Formal Methods in Computer-Aided Design 2025

Modeling the AWS Authorization Engine

Lee A. Barnett, Loris D'Antoni, Amit Goel, Rami Gökhan Kıcı,
Neha Rungta, Mary Southern, Chungha Sung
Amazon Web Services
Seattle, USA
{lbarnt,lorisd,amgoel,ramikici,rungta,masouth,chunghs}@amazon.com

***Abstract*—Cloud computing providers employ sophisticated authorization engines to decide when a request to access a resource should be allowed or denied. Several approaches have formalized the behavior of *individual authorization policies*, but authorization engines employ multiple types of policies that can interact in different ways. This paper presents a modular formalization of the Amazon Web Services (AWS) authorization engine and a corresponding analysis tool, called IAM-MULTIPOLICYANALYZER, for verifying properties pertaining to multiple policies of different types. IAM-MULTIPOLICYANALYZER adopts ZELKOVA [18]—i.e., the formalization of individual IAM policies—as a basic building block, and uses a new domain-specific language for modularly describing how the authorization engine composes individual uses of ZELKOVA. As a result, IAM-MULTIPOLICYANALYZER provides a trusted, reusable, human-readable, and performant SMT-backed model of AWS's authorization logic that is now used within multiple AWS applications. We have run conformance testing of our model against the engine implementation and its documentation; the corner cases identified by our testing have led to improvements and modifications to the official AWS documentation.**

I. INTRODUCTION

Cloud authorization engines enable secure, on-demand access to digital resources through policies that control who can access what resources and under which conditions. These policies form an essential layer of an organization's security framework, providing clear and enforceable access rules. Specifically, the AWS Identity and Access Management (IAM) policy language [10] governs access by allowing or denying permissions based on rules and conditions on request parameters such as source IP addresses [4] or encryption settings [14]. It provides a rich and expressive syntax that supports multiple policy types, including identity-based policies, resource-based policies, permissions boundaries, and more [6].

To help customers analyze properties of IAM policies, ZELKOVA [18] formalized the meaning of *individual policies* in the AWS IAM language. Using this formalization, IAM policies can be compiled into semantic logical representations that can be analyzed using Satisfiability Modulo Theories (SMT) solvers. For example, ZELKOVA is used to help AWS customers write IAM policies that align with their intent by flagging policies that authorize public access to a resource [21], or by automatically modifying a policy to remove unwanted access in a provably sound way [23].

While ZELKOVA's ability to analyze *individual policies* has found wide adoption and helped many users of AWS, many security applications require more precise analysis that involve understanding how *multiple policies* interact with each other. This paper formalizes how the AWS authorization engine handles arbitrary combinations of IAM policies, and therefore yields the **first logical model of the semantics of the AWS authorization engine that can be used to automatically analyze combinations of IAM policies**. This model allows one to analyze all authorization decisions for AWS resources, thus allowing new automated analysis within security applications at AWS.

Building a logical model of the semantics of the AWS authorization engine involves striking the right level of granularity:

1) *Code is too low level:* The engine is written in a general-purpose language with performance in mind (i.e., it needs to run about one billion authorizations per second [22]), thus directly compiling the code semantics into logical constraints will lead to constraints that are hard to analyze (both in terms of user understanding and SMT solver performance). However, the model should be faithful to the implementation and map cleanly to it.
2) *Documentation is too high level:* While the authorization engine is heavily documented in natural language [11], directly translating such a documentation to a logical formalism will yield a logical model that will not map well to the actual authorization code and is therefore hard to test for faithfulness.

To address the considerations above, our work provides a pragmatic approach for building a logical representation that is **analyzable**, **faithful to the code**, and **well-tested** by carefully threading the needle between the levels of granularity offered by the code and the documentation. By studying the structure of the AWS IAM authorization implementation, we design a *modular SMT-friendly domain-specific language (DSL) for formalizing IAM authorization* that separates modeling into two parts. First, our DSL provides modular constructs that capture the top-level structure of how the actual authorization engine handles multiple policies (e.g., order of evaluation and control flow). Second, the low-level details that pertain to the evaluation of individual policies are encoded as queries to ZELKOVA, which has been used in production for over seven years and heavily tested to be correct.

Thanks to this modular structure, the model of the engine written in the DSL can easily be mapped (line-by-line) to the

https://doi.org/

actual code. Aside from making the model human-readable, the above mapping allows us to design comprehensive test cases that capture all the combinations of policies that trigger all possible control-flow combinations in the authorization engine. Using these test cases, we can perform conformance testing between the engine implementation and the SMT encoding generated from the model of the engine written in our DSL. To further strengthen our conformance testing, we also consider the third (and customer-facing) way in which the authorization engine is described: the AWS documentation [11]. To automate testing of the documentation, we manually wrote a Scala implementation of the informal semantics described in it.

Our efforts have resulted in three key contributions.

1) We built a trusted, human-readable, and performant SMT-backed model of AWS authorization logic that can be used to automatically verify properties of IAM policies across applications (Section III).
2) Our modeling strategy has helped generate test cases that were used to improve the official AWS documentation and highlight edge cases customers should consider while authoring policies. (Section IV)

To fully understand the presentation in this paper we recommend a basic familiarity with IAM Policy concepts [6].

II. Modeling AWS Authorization

AWS customers can control access to their cloud resources by writing policies in the AWS IAM language [10]. AWS today supports several different kinds of policies, including identity-based and resource-based policies [8], service and resource control policies [16], [15], and permissions boundaries [9]. For example, a resource-based policy (i.e., a policy attached to a resource) can be attached to an Amazon S3 bucket [2] to specify which AWS principals are allowed to read the bucket's contents. On the other hand, an identity-based policy can be attached to a user, a group of users, or a role, to describe what those identities are allowed to access. For example, an identity-based policy attached to a role can specify resources the role is allowed to access, or which actions the role may perform.

