$$\mathsf{Allow}(\mathcal{P}_2) \doteq S_1 \wedge \neg S_2$$

Figure 5: SMT encoding of $\mathcal{P}_2$

and furthermore it was based on a fixed notion of public access not influenced by the policy under consideration. Instead, we develop a logical encoding of Trust Safety that can be compared against the SMT encoding of Allow($\mathcal{P}$).

4.2 A Logical Encoding of Trust Safety

Given a policy $\mathcal{P}$, a set of trusted keys $\mathsf{Tr}^{\mathcal{K}}$ and trusted values $\mathsf{Tr}^{\mathcal{V}}$ we can check whether a policy is Trust Safe by constructing a *verification condition* (VC), a first-order logic formula:

$$\mathsf{VC}(\mathcal{P}, \mathsf{Tr}^{\mathcal{K}}, \mathsf{Tr}^{\mathcal{V}}) \doteq \mathsf{Allow}(\mathcal{P}) \wedge \neg\mathsf{Trust}(\mathsf{Tr}^{\mathcal{K}}, \mathsf{Tr}^{\mathcal{V}}) \quad (3)$$

where Allow($\mathcal{P}$) describes *all* request contexts allowed by $\mathcal{P}$, and $\mathsf{Trust}(\mathsf{Tr}^{\mathcal{K}}, \mathsf{Tr}^{\mathcal{V}})$ describes *trusted* request contexts. We can then use an SMT solver to check the satisfiability of the formula (3). A satisfying assignment corresponds to an untrusted (public) access that is allowed, and hence, we reject $\mathcal{P}$ as not Trust Safe. Instead, if the formula is unsatisfiable, we can be sure that all allowed requests are trusted, and hence, we can accept the policy as Trust Safe.

Encoding the Trusted Requests Given the sets of trusted keys $\mathsf{Tr}^{\mathcal{K}}$ and their trusted values $\mathsf{Tr}^{\mathcal{V}}$, we generate the following formula that corresponds to the set of all *trusted* request contexts.

$$\mathsf{Trust}(\mathsf{Tr}^{\mathcal{K}}, \mathsf{Tr}^{\mathcal{V}}) \doteq \bigvee_{k \in \mathsf{Tr}^{\mathcal{K}}} \bigvee_{v \in \mathsf{Tr}^{\mathcal{V}}(k)} \left(var_k^{\exists} \wedge var_k = v \right)$$

where var_k denotes the variable in the SMT encoding corresponding to key **k**. Hence, if the VC formula (3) has any satisfying assignments then the policy allows untrusted (public) access, and the satisfying assignment can be used to compute an example of an untrusted request that would be allowed by the policy and can be presented to the user as an explanation.

4.3 Inferring Trusted Entities

The above procedure relies on having a given set of trusted values for the trusted keys. As discussed earlier (§ 3.3), the set of trusted keys can be determined once for each service. Hence, we need only determine the trusted values for each key. Instead of relying on users to provide a specification of trusted values, we must automatically infer the values that the user trusts. We achieve this with a syntactic analysis on the structure of the policy.

Extracting Trusted Values from Policies $\mathsf{Tr}^{\mathcal{V}}(key)(\mathcal{P})$ denotes the set of *trusted values* that can be inferred from policy $\mathcal{P}$. Intuitively, $\mathsf{Tr}^{\mathcal{V}}(key)(\mathcal{P})$ contains all the values of the *key* in the policy that cannot match an overly large set of values, for domain-specific definitions of "overly large." This set is computed syntactically from the policy. The following are examples of what we consider trusted key values for several keys:

$$\mathsf{Tr}^{\mathcal{V}}(\textbf{SourceVpc})(\mathcal{P}) = \{v \mid v \in \mathcal{P} \text{ is compared with } \textbf{SourceVpc} \text{ and } \text{"*"} \notin v \text{ and "?"} \notin v\}$$

$$\mathsf{Tr}^{\mathcal{V}}(\textbf{SourceArn})(\mathcal{P}) = \{v \mid v \in \mathcal{P} \text{ is compared with } \textbf{SourceArn} \text{ and } \text{"*"} \notin account(v) \text{ and "?"} \notin account(v)\}$$

