Semantic-based Automated Reasoning for AWS Access Policies using SMT

John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek,
Kasper Luckow, Neha Rungta, Oksana Tkachuk, Carsten Varming
Amazon Web Services

***Abstract*—Cloud computing provides on-demand access to IT resources via the Internet. Permissions for these resources are defined by expressive access control policies. This paper presents a formalization of the Amazon Web Services (AWS) policy language and a corresponding analysis tool, called ZELKOVA, for verifying policy properties. ZELKOVA encodes the semantics of policies into SMT, compares behaviors, and verifies properties. It provides users a sound mechanism to detect misconfigurations of their policies. ZELKOVA solves a PSPACE-complete problem and is invoked many millions of times daily.**

I. INTRODUCTION

Cloud computing provides on-demand access to IT resources via the Internet. The convenience of accessing resources in the cloud is made secure by user-specified *access control policies*. An access control policy is an expressive specification of what resources can be accessed, by whom, and under what conditions. Properly configured policies are an important part of an organization's security posture. The scale and diversity of cloud-based services is constantly growing (*e.g.*, serverless computing, streaming analytics, edge-computing devices), and each new offering used by an organization may require a different access policy configuration. Moreover, customers are combining these services, which means that the complexity is increasingly moving into policies. Thus the security challenge for many customers is becoming one of reasoning about static policies for their dynamic systems. Cloud customers want a tool that allows them to check policy configurations based on their security requirements.

Amazon Web Services (AWS) defines a policy language that lets users govern access to AWS resources. The permissions granted by a policy rely on the interactions of different statements and conditions. The policy language supports the interplay of statements that either grant access (*allow* statements) or revoke access (*deny* statements). Furthermore, conditions within statements can be based on access details such as the source address, encryption, and other configuration options.

Users want assurances that their policies grant the right permissions. To validate that policies express what is intended, some AWS users have implemented heuristic-based syntactic checks that detect certain patterns in policies, *e.g.*, the use of a wildcard that makes resources publicly accessible. Although helpful, heuristic-based syntactic checks are unsound, since they do not fully take into account the semantics of the policy language. Others attempt to explicitly enumerate all possible requests to a policy but quickly find this intractable.

In this paper, we present the development and application of ZELKOVA, a policy analysis tool designed to reason about the semantics of AWS access control policies. ZELKOVA translates policies and properties into Satisfiability Modulo Theories (SMT) formulas and uses SMT solvers to check the validity of the properties. We use off-the-shelf solvers and an in-house extension of Z3 called Z3AUTOMATA.

ZELKOVA reasons about all possible permissions allowed by a policy in order to verify properties. For example, ZELKOVA can answer the questions "Is this resource accessible by a particular user?" and "Can an arbitrary user write to this resource?". The property to be verified is specified in the policy language itself, eliminating the need for a different specification or formalism for properties. In addition, ZELKOVA provides many built-in checks for common properties.

The SMT encoding uses the theory of strings, regular expressions, bit vectors, and integer comparisons. The use of the wildcards $*$ (any number of characters) and ? (exactly one character) in the string constraints makes the decision problem PSPACE-complete. However, our experience with real-world policies is that 99% of policy questions can be answered in less than 160 milliseconds.

ZELKOVA is the underlying policy analysis engine for a growing number of AWS services. Used many millions of times every day, ZELKOVA analyzes policies attached to resources with compute, storage, messaging, search, analytics, and other capabilities. A sample of AWS services that integrate ZELKOVA includes Amazon S3 (object storage), AWS Config (change-based resource auditor), Amazon Macie (security service), AWS Trusted Advisor (compliance to AWS best practices), and Amazon GuardDuty (intelligent threat detection). Also, ZELKOVA is used by internal AWS Security auditing tools to enforce security best-practices for policy configurations, *e.g.*, public access to the resources is prohibited.

A. Related work

Policy languages have been used in a variety of domains, *e.g.*, trust management, distributed authorization, role-based access, access control of resources [1]–[6]. Several policy languages are defined as Datalog programs since it enables efficient verification of properties [2], [6]–[10]. 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. ZELKOVA combines all the components of the policy language in a single analysis tool.

Fisler *et al.* define a policy formalism that consists of transitions between different states of the environment that determine access control in policies [2]. 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. Other policy frameworks, *e.g.*, XACML, allow policies across different applications to be combined [11], [12]. In a closely related work, Hughes and Bultan 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. In contrast, the encoding to SMT in ZELKOVA is sound. The TRBAC policy model uses concrete units of time to grant or revoke access [13]. This is accomplished in the AWS policy language with conditions on date and time. Finally, the SecGuru tool [14] compares network connectivity policies using the SMT theory of bit vectors.

Our present work stands out most along three dimensions. First, we use an existing industrial policy language, which has evolved to suit the needs of millions users and use cases. The language is robust and flexible, with features that have arisen from practical needs. Second, we work closely with service teams to integrate our tool and to develop custom pre-built properties that are relevant to each service's users. Finally, we have reached an audience of many millions with our tool.

II. APPROACH

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 specific *action* being requested. A policy evaluation engine compares this request context against the policies for the user and the resource to determine if access is granted or denied.

ZELKOVA verifies AWS policies by reasoning over all possible request contexts. The fundamental mechanism of ZELKOVA is the ability to say if one policy is less-or-equally-permissive than another. Properties can be specified as *boundary policies* that represent either upper or lower bounds on desired behavior. ZELKOVA's less-or-equally-permissive check then establishes the correctness of these bounds or finds a counterexample.