When an attempt to access a resource is made, the AWS authorization engine evaluates all relevant policies, such as the policy attached to the resource and any policies attached to the identity of the principal making the request. The relevant policies, and the final authorization decision (i.e. whether the request is allowed or denied), can depend on the *hierarchical structure of AWS applications*. Resources and identities belong to AWS accounts, and accounts can belong to AWS organizations defining service or resource control policies. Access granted to an AWS account can also be *delegated* to users or roles within that account.

The goal of this paper is to build an analyzable model of the AWS authorization engine. Through an example demonstrating how the authorization engine processes requests involving both identity-based and resource-based policies, we illustrate why our goal requires a new modeling mechanism that differs from previous work on Zelkova.

Example (Cross-Account Authorization): *An AWS IAM role [7] named* `Ace` *is requesting to perform the action* `s3:GetObject` *on the S3 bucket named Photo.* `Ace` *belongs to the AWS account* `111` *while the Photo bucket belongs to account* `222`*. We consider three cases to demonstrate how the AWS authorization engine evaluates this request in the presence of different policies. Figure 1 shows a diagram of the request and the resulting authorization decisions for each case. We use notation* `111/Ace` *to represent a user or role named Ace that belongs to the account with identifier* `111`*. This is a short form way of representing Amazon Resource Names (ARNs), which are used within AWS to uniquely identify identities and resources [1].*

Case A. *The resource-based policy attached to the S3 bucket grants the action* `s3:GetObject` *to the principal account* `111`*, an alias for the administrator of the account* `111`*, but there is no identity-based policy attached to* `Ace`*. Because the principal making the request (*`Ace`*) is in a different AWS account than the resource, permission must also be granted to* `Ace` *by the policies within the account. Therefore the request is denied.*

Case B. *An identity-based policy attached to* `Ace` *grants permission to perform the action* `s3:GetObject` *on the Photo bucket, but there is no resource-based policy. The request is denied, since access is not granted by the bucket's resource-based policy.*

Case C. *As in Case B, an identity-based policy attached to* `Ace` *allows access to the Photo bucket. Additionally, the resource-based policy attached to the bucket grants access to the principal account* `111`*. Now the authorization request is allowed: although the bucket's policy does not name the* `Ace` *role specifically, the administrator of* `Ace`*'s account delegates access to* `Ace` *with the identity-based policy.*

The example illustrates how different types of policies can interact in non-trivial ways.

Given that the tool Zelkova [18] (described in Section III-A) provides SMT models of the semantics of individual policies, one might wonder if there is an easy way to model the combination of multiple policies using analysis results from Zelkova. As we illustrate next, just "combining" the Zelkova analyses of different policies cannot soundly capture how different policies interact because permissions can be *delegated* between multiple principals through policies attached to *different entities* (resources, roles, users, etc.).

Consider again the three authorization cases in Figure 1. There are two policies to consider in each case: one for the identity-based policy attached to the `Ace` role in account `111`, and another for the resource-based policy attached the `Photo` bucket in account `222`. Recall that the request is allowed in Case C, but not in Cases A and B. Let's analyze what happens if we analyze each policy separately using Zelkova, and see if there is a way to combine the results to model the right authorization outcomes.

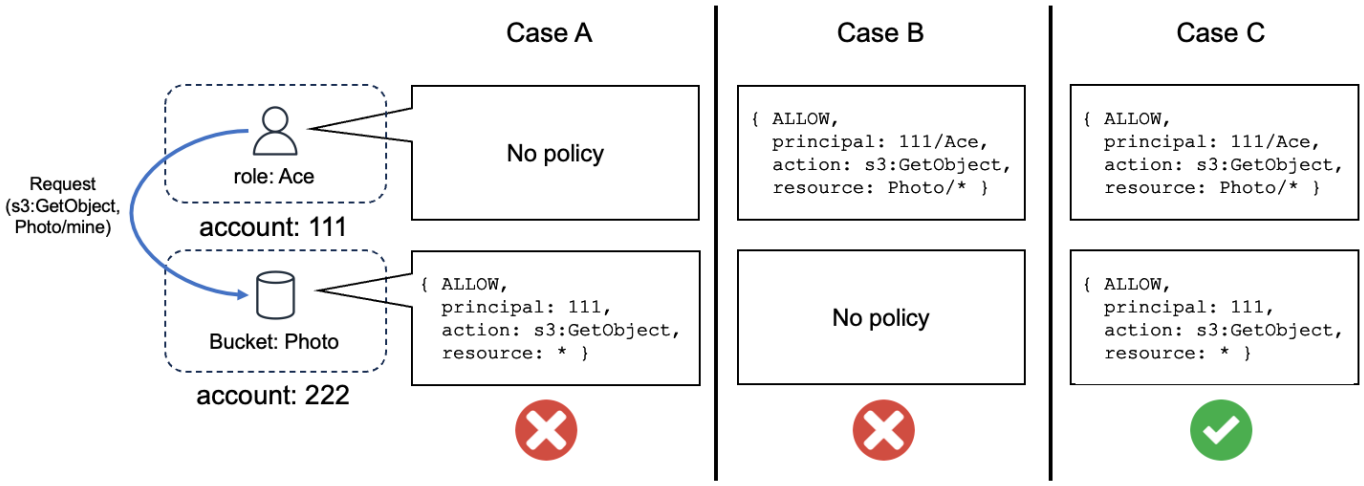

Fig. 1. Example of AWS authorization for two types of policies.

For Case A, ZELKOVA captures that the principal in the request does not match the permissions granted by the resource-based policy—i.e., the resource-based policy only grants access to principal `111`, with no mention of the case in which `Ace` role makes the request. Similarly, in Case B, ZELKOVA captures the fact that the identity-based policy allows the request but the resource-based policy denies—i.e., an empty policy does not allow any access.