For the **SourceVpc** key that stores the virtual private cloud identifier, we only trust values that do not contain any wildcard symbols. Because private cloud identifiers are assigned nondeterministically, any wildcard in this identifier will result in granting unintended access. However, other trusted identifiers may contain wildcards. Consider, for example, the Amazon Resource Name (ARN) that has the following structure:

arn:partition:service:region:account-id:type/resource

A wildcard in the region section of the ARN is safe, because a user may want to reference resources across multiple regions. A wildcard in the account-id section (denoted by *account(v)*) is not safe, because account IDs are nondeterministically assigned—even a wildcard replacing just one digit would allow public access, since the matched accounts are random. Coming up with the precise definition of $\mathsf{Tr}^{\mathcal{V}}(key)(\mathcal{P})$ for each **key** requires extensive domain-specific knowledge, but it need only be created once to be used many times.

Examples 1 & 2 Revisiting policy $\mathcal{P}_1$ in Figure 3(a), we infer the following set of trusted key value pairs:

{(**Principal**, "arn:aws:iam::123456789012:role/dev"),
(**Principal**, "arn:aws:iam::123456789012:role/support"),
(**SourceVpc**, "vpc-abcdef")}

Note that the (**Username**, "admin") key pair is not in the set of inferred keys because the **Username** value can be controlled by the issuer of the request. For the policy $\mathcal{P}_2$ in Figure 3(b), we infer just the following pair as a trusted value: (**PrincipalOrgID**, "o-1234").

5 EFFICIENT VERIFICATION

Given a policy $\mathcal{P}$, we can compute the set of trusted keys and values, and then use those to compute a VC (3). If the VC is (resp. not) satisfiable, then the policy allows (resp. prevents) untrusted request contexts, and hence is not (resp. is) Trust Safe.

Semantic Trust Elimination Recall that the VC (3) has two parts. The formula Allow($\mathcal{P}$) describes all *allowed* request context, and the second conjunct $\neg\mathsf{Trust}(\mathsf{Tr}^{\mathcal{K}}, \mathsf{Tr}^{\mathcal{V}})$ stipulates that we exclude all *trusted* requests. Intuitively, the second conjunct *semantically eliminates* trusted requests from the VC, forcing the SMT solver to find a satisfying assignment that corresponds to an untrusted (public) access. The wildcards and string comparisons that are crucial for making the policies flexible, yield VCs whose satisfiability checking is PSPACE-complete [5]. While SMT solvers have sophisticated heuristics to efficiently solve string constraints, in practice,

the VCs generated in § 4 are quite large for real-world policies, and hence, satisfiability checking is quite slow.

Syntactic Trust Elimination Our key observation is that instead of semantically eliminating trust using the SMT solver, we can exploit the structure of the policies to *syntactically* eliminate the trusted components from the policies, in a sound manner. This drastically shrinks the sizes of the policies and hence, the VCs, making verification efficient. We implement syntactic trust elimination via a set of *syntax-directed* rewrite rules that rewrite a policy $\mathcal{P}$ into a new, smaller $\mathcal{P}'$ that allows strictly *fewer* requests than $\mathcal{P}$, by denying some (or all) of $\mathcal{P}$'s trusted requests. Formally, given an input policy $\mathcal{P} \doteq \mathcal{P}_0$, we use a set of *local rewrite rules* to derive from $\mathcal{P}_0$ a sequence of increasingly restrictive policies $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n$, where, for each step $0 \leq i < n$, (1) $\mathsf{Allow}(\mathcal{P}_i) \supseteq \mathsf{Allow}(\mathcal{P}_{i+1})$, (2) and any request context $r \in \mathsf{Allow}(\mathcal{P}_i) \setminus \mathsf{Allow}(\mathcal{P}_{i+1})$ is a trusted request context. When no further rewrites can be applied, $\mathcal{P}' \doteq \mathcal{P}_n$ does not allow *any* trusted request contexts, but allows *only* the *untrusted* requests allowed by $\mathcal{P}$. Thus, if $\mathcal{P}'$ denies *all* requests, we can be sure that $\mathcal{P}$ *also* denies all *untrusted* requests, *i.e.* that $\mathcal{P}$ is Trust Safe. We can establish the last check, that $\mathcal{P}'$ denies all requests by comparing $\mathcal{P}'$ to the trivial policy that denies all requests using Zelkova's policy comparison engine.