A. Policy language overview

The AWS policy language is defined as serialized JSON[1], however, in this paper we describe the core constructs of the policy language in a simplified abstract syntax. The examples in this paper are also presented using this abstract syntax.

Fig. 1 shows the abstract syntax for the policy language. In this syntax, ? denotes optional elements and * denotes list valued elements. A policy is a list of statements. Each *Statement* consists of a tuple (*Principal*, *Effect*, *Action*, *Resource*, *Condition*?). The *Condition* is an optional element in the policy while the others are required.

```
   Policy → Statement*
Statement → (Effect, Principal, Action, Resource, Condition?)
   Effect → allow | deny
Principal → principal: string*
   Action → action: string*
 Resource → resource: string*
Condition → condition: Operator*
 Operator → (OpName, KeyName, Value*)
   OpName → StringEquals | StringEqualsIfExists | StringLike |
            StringNotEquals | IpAddress | ...
  KeyName → aws:sourceVpc | aws:sourceIp | s3:prefix | ...
    Value → string | num | bool
```

Fig. 1. Simplified abstract syntax for the AWS policy language

The *Effect* construct states whether the statement allows or denies access. By default, access to a resource is denied. Allow statements override the default permissions, and deny statements override the permissions granted by allow statements. In other words, to get access to a resource, there must be some allow statement that grants access and no deny statement that revokes that access. There is no ordering constraints on statements in a policy.

The *Principal* construct is used in policies to specify which users, accounts, services, or entities are granted or denied access to resources. The principals are identified uniquely by string values. The *Action* construct specifies the list of actions that are either allowed or denied on the corresponding resource. Various AWS services publish the set of actions that can be performed by the user for the resources specific to those services. The *Resource* construct specifies the list of service specific resources to which access is either granted or denied. Every AWS service has its own set of resources and each AWS resource is uniquely identified by a string value. String values for *Action* and *Resource* can contain the wildcard $*$ which matches any number of characters and the wildcard ? which matches exactly one character.

The *Condition* construct specifies conditions under which access is granted or denied. In the *Condition* construct expressions are constructed using *Operators* on condition key value pairs. The condition operators are grouped by their types: String, Numeric, Date and Time, Boolean, Binary, IP address, and others. The operator name (*OpName*) indicates the type and the comparator. String condition operators provide comparison on string conditions, *e.g.*, `StringEquals` checks string equality, `StringLike` checks a string against a pattern. The complete list of operators is defined in the IAM documentation[2] and is supported in our implementation. The operators are applied to condition keys (*ConditionKey*). Each condition key is mapped to a corresponding value. Certain condition keys are defined globally across all services, *e.g.*, `aws:sourceIp`, while other condition keys are service specific, `s3:prefix`.

[1]https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements.html

[2]https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_condition_operators.html

```
(( allow,
   principal   :   students,
   action      :   getObject,
   resource    :   cs240/Exam.pdf),
 ( allow,
   principal   :   tas,
   action      :   getObject,
   resource    :   (cs240/Exam.pdf,
                    cs240/Answer.pdf)))
```

(a) Policy X

```
(( allow,
   principal   :   *,
   action      :   getObject,
   resource    :   cs240/*),
 ( deny,
   principal   :   students,
   action      :   getObject,
   resource    :   cs240/Answer.pdf))
```

(b) Policy Y

Fig. 2. Example policies for students and TAs access to exams and answers.

$$
\begin{aligned}
X_0 &: a = \text{“}\textit{getObject}\text{”} \wedge p = \text{“}\textit{students}\text{”} \wedge r = \text{“}\textit{cs240/Exam.pdf}\text{”} \\
X_1 &: a = \text{“}\textit{getObject}\text{”} \wedge p = \text{“}\textit{tas}\text{”} \wedge \\
&\qquad (r = \text{“}\textit{cs240/Exam.pdf}\text{”} \vee r = \text{“}\textit{cs240/Answer.pdf}\text{”}) \\
X &: X_0 \vee X_1 \\[6pt]
Y_0 &: a = \text{“}\textit{getObject}\text{”} \wedge r = \text{“}\textit{cs240/*}\text{”} \\
Y_1 &: a = \text{“}\textit{getObject}\text{”} \wedge p = \text{“}\textit{students}\text{”} \wedge r = \text{“}\textit{cs240/Answer.pdf}\text{”} \\
Y &: Y_0 \wedge \neg Y_1
\end{aligned}
$$

Fig. 3. SMT encoding of policies X and Y from Fig. 2.

B. Example

A *policy* in the simplified abstract syntax for the Amazon Simple Storage Service (S3) is shown in Fig. 2. Amazon 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 through an Amazon Resource Name (ARN). The policy attached to the bucket controls access to the bucket and the objects in the bucket. The policy in Fig. 2(a) states that students can read the exam and teaching assistants can read both the exam and its answers. The other policy, shown in Fig. 2(b), says that everybody can access all the contents of *cs240/* except that students cannot access the answers.

Fig. 3 shows the encoding of the policies from Fig. 2. The encoding for each policy is a formula over three variables p, a, and r that correspond to the principal, action, and resource in the resource request context. The formula evaluates to true whenever the policy grants access. Since policy X has two allow statements that can grant access, it is represented by their disjunction. On the other hand, policy Y has one allow statement Y_0 and one deny statement Y_1. Thus policy Y only grants access if Y_0 allows access and Y_1 does not deny it: $Y_0 \wedge \neg Y_1$. Note that we are abusing notation in Y_0 to say $r =$ *“cs240/*”* since this, in fact, will correspond to a form of string matching rather than equality. We discuss the details of string matching in Section III-A.