From Case A and Case B, one might guess that the actual authorization result is `Allow` if and only if ZELKOVA concludes that both the identity-based and resource-based policies result in an `Allow` decision. However, such an approach is *not sound* (i.e., the model would not match the actual implementation): in Case C the request should be allowed, analyzing the resource-based policy in isolation using ZELKOVA would yield the same result as for Case A—i.e., that the request should not be allowed.

A similar issue can arise when a resource-based policy allows access to a specific role in the same account as the resource, but an identity-based policy in that role's account explicitly denies access. ZELKOVA's per-policy analysis might suggest allowing access (seeing the resource-based allow), while the actual AWS authorization engine would correctly deny it due to the explicit deny in the identity-based policy.

These examples demonstrate how one cannot model the AWS authorization engine by separately analyzing single policies with ZELKOVA and taking Boolean combinations of the analysis outcomes. The key missing element is the ability to model how *permissions flow through delegation relationships between principals*. A sound analysis needs to represent the hierarchical structure of AWS principals and resources, capturing how permissions are delegated from resources to accounts and from accounts to roles or users. The tool presented in this paper, IAM-MULTIPOLICYANALYZER, introduces a DSL that still uses ZELKOVA as a building block, but explicitly models permissions delegation and uses it to soundly capture all the features of the AWS authorization engine, including cross-account access requirements and the precedence of different policy types. In IAM-MULTIPOLICYANALYZER, the authorization decision for Case C is modeled using the DSL program illustrated in Figure 3. While we discuss the program in Figure 3 in detail in Section III, they key idea behind IAM-MULTIPOLICYANALYZER's model in this DSL is that delegation is treated explicitly by issuing separate calls to ZELKOVA using different principals (as illustrated in the third let-statement).

III. A MODULAR FORMAL MODEL FOR AWS AUTHORIZATION

Because our goal is to build a model of **any possible combination of all types of policies** with allow and deny statements, modeling the authorization engine requires a new modular mechanism to combine the decisions performed by individual policies. In this section, we present IAM-MULTIPOLICYANALYZER, a modular approach for translating sets of AWS policies into human-readable SMT terms that exactly capture the policies' semantics according to the AWS authorization engine implementation.

IAM-MULTIPOLICYANALYZER relies on ZELKOVA, a model for single policies that has withstood the test of time, to model individual policies (Section III-A) and a high level SMT friendly domain-specific language (DSL) that models the control flow of how the authorization engine dispatches authorization to individual policies and combines their results (Section III-C).

A. ZELKOVA: *From Individual Policies to SMT*

The key component that enables modularity is ZELKOVA [18], an existing service that compiles the semantics of individual IAM policies into SMT. The core of ZELKOVA's compilation is the deconstruction of policies into

$$p = \text{“111/Ace”} \wedge a = \text{“s3:GetObject”} \wedge r \texttt{ matches } \text{“Photo/*”} \quad (1)$$

$$p = \text{“111”} \wedge a = \text{“s3:GetObject”} \quad (2)$$

Fig. 2. ZELKOVA SMT Encoding of the Identity-based Policy (Equation (1)) and the Resource-based Policy (Equation (2)) from Case C of Figure 1.

a set of statements, where each statement has effect (ALLOW or DENY), principal, action, and resource components. Conditions can optionally be used to further restrict which requests a policy allows, but are not described here for space considerations (details of conditions in ZELKOVA are described by Backes et al. [18]).

The ZELKOVA SMT encoding of each policy statement s is a formula $\varphi_s(p, a, r)$ over three variables: principal (p), action (a), and resource (r). Satisfying assignments to the formula $\varphi_s(p, a, r)$ correspond to requests for which the policy grants access (i.e., what principals can perform what actions on what resources). Figure 2 shows the ZELKOVA SMT encodings of the two policies shown in Case C of Figure 1. In Equation (1), the principal and action must match the specific concrete values given in the policy, while the resource must match the expression “Photo/*” containing a wildcard * (indicating that it may be any string that begins with “Photo/”).

To meet the performance required by its applications within AWS, ZELKOVA employs many powerful optimizations, and taking advantage of such optimizations is one of the key design choices behind the modular approach of IAM-MULTIPOLICYANALYZER. Concretely, ZELKOVA supports many optimizations for policies involving string and set comparison, which we want to reuse in our design. For example, in Equation (2), the resource component is omitted as a ZELKOVA optimization detects that any string would match the regular expression containing only the wildcard.

When multiple statements appear in *the same policy*, the SMT formula produced by ZELKOVA is a conjunction of: (1) a disjunction of those statements with effect ALLOW, and (2) a conjunction of the negation of the statements with effect DENY—i.e., a request is allowed if it is explicitly allowed by some ALLOW statement and is not denied by any DENY statement. As discussed in Section II, this Boolean-combination approach is only sound (i.e., it accurately captures whether a request should be allowed) when considering a single policy, but fails when two statements belong to policies of different types.

B. AUTHENGINE: *AWS Authorization Engine*

In AWS, the authorization engine (which we call AUTHENGINE) handles the policy evaluation logic. At a high level, AUTHENGINE takes as input a set of policies $\{P_1, \ldots, P_n\}$ and a request r including details about the principal, action, resource, and the context about the environment/resource (e.g., IP address) [13], and decides whether the request should be allowed.

