RELIA: Accelerating the Analysis of Cloud Access Control Policies

Dan Wang*, Peng Zhang*, Zhenrong Gu*, Weibo Lin†, Shibiao Jiang†, Zhu He†, Xu Du†, Longfei Chen†, Jun Li†, and Xiaohong Guan*
*Xi'an Jiaotong University, †Huawei Cloud

***Abstract*—With the diversification of cloud services, cloud providers offer flexible access control by letting users apply fine-grained cloud access control policies to secure their cloud resources. However, flexibility comes with the cost that configuring cloud access control policies is error-prone. Therefore, cloud providers have developed SMT-based tools to formally analyze the user-defined policies. Unfortunately, we find these analyzers slow, due to the complex *regular expression matching* conditions in policies. To this end, this paper introduces RELIA, a general method to speed up the analysis of cloud access control policies. The key idea of RELIA is to pre-compute a set of *String Equivalence Classes (SECs)* based on the regular expressions in a policy, assign a unique integer to each SEC, and rewrite the regular constraints into equivalent integer constraints, which are easier to solve. We implement RELIA as a transparent layer between our in-house access analyzer and off-the-shelf SMT solvers. Based on real policies from a large public cloud provider, we show that: when enabling RELIA, our in-house portfolio solver (consisting of Z3, CVC4, and CVC5) can speed up the analysis process for nearly 95% of all cases, with an average speedup of 8.21×.**

I. INTRODUCTION

With the surge of public cloud services, many companies are outsourcing their IT infrastructure and tasks to the cloud. To secure those valuable digital assets, major cloud service providers (CSPs), *e.g.*, Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform, enable users to configure fine-grained access control policies. However, due to the flexible policy language, configuring cloud access control policies is complex and error-prone. A misconfiguration in the policies can lead to security risks, such as exposing private resources to the public [1]–[4], or allowing attackers to gain unauthorized access by escalating privileges [5].

To help cloud users ensure the correctness of their access control policies, the CSPs develop tools that can automatically analyze the correctness of policies. For example, ZELKOVA [6] from AWS encodes policies and user intents as SMT constraints, and uses off-the-shelf SMT solvers (*e.g.*, Z3, CVC4/5) to check whether the policies satisfy user intents, *e.g.*, a resource should not allow public access. Based on ZELKOVA, another tool ACCESSSUMMARY [7] computes a set of declarative statements named "findings" that summarize the access control policies.

In Huawei Cloud, we have developed similar tools (referred to as *analyzers*) for analyzing the correctness of cloud access control policies, also based on SMT solvers. Table I shows the five analyses that the analyzer supports. Evaluation results show that general-purpose SMT solvers (*e.g.*, Z3 [8], CVC4 [9], and CVC5 [10]) are slow when analyzing some complex policies. For example, Z3 timed out (after 3 seconds) for 26% of real-world policies; CVC4 and CVC5 timed out for all the 10 synthesized policies (see §VIII).

A major reason for the inefficiency of SMT solvers is that cloud access control policies extensively use regular expressions (RegExps) and string concatenations to define access conditions, which will be translated into complex string constraints. Figure 2(a) shows an example policy saying that "every member of the course `cs101` in the fall semester of 2025 (*i.e.*, `fall25`), except students, can access all answers to the exams of this course". This policy translates into two regular constraints: (1) the uniform resource name (urn) should satisfy a RegExp `"cs101/.*:fall25/.*"` (Line 7), and (2) the user identifier (id) should not start with `"fall25/stu"` (Line 10), and one concatenation constraint: *i.e.*, urn consists of org (user organization), `:`, and id.

Solving a combination of regular and concatenation constraints was shown to be PSPACE-complete [11]–[13], making string solvers quite inefficient. Researchers have proposed excellent string solvers for speedup. Some of them [14]–[25] use incomplete algorithms, *i.e.*, not guaranteed to terminate, while others [26]–[40] only work for a limited fragment, such as acyclic [29], [30], straight-line [35], [40], and chain-free [31]. The string logic used by cloud access control (see §IV-B) falls in the chain-free fragment, which Z3-NOODLER [34], the QF_S track winner in SMT-COMP 2024, targets. Evaluation results (§VIII) show that Z3-NOODLER is generally faster than Z3 and CVC4/5 when analyzing real-world policies. However, it is still not supportive and efficient enough for analyzing cloud access control policies. Specifically, it times out on 5 real policies, aborts on both real and synthesized policies due to not supporting arrays and quantifiers, and returns incorrect answers when getting variable assignments on satisfiable constraints.

This paper presents RELIA, a general method for accelerating the analysis of cloud access control policies. Instead of optimizing the analyzer or the solver, RELIA is a layer sitting in-between them, which rewrites regular constraints produced by the analyzer into equisatisfiable integer constraints that can be solved faster by off-the-shelf solvers.

Such a constraint rewriting is based on the following observation: even though the space of all strings is infinite, there are only a small number of equivalence classes, termed *string equivalence classes (SECs)*, such that strings in the same class are "equivalent", i.e., satisfying the same set of regular constraints. Based on this observation, we can pre-compute the SECs, each of which is identified by a unique integer, and then rewrite a regular constraint $s \in re$ in an equivalent form of integer constraint $s_{int} \in SECs(re)$. Here, s_{int} is the integer, and $SEC(re)$ is a set of integers representing those SECs that satisfy re. Based on 506 real policies from Huawei Cloud, we observe that the number of SECs is always less than 12.

The above idea of pre-computing SECs and is inspired by *network verification*, where the space of all packets is too huge to verify (*e.g.*, 2^{104} for IPv4 5-tuple). Researchers observed that the number of different forwarding behaviors in a network is often small (*e.g.*, $\leq 10^3$). Based on this observation, network verifiers choose to pre-compute the *packet equivalence classes (PECs)*, and check the forwarding behaviors of these PECs, rather than all possible packets [41], [42].

A hurdle to rewriting regular constraints is that those constraints are often applied on a concatenation of multiple string variables. As an example, consider two constraint $s \in re$ and $s = s_1 \cdot s_2$, where $\cdot$ means string concatenation. Simply rewriting $s \in re$ to $s_{int} \in SECs(re)$ is problematic since $s \in re$ implicitly places some constraints on s_1 or s_2, and if s_1 or s_2 already have their own constraints, say $s_1 \in re'$, we will miss the potential conflicts between $s_1 \in re'$ and the implicit constraints. To handle this challenge, we propose an efficient method to split the RegExp for a concatenated string s into a set of sub-RegExps, so as to equivalently transform the regular constraint on s to a conjunction of regular constraints on sub-strings of s.

We realized RELIA as a transparent layer sitting in-between the analyzer and the SMT solver: it intercepts all SMT constraints from the analyzers, rewrites those string constraints into integer constraints, and invokes the underlying SMT solvers to solve the new set of constraints. Such a realization makes RELIA agnostic of both the analyzer and the SMT solver, so that the analyzer can evolve independently of RELIA, and RELIA can be applied to all SMT solvers conforming to the SMT-LIB2 standard [43] (*e.g.*, Z3, CVC4/5).

Contributions. In summary, we make the following contributions:

- Our evaluation results show that current analyzers of cloud access control policies are slow and even time out after several seconds. We identify the major reason is that general-purpose SMT solvers are inefficient in solving complex regular constraints on concatenated strings.
- We propose RELIA, a general way to accelerate the analysis of cloud access control policies, by first decomposing the regular constraints on concatenated strings into equivalent regular constraints, and then rewriting the regular constraints (RE) into equivalent integer constraints (LIA).
- We implement RELIA based on an in-house portfolio SMT solver consisting of Z3, CVC4, and CVC5. We use both real and synthesized policies to show that (1) for 60% of cases, RELIA speeds up Z3, CVC4, and CVC5 when analyzing synthesized policies, by 2.68×, 7.99×, and 1.72×, respectively; (2) for nearly 95% of cases, RELIA speeds up the portfolio solver when analyzing real policies, with an average speedup ratio of 8.21×.

Policy	$\rightarrow$	`Statements:` [*Statement**]
Statement	$\rightarrow$	{*Effect, Action, Resource, Condition?*}
Effect	$\rightarrow$	`Effect:` *"Allow"* | `Effect:` *"Deny"*
Action	$\rightarrow$	`Action:` [*string**]
Resource	$\rightarrow$	`Resource:` [*string**]
Condition	$\rightarrow$	`Condition:` {*Operator**}
Operator	$\rightarrow$	*Quantifier? Op*: {*Attr*: [*Value**]}
Op	$\rightarrow$	`StringEquals` | `StringMatch` | $\cdots$
Quantifier	$\rightarrow$	`ForAllValue:` | `ForAnyValue:`
Attr	$\rightarrow$	*urn* | *org* | *id* | *calledVia* | $\cdots$
Value	$\rightarrow$	*string* | *num* | *bitvector* | *bool* | $\cdots$

Fig. 1: Simplified abstract syntax for the cloud access control policy language of Huawei Cloud. '?' denotes optional elements and '*' denotes a list of valued elements.

TABLE I: The five different analyses of cloud control policy.

Analysis	Explanation
`SomeAccess`	Does a policy permit at least one access?
`PublicAccess`	Does a policy permit public access?
`NoNewAccess`	Does an updated policy permit any new access?
`Compare`	Which of two policies is more permissive?
`Findings`	Concise summary of permitted accesses.

II. ANALYSIS OF CLOUD ACCESS CONTROL POLICIES

A. Cloud Access Control Policy Language

Figure 1 shows a simplified syntax for the cloud access control policy language in Huawei Cloud. The syntax is mostly the same as that of AWS [6], with some minor differences in also including quantifiers, arrays, and bit vectors. As shown in Figure 1, a policy is a list of *statements*, where each statement defines a set of constraints over some *attributes* (*e.g.*, urn), meaning that when the constraints are satisfied, some specific *actions* on the corresponding *resources* will be *allowed/denied*.