Our syntactic elimination procedure drastically shrinks the size of the VC in two ways. First, it yields a much slimmer $\mathcal{P}'$ that excludes all the trusted elements of $\mathcal{P}$. Second, it removes the need for the additional $\neg\mathsf{Trust}(\mathsf{Tr}^{\mathcal{K}}, \mathsf{Tr}^{\mathcal{V}})$ clause that semantically eliminates trusted keys and values. Together, the rewrites make verification efficient as shown in § 7.

Rewriting Strategy Our rewriting rules implement the following high-level strategy: each rule removes trusted values from (a) *positive* conditions in **Allow** statements, or, (b) *negative* conditions in **Deny** statements. Intuitively, the former reduces the set of allowed requests by directly eliminating the conditions that allow those requests; the latter amounts to removing exceptions to the **Deny** statements, thereby increases the set of requests that are explicitly denied. Note that the the transformed policy is not *used* in place of the original, user-supplied policy for the purpose of access control: it is simply used to determine if the original policy is Trust Safe, and then discarded.

Rewriting Policies Recall, from Figure 2, that a policy $\mathcal{P}$ is defined as a set of *statements* $\{s_1, \ldots, s_n\}$. We define our rewrite system via two sets of rules. Rules of the form $\mathsf{OPT}(s) \doteq s'$ rewrite the statement s into s'. Rules of the form $\mathsf{OPT}(s) \doteq \bot$ indicate the statement s is trivial and can be removed from the policy. We write $\mathsf{OPT}^*(s)$ for the result of repeatedly applying the rules until no further rules can be applied or we have derived $\bot$. The rules are syntax-directed and hence, deterministic, making $\mathsf{OPT}^*(s)$ well-defined. Thus, we rewrite a policy as:

$$\mathsf{OPT}^*(\{s_1, \ldots, s_n\}) \doteq \{s' \mid \exists s_i.\ \mathsf{OPT}^*(s_i) = s' \neq \bot\}$$

Rewriting Statements Figure 6 shows a subset of the rules for rewriting different statements. Recall, from Figure 2 that each statement is a tuple of the form (E, P, A, R, C) where $E \in \{\textbf{Allow}, \textbf{Deny}\}$ is the *effect*, P is the set of *principals*, A is the set of *actions*, R is the set of *resources*, and C is the set of *conditions* under which the statement is matched. The values of A and R are not used in our rewrite rules, so we elide them and write simply (E, P, C). We will also drop the literal **Condition** in our presentation of the rules. The derivation rules from Figure 6 remove trusted principals and trusted condition values from the policy.

Principal Matching Rules The [PrincipalVal] rule removes a trusted principal from the set of principals of an **Allow** statement. The rule can be read as: given a statement

$$(\textbf{Allow}, \textbf{Principal} : \{p\} \cup P, C)$$

if the principal p is trusted, then p can be removed from the set of (allowed) principals, resulting in the simplified statement:

$$(\textbf{Allow}, \textbf{Principal} : P, C)$$

The [PrincipalElim] rule determines that **Allow** statements with an empty principal list can be rewritten to $\bot$, *i.e.* deleted, as they cannot match any requests.

Condition Matching Rules *Op* denotes the condition operation such as *StringLike* or *ArnEquals*. *NotOp* denotes the negated condition operators such as *StringNotLike* or *ArnNotEquals*. The [OpVal] rule removes trusted values that occur under a positive operator in the condition of an allow statement. The resulting policy is more restrictive, as it does not necessarily allow request contexts containing $(\mathbf{k}, v)$. The [NotOpVal] removes trusted values that occur under a negative operator in the condition of a **Deny** statement. The intuition is that we are removing trusted exceptions to the **Deny** statement, therefore the resulting policy allows fewer request contexts. Removing trusted values can result in policies with an empty set of values. The [OpElim] rule removes **Allow** statements with an empty set of values under a positive condition, because this condition, and hence the entire statement, could never match. Similarly, empty values under a negative condition in a **Deny** statement always evaluates to *true*, and so we remove the condition.

Key-Existence Rules Recall that if the condition operator contains the IfExists suffix, the rule matches if the key is not present in the context. The trusted value elimination rules work the same as the regular operators, but the operator elimination rule [OpIfExElim] transforms the condition to match if the key is not present (checked using the Null operator).