Algorithm 1 AUTHENGINE algorithm snippet for Section II

Require: req is the authorization request for role Ace; req_A is the authorization request for account 111; $evaluate$ evaluates a policy given a request and returns $Allow$, $ImpDeny$ or $Deny$; P_I is the identity policy; P_R is the resource-based policy

Ensure: Returns $true$ iff cross-account access request is allowed by policies

if $evaluate(P_R, req_A) = Deny \vee evaluate(P_R, req) = Deny \vee evaluate(P_I, req) = Deny$ **then**
  return $false$ ▷ Deny trumps all
end if
$delegated \leftarrow evaluate(P_R, req_A) = Allow$ ▷ Trusting account delegates to the trusted account
$trusting \leftarrow evaluate(P_R, req) = Allow$
$trusted \leftarrow evaluate(P_I, req) = Allow$
return $trusted \wedge (trusting \vee delegated)$

The key mechanism employed by the AUTHENGINE (and the one that we model explicitly in our DSL) is the *delegation* of permissions across principals. This mechanism allows permissions to flow automatically through chains of principals (e.g., from organization to account to role) without requiring explicit definition of each transfer. This mechanism is enabled by the interaction between the so-called trusted and trusting principals, where if a trusting principal *A*, e.g. an account, can access a resource and delegates permission to principal *B*, e.g. a role within that account, then *B* becomes a trusted principal who can further delegate those permissions down the chain [5].

A common and practical use case for delegation is inter-account access, where the authors of the resource-based and identity-based policies are different actors. Using permission delegation, a resource policy can allow access to a trusted account without the resource owner needing to know the details of the identities within that account. The trusted account is then responsible for further delegating appropriate permissions to the users or roles that should have access. A key point is that a principal is only able to delegate the access they themselves have. For example, if a role *A* cannot access a resource *R*, that same role cannot delegate access to resource *R* to any principal.

While the internals of AUTHENGINE are closed-source and cannot be described in detail, the reader can refer to AWS documentation for an overview of AUTHENGINE [12]. Additionally, Algorithm 1 in contains a simplified snippet that corresponds to how AUTHENGINE would operate for the case described in Figure 1.

C. IAM-MULTIPOLICYANALYZER: *From Sets of Policies to SMT via* ZELKOVA

In this section, we describe how IAM-MULTIPOLICYANALYZER models multiple policies and delegation of permissions. IAM-MULTIPOLICYANALYZER follows two key principles:

- *Modularity:* IAM-MULTIPOLICYANALYZER should use ZELKOVA (a well-tested and optimized trusted component) as a key primitive whenever it models an individual policy.
- *Faithfulness:* IAM-MULTIPOLICYANALYZER's model should follow the original implementation of AUTHENGINE as closely as possible—i.e., we should be able to map each line of the model to a corresponding line in AUTHENGINE.

IAM-MULTIPOLICYANALYZER uses a DSL (Figure 4) to model how permissions are delegated by the AUTHENGINE. Concretely, given a set of policies $\mathcal{P}$, IAM-MULTIPOLICYANALYZER builds a program X in the DSL (which can then be translated to an SMT formula), such that a request r is a valid model of X iff AUTHENGINE allows the request r for policies $\mathcal{P}$. In the rest of the section, we describe the components of the DSL.

1) Main Program: A program in the IAM-MULTIPOLICYANALYZER DSL is a predicate that describes an authorization decision. Complex predicates can be built using let-statements, Boolean operations, and most importantly, the three specialized predicates used for handling delegation listed under `IRPredicate` in Figure 4.

For example, the snippet of DSL program in Figure 3 models how AUTHENGINE handles the policies in Case C of Section II. The first let expression models the access through some identity policy which is attached to the principal, in this case `111/Ace`. The second let expression models requests allowed by the resource-based policy directly. The final let expression models that a resource may delegate permissions through the account when the acting principal has an appropriate principal type (`Role` or `User`). To keep the example manageable we do not include the clause for principal types with more complex delegation chains, such as `AssumedRole`, as our running example requires only the `Role` type. However, IAM-MULTIPOLICYANALYZER does support these multi-jump delegation chains as well. This example demonstrates how this DSL creates a human-readable program that expresses the model of access control in AUTHENGINE and is closely aligned with the authorization algorithm (Algorithm 1).

We now describe each specialized predicate and how their semantics is translated to SMT terms to enable automated reasoning on policies.

2) Evaluation under Context: The main primitive in the DSL is the predicate $\texttt{EvalCtx}(\texttt{IRCtx}, \texttt{AuthPolicy}, \texttt{Outcome})$, which is used to dispatch the modeling of individual policies to ZELKOVA. This predicate has the following intended behavior (i.e., set of satisfying assignments): under context `IRCtx`, the *single policy* `AuthPolicy` evaluates to the outcome `Outcome`.

As we have seen in Section III-A, single-policy evaluation results in a formula $\varphi(p, a, r)$. The context `IRCtx` represents the variable valuations relevant to this single policy evaluation within the larger evaluation—e.g., who is the principal p, what resource r are they accessing, etc. For simplicity we will restrict the presentation of `IRCtx` to just the principal, action, and resource; eliding condition keys. Intuitively, an `IRCtx` is a predicate acting as a symbolic representation of a set of actual concrete requests.

Let AllowI $=$
$\quad \texttt{PrHasArn}(\{\texttt{111/Ace}, a, r\}, \texttt{111/Ace}) \wedge$
$\quad \texttt{EvalCtx}(\{\texttt{111/Ace}, a, r\}, P_I, \texttt{ALLOW})$
in
Let AllowR $=$
$\quad \texttt{EvalCtx}(\{\texttt{111/Ace}, a, r\}, P_R, \texttt{ALLOW})$
Let AllowDelegateR $=$
$\quad (\texttt{ActPrHasType}(\{\texttt{111/Ace}, a, r\}, \texttt{Role}) \vee$
$\quad\quad \texttt{ActPrHasType}(\{\texttt{111/Ace}, a, r\}, \texttt{User})) \wedge$
$\quad \texttt{EvalCtx}(\{\texttt{111}, a, r\}, P_R, \texttt{ALLOW})$
in
$\quad \text{AllowI} \wedge (\text{AllowR} \vee \text{AllowDelegateR})$

Fig. 3. DSL program modeling the authorization of Case C from Figure 1 in IAM-MULTIPOLICYANALYZER.