Figure 2(a) shows an example policy, which grants access to exam answers to non-student members of the course `"cs101"` in fall 2025. As we can see, the statement specifies the conditions to `allow` (effect) the `"GetObject"` (action) on the exam answers `"exams/answer.*"` (resource). The `Condition` requires that the uniform resource name (urn) of the user should match `"cs101/.*:fall25/.*"`, while her id must not start with `"fall25/stu"`. Here, urn is a concatenation of org, `:`, and id.

A cloud access control policy can also have some complex string constraints with array or quantifier [44]. For example, when a user initiates an access request to a cloud service, the service may forward the request to another service. Then, an attribute termed $calledVia$ is defined as a list of services in the chain that send requests on behalf of the user. Operators can use quantifiers like `ForAllValues:*` and `ForAnyValue:*` on the $calledVia$ array [45].

B. Analysis of Cloud Access Control Policies

Table I shows the five types of analysis supported by Huawei Cloud, including: `SomeAccess`, `PublicAccess`, `NoNewAccess`, `Compare`, and `Findings`. All these analyses depend on the ability to encode the semantics of a policy into a set of SMT constraints. Formally, given a policy P, its semantics $[\![P]\!]$ is defined as:

$$[\![P]\!] := [\![I]\!] \wedge [\![U]\!], \qquad (1)$$

where $[\![I]\!]$ is a set of *intrinsic constraints* for the policy language shared by all policies; and $[\![U]\!]$ is set of *user-defined constraints* that are specific to policy P, defined as:

$$[\![U]\!] := \left(\bigvee_{S \in Allow} [\![S]\!] \right) \wedge \neg \left(\bigwedge_{S \in Deny} [\![S]\!] \right), \qquad (2)$$

where $Allow$ (resp. $Deny$) represents the set of statements whose effect is `allow` (resp. `deny`); and $[\![S]\!]$ encodes the semantics of a statement S. Seeing *Action* and *Resource* as another two *Attr*s, we encode the semantics of a statement as:

$$[\![S]\!] := \bigwedge_{(Attr, Values) \in S} \left(\bigvee_{v \in Values} Op(Attr, v) \right), \qquad (3)$$

where we encode $Op(Attr, v)$ according to specific theories. For example, we encode `StringMatch` as regular constraints. In the following, we use an example to illustrate.

Figure 2(b) shows a snippet of the SMT encoding for the policy in Figure 2(a). We omitted the declaration of string variables for simplicity of illustration here. The encoding consists of two parts: (1) Line 1 is an *intrinsic constraint* which states that urn is a concatenation of two strings org and id, with a semicolon `:` in between. (2) Lines 2-13 are a set of *user-defined constraints*. For example, Lines 7-9 and Lines 10-12 are two constraints for strings urn and id, specified with two regular expressions. We term such constraints with regular expressions as *regular constraints*. In addition to explicit constraints expressed with regular expressions, many other string constraints, *e.g.*, $\texttt{prefixof}(\omega, s)$, can also be seen as a regular constraint that s matches any string starting with ω.

III. Overview

A. Key Idea

We propose Relia based on the following observation. Even though the space of all strings is huge, a lot of strings satisfy the same set of regular constraints and thus can be merged into an equivalence class, termed *string equivalence class (SEC)*. We found that the total number of SECs is small for cloud access control policies: on 506 real policies from Huawei Cloud, the total number of SECs is always less than 12.

Based on the above observations, the key idea of Relia is to pre-compute the SECs for each string variable s, based on all regular constraints on s, represent each SEC with a unique integer, and rewrite all regular constraints on s into equivalent integer constraints. For example, let $s \in re$ be such a regular constraint of s, then its equivalent integer constraint is $s_{int} \in SECs(re)$, where s_{int} is the integer, and $SEC(re)$ is a set of integers of SECs that satisfy re. Compared with regular constraints, such integer constraints are relatively easier to solve, assuming that the total number of integers (SECs) is small.

The above idea of pre-computing SECs is inspired by how network verifiers compute equivalence classes over packet headers, in order to speed up the verification [41], [42].

B. Workflow of Relia

We use Figure 2 as an example to walk through the process of Relia. There are two steps.

Step 1: Eliminate concatenations (§V). In cloud access control policies, regular constraints can be over what we call a *compound string*, which is a concatenation of *single strings*. For example, urn is a compound string since $urn = org \cdot \texttt{":"} \cdot id$. If there is a regular constraint on urn, it will impose implicit regular constraints on org and id, and interfere with other regular constraints on them. To handle such concatenations, Relia first splits the regular expressions over a compound string str into multiple sub-expressions over each single string of str. As shown in Figure 2(c), we split the regular expression `"cs101/.*:fall25/.*"` into (`"cs101/.*"`, `":"`, `"fall25/.*"`)[1], and convert the regular constraints on urn into new constraints over its component strings, *i.e.*, org and id. Here, two new constraints, $org \in$ `"cs101/.*"` and $id \in$ `"fall25/.*"`, are added.

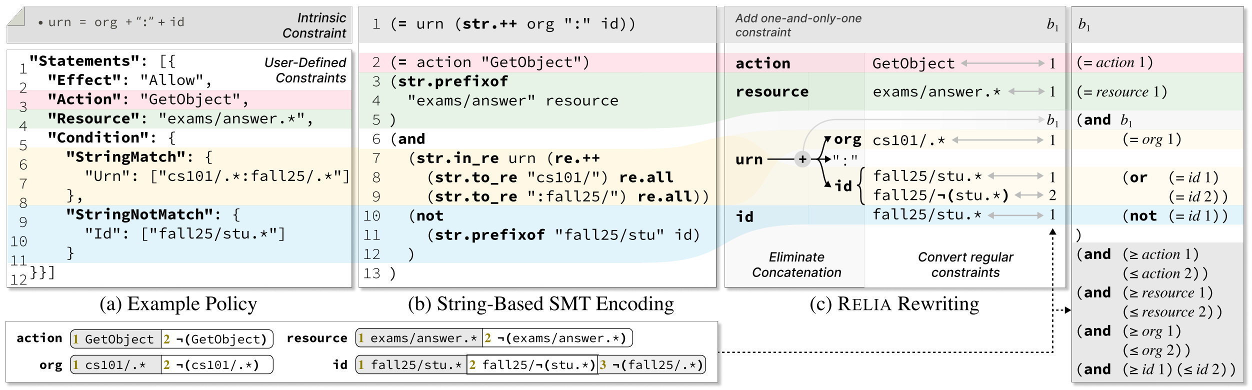
Step 2: Convert regular constraints (§VI). Next, for each string variable, Relia computes the SECs based on the RegExps from the regular constraints for it, and rewrites those regular constraints into integer constraints. Taking id for example, based on the two RegExps for it, *i.e.*, `"fall25/.*"` and `"fall25/stu.*"`, we will obtain three SECs and assign each SEC a unique integer. That is, 1 for `"fall25/stu.*"`, 2 for `"fall25/¬(stu.*)"`, and 3 for `"¬(fall25/.*)"`. Since `"fall25/.*"` = `"fall25/stu.*"` ∪ `"fall25/¬(stu.*)"`, we can represent `"fall25/.*"` as a set of two integers $\{1, 2\}$. Finally, we can rewrite the original regular constraint of id as an equivalent constraint $((id = 1) \vee (id = 2)) \wedge (1 \leq id \leq 3)$.

C. Merits of Relia

Efficiency. By reducing all regular constraints into equivalent LIA constraints, Relia makes the original problems easier to solve with off-the-shelf SMT solvers. For example, Relia can solve some hard instances that time out when using Z3, and CVC4/5 (Figure 4 provides such an instance).

Generality. The conversion of Relia happens after the analyzers have generated the constraints, but before invoking an SMT solver. Therefore, Relia can be generally applied to any off-the-the-shelf SMT solvers (*e.g.*, Z3, Cvc4/5), while independent of the analyzer's application logic.

[1]There's only one split case since in our policy language, org and id do not contain `:`.


• urn = org + ":" + id
Intrinsic Constraint
User-Defined Constraints
Add one-and-only-one constraint
b_1
Eliminate Concatenation
Convert regular constraints
action
resource
urn
org
":"
id
GetObject 1
exams/answer.* 1
cs101/.* 1
fall25/stu.* 1
fall25/¬(stu.*) 2
fall25/stu.* 1
action: 1 GetObject, 2 ¬(GetObject)
resource: 1 exams/answer.*, 2 ¬(exams/answer.*)
org: 1 cs101/.*, 2 ¬(cs101/.*)
id: 1 fall25/stu.*, 2 fall25/¬(stu.*), 3 ¬(fall25/.*)

```
"Statements": [{
  "Effect": "Allow",
  "Action": "GetObject",
  "Resource": "exams/answer.*",
  "Condition": {
    "StringMatch": {
      "Urn": ["cs101/.*:fall25/.*"]
    },
    "StringNotMatch": {
      "Id": ["fall25/stu.*"]
    }
}}]
```

(a) Example Policy

```
(= urn (str.++ org ":" id))
(= action "GetObject")
(str.prefixof
  "exams/answer" resource
)
(and
  (str.in_re urn (re.++
    (str.to_re "cs101/") re.all
    (str.to_re ":fall25/") re.all))
  (not
    (str.prefixof "fall25/stu" id)
  )
)
```

(b) String-Based SMT Encoding

(c) RELIA Rewriting

```
b1
(= action 1)
(= resource 1)
(and b1
     (= org 1)
     (or  (= id 1)
          (= id 2))
     (not (= id 1))
)
(and (≥ action 1)
     (≤ action 2))
(and (≥ resource 1)
     (≤ resource 2))
(and (≥ org 1)
     (≤ org 2))
(and (≥ id 1) (≤ id 2))
```

(e) String Equivalence Class (SEC)

(d) LIA-Based SMT Encoding

Fig. 2: An illustration of RELIA's constraint rewrite process. The intrinsic constraints that org and id should not contain `":"` are omitted for simplicity. Similarly, we omitted the keyword `assert` in each constraint.