To determine if policy X is less-or-equally-permissive than policy Y, ZELKOVA uses SMT solvers to check if

$$(X_0 \vee X_1) \implies (Y_0 \wedge \neg Y_1)$$

is valid, which is true. The result of this check states that all requests allowed by policy X are allowed by policy Y.

However, policy Y allows additional permissions. The resource *“cs240/*”* in the allow statement in policy Y allows the *“students”* and *“tas”* principals access to objects (files) *other* than *“Exam.pdf”* and *“Answer.pdf”*, such as *“Class-Roster.pdf”*. Policy Y additionally grants principals other than *“students”* and *“tas”* access to the resources in the bucket *“cs240”*, since the deny statement only denies *“students”* access to the *“Answer.pdf”*. This leads to a publicly readable bucket since any other principal can perform the *getObject* action on the contents of the bucket. Thus this policy does not represent the user's intentions, and it violates security best practices. This shows the need for sound analysis of policies. ZELKOVA provides this by reducing policies to mathematical formulas and verifying their properties using SMT solvers.

III. SMT ENCODING

In this section, we describe ZELKOVA's SMT encoding. The encoding uses the theory of strings, regular expressions, bit vectors, and integer comparisons. The policy language is declarative, with no programming constructs such as loops or dynamically allocated arrays. The semantics of the policy language are encoded as an SMT formula. The permissions granted by the policy are encoded as all the permissions granted by allow statements and not revoked by deny statements:

$$\left(\bigvee_{S \in \mathit{Allow}} [\![S]\!] \right) \wedge \neg \left(\bigvee_{S \in \mathit{Deny}} [\![S]\!] \right) \tag{1}$$

Here *Allow* and *Deny* are the set of allow and deny statements in a policy. The semantic meaning of each statement, $[\![S]\!]$, is the set of permissions granted by an allow statement or the set of permissions revoked by a deny statement.

Each statement in a policy encodes the constraints over the principal, action, resource, and conditions:

$$
\begin{aligned}
[\![S]\!] := \left(\bigvee_{v \in P(S)} p = v \right) \wedge \left(\bigvee_{v \in A(S)} a = v \right) \wedge \\
\left(\bigvee_{v \in R(S)} r = v \right) \wedge \left(\bigwedge_{O \in C(S)} [\![O]\!] \right)
\end{aligned} \tag{2}
$$

The function $P(S)$ returns all the string values specified for a principal. Similarly, $A(S)$ and $R(S)$ return the string values for the actions and resources in the statement. The function $C(S)$ returns the set of condition operators for a given statement. The variables p, a, and r map respectively to the principal, action, and resource values. The permissions in a statement are granted as a disjunction over string values of the principal, action, and resource values as well as a conjunction over the conditions as shown in Eq. (2).

```
( allow,
  principal   :   *,
  action      :   getObject,
  resource    :   cs240,
  condition   :   (StringEquals, aws:sourceVpc, vpc-111bbb222),
                  (StringLike, s3:prefix, cs240/Exam*))
```

Fig. 4. Example policy with two conditions.

Each condition in a policy encodes a constraint over the corresponding condition key:

$$[\![O]\!] := \bigwedge_{\langle op,k,V\rangle \in CO(O)} \left(condExists_k \wedge \left(\bigvee_{v \in V} op(k,v) \right) \right) \quad (3)$$

Each condition maps to an operator name, a key name, and a list of values via the function $CO(O)$. The meaning of a condition is encoded by a disjunction over all the listed values. The Boolean variable $condExists_k$ states that condition key, k, must exist in the request context. The variable k represents the value of the condition key when it exists. The operator (*op*) defines the operations on the key and value pair (k, v), *e.g.*, equality or inequality.

Next, we present the encoding of a few important classes of condition operators.

A. String constraints

The encoding of policies in ZELKOVA is largely through the use of string constraints. This includes both string equality and inequality constraints, as well as pattern matching against regular expressions. The principal, action, and resources constructs in the policy are encoded as string constraints. String operators and their corresponding condition keys are also encoded as string constraints. An example policy with conditions is shown in Fig. 4. The operator `StringEquals` is applied to the condition key `aws:sourceVpc` with a value of *"vpc-111bbb222"*, which restricts access to a specific virtual private network (VPC) in the AWS cloud[3]. The string operator `StringLike` is applied to the condition key `s3:prefix` with a value of *"grades/∗"*, which limits access so that only objects under the *"grades/"* directory may be listed.