To model delegation, the `IRCtx` will have different principal "assignments" when performing different evaluations. For example, the two instances of `EvalCtx` in the second and third let statements in Figure 3 use different contexts $\{\texttt{111}, a, r\}$ and $\{\texttt{111/Ace}, a, r\}$ to model the two possible delegation paths for the role `111/Ace`. The resulting authorization decision is the disjunction of the two predicates.

To summarize, a satisfying assignment for a predicate $\texttt{EvalCtx}(\texttt{IRCtx}, P, \texttt{Allow})$ is any evaluation context that is consistent with the predicate `IRCtx` and that results in an allow-decision when evaluated by AUTHENGINE on the single policy P. While in our example, the principal value is set to different constants corresponding to the possible delegated principal values in each `IRCtx`, it is also possible to assign delegated principal values symbolically through predicates over p.

3) Predicates for Constraining Principals: While the snippet illustrated in Figure 3 only contains the case in which the acting principal has type `Role` or `User`, the full model of the example in Figure 1 will also contain cases for other types of delegation, with the corresponding evaluation behavior.

In AUTHENGINE, the delegation mechanism depends on the type of a principal (e.g., `Root`, `User`, `Role`) and the structure of its ARN (a format used by AWS to locate resources). It is not important to understand the specifics of ARNs for this work and so specific discussion of ARNs is left to the existing AWS documentation [1]. For the sake of this explanation, it is just relevant to know that AWS imposes constraints of what structures of ARNs one can use for different types of principal types.

To correctly model delegation, the DSL uses two pred-

```
IRExpr ::= Let id = IRExpr in IRExpr |
    IRPredicate |
    TRUE | FALSE | ¬IRExpr |
    IRExpr ∧ IRExpr | IRExpr ∨ IRExpr

IRPredicate ::=
    EvalCtx(IRCtx, AuthPolicy, Outcome) |
    PrHasArn(IRCtx, Arn) |
    ActPrHasType(IRCtx, PrncplType)

PrncplType ::=
    Root | Role | User |
    AssumedRole | FederatedUser |
    FederatedPrincipal
```

Fig. 4. Simplified Syntax of IAM-MULTIPOLICYANALYZER DSL.

icates, `PrHasArn` and `ActPrHasType`, that capture the requirements of different uses of principal in AUTHENGINE. The predicate `PrHasArn` (`IRCtx`, `Arn`) holds when the acting principal in `IRCtx` has the ARN `Arn`, while `ActPrHasType` (`IRCtx`, `PrncplType`) holds when the acting principal in `IRCtx` has type `PrncplType`.

In our example, the principal is a constant and therefore one can trivially infer the account and roles, but in general one may require string predicates to do so. In fact, while Figure 3 only illustrates the snippet of DSL program corresponding to the case in which the principal has type `Role` or `User`, IAM-MULTIPOLICYANALYZER will generate similar snippets for other types of principals. Because for such snippets the predicate `ActPrHasType` will be false, these snippets will never affect the authorization decision in this case.

4) Translation to SMT: For an IAM-MULTIPOLICYANALYZER model (i.e., a program in the DSL) to be analyzable, we must generate an SMT term corresponding to it. We cannot share the exact details in this work, but implementing this translation requires maintaining the shared variable state across different ZELKOVA invocations issued by `EvalCtx` to ensure consistency across the dependent evaluations. Intuitively, if $irCtx$ is the `IRCtx` and φ is the formula encoding the ZELKOVA invocation, then the formula $\varphi(irCtx)$ encodes the IR expression $\texttt{EvalCtx}(irCtx, P, \texttt{ALLOW})$. The final SMT formula is otherwise generated by composing ZELKOVA SMT terms according to the structure of the DSL term. The check of whether the request from Figure 1 matches the DSL program in Figure 3 is captured by the following SMT formula.

$$\begin{pmatrix} \text{“111/Ace”} = \text{“111/Ace”} \wedge \\ a = \text{“s3:GetObject”} \wedge \\ r \texttt{ matches } \text{“Photo/*”} \end{pmatrix} \wedge \left(\begin{pmatrix} \text{“111/Ace”} = \text{“111”} \wedge \\ a = \text{“s3:GetObject”} \wedge \\ \text{r } \texttt{matches } \text{“*“} \end{pmatrix} \vee \begin{pmatrix} \text{“111“} = \text{“111“} \wedge \\ a = \text{“s3:GetObject”} \wedge \\ \text{r } \texttt{matches } \text{“*“} \end{pmatrix} \right)$$

The generated SMT formula has the desired behavior described in Section II: any request with principal 111/Ace to perform action s3:GetObject on the Photo bucket corresponds to an instantiation which makes this formula true. The vacuous string equalities are due to the fact that we are instantiating the principal with a specific constant.

Using an intermediate DSL to build SMT models enables multiple important optimizations in IAM-MULTIPOLICYANALYZER. The system applies constant folding to simplify expressions, caches ZELKOVA models of individual policies for reuse, and filters irrelevant statements from resource-based and resource-control policies based on the requesting principal. The DSL approach also produces human-readable models, as shown in Figure 3, making it straightforward to confirm that they capture the AUTHENGINE logic.

IV. EVALUATION

In this section we evaluate IAM-MULTIPOLICYANALYZER by assessing its correctness with respect to AUTHENGINE, as well its performance and usability within applications in Amazon.

In Section IV-B we evaluate the faithfulness of IAM-MULTIPOLICYANALYZER to the implementation of AUTHENGINE, additionally comparing against the publicly available documentation about AWS authorization, using a three-way conformance testing approach. In Section IV-C we examine applications within Amazon using IAM-MULTIPOLICYANALYZER and in Section IV-D we assess its runtime performance responding to service requests.