IV. Notations and Definitions

In this section, we introduce common notations used in this paper, and define the string logic used in cloud access control policies.

A. Notations

Strings. An *alphabet* Σ is a finite set of *symbols/letters*. A *string* ω on the alphabet Σ is a sequence of symbols $a_1 \cdots a_n$, where $a_i \in \Sigma$. As a special case, we use $\varepsilon \notin \Sigma$ to represent an *empty string*, and Σ^* to represent the set of all strings over Σ, including ε. $\omega_1 \cdot \omega_2$ denotes the *concatenation* of two strings ω_1, ω_2. $|\omega|$ denotes the length of a string ω.

Regular Expressions (RegExps). A regular expression R over a finite alphabet Σ is recursively defined as:

$$R \overset{\text{def}}{=} \emptyset \mid \varepsilon \mid a \mid \texttt{.} \mid R^* \mid \neg(R) \mid (R|R) \mid R \cdot R, \tag{4}$$

where $\emptyset$ represents the empty regexp accepting no string; ε represents the empty string; a is a single letter in Σ; $\texttt{.}$ matches any letter in Σ; R^* and $\neg(R)$ represent the Kleene star and negation of a regex, respectively. $(R|R)$ and $R \cdot R$ represents the union and concatenation of two regexps, respectively; The concatenation operator $\cdot$ may be omitted for abbreviation.

Finite Automaton (FA). An FA over a finite alphabet Σ is a tuple $A = (\Sigma, Q, q_0, F, \delta)$, where Q is a finite set of *states*, $q_0 \in Q$ is the *initial* state, $F \subseteq Q$ is a set of *accepting (final)* states, and $\delta \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. An FA is said to be *deterministic* (*i.e.*, DFA) if there is only one possible transition from one state on the same input symbol, or *non-deterministic* (*i.e.*, NFA) if without such a requirement. In this paper, we will use some operations over DFAs, including intersection ($A_1 \cap A_2$), complementation ($\bar{A}_1$), and difference ($A_1 - A_2 = A_1 \cap \bar{A}_2$). A string $\omega = a_1 \dots a_n$ is *accepted* by A if there is a sequence of states $q_0, \dots, q_n$ such that $(q_{i-1}, a_i, q_i) \in \delta$ for every $i \in [1, n]$ and $q_n \in F$.

Languages. Given a finite alphabet Σ, a *language* over Σ is defined as a subset of Σ^*. Suppose L_1 and L_2 are two languages of Σ. we use $L_1 \cup L_2$, $L_1 \cap L_2$, and $L_1 \cdot L_2$ to denote their union (*i.e.*, $\{\omega \mid \omega \in L_1 \vee \omega \in L_2\}$), intersection (*i.e.*, $\{\omega \mid \omega \in L_1 \wedge \omega \in L_2\}$), and concatenation (*i.e.*, $\{\omega_1 \cdot \omega_2 \mid \omega_1 \in L_1, \omega_2 \in L_2\}$) of them, respectively. It is well-known that finite automata and regular expressions generate the same class of languages termed as *regular languages* [46]. In particular, given a RegExp R which generates a language $\mathcal{L}(R)$, this is a linear-time algorithm to construct an FA A, such that the regular language $\mathcal{L}(A)$ accepted by A is the same with $\mathcal{L}(R)$.

B. String Logic in Cloud Access Control Policy

In this section, we introduce a logic, which we call $\mathcal{E}_{\zeta,\gamma}$ for $\underline{\zeta}$oncatenations and $\underline{\gamma}$egular constraints (short for membership constraints in regular languages). We assume a finite alphabet Σ and work with a set $\mathbb{S}$ of string variables.

Syntax. With $s \in \mathbb{S}, c \in \Sigma^*$, the syntax of formulae in $\mathcal{E}_{\zeta,\gamma}$ is defined as follows:

$$\begin{array}{rcll}
\Psi & ::= & \psi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \mid \neg\Psi \mid \exists x.\Psi(x) \mid \forall x.\Psi(x) & \\
\psi & ::= & \zeta \mid \gamma & \\
\zeta & ::= & s = tr & \text{concatenations} \\
\gamma & ::= & s \in R & \text{regular constraints} \\
tr & ::= & c \mid s \mid tr \cdot tr &
\end{array}$$

Every formula Ψ in $\mathcal{E}_{\zeta,\gamma}$ is a Boolean combination of two types of *atoms*, *i.e.*, concatenations (*i.e.*, ζ) and regular constraints (*i.e.*, γ). We denote by $\mathcal{A}(\Psi)$ the set of atoms occurring in Ψ, by $\mathcal{V}(\Psi)$ the set of variables occurring in Ψ, and by $\Sigma(\Psi)$ the set of constant symbols occurring in Ψ.

Substitution. We write $\Psi[\psi/\Psi']$ to denote the formula obtained by substituting in the formula Ψ each occurrence of the atom ψ by the formula Ψ'.

Interpretation and Satisfiability. An *interpretation* is a mapping $\eta : \mathbb{S} \to \Sigma^*$ from string variables to words. If $\eta(\Psi) = \texttt{True}$ then η is a *solution* (also called a *model*) of Ψ, written $\eta \models \Psi$. The formula Ψ is *satisfiable* iff it has a solution, otherwise it is *unsatisfiable*.

In the following, we use $\mathcal{E}_\zeta$ (resp. $\mathcal{E}_\gamma$) to represent the string logic containing only concatenation (resp. regular) atoms.

V. Handling String Concatenations

This section shows how Relia transforms a formula with regular and concatenation constraints (*i.e.*, in $\mathcal{E}_{\zeta,\gamma}$) into an equisatisfiable formula without concatenations (*i.e.*, in $\mathcal{E}_{\gamma}$).

A. Splitting Regular Expressions

To handle concatenation constraint ζ of the form $s = s_1 s_2 \cdots s_m$, where s has a regular constraint $s \in R$, Relia would first split R into a set $\{R_1, R_2, \dots, R_m\}$ such that $s \in R$ is equivalent to $(s_1 \in R_1) \wedge (s_2 \in R_2) \wedge \cdots \wedge (s_m \in R_m)$. Suppose there are N different ways to split R, the function for such a split can be defined as:

$$\texttt{Split}(s, R, \zeta) := \bigvee_{i \in [1,N]} \left(\bigwedge_{j \in [1,m]} s_j \in R_i^j \right) \tag{5}$$

Norn [29] proposed an algorithm for such a split. Suppose we want to transform a regular constraint $s_1 \cdot s_2 \in R$ into a disjunction form $\bigvee_{i=1}^{n}(s_1 \in R_i^1 \wedge s_2 \in R_i^2)$, where the set $\{(R_i^1, R_i^2)\}_{i=1}^{n}$ is finite and only depends on the regular expression R. Let $A = (\Sigma, Q, q_0, F, \delta)$ be the finite automaton for $\mathcal{L}(R)$. The algorithm iterates over all states of the automaton A, and for each state $q_i \in Q$, constructs two automata A_i^1 and A_i^2 whose accepting and initial state is both q_i, respectively. That is, $A_i^1 = (\Sigma, Q, q_0, \{q_i\}, \delta)$ and $A_i^2 = (\Sigma, Q, q_i, F, \delta)$. Then, the algorithm computes $\mathcal{L}(R_i^1) = \mathcal{L}(A_i^1)$ and $\mathcal{L}(R_i^2) = \mathcal{L}(A_i^2)$ for each i.

The above algorithm is general but inefficient, due to the large number of splitting cases. In the worst case, $N = n^m$, where n is the total number of states in the corresponding automaton of R. For our cloud access control policies, however, concatenations are all in the following form:

$$s = \omega_1 \cdot s_1 \cdot \omega_2 \cdot s_2 \cdot \ldots \cdot \omega_n \cdot s_n, \tag{6}$$

where s and s_i ($i \in [1, n]$) are string variables, and ω_i ($i \in [1, n]$) are non-empty constant strings (termed *anchors*) that are supposed to never appear in any of s_i's. Due to the existence of anchors, the number of split cases can be reduced.

As an example, consider $\Psi := (s = s_1 \cdot \texttt{":"} \cdot s_2) \wedge (s \in R)$, where $R = \texttt{"b:..*a"|"x:.*a"}$. The automaton for R is shown in Figure 3(a). Norn's algorithm will consider every two states q_i, q_j ($0 \le i < j \le 5$) as the splitting point between s_1 and $:$, and between $:$ and s_2, respectively. Considering that $\texttt{":"}$ is an anchor that never appears in s_1 or s_2, there are only two split cases, as shown in Figure 3(b)(c), and the equisatisfiable concatenation-free formula of Ψ is $\mathcal{S}(\Psi) := ((s_1 \in \texttt{"b"}) \wedge (s_2 \in \texttt{"..*a"})) \vee ((s_1 \in \texttt{"x"}) \wedge (s_2 \in \texttt{".*a"}))$.