Fig. 5 shows the SMT encoding for this example. The Boolean variables *vpcExists* and *s3PrefixExists* encode whether the conditions `aws:sourceVpc` and `s3:prefix` are present in the request context. The constraint *"grades/" prefixOf s3Prefix* encodes that *"grades/"* is a prefix of the variable *s3Prefix*. The following request context corresponds to a satisfying assignment to the set of constraints in Fig. 5:

```
{principal: bob,
 action : listBucket,
 resource : cs240,
 condition: {aws:sourceVpc: vpc-111bbb222,
             s3:prefix: grades/2018/final/}}
```

In order to encode $*$ wildcards in strings we use the *prefixOf*, *suffixOf*, and *contains* string operators. With this encoding we

[3]https://aws.amazon.com/vpc/

$$a = \texttt{"listBucket"} \wedge r = \textit{"cs240"} \wedge \\ vpcExists \wedge vpc = \textit{"vpc-111bbb222"} \wedge \\ s3PrefixExists \wedge \textit{"grades/"}\ prefixOf\ s3Prefix$$

Fig. 5. SMT encoding of policy in Fig. 4

```
( allow,
  principal   :   *,
  action      :   listBucket,
  resource    :   *,
  condition   :   (StringEquals, s3:prefix, UpLoads),
                  (StringEqualsIgnoreCase, s3:prefix, Uploads))
```

Fig. 6. Example policy with mixed case conditions.

can support up to two $*$ wildcards. Later we will see a different encoding for additional wildcards. Examples of the current encoding are given in (4).

$$\begin{aligned} &\textit{"cs2*/Exam*"} \mapsto \textit{"cs2"}\ \texttt{prefixOf}\ Var \wedge Var\ \texttt{contains}\ \textit{"/Exam"} \\ &\textit{"cs2*/*Exam"} \mapsto \\ &\quad \textit{"cs2"}\ \texttt{prefixOf}\ Var \wedge Var\ \texttt{contains}\ \textit{"/"} \wedge \textit{"Exam"}\ \texttt{suffixOf}\ Var \\ &\textit{"*240/*Exam"} \mapsto Var\ \texttt{contains}\ \textit{"240/"} \wedge \textit{"Exam"}\ \texttt{suffixOf}\ Var \end{aligned} \quad (4)$$

When different parts of a pattern can overlap, we disallow the possible overlaps. For example, *"ab∗bc"* translates to *"ab"* `prefixOf` *Var* ∧ *"bc"* `suffixOf` *Var* ∧ *Var* ≠ *"abc"*. Note that *"abc"* would otherwise satisfy the prefix and suffix constraints, yet it does not match the pattern *"ab∗bc"*.

B. Regular expression constraints

More complicated string constraints require a more powerful encoding. In particular, the encoding described above is unable to represent constraints with the ? wildcard or more than two $*$ wildcards. For example, the following encoding fails because it does not restrict *"b"* to appear before *"c"*.

$$\begin{aligned} \textit{"a*b*c*d"} \mapsto\ & \textit{"a"}\ \texttt{prefixOf}\ Var \wedge Var\ \texttt{contains}\ \textit{"b"} \wedge \\ & Var\ \texttt{contains}\ \textit{"c"} \wedge \textit{"d"}\ \texttt{suffixOf}\ Var \end{aligned} \quad (5)$$

In such cases, we use regular expressions to encode these constraints. For example, (6) shows two encodings based on the traditional regular expression pattern format where "." stands for any single character and "*" is the Kleene star operator representing zero or more occurrences of the previous character.

$$\begin{aligned} &\textit{"cs???/Exam*"} \mapsto Var\ \texttt{matches}\ \textit{"cs.../Exam.*"} \\ &\textit{"cs2*/Exam/*/Results/*"} \mapsto \\ &\qquad Var\ \texttt{matches}\ \textit{"cs2.*/Exam/.*/Results/.*"} \end{aligned} \quad (6)$$

Some condition operators are case sensitive (*StringEquals*, *StringLike*) while others are case insensitive (*StringEqualsIgnoreCase*, *Bool*). Which type of operators are used on the same condition key determines the exact encoding for case sensitivity. When a condition key is constrained with only case sensitive operators, nothing special needs to be done. When a condition key is constrained with only case insensitive operators, the targets of all those comparisons are converted to lowercase which solves the problem. The difficult case is

$$\begin{aligned}&a = \texttt{"listBucket"} \wedge \textit{s3PrefixExists} \wedge \\ &\textit{s3Prefix}\ \texttt{matches}\ \textit{"UpLoads"} \wedge \\ &\textit{s3Prefix}\ \texttt{matches}\ \textit{"[uU][pP][lL][oO][aA][dD][sS]"}\end{aligned}$$

Fig. 7. SMT encoding of policy in Fig. 6

when a condition key is constrained with both case sensitive and case insensitive operators. The previous method of converting to lowercase all targets of case insensitive operators would interfere with the case sensitive operators. Instead, case sensitive comparisons are treated normally while the targets of case insensitive comparisons are encoded into a regular expression that represents all possible case combinations. For example, consider the contrived combinations of conditions in shown in Fig. 6. Here there is both a case sensitive and a case insensitive constraint on the `s3:prefix` condition key. The ZELKOVA encoding of these constraints is shown in Fig. 7 where we use character classes of the form *[xX]* to represent a regular expression which matches a single character, either *"x"* or *"X"*.