A. Implementation

IAM-MULTIPOLICYANALYZER intentionally relies upon the existing ZELKOVA model of single-policy semantics, which has been available through an Amazon internal web service used by tens of applications for several years. We implemented IAM-MULTIPOLICYANALYZER as a significant addition on top of ZELKOVA, extending this web service with novel operations that use IAM-MULTIPOLICYANALYZER's model to check properties about the effective permissions of sets of policies; for example, whether any kind of access is possible from a specific principal to an S3 Bucket given all associated policies. Like ZELKOVA, IAM-MULTIPOLICYANALYZER runs on AWS Lambda and relies upon a portfolio of solvers comprising Z3 [24], CVC4 [20], CVC5 [19], and NFA2SAT [29]. IAM-MULTIPOLICYANALYZER was designed to reuse existing ZELKOVA code as much as possible,

so that it benefits from the correctness and efficiency already built into ZELKOVA, as well as any future improvements made to ZELKOVA.

B. Conformance of the Model to the Code and Documentation

We performed 3-way conformance testing of the following implementations:

- IAM-MULTIPOLICYANALYZER*'s model:* the model G contributed by this paper.
- AUTHENGINE *implementation:* the Java implementation I of the AWS authorization engine modeled in this paper.
- *Informal IAM documentation:* a Scala program D encoding the informal specification of the AWS authorization logic presented in the IAM documentation [12]. Like I, the program D takes as input AWS policies along with an authorization request, and returns a yes/no answer representing an authorization decision. Unlike I, it does not include any optimizations, intended only to correspond cleanly to how AWS authorization is explained in the documentation. Note that D is quite distinct from G: it does not produce an SMT formula or other formal model of authorization, but is simply a less efficient version of I.

To guarantee complete and exhaustive evaluation of conformance, we need test inputs that exercise all control-flow paths of the AUTHENGINE implementation. Each test input should include a set of policies P, also specifying the entities to which they are attached, as well as an authorization request r. We say that the three systems G, I, and D are consistent on a given test (P, r) if they all agree on whether the request r should be allowed by the set of policies P. However, generating such inputs is a hard task because (1) randomly generating pairs of the form (P, r) results in many meaningless policies and requests, whereas (2) automated symbolic (or concolic) execution is difficult due to the complexity of the involved datatypes; i.e., complex JSON objects with string variables.

To sidestep this problem, we automatically generated tests that consider only the dimensions of policies that could affect the overall control-flow of how an authorization request r is evaluated. Specifically, we wanted to account for whether each policy in the set P would individually allow or deny the request r, but did not want to account for all the many different ways in which one policy could be written to allow or deny r. We think this design choice is justified by the fact that IAM-MULTIPOLICYANALYZER relies on ZELKOVA's trusted model of single-policy semantics (i.e., single-policy semantics have already been heavily tested). Hence, we were able to focus testing on the high-level policy combining logic within AUTHENGINE without worrying about the details of intra-policy evaluation.

Concretely, our generated test cases comprise all potential combinations of policy types as well as combinations of whether policies allow, implicitly deny, or explicitly deny potential principals. The final set of test cases contained approximately 1,500 (P, r) pairs.

Findings. IAM-MULTIPOLICYANALYZER's model G and AUTHENGINE's implementation I agreed on all the test cases, thus providing us with high assurance that IAM-MULTIPOLICYANALYZER's model is sound. Additionally, our tests highlighted two edge cases that the Scala program D did not capture, and have since resulted in changes to IAM documentation [3]. These changes improve the description of authorization presented to AWS customers. For example, one of these edge cases showed that the documentation did not fully capture how the "NotPrincipal" policy element in resource-based policies works in combination with permission boundary policies. Today, the documentation contains a clear warning highlighting that customers should avoid including a NotPrincipal policy element with a Deny effect for IAM users or roles that have a permissions boundary policy attached.

C. Applications of IAM-MULTIPOLICYANALYZER

We discuss three applications running within Amazon that depend on IAM-MULTIPOLICYANALYZER, chosen to demonstrate the range of use-cases for which this precise, multi-policy model of authorization is beneficial.

Security When necessary, Amazon may store customer data, which can in some cases include highly-sensitive information such as payment and health records. Amazon's top priority in handling this data is keeping it secure, ensuring it is available only to the small number of applications where it is absolutely needed. One of the ways in which access to sensitive data is safeguarded is through a security tool that uses IAM-MULTIPOLICYANALYZER to audit permissions granted to these critical resources by AWS policies. The tool sends regular requests to the IAM-MULTIPOLICYANALYZER web service to determine all principals that are granted permission to access any of this sensitive data, considering the identity-based policies of those principals in combination with the critical data's resource-based policies and any organization-level policies. IAM-MULTIPOLICYANALYZER enables the security tool to produce a complete picture of all possible ways access could be granted to these sensitive resources by AWS policies, helping detect possible security risks.

Availability Service-control policies (SCPs) are guardrails that define the maximum permissions which can be granted to members at different levels of an AWS Organization. Modifications to SCPs can have large-scale effects, possibly affecting access for all principals within an organizational unit. IAM-MULTIPOLICYANALYZER is used by a tool within Amazon that checks whether proposed changes to their organization's SCPs are safe before applying them. The tool sends requests to the IAM-MULTIPOLICYANALYZER web service that include the proposed new SCPs along with identity-policies for principals to ensure that (1) the new SCPs do not permit new accesses that should never be allowed, and (2) the new SCPs do not cause an availability risk by preventing needed permissions that are necessary for their applications' intended functions.