Correctness Each rule for $\mathsf{OPT}(s)$ preserves (un)satisfiability over untrusted requests, *i.e.* modulo the assumption $\neg\mathsf{Trust}(\mathsf{Tr}^{\mathcal{K}}, \mathsf{Tr}^{\mathcal{V}})$. This leads to the following correctness result for our optimization:

Theorem 1. *$\mathcal{P}$ is Trust Safe iff $\mathsf{OPT}^*(\mathcal{P})$ is Trust Safe*

Example Figure 7 shows the policies obtained by rewriting the examples in Figure 3. For $\mathcal{P}_1$, this entails the following steps:

(1) applying the [PrincipalVal] rule twice, until the principal set is empty; then
(2) applying the [PrincipalElim] rule to remove the **Allow** statement; and
(3) using the [NotOpVal] rule to remove all trusted exceptions to the deny for the **SourceVpc** key in the StringNotEquals condition.

After the above, no values remain in the **Deny** condition, and we apply the [NotOpElim] rule to remove the condition entirely. This results in a derived policy that still allows untrusted requests from any "admin" user for resources in "my-bucket/*" but not in

$$\mathsf{OPT}((\textbf{Allow}, \textbf{Principal}: \{p\} \cup P, C)) \doteq (\textbf{Allow}, \textbf{Principal}: P, C)) \quad \text{if } p \in \mathsf{Tr}^{\mathcal{V}}(\textbf{Principal}) \quad [\textsc{PrincipalVal}]$$

$$\mathsf{OPT}((\textbf{Allow}, \textbf{Principal}: \emptyset, C)) \doteq \bot \quad [\textsc{PrincipalElim}]$$

$$\mathsf{OPT}((\textbf{Allow}, P, \{(Op, \mathbf{k}, \{v\} \cup V)\} \cup C)) \doteq (\textbf{Allow}, P, \{(Op, \mathbf{k}, V)\} \cup C) \quad \text{if } \mathbf{k} \in \mathsf{Tr}^{\mathcal{K}}, v \in \mathsf{Tr}^{\mathcal{V}}(\mathbf{k}) \quad [\textsc{OpVal}]$$

$$\mathsf{OPT}((\textbf{Allow}, P, \{(Op, \mathbf{k}, \emptyset)\} \cup C)) \doteq \bot \quad [\textsc{OpElim}]$$

$$\mathsf{OPT}((\textbf{Deny}, P, \{(NotOp, \mathbf{k}, \{v\} \cup V)\} \cup C)) \doteq (\textbf{Deny}, P, \{(NotOp, \mathbf{k}, V)\} \cup C) \quad \text{if } \mathbf{k} \in \mathsf{Tr}^{\mathcal{K}}, v \in \mathsf{Tr}^{\mathcal{V}}(\mathbf{k}) \quad [\textsc{NotOpVal}]$$

$$\mathsf{OPT}((\textbf{Deny}, P, \{(NotOp, \mathbf{k}, \emptyset)\} \cup C)) \doteq (\textbf{Deny}, P, C) \quad [\textsc{NotOpElim}]$$

$$\mathsf{OPT}((\textbf{Allow}, P, \{(OpIfEx, \mathbf{k}, \{v\} \cup V)\} \cup C)) \doteq (\textbf{Allow}, P, \{(OpIfEx, \mathbf{k}, V)\} \cup C) \quad \text{if } \mathbf{k} \in \mathsf{Tr}^{\mathcal{K}}, v \in \mathsf{Tr}^{\mathcal{V}}(\mathbf{k}) \quad [\textsc{OpIfExVal}]$$

$$\mathsf{OPT}((\textbf{Allow}, P, \{(OpIfEx, \mathbf{k}, \emptyset)\} \cup C)) \doteq (\textbf{Allow}, P, \{(\mathsf{Null}, \mathbf{k}, \mathit{true})\} \cup C) \quad [\textsc{OpIfExElim}]$$

Figure 6: Sample Rules for Simplifying Policies.

```
{(Allow,
  Principal  :  "*",
  Action     :  {"s3:GetObject, s3:PutObject"},
  Resource   :  "arn:aws:s3:::my-bucket/*",
  Condition  :  (StringEquals, Username, "admin")),
 (Deny,
  Principal  :  "*",
  Action     :  "s3:*",
  Resource   :  "arn:aws:s3:::my-bucket/accounts/*")}
```