C. Bit vector constraints

The `IpAddress` condition operator allows users to restrict access based on IP addresses. The `IpAddress` operator is used in combination with the `aws:SourceIp` condition. The values of `aws:SourceIp` have to be in the Classless Inter-Domain Routing (CIDR) format. The CIDR format associates net masks as part of the IP address specification. For example, the IPv4 in CIDR notation 11.22.33.0/24 means that the first 24 bits of the IP address are considered significant. Consider the translation of two conditions, one where `aws:SourceIp` is set to 11.22.33.0/24 and the other set to 11.22.0.0/16:

$$\begin{aligned}&C_0 : (\texttt{IpAddress}, \texttt{aws:SourceIp}, 11.22.33.0/24) \mapsto \\ &\qquad \textit{ipV4Exists} \wedge (\textit{0x0B162100} = (\textit{ipV4}\ \&\ \textit{0xFFFFFF00})) \\ &C_1 : (\texttt{IpAddress}, \texttt{aws:SourceIp}, 11.22.0.0/16) \mapsto \\ &\qquad \textit{ipV4Exists} \wedge (\textit{0x0B160000} = (\textit{ipV4}\ \&\ \textit{0xFFFF0000}))\end{aligned} \tag{7}$$

The Boolean variable *ipV4Exists* encodes the existence of condition `aws:SourceIp`, and the bit vector variable *ipV4* encodes the actual value. A bitwise AND operation is used to mask the insignificant bits of the IP address in the constraint.

With this encoding we have $[\![C_0]\!] \implies [\![C_1]\!]$ is valid. There are 24 significant bits in the IP address in constraint C_0 and only 16 significant bits in the IP address in the constraint C_1. The common routing prefix is the same. Thus, request contexts that are allowed by C_0 are also allowed by C_1.

D. Other operators

The conditions on numeric operators only perform integer comparisons. There are no arithmetic operations in the policy language and no interactions between numeric values and string values, *e.g.*, you cannot take the length of a string. The conditions applicable to the Boolean operators are simply encoded as Boolean constraints. Conditions with the `IfExists` suffix check existence of the condition key in the request context. This suffix can be added to other condition operators such as `StringEquals` which results in a new operator `StringEqualsIfExists`. The resulting operator can be applied to the `aws:sourceVpc` condition key. For example:

$$\begin{aligned}&(\texttt{StringEqualsIfExists}, \texttt{aws:sourceVpc}, \textit{"vpc-111bbb222"}) \mapsto \\ &\quad \textit{awsSourceVpcExists} \implies \textit{awsSourceVpc} = \textit{"vpc-111bbb222"}\end{aligned} \tag{8}$$

	Z3	CVC4	Z3AUTOMATA
UNSAT	965,092	34,908	0
SAT	959,543	39,932	525

Fig. 8. Number of times each solver was the fastest for one million UNSAT and one million SAT property checks.

IV. Z3AUTOMATA

Z3AUTOMATA is an in-house extension of Z3 designed to provide a complete decision procedure for the theory of regular expressions. As described in Section III, ZELKOVA uses the regular expressions for problems that involve more than two $*$ wildcards, any ? wildcards, or tricky combinations such as mixing case-sensitive and case-insensitive string comparisons. Such cases are rare in general, but common at our scale where we receive many millions of queries every day.

Z3 and CVC4 aim to efficiently solve problems over word equations, a strictly more general problem than regular expression matching. This sometimes results in degraded performance for pure regular expression problems. For example, both fail to answer the query "Does there exist a string that matches *'ab∗b∗b∗b'* but not *'a∗b∗b∗b'*?". More generally, both solvers seem very sensitive to small changes in the input encoding, where a quickly solved problem in our domain becomes non-terminating. Yet, the theory of regular expressions is decidable, and our problems stay within that theory. Thus Z3AUTOMATA fills an important niche for our domain.

Fig. 8 shows which solver was the fastest for one million UNSAT and one million SAT Zelkova property checks, both randomly selected. Note that for UNSAT problems, Z3AUTOMATA is never the fastest solver. The SMT problems that ZELKOVA generates contain a mix of both simple and complex string constraints. For the properties that ZELKOVA checks, an UNSAT result is, in our experience, always due to the simple string constraints being unsatisfiable. Z3 and CVC4 can easily and efficiently handle that case, thus Z3AUTOMATA never wins. In the case where the constraints are satisfiable, all the constraints must be considered including the complex ones. Here, Z3AUTOMATA is able to win, often in cases where Z3 and CVC4 are non-terminating.

Z3AUTOMATA solves regular expression problems using the standard translation to deterministic finite automata (DFAs) via non-deterministic finite automata (NFAs). It uses Hopcroft's algorithm for DFA minimization [15]. Z3AUTOMATA is parametric with respect to the character set and strives to produce strings using only the printable subsets of a character set. The

```
( allow,
  principal  :  *,
  action     :  getObject,
  resource   :  *,
  condition  :  (StringEquals, aws:sourceVpc, vpc-111bbb222)))
( allow,
  principal  :  *,
  action     :  putObject, listBucket, ...,
  resource   :  *)
```

Fig. 9. A policy check that allows *getObject* requests only from *vpc-111bbb222*.

full range of regular expression (and automata) features are supported including intersection, union, and complement.

Z3AUTOMATA currently integrates with Z3 only on the SAT level and treats each regular expression match as an atom. A good future challenge for the SMT community to solve is how to integrate this into the traditional Nelson-Oppen framework.

V. ZELKOVA PROPERTIES

Organizations using cloud services want assurances that policies being authored or modified by users do not violate general security best-practices, adhere to the security guidelines defined by the organization, and do not deny access to the intended users. Examples of these properties are as follows: *"Ensure that unrestricted public write is not allowed to a particular resource."* (security best-practice), *"Ensure access to a resource is only allowed from a certain range of IP addresses."* (organizational security check), and *"Ensure a particular user is allowed to perform a specific action on a resource"* (availability property). These properties can be specified in the policy language and checked by ZELKOVA. Verification of properties by ZELKOVA provides assurance that there are no inappropriately configured resources within an organization.