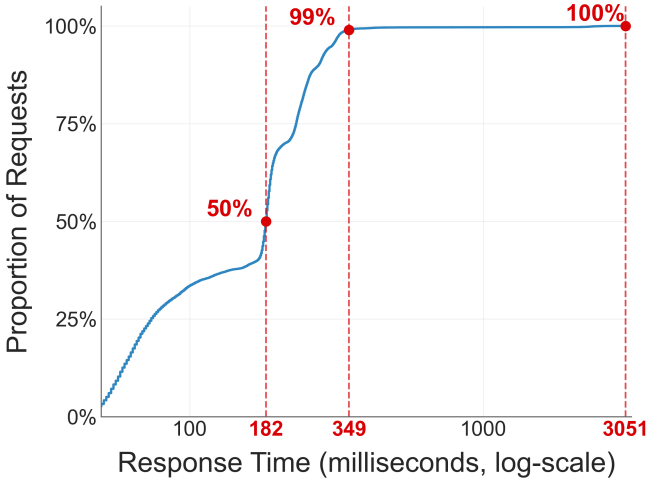

Fig. 5. Performance of IAM-MULTIPOLICYANALYZER on one million policy analysis requests.

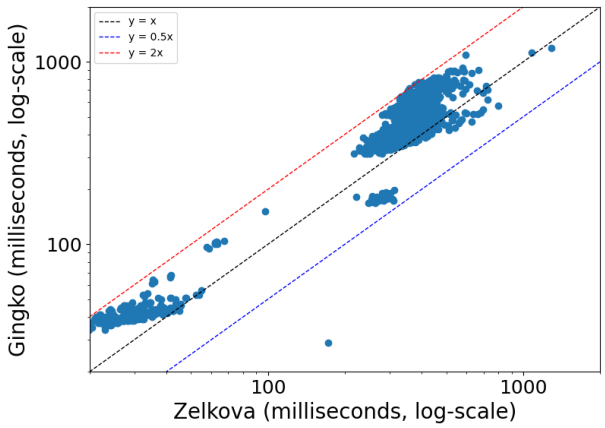

Fig. 6. Performance of IAM-MULTIPOLICYANALYZER compared to ZELKOVA on 3819 security tool requests involving identity-based and service control policies.

Automation Some services within AWS help customers onboard to their cloud offerings by automatically performing actions on behalf of the customer, such as the deployment of new resources within their account. This automation can be done by assuming a role with permissions to perform these actions in the customer's account, but in some cases there can be additional policies such as SCPs or resource-based policies that prevent the automation from succeeding. One service that relies heavily on this automation uses IAM-MULTIPOLICYANALYZER to determine whether their automation will succeed before attempting it, calling the IAM-MULTIPOLICYANALYZER web service with all relevant policies to check that each needed action is allowed. This check enables them to work with customers ahead of time to identify any policies that could cause problems for the onboarding, reducing friction around the adoption of their service.

D. Performance

General metrics The total number of invocations of IAM-MULTIPOLICYANALYZER ranges from hundreds of thousands to over a million in a single day. Figure 5 shows the performance of the analysis on a randomly-chosen set of one million requests to IAM-MULTIPOLICYANALYZER from multiple services. The y-axis represents the percentage of requests solved within the time on the x-axis. The graph shows that 99% of requests are solved within 349 milliseconds, with a median response time of 182 milliseconds.

Comparison to prior approaches While ZELKOVA can answer questions about what could be allowed by a single policy, IAM-MULTIPOLICYANALYZER is used to analyze the effective permissions of multiple policies with respect to AUTHENGINE. The only policy-analysis problems both tools can solve are ones involving single policies, but in this setting, the two tools are identical as IAM-MULTIPOLICYANALYZER simply encodes a single policy using ZELKOVA.

To perform a meaningful comparison between the two tools, we retrofit a set of benchmarks that were created by the security tool described in Section IV-C, which was in fact using ZELKOVA before the development of IAM-MULTIPOLICYANALYZER. Though ZELKOVA is intended to only analyze a single policy at a time, this application relied on custom compositions of calls to ZELKOVA that combined multiple policies into one for analysis (i.e., they manually implemented a special case of the IAM-MULTIPOLICYANALYZER's semantics). The tool's ad hoc policy-combining logic applied several heuristics to improve the performance of ZELKOVA on their requests.

When using IAM-MULTIPOLICYANALYZER to perform the same analysis, we observed no differences in the outcomes of the analysis (i.e., the two tools agreed in all cases) and observed similar performance. Figure 6 shows a scatter plot of the performance of IAM-MULTIPOLICYANALYZER on 3,819 requests from the security tool, compared to the performance of the existing ad-hoc implementation using ZELKOVA. On average, the response time of IAM-MULTIPOLICYANALYZER on this set of requests was 99.94 milliseconds (or 1.31 times) longer than the response time of ZELKOVA, with 99% of requests taking at most 331.90 milliseconds (or 1.87 times) longer than the response time of ZELKOVA.

While the ad-hoc ZELKOVA encoding could take advantage of the problem structure to achieve better performance, the security tool still decided to switch to IAM-MULTIPOLICYANALYZER because (1) conformance testing provides assurance that IAM-MULTIPOLICYANALYZER faithfully captures the AWS authorization logic, and (2) the performance was largely comparable.

Furthermore, because IAM-MULTIPOLICYANALYZER can model more policy types than just identity-based policies and

service control policies, this tool was able to extend its usage to include additional policy types, e.g., resource-based policies, thus enabling more precise and generalizable analyses.

V. Related Work

ZELKOVA [18] has been used in a variety of applications. For example, the *stratified predicate abstraction* algorithm was built on top of ZELKOVA to summarize all permissions granted by an IAM policy [17], while Bouchet et al. [21] introduced an approach built on ZELKOVA to detect IAM policies that grant overly broad or *public* access. These extensions focus on identifying which permissions are allowed or denied by single IAM policies; in a similar vein, D'Antoni et al. [23] describe a technique for automatically modifying IAM policies to remove undesired permissions.