(a) Final policy derived from $\mathcal{P}_1$.

```
{(Allow,
  Principal  :  "*",
  Action     :  "s3:*",
  Resource   :  "arn:aws:s3:::my-bucket/*")
 (Deny,
  Principal  :  "*",
  Action     :  "s3:*",
  Resource   :  "arn:aws:s3:::my-bucket/*")}
```

(b) Final policy derived from $\mathcal{P}_2$.

Figure 7: Final policies after applying derivation rules.

"my-bucket/accounts/*", *e.g.* requests like 2. Similarly, we transform the policy in Figure 7(b) by first removing the safe exceptions and then dropping the condition for the **Deny** statement altogether. The resulting **Deny** statement unconditionally denies all requests, as does the policy.

Note that derivation rules for handling *both* **Allow** and **Deny** statements are necessary. Consider, simply using the syntactic check from $\mathsf{Tr}^{\mathcal{V}}(k)$ to ensure that every **Allow** statement only allows syntactically trusted values. By ignoring **Deny** statements, this approach would result in a *false positive* for the policy in Figure 3(b): while the **Allow** statement is very broad, the **Deny** statement crucially narrows down the set of allowed request contexts to only those coming from a specific organization.

6 IMPLEMENTATION

In this section we describe the overall system architecture, and show how Block Public Access is integrated in S3. This integration must satisfy the same operational requirements as the S3 service: security, durability, availability, and robustness.

A key concept in a service such as S3 is the separation between the control plane and the data plane. The control plane is responsible for configuring the service, *e.g.* creating buckets, setting the bucket policy, and deleting buckets. The data plane is responsible for the primary workload, *e.g.* read and writing objects to buckets. The data plane of S3 is used several orders of magnitude more often than the control plane. In turn, the latency requirements for the data plane are an order of magnitude more stringent than on the control plane.

Block Public Access is designed to prohibit public access to both control and data plane operations. While Block Public Access is fast enough to be integrated into the control plane of S3, it is too slow and high variability for data plane integration. To solve this, the crucial insight is that Block Public Access only needs to run its analysis in the control plane when policies are updated. A cached result can be used in the data plane since Block Public Access analysis decides if a policy is Trust Safe rather than deciding if individual requests are public or not.

The computational complexity of the problem still makes time-outs on pathological examples unavoidable. To maintain soundness, we choose to fail closed: if the analysis result is unknown due to a time-out, we say the policy is not Trust Safe.

Block Public Access provides two checks which can be enabled independently. The first blocks any attempt to attach a policy that grant public access. The second blocks access to existing buckets that already have a public policy. While sometimes it is reasonable to allow public access, for example sharing public data, to ensure we are safe by default the recommendation is to enable both options.

Figure 8 shows simplified S3 authorization code with these two options. The `updatePolicy` routine calls Block Public Access to check if the policy allows public access. If it does and the `s3NewPublic` option is enabled, then the policy is never attached. Otherwise the result of the Block Public Access analysis is stored in the cache and the policy is attached to the bucket. The `authS3` authorization check uses this cached value to block access if the `s3LegacyPublic` option is enabled. After retrieving the cached value we enhance the request context by adding a special key to the context `s3:isPublic` and mapping it to the cached value. If the `s3LegacyPublic` option is enabled we create a lock-down policy that will deny access if the original bucket policy was public. The lock-down policy has the following structure:

```
boolean updatePolicy(Bucket bucket, Policy policy){
  boolean isPublic = BlockPublicAccess.
     checkPublicPolicy(policy);
  Account account = bucket.getAccount();
  if (isPublic && account.s3NewPublic) {
    return false;
  }
  policyCache.store(policy, isPublic);
  bucket.attachPolicy(policy);
  return true;
}

boolean authS3(Bucket bucket, Request req){
  if (!IAM.authenticate(req)) {
    return false;
  }
  Policy policy = bucket.getPolicy();
  boolean isPublic = policyCache.get(policy);
  req.context.add("s3:isPublic", isPublic);
  Account account = bucket.getAccount();
  List<Policy> policies = new ArrayList<>();
  policies.add(policy);
  if (account.s3LegacyPublic) {
    policies.add(generateLockDownPolicy(account));
  }
  return IAM.evaluate(policies, req);
}
```

Figure 8: Block Public Access integration in S3 authorization process.