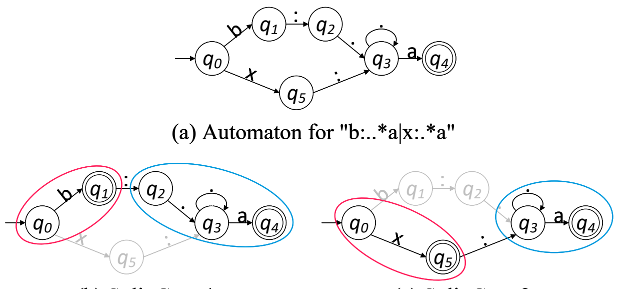
Based on the above idea, we propose a more efficient algorithm, which works as follows. First, we traverse the automaton to determine the location of each anchor ω_i, denoted as a tuple (q_{p_i}, q_{k_i}), with ω_i accepted by $(\Sigma, Q, q_{p_i}, \{q_{k_i}\}, \delta)$. Then, we can build $A_i = (\Sigma, Q, q_{k_i}, \{q_{p_{i+1}}\}, \delta)$ for each $s_i, i \in [1, n)$, and $A_n = (\Sigma, Q, q_{k_n}, F, \delta)$.

B. Eliminating Concatenation Constraints

Given a formula Ψ in $\mathcal{E}_{\zeta,\gamma}$, Algorithm 1 shows how to eliminate all concatenation constraints for a specific string variable s. Before introducing the algorithm, first note that a string s can have multiple ways for concatenation, each represented by a concatenation constraint (*e.g.*, see Figure 4). Let Ψ_ζ be the set of all concatenation constraints on s. Similarly, we use Ψ_γ to denote the set of all regular constraints on s. For each concatenation constraint $\zeta \in \Psi_\zeta$, Algorithm 1 creates a new Boolean variable b_ζ, representing whether s follows the concatenation ζ (line 4), and then replaces the constraint ζ with the boolean variable b_ζ (line 5). Then, for each regular constraint $\gamma_R \in \Psi_\gamma$, the algorithm splits γ_R with each possible ζ (see §V-A), and adds a new constraint (line 7):

$$\mathcal{S}(\gamma_R) = \bigvee_{\zeta \in \Psi_\zeta} \left(b_\zeta \wedge \texttt{Split}(s, R, \zeta)\right). \tag{7}$$

Then, the algorithm substitutes the regular constraint with the above new constraint $\mathcal{S}(\gamma_R)$ (line 8). Finally, we add constraints on b_ζ's to ensure that one and only one concatenation constraint is satisfied (lines 9-10). By executing Algorithm 1 for all string variables, we get an equisatisfiable formula without any concatenation constraints. That is, regular constraints are only placed on simple string variables that are not concatenated by any other string variables.



(a) Automaton for "b:..*a|x:.*a"

(b) Split Case 1

(c) Split Case 2

Fig. 3: The splitting of $s \in R$ ($R = \texttt{"b:..*a"|"x:.*a"}$), given $s = s_1 \cdot \texttt{":"} \cdot s_2$, and s_1, s_2 do not contain $\texttt{":"}$.

Theorem 1. *Given a formula Ψ in $\mathcal{E}_{\zeta,\gamma}$, $\mathcal{S}(\Psi)$ computed by Algorithm 1 is satisfiable if and only if Ψ is satisfiable, assuming that: if there are multiple concatenation constraints on s, then for each solution (if any) of Ψ, one and only one concatenation constraint is satisfied.*

Proof 1. *See Appendix A [47].*

The above requirement of "one and only one concatenation constraint is satisfied" is naturally satisfied by our cloud access control policy language: which concatenation takes effect only depends on the user type, a unique constant in the string (*e.g.*, see Figure 4).

VI. Converting Regular Constraints

This section shows how Relia converts a formula with regular constraints (*i.e.*, in $\mathcal{E}_\gamma$), into an equisatisfiable formula without regular constraints.

```
Algorithm 1: EliminateConcatenation($\Psi, s$)
  Input: A formula $\Psi \in \mathcal{E}_{\zeta,\gamma}$, and a string variable
         $s \in \mathcal{V}(\Psi)$. We use $\Psi_\zeta$ and $\Psi_\gamma$ to denote the set
         of concatenations and regular constraints for
         string $s$, respectively.
  Output: $\mathcal{S}(\Psi)$, an equisatisfiable formula for $\Psi$
          without concatenation constraints on $s$.
1  $\mathcal{S}(\Psi) \leftarrow \Psi$
2  foreach $\zeta \in \Psi_\zeta$ do
3      $b_\zeta \leftarrow$ NewBoolVar()
4      $B \leftarrow B \cup \{b_\zeta\}$
5      $\mathcal{S}(\Psi) \leftarrow \mathcal{S}(\Psi)[\zeta/b_\zeta]$
6  foreach $\gamma_R \in \Psi_\gamma$ do
7      $\mathcal{S}(\gamma_R) \leftarrow \bigvee_{\zeta \in \Psi_\zeta} (b_\zeta \wedge \texttt{Split}(s, R, \zeta))$   // Eq (7)
8      $\mathcal{S}(\Psi) \leftarrow \mathcal{S}(\Psi)[\gamma_R/\mathcal{S}(\gamma_R)]$
9  $\phi \leftarrow \bigvee_{b \in B} b$        // satisfy at least one concatenation
10 $\phi \leftarrow \phi \wedge \left(\bigwedge_{b_1, b_2 \in B, b_1 \neq b_2} \neg(b_1 \wedge b_2)\right)$   // at most one
11 $\mathcal{S}(\Psi) \leftarrow \mathcal{S}(\Psi) \wedge \phi$
12 return $\mathcal{S}(\Psi)$
```

A. String Equivalence Class (SEC)

Given a finite alphabet Σ, every language L defined over Σ ($L \subseteq \Sigma^*$) can be seen as a classifier that divides Σ^* into two classes $C_1 = \{s | s \in L\}$ and $C_2 = \{s | s \notin L\}$, where strings in the same class have equivalent *membership w.r.t.* L (*i.e.*, contained in the language L or not). In the following, we generalize this classification to multiple languages by first defining *string membership* and then introducing *string equivalence class (SEC) w.r.t.* a set of languages.

DEFINITION 1 (STRING MEMBERSHIP). *Given a finite alphabet Σ, and a set of languages $\mathbb{L} \subseteq 2^{\Sigma^*}$, the membership of a string $s \in \Sigma^*$ w.r.t. $\mathbb{L}$, denoted as $\mathcal{M}_{\mathbb{L}}(s, \cdot)$, is a map from each $L \in \mathbb{L}$ to a boolean value, with $\mathcal{M}_{\mathbb{L}}(s, L) = True$ or $False$ representing s is a member of L ($s \in L$) or not ($s \notin L$)*

DEFINITION 2 (STRING EQUIVALENCE CLASS (SEC)). *Given a finite alphabet Σ, and a set of languages $\mathbb{L} \subseteq 2^{\Sigma^*}$, we say $\mathbb{E}(\mathbb{L}) = \{E_1, \ldots, E_n\}$, $E_i \subseteq \Sigma^*$, are a set of string equivalence classes (SECs) w.r.t. $\mathbb{L}$ if:*

1) *$\forall i, 1 \leq i \leq n : E_i \neq \emptyset$. (each SEC should be non-empty.)*
2) *$\bigcup_{1 \leq i \leq n} E_i = \Sigma^*$ (each string should be in at least one SEC.)*
3) *$\forall i, j, 1 \leq i \neq j \leq n : E_i \cap E_j = \emptyset$ (each string should be in only one SEC.)*
4) *$\forall i, 1 \leq i \leq n : s_1 \in E_i, s_2 \in E_i \Rightarrow \mathcal{M}_{\mathbb{L}}(s_1) = \mathcal{M}_{\mathbb{L}}(s_2)$. (all strings in the same SEC should have the same membership.)*

Given a set of languages $\mathbb{L}$, there can be multiple sets of SECs according to Definition 2. In this paper, we are only interested in the *minimum set of SECs*, defined as: *A set of SECs is said to be the minimum set of SECs if the reverse direction of Condition (4) also holds, i.e., if two strings s_1, s_2 have the same membership, then they must be in the same SEC.* The minimum set of SECs is unique, since otherwise the reverse direction of Condition (4) will be violated. Without special notation, "SECs" in this paper refer to "the minimum set of SECs".

LEMMA 1. *Every language L in $\mathbb{L}$ is equal to a union of a subset of SECs,*

$$L = \bigcup_{i \in \mathbb{E}(\mathbb{L}, L)} E_i, \text{ where } \mathbb{E}(\mathbb{L}, L) = \{i \mid E_i \in \mathbb{E}(\mathbb{L}), E_i \subseteq L\} \tag{8}$$

In the following, without special notation, we omit $\mathbb{L}$ in $\mathbb{E}(\mathbb{L})$ and $\mathbb{E}(\mathbb{L}, L)$ for simplicity.