A. Organizational security checks

We use the example in Section II to describe how an organization can specify a property in the policy language such that it can be checked by ZELKOVA. The example in Fig. 2(b) allows principal "$*$" access to the *cs240* resource and denies *students* access to *Answer.pdf*. The principal being set to a wildcard can lead to unauthorized access of objects by users who are not members of the University as described in Section II. As a safeguard measure, suppose, the University administrator wants to ensure that there is no unauthorized access to data in the buckets. The administrator and the security lead of the University decide that an appropriate property to check would be "*the getObject action on the CS department S3 buckets is only allowed on requests from vpc-111bbb222.*" The VPC is owned by the University, and so access requests from within the VPC are trusted.

A policy that specifies the property, "*getObject actions are only allowed from vpc-111bbb222*" is shown in Fig. 9. The first allow statement in Fig. 9 permits *getObject* only when the request comes from *vpc-111bbb222*. The second allow statement permits all other unrelated actions that are not relevant to the comparison. The policy in Fig. 9 represents a desired upper bound on the set of request contexts that should be allowed. This bound will only be violated if the input policy allows a request which Fig. 9 does not allow. In such a case, the request must be a *getObject* request (since all other requests are allowed by the second allow statement in Fig. 9) and it must come from outside of *vpc-111bbb222* (since all *putObject* requests inside the VPC are allowed by the first allow statement). Such a request would indeed violate the proposed property. On the other hand, if ZELKOVA shows that the input policy implies the policy in Fig. 9 then the upper bound is establish and the proposed property holds true.

```
(( allow,
   principal  :  *,
   action     :  sendMessage,
   resource   :  *,
   condition  :  (ArnEquals, aws:sourceArn, mytopic)))
```

(a)

```
(( allow,
   principal  :  *,
   action     :  sendMessage,
   resource   :  *,
   condition  :  (ForAllValues:ArnEquals, aws:sourceArn, mytopic)))
```

(b)

Fig. 10. Policies constrained by `aws:sourceArn`. (a) Policy does not allow world writability. (b) `ForAllValues` semantics allow world writability.

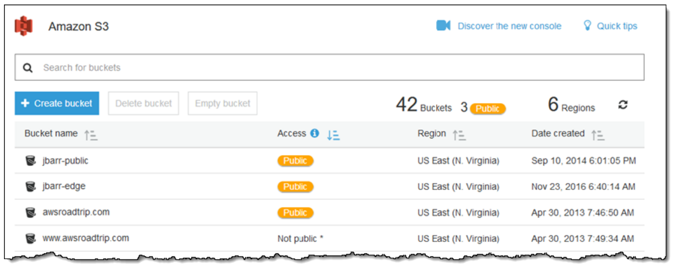
B. Security best-practices

ZELKOVA supports several built-in checks that can be leveraged to check a variety of security best-practices. Examples of these include checking whether a policy allows world accessibility for services such as Amazon S3, Amazon SQS, Amazon SNS, Amazon Glacier, Amazon Elasticsearch, and AWS Lambda. These AWS services provide compute, storage, messaging, and search capabilities. These checks are used internally by AWS to check adherence to security best practices and also available to external customers through services such as Amazon Macie, AWS Config, AWS Trusted Advisor, and the Amazon S3 console. The built-in checks provide greater security assurances without requiring the users to define the properties.

Consider the case of Amazon SQS, a fully managed message queueing service. ZELKOVA provides a built-in check for whether an Amazon SQS policy is world accessible. Fig. 10(a) shows an example SQS policy which which allows *sendMessage* to any resource by any principal, predicated on a condition. The condition restricts the source (`aws:sourceArn`) of the message to be a specific source (*mytopic*). A similar policy is shown in Fig. 10(b). Here, the operator `ForAllValues:ArnEquals` is applied to the condition `aws:sourceArn` whose value is restricted to *mytopic*. The semantics for the operator prefix `ForAllValues` states that if the condition *aws:sourceArn* exists, then its value is *mytopic*. The SMT formula for that is as follows:

$$\mathtt{awsSourceArnExists} \implies (\mathtt{awsSourceArn} = \mathit{mytopic})$$

When a request context does not have the condition key `aws:sourceArn` set, the above formula is true. Thus any


Amazon S3

Discover the new console · Quick tips

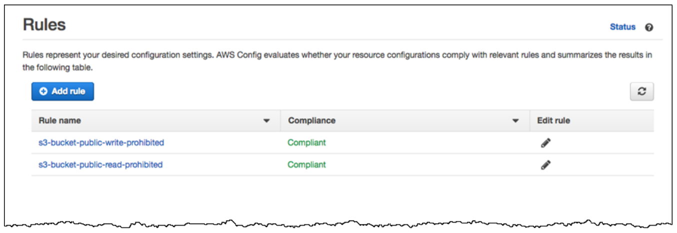
Search for buckets

Create bucket · Delete bucket · Empty bucket

42 Buckets 3 Public 6 Regions

Bucket name	Access	Region	Date created
jbarr-public	Public	US East (N. Virginia)	Sep 10, 2014 6:01:05 PM
jbarr-edge	Public	US East (N. Virginia)	Nov 23, 2016 6:40:14 AM
awsroadtrip.com	Public	US East (N. Virginia)	Apr 30, 2013 7:46:50 AM
www.awsroadtrip.com	Not public *	US East (N. Virginia)	Apr 30, 2013 7:49:34 AM

Fig. 11. S3 Console: Buckets marked Public or Not Public using ZELKOVA checks.