(**Deny**,		
NotPrincipal	:	*acccount*,
Action	:	"s3:*",
Resource	:	"*",
Condition	:	(BoolIfExists, **s3:isPublic**, *true*)

Here *account* is the account owner of the bucket. This policy will deny all requests for a public policy except from the owner's account. Using a lock-down policy with an extra request context key, ensures we do not have to modify the IAM policy evaluation logic.

Figure 9 shows the system design when Block Public Access is enabled. S3 API calls that result in policy updates trigger a request to a fleet dedicated to running Block Public Access. The result is stored with the policy in the policy cache database. All other requests consult this cache before delegating the authorization check to the IAM authorization engine. When a user issues a request req, S3 uses the policy cache to augment the req with the information whether the bucket policy allows public access (req'). If the Block Public Access option is enabled, it will also add a lock-down policy (p') and then use the IAM policy evaluation engine to determine whether access should be granted.

In this architecture S3 manages a high availability silo-ed fleet of hosts dedicated to handling policy analysis requests. This gives flexibility in throttling requests as needed. One of the reasons for using a dedicated fleet as opposed to reusing the existing S3 infrastructure is that SMT queries have a very different CPU profile compared to most S3 calls. Because SMT solvers are CPU intensive, using the same hosts to handle both S3 API calls and Block Public Access requests would result in unacceptable slow-downs. A Block Public Access specific S3 fleet is deployed for each region the service runs

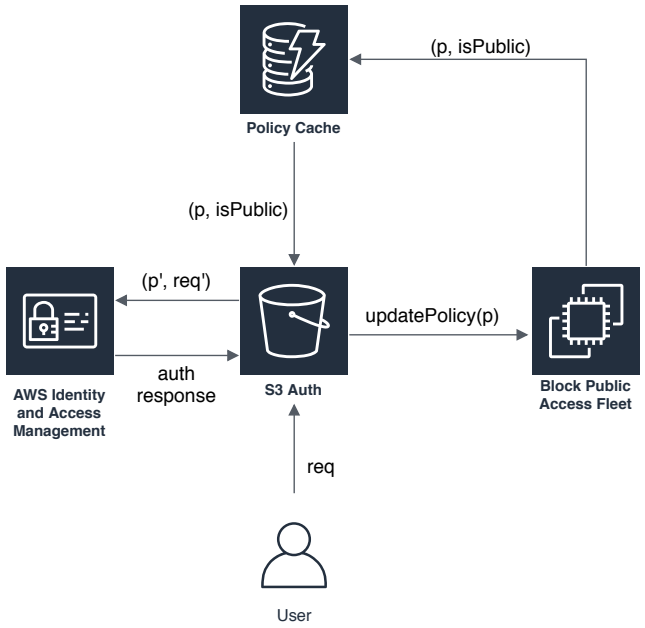

Figure 9: System diagram for Block Public Access integration in S3 request path.

in. To ensure high availability, fleets are spread over 3 availability zones and designed to work even if one of availability zones goes down. The fleet can be dynamically scaled to meet demand.

To be integrated in a high-availability, widely-used service like S3, our Block Public Access checks must meet the same operational bar as S3. In addition to exhaustive testing, we monitor the following metrics across all regions: latency, availability, and throughput. If any of these metrics are outside of predefined bounds, an automatic system pages our on-call engineer.

The policy encoding in Zelkova (on which Block Public Access is based) is designed to be robust to changes in the AWS environment. However, sometimes we do need to update the encoding to support additional functionality. One such example is the addition of new trusted context keys. To avoid false positives, Zelkova's encoding needs to be updated with knowledge of the syntax and semantics of these keys. Due to backwards compatibility these upgrades should not affect the way existing policies are evaluated. To ensure that our updates will not change the way existing policies are analyzed, we have an exhaustive corpus of policies that we use to benchmark our Zelkova analysis results. Before releasing the upgraded version of our Zelkova checks, we re-analyze these policies to ensure there are no regressions.