B. Computing SECs

Automaton-based Approach. We show how to compute a set of SECs $\mathbb{E}$ for languages $\{L_1, \ldots, L_n\}$ in Σ^*, based on operations on finite automaton (specifically, DFA). This approach is inspired by how Batfish [42] computes equivalence classes over BGP attributes (specifically, communities and AS path). As shown in Algorithm 2, we compute the automaton A_* for the universe Σ^* (line 1), and initialize the SECs $\mathbb{E}$ with a singleton set of A_* (line 2). Then, for $i \in [1, n]$, we compute the automaton A_i for L_i (line 4). Afterward, we update the current set of SECs $\mathbb{E}$ by replacing each $A \in \mathbb{E}$ with its intersection with A_i, *i.e.*, $A \cap A_i$, and its difference of A_i, *i.e.*, $A - A_i = A \cap \bar{A}_i$ (line 5). Note that in line 6, we remove the FA for $\emptyset$ from the set of SECs, to conform to the first condition of SECs, *i.e.*, each SEC should be non-empty.

```
Algorithm 2: ComputeSEC($\{L_1, L_2, \ldots, L_n\}$)
  Input: A set of languages $\{L_1, L_2, \ldots, L_n\}$
  Output: A set of SECs $\mathbb{E}$
1  $A_* \leftarrow$ ToDFA($\Sigma^*$)
2  $\mathbb{E} \leftarrow \{A_*\}$
3  foreach $i \in [1, n]$ do
4      $A_i \leftarrow$ ToDFA($L_i$)
5      $\mathbb{E} \leftarrow \{A \cap A_i, A \cap \bar{A}_i | A \in \mathbb{E}\}$
6      $\mathbb{E} \leftarrow \mathbb{E} \setminus \{A_\emptyset\}$
7  return $\mathbb{E}$
```

Given two RegExps R_1, R_2, the complexity of computing SECs with Algorithm 2 is $O(2^m)$ in the worst case [48], where m is the maximum total length of strings in R_1 and R_2. However, for most policies we have encountered, the RegExps are *trivial*, meaning that the RegExps are just (unions of) some concrete strings. Take Figure 2(a) for example, the RegExp for the `action` field and `resource` field is `"GetObject"` and `"Answer"`, respectively. For such trivial RegExps, the above automaton-based approach (Algorithm 2) seems an over-kill, and the following set-based approach will be more efficient.

Set-based Approach. For trivial RegExps, instead of converting each RegExp into an automaton (lines 1, 4), we can represent them with sets. Specifically, for RegExp R, we represent it as a tuple (S, c), where S is a set of strings and c is a Boolean flag indicating whether $\mathcal{L}(R)$ is the complementation of S (*i.e.*, $\Sigma^* \setminus S$) or not. For example, the trivial RegExp `"Answer"` can be represented as $(\{\texttt{"Answer"}\}, \texttt{False})$, and Σ^* can be represented as $(\emptyset, \texttt{True})$. Then, the DFA operations in line 5 can be replaced with simpler set operations. Compared to the automaton-based approach as in Algorithm 2, the set-based approach can reduce the complexity of computing SECs. For two trivial RegExps R_1, R_2, the complexity of computing SECs is $O(n)$, where n is the minimum number of strings in R_1 and R_2.

In practice, since there are both trivial and non-trivial RegExps in our cloud access control policies, RELIA adopts a hybrid of automaton-based and set-based approaches. Before computing SECs, RELIA sorts RegExps such that trivial RegExps always appear before non-trivial ones. Then, RELIA uses the set-based approach by default, and "lazily" switches to the automaton-based approach when encountering the first non-trivial RegExps.

Theoretically, the number of SECs can grow exponentially with the number of RegExps (according to line 5). However, for cloud access control policies, the number is often quite small. For example, in our dataset of real policies, the maximum number of SECs is 12, and for 97% of all cases, the number of SECs is less than or equal to 5. Such a small number of SECs is due to the fact that users tend to define meaningful policies with a limited number of RegExps for each string (*e.g.*, 96.7% of the variables have less than or equal to 5 RegExps in our dataset), which rarely overlap.

C. Rewriting Regular Constraints

Given a formula Ψ in $\mathcal{E}_\gamma$, we propose Algorithm 3 to convert all regular constraints of Ψ on variable s into LIA constraints. After executing Algorithm 3 on all string variables in $\mathcal{V}(\Psi)$, we get a regular-free formula, equisatisfiable to Ψ

During conversion, we first initialize a new integer variable s_{int} (line 2). Then, we compute SECs for all RegExps in Ψ_γ (*i.e.*, the set of all regular constraints on s), denoted as $\mathbb{E}$ (line 3). Subsequently, we substitute each regular constraint $\gamma_R \in \Psi_\gamma$ with an equisatisfiable LIA constraint $\mathcal{I}(\gamma_R)$ (lines 6-8). Finally, we bound the range of s_{int}, *i.e.*, $1 \leq s' \leq |\mathbb{E}|$ (line 7). The correctness of this algorithm is given by Theorem 2.

THEOREM 2. *Given a formula Ψ in $\mathcal{E}_\gamma$, $\mathcal{I}(\Psi)$ computed by Algorithm 3 is satisfiable if and only if Ψ is satisfiable.*

PROOF 2. *See Appendix B [47].*

Algorithm 3: `ConvertRegularConstraints`(Ψ, s)

```
Input: A formula $\Psi \in \mathcal{E}_\gamma$, and a string variable
       $s \in \mathcal{V}(\Psi)$. We use $\Psi_\gamma$ to denote the set of
       regular constraints on $s$.
Output: $\mathcal{I}(\Psi)$, an equisatisfiable formula of $\Psi$ with
        regular constraints on $s$ converted to LIA
        constraints.
1 $\mathcal{I}(\Psi) \leftarrow \Psi$
2 $s_{int} \leftarrow$ NewIntVar()
3 $\mathbb{E} \leftarrow$ ComputeSECs $(\Psi_\gamma)$          // Algorithm 2
4 foreach $\gamma_R \in \Psi_\gamma$ do
5     $\mathcal{I}(\gamma_R) \leftarrow \bigvee_{i \in \mathbb{E}(R)} (s_{int} = i)$          // Lemma 1
6     $\mathcal{I}(\Psi) \leftarrow \mathcal{I}(\Psi)[\gamma_R / \mathcal{I}(\gamma_R)]$
7 $\mathcal{I}(\Psi) \leftarrow \mathcal{I}(\Psi) \land (1 \leq s_{int} \leq |\mathbb{E}|)$
8 return $\mathcal{I}(\Psi)$
```

VII. IMPLEMENTATION

Inside Huawei Cloud, we have already developed an *access analyzer* and a *portfolio SMT solver* in Rust, to help cloud users analyze their policies. In the following, we briefly introduce these two software, and show how RELIA is implemented on top of them.

The access analyzer supports five types of analysis shown in Table I. It encodes the policies and properties with SMT formulas conforming to SMT-LIB2, and uses the *portfolio solver* to check the satisfiability of SMT formulas.

The portfolio SMT solver supports three solvers, *i.e.*, Z3, CVC4, and CVC5. In order to let our access analyzer invoke each SMT solver in a unified way, the portfolio solver uses a normalized representation of SMT constraints (called `Term`), and defines a Rust trait (called `Solver`), an interface defining common functions like `assert(term: &Term)`, `check_sat()`, etc. For each SMT solver like Z3, we implemented the `Solver` trait with the APIs of that solver.

RELIA. As one of our design goals, RELIA should be agnostic of both the access analyzers and off-the-shelf SMT solvers, and we achieved this by making RELIA a transparent layer in-between the analyzers and solvers. The portfolio solver provides an ideal place to implement RELIA, since it takes the terms (SMT constraints) from the analyzer as input, and transforms those terms into API calls to individual solvers. Therefore, we choose to disguise RELIA as another solver by implementing the `Solver` trait for it. The RELIA solver takes a set of `Terms` as input, applies constraint rewrite, and outputs another set of `Terms`, which are fed into a real solver instance for constraint solving. The following shows how we implement the functions (defined by the `Solver` trait) for RELIA.

- `assert(term: &Term)`. This function simply adds `term` into the current set of terms, without any further processing.
- `check_sat()`: The function works in 3 step. (1) It first pre-processes the set of terms to filter some terms, via circuit propagation [49]. Then, it solves the propositional SAT problem, and returns if the problem is already USAT. (2) The function transforms the terms by eliminating concatenation constraints (§V) and rewriting the regular constraints (§VI). (3) The function calls the `assert` and `check_sat` of the backend solver (*e.g.*, Z3, CVC4/5) with the transformed terms.
- `get_model()`. The function first calls `get_model()` of the backend solver to get the assignment of the integer

variables, and then translates it into assignment of the original string variables (§VI). For each integer variable, there are two cases to consider depending on what string variables that the integer variable represents: (1) if the string variable is not an concatenation of other strings (*e.g.*, id), then it just need to sample a string from the corresponding SEC; (2) if the string variable is a concatenation of other string variables (*e.g.*, urn), then it first determines what concatenation is used according to the assignment of b_ζ (§V-B), sample a string for each sub-string as in (1), and finally apply concatenation on these sampled strings.

VIII. Evaluation

In this section, we evaluate RELIA to answer the following questions: (**RQ1**) Can RELIA accelerate the access analysis of real policies? (**RQ2**) Can RELIA help analyze harder policies for which using Z3, CVC4/5 times out? (**RQ3**) Can RELIA uniformly accelerate all five types of access analysis? (**RQ4**) Can RELIA generalize to public datasets?

A. Dataset and Setup

In-house dataset. Our in-house dataset includes 506 real policies configured by some cloud tenants (denoted by **"real"**), and 309 policies synthesized by developers of the access analyzer. Some of these synthesized policies (denoted by **"test"**) are used to test the functional correctness of analyses shown in Table I. Some of these synthesized policies (denoted by **"hard"**) are intentionally made to be extremely complex for stress testing purposes. Those policies will be input to our in-house access analyzer, and depending on the specific analysis, the access analyzer generates an encoding for the analysis.

Figure 4 shows an example encoding for `SomeAccess` analysis of a hard policy. Each assert here is a `Term` (SMT constraint), which is either intrinsic or user-defined.