Rules

Status

Rules represent your desired configuration settings. AWS Config evaluates whether your resource configurations comply with relevant rules and summarizes the results in the following table.

Add rule

Rule name	Compliance	Edit rule
s3-bucket-public-write-prohibited	Compliant	✎
s3-bucket-public-read-prohibited	Compliant	✎

Fig. 12. Rules in AWS Config that check public read is prohibited (s3-bucket-public-read-prohibited) and public write is prohibited (s3-bucket-public-write-prohibited) for an S3 bucket using ZELKOVA.

principal can send a message to the SQS queue. The ZELKOVA built-in check for SQS world accessibility correctly marks Fig. 10(a) as not world accessible and Fig. 10(b) as world accessible.

VI. INDUSTRIAL EXPERIENCE

ZELKOVA is integrated in many AWS services including Amazon S3, AWS Config, Amazon Macie, AWS Trusted Advisor, and Amazon GuardDuty. In addition, ZELKOVA is used by an internal security auditor by the AWS Security team.

The Amazon S3 Console is a web-based interface where users can provision buckets; manage buckets, objects, and folders; and set permissions to buckets and objects. A recent release of the console added a view showing whether a bucket is publicly accessible (`Public`) or not (`Not Public`). The underlying check is performed by ZELKOVA. Fig. 11 shows an example of this view.

AWS Config currently supports several managed rules based on ZELKOVA[4], such as a check for AWS Lambda Functions granting unrestricted access, a check for S3 buckets granting unrestricted read access, a check for S3 buckets granting unrestricted write access, deny *putObject* requests that do not have server side encryption, and deny actions that do not allow https traffic. Config will trigger a new ZELKOVA-based check whenever a new resource is created or the policy attached to it is changed. Using the Config console, customers can determine compliance of their S3 buckets against these rules, as shown in Fig. 12, and receive notifications when permissions change or view the permissions history in the console. The checks

[4]https://docs.aws.amazon.com/config/latest/developerguide/managed-rules-by-aws-config.html

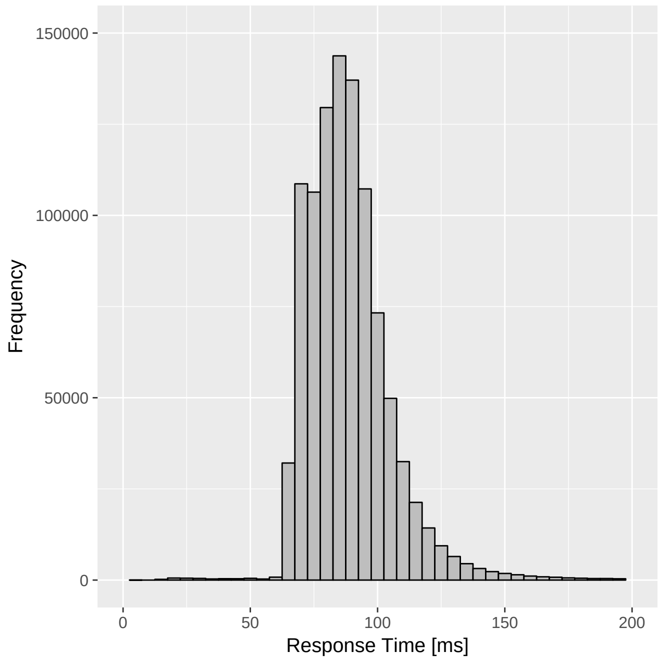


Fig. 13. Performance of ZELKOVA on one million random policy questions

available in the Amazon Macie and AWS Trusted Advisor services are similar to those in AWS Config.

ZELKOVA is used by internal security auditing tools, owned by the AWS Security team, that scan all *internal* AWS accounts to check for unintended configurations of resources. Internal accounts are all AWS accounts owned by the AWS development teams and personnel. These include policies attached to various resources such as S3 buckets, SQS queues, SNS topics, Glacier Vaults, KMS Keys, ElasticSearch Domains, and AWS Lambda Functions. The security auditing tools periodically scan all the resources and check compliance of the resources policies according to the security best practices. Violations of checks are automatically ticketed as discovered, assigned to the owners, and automatically resolved when policies are fixed. The auditing tools require no manual intervention by the security engineering team.

While the checks available in Amazon Macie, AWS Config, Amazon S3 Console, and AWS Trusted Advisor check safety properties, the ZELKOVA integration in Amazon GuardDuty checks for an availability property. ZELKOVA ensures that the requisite permissions are enabled in a user's policy when they are on-boarding onto the service.

A. *Implementation*

ZELKOVA runs on AWS Lambda, a serverless computing platform that runs applications without users needing to provision or manage servers. The input to ZELKOVA is a JSON structure that consists of the policies that are being compared, or a policy and the name of a built-in ZELKOVA check. The response from ZELKOVA is also a JSON structure with the answer to the query. For a comparison of policies, it returns whether the first policy in the payload is *less permissive*, *more permissive*, *equivalent*, or *incomparable* with respect to the second policy in the payload. For each of the built-in checks, ZELKOVA takes a policy and returns *true* or *false* based on whether the check is satisfied. If ZELKOVA is unable to handle any construct in the policy or the solver times out, it returns *unknown*.