While ZELKOVA models the permissions granted by individual access control policies in isolation, IAM-MULTIPOLICYANALYZER focuses on the combined effect of multiple policies, which may be attached to distinct, distributed cloud resources, on authorization decisions. The applications that use ZELKOVA can now be lifted to supporting multiple policies by adopting IAM-MULTIPOLICYANALYZER.

Other approaches have also looked at modeling permissions granted by access control policies expressed in different languages. Fisler et al. [26] use transitions between evaluation states to determine access control in policies. Other applications also use an SMT encoding to model authorization, such as the work on verification of NGAC policies by Dubrovenski et al. [25]. These applications target different languages and authorization mechanisms and cannot be applied to our domain. SecGuru [28] also uses SMT; specifically, the SMT theory of bit vectors, to compare network connectivity policies. Hughes and Bultan [27] transform XACML policies into Boolean satisfiability problems and use a SAT solver to check partial orders between policies using a bounded analysis. This bounded analysis is however unsound, while both ZELKOVA and IAM-MULTIPOLICYANALYZER are based on a sound SMT encoding that ensures all allowed accesses are identified.

VI. Conclusion

We proposed a modular approach for analyzing how AWS IAM authorization handles sets of policies and implemented it in IAM-MULTIPOLICYANALYZER. IAM-MULTIPOLICYANALYZER has launched as an internal service in AWS, serving about two million requests per week showing the scale at which the approach operates in practice. Customers use IAM-MULTIPOLICYANALYZER in many tasks and applications, from checking that the service control policies used within their organizations are not overly permissive, to detecting ahead of deployment when overly-restrictive service control policies can cause access-denied issues. As next steps, we are building solutions that use IAM-MULTIPOLICYANALYZER for AWS customers—e.g., allowing customers to audit who has access to what within their accounts and organizations.

References

[1] Amazon resource names. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference-arns.html.

[2] Amazon simple storage service documentation. https://docs.aws.amazon.com/s3/. Accessed: January 2025.

[3] AWS JSON policy elements: NotPrincipal. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_notprincipal.html.

[4] Denies access to AWS based on the source IP. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_examples_aws_deny-ip.html.

[5] Examples of policies for delegating access. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_create_policy-examples.html.

[6] IAM JSON policy reference. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies.html.

[7] IAM roles. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html.

[8] Identity-based policies and resource-based policies. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_identity-vs-resource.html.

[9] Permissions boundaries for IAM entities. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_boundaries.html.

[10] Policies and permissions in AWS identity and access management. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html.

[11] Policy evaluation logic. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic.html.

[12] Policy evaluation logic. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic.html.

[13] Processing the request context. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic_policy-eval-reqcontext.html.

[14] Protecting data with server-side encryption. Accessed: January 2025. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/serv-side-encryption.html.

[15] Resource control policies (RCPs). Accessed: January 2025. URL: https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_rcps.html.

[16] Service control policies (SCPs). Accessed: January 2025. URL: https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scps.html.

[17] J. Backes, U. Berrueco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. Luckow, S. McLaughlin, M. Menon, D. Peebles, U. Pugalia, N. Rungta, C. Schlesinger, A. Schodde, A. Tanuku, C. Varming, and D. Viswanathan. Stratified abstraction of access control policies. In *Computer Aided Verification–CAV*, pages 165–176, 2020. doi:10.1007/978-3-030-53288-8_9.

[18] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *Formal Methods in Computer Aided Design–FMCAD*, pages 1–9, 2018. doi:10.23919/FMCAD.2018.8602994.

[19] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems–TACAS*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

[20] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd Intl Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.

[21] M. Bouchet, B. Cook, B. Cutler, A. Druzkina, A. Gacek, L. Hadarean, R. Jhala, B. Marshall, D. Peebles, N. Rungta, et al. Block public access: trust safety verification of access control policies. In *Proceedings of the*

28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 281–291, 2020.

[22] A. Chakarov, J. Geldenhuys, M. Heck, M. Hicks, S. Huang, G. A. Jaloyan, A. Joshi, R. Leino, M. Mayer, S. McLaughlin, A. Mritunjai, C. P. Claudel, S. Porncharoenwase, F. Rabe, M. Rapoport, G. Reger, C. Roux, N. Rungta, R. Salkeld, M. Schlaipfer, D. Schoepe, J. Schwartzentruber, S. Tasiran, A. Tomb, E. Torlak, J. Tristan, L. Wagner, M. Whalen, R. Willems, J. Xiang, T. J. Byun, J. Cohen, R. Wang, J. Jang, J. Rath, H. T. Syeda, D. Wagner, and Y. Yuan. Formally verified cloud-scale authorization. 2025. URL: https://www.amazon.science/publications/formally-verified-cloud-scale-authorization.

[23] L. D'Antoni, S. Ding, A. Goel, M. Ramesh, N. Rungta, and C. Sung. Automatically reducing privilege for access control policies. 8, Oct. 2024. doi:10.1145/3689738.

[24] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[25] V. Dubrovenski, E. Chen, and D. Xu. SMT-based verification of NGAC policies. In *IEEE Computers, Software, and Applications Conference–COMPSAC*, pages 860–869, 2023. doi:10.1109/COMPSAC57700.2023.00115.

[26] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings 27th Intl Conference on Software Engineering–ICSE.*, pages 196–205, 2005. doi:10.1109/ICSE.2005.1553562.

[27] G. Hughes and T. Bultan. Automated verification of access control policies using a SAT solver. *Intl journal on software tools for technology transfer*, 10(6):503–520, 2008.

[28] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman. Automated analysis and debugging of network connectivity policies. *Microsoft Research*, 1:1–11, 2014.

[29] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, R. Majumdar, and D. Nowotka. Solving string constraints using sat. In C. Enea and A. Lal, editors, *Computer Aided Verification–CAV*, pages 187–208, 2023.