- Lines 1-13 show the intrinsic constraints that are shared by all policies. Line 1 says that *principal_type* should be one of the three types: `User`, `AssumedAgency`, or `ExternalUser`. Lines 2-5 say that if *principal_type* is `User`, then $principal\_urn$ should be a concatenation of `"iam::"`, *principal_account*, `":user:"`, and $user\_name$; and *principal_id* should equal $user\_id$. Lines 6-13 define the concatenations for the other two types in a similar way.
- Lines 14-20 show user-defined constraints that are specific to a policy and an analysis. Lines 14-17 are a regular constraint saying that $principal\_urn$ should match a regular expression (RegExp), with the `str.in_re` operator. The RegExp is complex, specified with a lot of operators like `str.++` (concatenation), `re.all` (RegExp for arbitrary string), `re.allchar` (RegExp for arbitrary character). To make it easier to read, we also write the RegExps in colored strings besides the constraints.

Public dataset of access control policies. We use the AWS IAM policies released by Quacky [50]. This dataset consists of 587 policies, including 41 real policies collected from websites, and another 546 policies that are synthesized by mutating the real policies. Among all the 587 policies, there are 34 AWS Identity and Access Management (IAM) policies, 253 AWS Elastic Compute Cloud (EC2) policies, and 300 AWS Simple Storage Service (S3) policies.

Public dataset of string constraints. To test the ability of RELIA to solve general string constraints, we use the AutomatArk dataset from the SMT-LIB non-incremental QF-S Benchmark [51] Specifically, we selected 1481 out of 15995 instances that contain only string and RegExp operators supported by our in-house portfolio SMT solver. For those 1481 instances, most of them contain only one (88.59%) or two (9.05%) regular membership constraints.

*We released the **"hard"** policies of the in-house dataset and a Java prototype of* RELIA *in [47].*

Setup. Our portfolio solver uses Z3 (v4.12.2), CVC4 (v1.8), and CVC5 (v1.7.0) as backend solvers. We also compare RELIA with Z3-NOODLER (v1.3.0) for the cases Z3-NOODLER supports. All evaluations are done on a Linux server with an Intel(R) Core(TM) i7-10700 CPU running at 2.90GHz and 32GB RAM. Note that performing one analysis on a policy may invoke multiple `check_sat()` calls. All times reported in the figures and tables are for single `check_sat()` calls.

B. (RQ1) Performance for Real Policies

We perform `SomeAccess` and `Findings` analyses on the 506 real policies. We do not perform the other three analyses, since they require additional input which are absent in the real policies. For example, `NoNewAccess` requires an older version of a policy.

Figure 5 shows the *speedup ratio* for each run of `check_sat()` when enabling RELIA in the portfolio solver. Since the real policies are simpler than the synthesized ones, the portfolio solver finishes in 25ms for most of the cases. For 94.96% of all cases, RELIA can speed up the two analyses, with an average speedup of 8.21×, when enabled in the portfolio solver.

C. (RQ2) Performance for Hard Policies

TABLE II: Number of solved hard cases in $10, 10^2, 10^3, 10^4$ milliseconds using different SMT solvers (Z3, CVC4, CVC5, with and without RELIA).

Solver	**Cases solved in X milliseconds (10 cases in total)**			
	X=10	X=10^2	X=10^3	X=10^4
Z3	0	0	0	1
Z3-RELIA	0	4	7	8
CVC4	0	0	0	0
CVC4-RELIA	1	4	7	8
CVC5	0	0	0	0
CVC5-RELIA	1	4	7	8
Z3-NOODLER	0	2	5	5

In order to evaluate whether RELIA is helpful in analyzing much harder policies, we perform `SomeAccess` analysis on

```
(assert (or (= principal_type "User") (= principal_type "AssumedAgency") (= principal_type "ExternalUser"))
(assert (=> (= principal_type "User")
            (and (= principal_urn
                    (str.++ "iam::" principal_account ":user:" user_name))
                 (= principal_id user_id))))
(assert (=> (= principal_type "AssumedAgency")
            (and (= principal_urn
                    (str.++ "sts::" principal_account ":assumed-agency:" agency_name "/" agency_session_name))
                 (= principal_id (str.++ agency_id ":" agency_session_name)))))
(assert (=> (= principal_type "ExternalUser")
            (and (= principal_urn
                    (str.++ "sts::" principal_account ":external-user:" idp_id "/" idp_session_name))
                 (= principal_id (str.++ idp_id ":" idp_session_name)))))
(assert (and (str.in_re principal_urn                sts.*1a2b3c4d:.*assumed-agency..*example.*/ab.*b.*b.*b.*b.*b.*b.*b
                        (re.++ (str.to_re "sts") re.all (str.to_re "1a2b3c4d:") re.all (str.to_re "assumed-agency")
re.allchar re.all (str.to_re "example") re.all (str.to_re "/ab") re.all (str.to_re "b") re.all (str.to_re "b") re.all
(str.to_re "b") re.all (str.to_re "b") re.all (str.to_re "b") re.all (str.to_re "b") re.all (str.to_re "b"))))
             (not (str.in_re principal_id                .*:a.*b.*b.*b.*b.*b.*b.*b
                             (re.++ re.all (str.to_re ":a") re.all (str.to_re "b") re.all (str.to_re "b") re.all
(str.to_re "b") re.all (str.to_re "b") re.all (str.to_re "b") re.all (str.to_re "b") re.all (str.to_re "b")))))))
```

Fig. 4: A snippet of SMT encoding for the `SomeAccess` analysis of a hard policy, in SMT-LIB2 language. All variables in the constraints are string variables, whose declarations are omitted for simplicity.

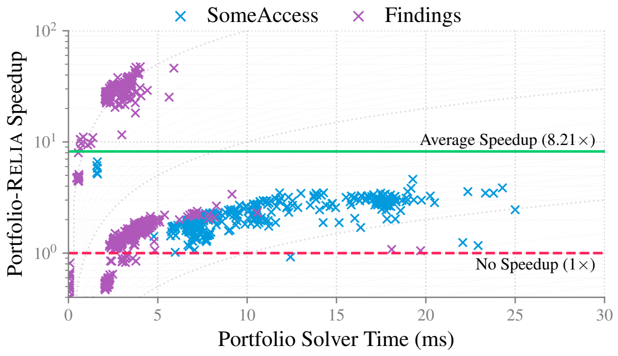


Fig. 5: The speedup ratio for each `check_sat()` when enabling RELIA on top of the portfolio solver. Two types of analysis (`SomeAccess` and `Findings`) are performed on the 506 real policies. The x-axis is the time for the portfolio solver without RELIA.

the 10 hard policies (Figure 4 shows one example). As shown in Table II, Z3 can only solve one policy within 10 seconds, while CVC4 and CVC5 time out for all 10 policies after 10 seconds. In contrast, after applying RELIA, Z3 and CVC4/5 can solve 8 (resp. 7) of these 10 policies in 10 (resp. 1) seconds. Also note that Z3-NOODLER can solve 5 of the 10 policies. For 2 of the 10 policies, Z3-NOODLER aborts since these 2 policies contain arrays or quantifiers, which are not supported by Z3-NOODLER[2].

D. (RQ3) Performance for Different Analyses

To evaluate whether RELIA can speed up different analyses, we compare the performance of Z3, CVC4, and CVC5, with and without enabling RELIA. In addition, we also consider Z3-NOODLER [34], a more related solver specialized for the chain-free fragment [31]. We use these solvers to perform all five analyses (listed in Table I) on both real and synthesized policies.

Note that each analysis task may invoke multiple calls of `check_sat()`. For example, each `Compare` analysis consists of two calls of `check_sat()`. The total number of `check_sat()` calls is 3027.

As shown in Figure 6, RELIA can speed up different analyses when enabled for Z3, CVC4, CVC5, and Z3-NOODLER (by an average of 123×, 9.79×, 5.07×, and 5.06× respectively). In particular, RELIA solves all the cases for which the three vanilla solvers timed out: Z3, CVC4, and CVC5 timed out for 249, 11, and 11 out of the 1539 cases, respectively, but finishes within 232ms when RELIA is enabled. Also note that for the three analyses, `SomeAccess`, `Compare`, and `NoNewAccess`, RELIA speeds up almost all of the cases; for the `PublicAccess` and `Findings` analyses, however, enabling RELIA may slow down the vanilla solvers, especially CVC5, on some simple cases for which these solver can already finishes in 10ms. This implies that RELIA is more suitable for handling harder cases, where the overhead of constraint rewriting pays off.

After further examination, we find that the reason for the degradation on the `PublicAccess` and `Findings` analyses of some simple cases is mainly due to *incremental solving* That is, during each analysis, multiple `check_sat()`'s are called, and the SMT constraints are mostly the same. For example, after generating one finding, the analyzer will add a new constraint which negates this finding, in order to explore new findings. SMT solvers can solve those constraints incrementally, therefore, runs faster after the first call of `check_sat()`. However, since RELIA currently maintains no context of the constraint rewrite, it needs to redo the rewrite for each `check_sat()` call, which imposes a constant overhead on all calls. As a result, after enabling RELIA, these solvers become faster in the first `check_sat()` call, while are even slower in the following several calls. Actually, it is possible for RELIA to re-use the constraint rewrite in the last call, such that it can incrementally rewrite modified or new

[2]We do not enable RELIA on Z3-NOODLER, since after RELIA's rewrite, there will be no string variables, making Z3-NOODLER's optimizations on string variables ineffective.