ZELKOVA uses the solvers Z3, Z3AUTOMATA, and CVC4 in the backend to solve queries [16], [17]. The solvers provide a combination of string, regular expression, bit vector, and integer comparison theories. ZELKOVA invokes the solvers in parallel and returns the results as soon as one of the solvers provides the answer. We use the Z3 solver with its traditional sequence string solver. Experiments with other solvers such as Z3Str3 [18] and other automata-based solvers [19] is part of our future work.

B. Usage statistics

The total number of invocations of ZELKOVA ranges from a few million to tens of millions in a single day. The number of invocations varies based on the services invoking ZELKOVA. Certain services invoke ZELKOVA at some regular cadence, *e.g.*, the internal security auditing tools, while other services, *e.g.*, AWS Config, invokes ZELKOVA when a change is detected in the policies.

Fig. 13 shows the performance of ZELKOVA on one million randomly selected policy questions. These contain both policy comparisons and built-in checks. The total time includes time to parse the input JSON, encode the policies into SMT, perform the check, and construct the resulting JSON that is returned. The y-axis represent the count, i.e., number of policies solved within the time. The graph shows that 99% of policies are solved within 160 milliseconds.

VII. CONCLUSION

In this paper, we have presented a formalization of the AWS policy language that controls access to resources. This is the first instance of formalizing the AWS policy language as SMT formulas. The advantage of this approach is that it allows us to use off-the-shelf SMT solvers to verify safety and availability properties. Given the distributed nature of the policy language where different services establish their own list of condition keys, this work provides a single consolidated service to reason about the semantics of policies applicable across different services in AWS. The previous state of the art in policy checks for AWS services used syntactic checks for policies. Alternatively, given a concrete request context, the policy evaluation engine allows users to test access control. In contrast, our formalization into SMT provides the ability to soundly reason about properties of a policy for all valid request contexts.

For customers of AWS services, ZELKOVA provides deeper insights into the policy language, its semantics, and its implications. The tool enables customers to automatically maintain their security posture. For people in the SMT and verification community, this work shows how SMT can verify properties of a complex industrial policy language that is used by millions on a daily basis. Moreover, this work is one of the largest and most widespread uses of formal methods in industry.

There are two avenues of future work. One avenue is to improve the existing functionality provided in ZELKOVA. This includes further work on Z3AUTOMATA to make it more competitive. The second avenue is to enhance the functionality of the ZELKOVA engine itself. For example, we want to add support in ZELKOVA to return to the user a concrete request context using the model generated by the SMT solver when performing the check. The concrete request context will provide information to the user on why a certain check passed or failed. We also want to add support for recommending policy repairs in cases when the policy fails a certain check.

REFERENCES

[1] D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Specifying and reasoning about dynamic access-control policies," in *IJCAR*, vol. 4130. Springer, 2006, pp. 632–646.

[2] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on.* IEEE, 2005, pp. 196–205.

[3] D. P. Guelev, M. Ryan, and P.-Y. Schobbens, "Model-checking access control policies," in *ISC*, vol. 3225. Springer, 2004, pp. 219–230.

[4] G. Hughes and T. Bultan, "Automated verification of access control policies using a SAT solver," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 10, no. 6, pp. 503–520, 2008.

[5] G. Kolaczek, "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, 2003, pp. 190–195.

[6] N. Li, B. N. Grosof, and J. Feigenbaum, "Delegation logic: A logic-based approach to distributed authorization," *ACM Transactions on Information and System Security (TISSEC)*, vol. 6, no. 1, pp. 128–171, 2003.

[7] M. Y. Becker and P. Sewell, "Cassandra: Flexible trust management, applied to electronic health records," in *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE.* IEEE, 2004, pp. 139–154.

[8] J. DeTreville, "Binder, a logic-based security language," in *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on.* IEEE, 2002, pp. 105–113.

[9] T. Jim, "SD3: A trust management system with certified evaluation," in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on.* IEEE, 2001, pp. 106–115.

[10] N. Li and J. C. Mitchell, "Datalog with constraints: A foundation for trust management languages," in *Padl*, vol. 3. Springer, 2003, pp. 58–73.

[11] A. Anderson, A. Nadalin, B. Parducci, D. Engovatov, H. Lockhart, M. Kudo, P. Humenn, S. Godik, S. Anderson, S. Crocker *et al.*, "Extensible access control markup language (xacml) version 1.0," *OASIS*, 2003.

[12] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati, "An algebra for composing access control policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 5, no. 1, pp. 1–35, 2002.

[13] E. Bertino, P. A. Bonatti, and E. Ferrari, "TRBAC: A temporal role-based access control model," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 191–233, 2001.

[14] K. Jayaraman, N. Bjorner, G. Outhred, and C. Kaufman, "Automated analysis and debugging of network connectivity policies," Microsoft Research, Tech. Rep. MSR-TR-2014-102, 2014.

[15] J. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds. Academic Press, New York, 1971, pp. 189–196.

[16] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *International Conference on Computer Aided Verification.* Springer, 2011, pp. 171–177.

[17] L. de Moura and N. Bjørner, “Z3: An efficient SMT solver,” *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[18] M. Berzish, V. Ganesh, and Y. Zheng, “Z3str3: A string solver with theory-aware heuristics,” *Formal Methods in Computer-Aided Design FMCAD 2017*, vol. 10, no. 14, p. 55, 2017.

[19] A. Aydin, L. Bang, and T. Bultan, “Automata-based model counting for string constraints,” in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 255–272.