The enhanced S3 authorization system described here helps customers secure their S3 buckets. Using Block Public Access, customers can override S3 permissions that allow public access, making it easy for the account administrator to set up centralized control and prevent variation in security configuration regardless of how an object is added or a bucket is created. By integrated policy analysis in the control plane and using cached results in the data plane, S3 can use our Block Public Access checks as a sound mechanism to prevent public access to S3 resources.

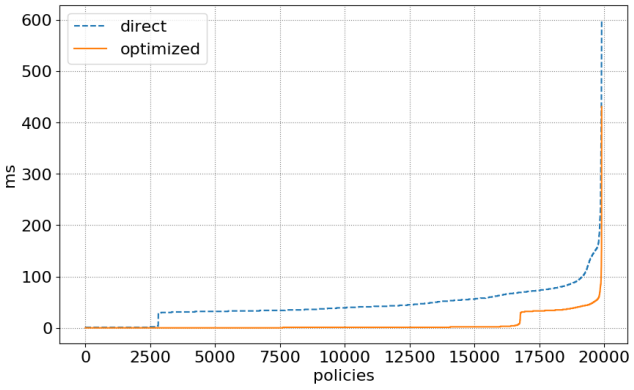

Figure 10: Block Public Access runtime using direct and optimized algorithms.

7 EVALUATION

Finally, we present an evaluation of the efficiency and precision of Block Public Access.

Efficiency To evaluate the improvement in the overall runtime of the policy analysis by our optimized (syntactic) algorithm, we randomly selected 19,901 real-world policies and compared the results to the direct (semantic) approach. The results are shown in Figure 10. Each line in the graph is sorted to be monotonic, *i.e.* the order of policies is different between the two lines and the graph does not directly compare the runtime of each policy.

There is an 85.2% average speedup in runtime using the optimized algorithm. The bulk of the optimized algorithm speedup comes from avoiding the SMT solver. This can be seen in Figure 10 as short vertical segments of the graph. The optimized algorithm is able to solve 88.0% of problems without even invoking the SMT solver. Instead, Block Public Access detects that the generated SMT formula is "trivial" *i.e.* it either asserts *false* or asserts a single atom both positively and negatively. The direct algorithm produces only 10.7% trivial SMT problems by comparison. Among the 19,901 policies, there were 167 problems that were non-trivial for both approaches. Even across these non-trivial problems, the optimized approach produces an average 65.7% reduction in SMT problem size and a corresponding 10.1% speedup.

Precision We also evaluate the precision of Block Public Access by measuring our rate of false positives when evaluating policies. Recall that to ensure that a public resource is never marked as *non-public* we chose to fail closed: if Block Public Access times out on a request, we say the policy is not Trust Safe. False positives and unknowns could result in denied access for legitimate requests. Because we are solving a problem in the PSPACE complexity class, it is inevitable that we will occasionally run into pathological examples that cannot be solved quickly. From our 19,901 sample policies we had zero unknowns. From our own use of Zelkova over the past three years, we estimate the unknown rate to be approximately one unknown every two million policies. Additionally both internal and external feedback we have received indicates a similarly low rate of false positives.

8 RELATED WORK

We position Block Public Access with respect to some lines of related work.

Policy DSLs Policy languages have been used in a variety of domains, including trust management, distributed authorization, role-based access, and access control of resources [8–11, 14, 15]. Several policy languages are defined as Datalog programs, as Datalog enables efficient verification of properties [3, 7, 9, 13, 15, 16]. The TRBAC policy model uses concrete units of time to grant or revoke access [4]. This is accomplished in the AWS policy language with conditions on date and time. Fisler *et al.* define a policy formalism that consists of transitions between different states of the environment that determine access control in policies [9]. The access control model in AWS also uses a policy and a dynamic environment request context to determine permissions, but the environment does not evolve during a single access request.

The AWS policy language is defined with respect to a JSON serialization and is designed to be used across various cloud services and scenarios of access control.