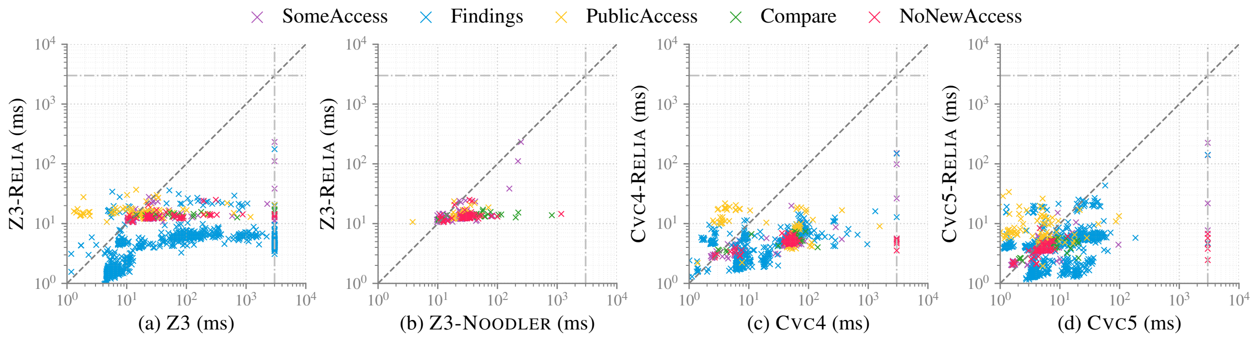


Fig. 6: The runtime of different solvers for the 5 types of analysis, with and without RELIA. Results for Z3-NOODLER on `Findings` are missing, since the analysis needs to invoke `get_model()`, for which Z3-NOODLER returns incorrect answers.

constraints, rather than redo the rewrite from scratch. We leave this as a future work.

E. (RQ4) Performance on Public Datasets

RELIA can generalize to check cloud access control policies of other cloud service providers (CSPs), e.g., AWS, whose policy languages are highly similar to Huawei Cloud's. We show this by using RELIA to check the AWS access control policies released by Quacky [50]. Note there are some minor differences in grammar, say, AWS used *arn* (*i.e.*, amazon resource name) while Huawei Cloud uses *urn* (*i.e.*, uniform resource name) to identify a resource canonically. Therefore, we first convert them into the format of Huawei Cloud's policy, and run `SomeAccess` analysis on them. As shown in Figure 7, RELIA accelerates Z3, CVC4, CVC5, and Z3-NOODLER, by an average of 9.94×, 12.79×, 2.75×, and 6.85×, respectively, when analyzing the policies.

RELIA can also be used to solve general string constraints from the AutomatArk dataset. As shown in Figure 7(f), (g), and (h), RELIA accelerates Z3, CVC4, and CVC5 for 31.3%, 44.9%, and 62.5% of all instances, respectively. However, for Z3-NOODLER, the state-of-the-art solver optimized for the chain-free string fragment, RELIA only accelerates it for 6.8% of all instances. This is mainly because most of the instances are quite simple, consisting of one or two RegExps, and the satisfiability can be determined by checking the non-emptiness of the RegExps. Since Z3, CVC4, and CVC5 check the emptiness of RegExps without using automata, RELIA's automata-based technique complements these solvers on some cases where automata shows better performance. In contrast, Z3-NOODLER also uses automata-based techniques, making the improvement of using RELIA almost diminish, especially considering the overhead that Z3 has to solve the converted LIA constraints from RELIA.

IX. RELATED WORK

Cloud Access Control Policy Analyzers. Besides ZELKOVA [6] and ACCESSSUMMARY [7] mentioned in §I, there are other access control policy analyzers. Shevrin et al. [52] propose a model-checking method to check whether the access control policies permit an attacker to escalate privileges and gain unauthorized access to resources through one or more role manipulations. The method builds a finite-state SMT model of the Identity and Access Management (IAM) system and employs a customized model checker to exhaustively explore all possible execution traces. For each trace, the model checker uses an SMT solver to check whether it allows unauthorized access. Eiers et al. [53] propose a quantitative and differential policy analysis framework that can quantify the relative permissiveness of access control policies, based on the model-counting constraint solver ABC [54], [55]. As we can see, these analyzers also rely on SMT solvers and may benefit from RELIA's constraint optimization.

String Solvers. In string theory, word equations and regular constraints are the most essential constraints and the primary source of difficulty. Their combination is PSPACE-complete [11], [13], decidable by Makanin [56] and Jeż's recompression algorithms [13]. Since it is not known how these general algorithms may be implemented efficiently, string solvers either use incomplete algorithms (*e.g.*, CVC4 [14], the Z3str series [15]–[19], S3 [20], [21], Stranger [22], [23], HAMPI [24], and Kudzu [25]) or work only with restricted fragments (*e.g.*, Norn [29]–[31], Trau [26]–[28], [32], Z3STR3RE [33], SLOTH [39], Ostrich [35], [36], [40], and Z3-NOODLER [34], [37], [38]). Since string constraints for cloud access control policies fall in the chain-free fragment, we expect the fragment-specific string solvers to be more efficient by applying dedicated optimizations. However, we found them still not efficient enough. For example, Z3-NOODLER, QF_S single query track winner in SMT-COMP 2024, timed out when analyzing some of our cloud access control policies.

Network verifiers. Network verification refers to a set of tools that apply formal methods to ensure the correctness of network configurations. An important idea underlying many network verifiers [41], [42], [57]–[60] is *packet equivalence class (PECs)*. Each PEC represents a set of packets with the same forwarding behaviors in a network. Especially, AP Verifier [41], a pioneer network verifier, proposed to compute the minimum number of PECs (termed as *atomic predicate*), based on Binary Decision Diagrams (BDDs). Then, Batfish

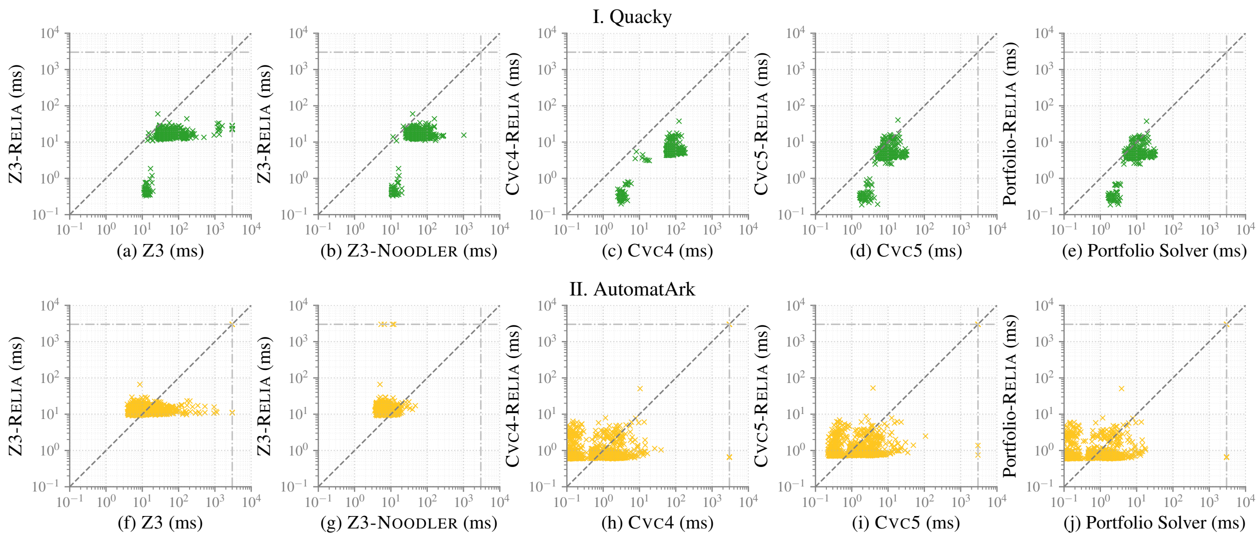


Fig. 7: The runtime of different solvers on public datasets, with and without RELIA. (a)-(e) are for AWS policies from Quacky, and (f)-(j) are for string constraints from AutomatArk.

[42], one state-of-the-art network verifier, leverages this idea to represent route filtering conditions (which extensively use regular expressions) with FA, and computes the atomic predicates with conjunctions, disjunctions, and negations on FA. Both AP Verifier and Batfish inspire us to compute the String Equivalence Classes (SECs), and based on which rewrite regular constraints.

X. DISCUSSION

Where to implement RELIA? Currently, we implement RELIA as a layer in-between the analyzer and the SMT solvers, in order to make RELIA agnostic of upgrades of the analyzer and solver. Besides, this way enables RELIA to generalize to other cloud access control policy analyzers like Zelkova. There are other choices. For example, we can implement it as another string solver or as a preprocessor in a specific SMT solver. This allows RELIA to be applied to other scenarios besides cloud access control analysis. We leave this as one of our future works.

Support hierarchical concatenations. RELIA can be extended to handle hierarchical concatenations by first flattening the concatenations, then eliminating them according to Algorithm 1. For example, suppose we have $s = s_1 \cdot \texttt{":"} \cdot s_2$, and $s_1 = s_{11} \cdot \texttt{"/"} \cdot s_{12}$. We first flatten s to $s_{11} \cdot \texttt{"/"} \cdot s_{12} \cdot \texttt{":"} \cdot s_2$. Then, we convert regular constraints on s to regular constraints on s_{11}, s_{12}, s_2 using Algorithm 1.

Support general word equations. Currently, RELIA supports a special case of word equation, *i.e.*, at least one side of '=' must be a single variable (*e.g.*, $s = s_1 \cdot s_2$). We claim that RELIA can be extended to support general word equations (*e.g.*, $s_1 \cdot s_2 = s_3 \cdot s_4$), which introduce more complex dependencies among string variables. Here, we simply outline the idea. We can first partition string variables into classes according to "=" relation, and then apply the same rewrite (collect RegExes, compute SECs, and rewrite regular constraints into integer constraints) for all string variables of the same class, rather than for each string variable as RELIA current does. We leave this as a future work.

Incremental rewriting. Currently, when a single policy analysis invokes multiple `check_sat()` calls, RELIA processes the constraints of each call independently. However, those constraints are mostly the same, where the latter `check_sat()` call only has several additional regular constraints to the previous `check_sat()` call. Therefore, we believe that implementing incremental rewriting in RELIA will further enhance its performance. Specifically, we will incrementally update the SECs by executing lines 4-6 of Algorithm 2 for the newly added constraints. We leave this as a future work.

Limitations. As shown in §V, the correctness of the concatenation elimination algorithm Algorithm 1 requires that for each string variable, if there are multiple concatenation constraints, then one and only one of them can be satisfied.

XI. CONCLUSION

This paper proposed RELIA, a general method to speed up the analysis of access control policies on the public cloud. RELIA achieves the speedup by computing string equivalence classes, and rewriting the string constraints into integer constraints, therefore bypassing the slow string solvers. Based on real and synthetic dataset, we showed that by enabling RELIA on top of Z3, CVC4, CVC5, and a portfolio solver consisting of these three solvers, the analyses of cloud access policies enjoy considerable speedup.

Acknowledgement. We thank the anonymous ASE reviewers for their valuable comments and suggestions. This work is supported by the National Natural Science Foundation of China (No. 62272382). Peng Zhang is the corresponding author of this paper.

References

[1] U. Team, "Cloud leak: Wsj parent company dow jones exposed customer data," https://www.upguard.com/breaches/cloud-leak-dow-jones.

[2] R. Abel, "Another misconfigured amazon s3 server leaks data of 50,000 australian employees," https://www.scmagazine.com/news/another-misconfigured-amazon-s3-server-leaks-data-of-50000-australian-employees.

[3] I. Thomson, "14 million verizon subscribers' details leak from crappily configured aws s3 data store," https://www.theregister.com/2017/07/12/14m_verizon_customers_details_out/.

[4] T. Ricke, "Microsoft azure cloud vulnerability is the 'worst you can imagine'," https://www.theverge.com/2021/8/27/22644161/microsoft-azure-database-vulnerabilty-chaosdb.

[5] R. S. Labs, "Aws iam privilege escalation – methods and mitigation," https://rhinosecuritylabs.com/aws/aws-privilege-escalation-methods-mitigation/.

[6] 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 *2018 Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–9.

[7] J. Backes, U. Berrueco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. Luckow, S. McLaughlin, M. Menon *et al.*, "Stratified abstraction of access control policies," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*. Springer, 2020, pp. 165–176.

[8] Microsoft, "Z3," https://github.com/Z3Prover/z3.

[9] Barrett, C., et al., "CVC4," https://cvc4.github.io/.

[10] ——, "CVC5," https://cvc5.github.io/.

[11] W. Plandowski, "Satisfiability of word equations with constants is in nexptime," in *Proceedings of the thirty-first annual ACM symposium on Theory of Computing*, 1999, pp. 721–725.

[12] P. Barceló, D. Figueira, and L. Libkin, "Graph logics with rational relations," *Logical Methods in Computer Science*, vol. 9, 2013.

[13] A. Jeż, "Recompression: a simple and powerful technique for word equations," *Journal of the ACM (JACM)*, vol. 63, no. 1, pp. 1–51, 2016.

[14] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, "A dpll (t) theory solver for a theory of strings and regular expressions," in *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 2014, pp. 646–662.

[15] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 114–124.

[16] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang, "Effective search-space pruning for solvers of string equations, regular expressions and length constraints," in *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I 27*. Springer, 2015, pp. 235–254.

[17] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang, "Z3str2: an efficient solver for strings, regular expressions, and length constraints," *Formal Methods in System Design*, vol. 50, pp. 249–288, 2017.

[18] M. Berzish, V. Ganesh, and Y. Zheng, "Z3str3: A string solver with theory-aware heuristics," in *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2017, pp. 55–59.

[19] F. Mora, M. Berzish, M. Kulczynski, D. Nowotka, and V. Ganesh, "Z3str4: A multi-armed string solver," in *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*. Springer, 2021, pp. 389–406.

[20] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1232–1243.

[21] ——, "Progressive reasoning over recursively-defined strings," in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28*. Springer, 2016, pp. 218–240.

[22] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for php," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2010, pp. 154–157.

[23] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra, "Automata-based symbolic string analysis for vulnerability detection," *Formal Methods in System Design*, vol. 44, pp. 44–70, 2014.

[24] A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: A solver for word equations over strings, regular expressions, and context-free grammars," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, pp. 1–28, 2013.

[25] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 513–528.

[26] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, J. Dolby, P. Janků, H.-H. Lin, L. Holík, and W.-C. Wu, "Efficient handling of string-number conversion," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 943–957.

[27] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holík, D. Hu, W.-L. Tsai, Z. Wu, and D.-D. Yen, "Solving not-substring constraint withflat abstraction," in *Programming Languages and Systems: 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17–18, 2021, Proceedings 19*. Springer, 2021, pp. 305–320.

[28] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, "Flatten and conquer: a framework for efficient analysis of string constraints," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 602–617, 2017.

[29] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "String constraints for verification," in *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 2014, pp. 150–166.

[30] ——, "Norn: An smt solver for string constraints," in *International conference on computer aided verification*. Springer, 2015, pp. 462–469.

[31] P. A. Abdulla, M. F. Atig, B. P. Diep, L. Holík, and P. Janků, "Chain-free string constraints," in *Automated Technology for Verification and Analysis: 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28–31, 2019, Proceedings 17*. Springer, 2019, pp. 277–293.

[32] P. A. Abdulla, M. Faouzi Atig, Y.-F. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, "Trau: Smt solver for string constraints," in *2018 Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–5.

[33] M. Berzish, M. Kulczynski, F. Mora, F. Manea, J. D. Day, D. Nowotka, and V. Ganesh, "An smt solver for regular expressions and linear arithmetic over string length," in *International Conference on Computer Aided Verification*. Springer, 2021, pp. 289–312.

[34] F. Blahoudek, Y.-F. Chen, D. Chocholatỳ, V. Havlena, L. Holík, O. Lengál, and J. Síč, "Word equations in synergy with regular constraints," in *International Symposium on Formal Methods*. Springer, 2023, pp. 403–423.

[35] T. Chen, Y. Chen, M. Hague, A. W. Lin, and Z. Wu, "What is decidable about string constraints with the replaceall function," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–29, 2017.

[36] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu, "Decision procedures for path feasibility of string-manipulating programs with complex operations," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[37] Y.-F. Chen, D. Chocholatỳ, V. Havlena, L. Holík, O. Lengál, and J. Síč, "Solving string constraints with lengths by stabilization," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 2112–2141, 2023.

[38] ——, "Z3-noodler: An automata-based string solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2024, pp. 24–33.

[39] L. Holík, P. Janků, A. W. Lin, P. Rümmer, and T. Vojnar, "String constraints with concatenation and transducers solved efficiently," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–32, 2017.

[40] A. W. Lin and P. Barceló, "String solving with word equations and transducers: towards a logic for analysing mutation xss," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 123–136.

[41] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," in *IEEE ICNP*, 2013.

[42] INTENTIONET, "Batfish," https://github.com/batfish/batfish.

[43] C. Barrett, A. Stump, C. Tinelli *et al.*, “The smt-lib standard: Version 2.0,” in *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, vol. 13, 2010, p. 14.

[44] AWS, “Multivalued context key examples,” https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_condition_examples-multi-valued-context-keys.html.

[45] ——, “aws::calledvia,” https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_condition-keys.html#condition-keys-calledvia.

[46] J. Hopcroft, “Introduction to automata theory, languages, and computation,” 2001.

[47] D. Wang, P. Zhang, Z. Gu, W. Lin, S. Jiang, Z. He, X. Du, L. Chen, J. Li, and X. Guan, “Artifact for ase 2025 paper ”relia: Accelerating analysis of cloud access control policies”,” Sep. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17222676

[48] M. O. Rabin and D. Scott, “Finite automata and their decision problems,” *IBM journal of research and development*, vol. 3, no. 2, pp. 114–125, 1959.

[49] C. Thiffault, F. Bacchus, and T. Walsh, “Solving non-clausal formulas with dpll search,” in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2004, pp. 663–678.

[50] W. Eiers, G. Sankaran, A. Li, E. O’Mahony, B. Prince, and T. Bultan, “Quacky: Quantitative access control permissiveness analyzer,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[51] M. Preiner, H.-J. Schurr, C. Barrett, P. Fontaine, A. Niemetz, and C. Tinelli, “Smt-lib release 2025 (non-incremental benchmarks),” Aug. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16740866

[52] I. Shevrin and O. Margalit, “Detecting {Multi-Step}{IAM} attacks in {AWS} environments via model checking,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6025–6042.

[53] W. Eiers, G. Sankaran, A. Li, E. O’Mahony, B. Prince, and T. Bultan, “Quantifying permissiveness of access control policies,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1805–1817.

[54] 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.

[55] A. Aydin, W. Eiers, L. Bang, T. Brennan, M. Gavrilov, T. Bultan, and F. Yu, “Parameterized model counting for string and numeric constraints,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 400–410.

[56] G. S. Makanin, “The problem of solvability of equations in a free semigroup,” *Matematicheskii Sbornik*, vol. 145, no. 2, pp. 147–236, 1977.

[57] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, “APKeep: Realtime verification for real networks,” in *USENIX NSDI*, 2020.

[58] R. Beckett and A. Gupta, “Katra: Realtime verification for multilayer networks,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 617–634.

[59] P. Zhang, D. Wang, and A. Gember-Jacobson, “Symbolic router execution,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 336–349.

[60] D. Wang, P. Zhang, and A. Gember-Jacobson, “Expresso: Comprehensively reasoning about external routes using symbolic simulation,” in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 197–212.