Formal Policy Analysis The Fireman system [21] uses Binary Decision Diagrams to analyze access control lists (ACL) in firewall configurations. The ACL configuration language is more restricted than ours and the tool answers a fixed set of queries about which accesses (packets) are allowed. Hughes and Bultan [11] transform XACML policies into Boolean satisfiability problems and use a SAT solver to check partial orders between policies using a bounded analysis. Bounding the analyses, however, makes it unsound. The Margrave system [18] encodes firewall policies as propositional logic formulas and uses SAT engines to analyze policies. Finally, the SecGuru tool [12] compares network connectivity policies using the SMT theory of bit vectors. Most similar to our approach is the work in [1], which checks properties of AWS access control policies by encoding their semantics in first-order logic.

None of the above explores any notion like Trust Safety. [1] has heuristics to detect (not prevent) public access that are integrated in monitoring mechanisms like AWS Config and the S3 Console that have relaxed latency requirements compared to the request path of S3. In contrast, Block Public Access implements a run-time preventative check that improves on the response time of [1] by an order of magnitude, which was achieved with an efficient algorithm based on policy rewriting. Moreover, Block Public Access ensures that no public access is ever allowed throughout the lifetime of the protected cloud resources. Finally, the presentation in [1] lacks any description of Trust Safety and any algorithm for checking it. We provide a formal definition and a detailed algorithm describing how to check this property without requiring user input.

9 CONCLUSION

Previous attempts at solving the problem of detecting misconfigured policies have relied on monitoring logs, using machine learning to find suspicious access patterns, or providing console warnings. All of these techniques detect the problem after access to the bucket has already been granted. In this work, we developed a practical solution that prevents public access to S3 buckets due to misconfigured policies. Our solution is sound, fully automated, scales to

cloud-scale request rates, and handles all the intricacies of an industrial, widely used access control policy language. This solution is currently deployed as part of S3. It analyzes millions of policies a day to prevent unintended public access.

REFERENCES

[1] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–9.

[2] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

[3] Moritz Y Becker and Peter Sewell. 2004. Cassandra: Flexible trust management, applied to electronic health records. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*. IEEE, 139–154.

[4] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. 2001. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)* 4, 3 (2001), 191–233.

[5] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A String Solver with Theory-aware Heuristics. *Formal Methods in Computer-Aided Design FMCAD 2017* 10, 14 (2017), 55.

[6] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

[7] John DeTreville. 2002. Binder, a logic-based security language. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*. IEEE, 105–113.

[8] Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2006. Specifying and reasoning about dynamic access-control policies. In *IJCAR*, Vol. 4130. Springer, 632–646.

[9] Kathi Fisler, Shriram Krishnamurthi, Leo A Meyerovich, and Michael Carl Tschantz. 2005. Verification and change-impact analysis of access-control policies. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 196–205.

[10] Dimitar P Guelev, Mark Ryan, and Pierre-Yves Schobbens. 2004. Model-checking access control policies. In *ISC*, Vol. 3225. Springer, 219–230.

[11] Graham Hughes and Tevfik Bultan. 2008. Automated verification of access control policies using a SAT solver. *International Journal on Software Tools for Technology Transfer (STTT)* 10, 6 (2008), 503–520.

[12] Karthick Jayaraman, Nikolaj Bjorner, Geoff Outhred, and Charlie Kaufman. 2014. *Automated Analysis and Debugging of Network Connectivity Policies*. Technical Report MSR-TR-2014-102. Microsoft Research.

[13] Trevor Jim. 2001. SD3: A trust management system with certified evaluation. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 106–115.

[14] Grzegorz Kolaczek. 2003. Specification and verification of constraints in role based access control. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003x2. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*. IEEE, 190–195.

[15] Ninghui Li, Benjamin N Grosof, and Joan Feigenbaum. 2003. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)* 6, 1 (2003), 128–171.

[16] Ninghui Li and John C Mitchell. 2003. Datalog with constraints: A foundation for trust management languages. In *Padl*, Vol. 3. Springer, 58–73.

[17] Anthony W Lin and Pablo Barceló. 2016. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 123–136.

[18] 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 (LISA'10)*. USENIX Association, USA, 1–8.

[19] Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. 2017. Scaling up DPLL (T) string solvers using context-dependent simplification. In *International Conference on Computer Aided Verification*. Springer, 453–474.

[20] u/DoersOfTheWord. 2016. https://www.reddit.com/r/aws/comments/3recc9/this_iam_policy_did_not_do_what_i_thought/

[21] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. 2006. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. 199–213. https://doi.org/10.1109/SP.2006.16