LEOPARD: Accelerating Cloud-based Access Control Policy Verification Using Logical Encoding Optimization

Xing Fang†, Feiyan Ding†, Mingyuan Song†, Yuntao Zhao†, Lizhao You⋆,
Qiao Xiang†, Linghe Kong◇, Jiwu Shu†‡, Xue Liu§,
†Xiamen Key Laboratory of Intelligent Storage and Computing, School of Informatics, Xiamen University,
⋆School of Informatics, Xiamen University, ◇Shanghai Jiao Tong University,
‡Minjiang University, §McGill University

***Abstract*—Cloud providers offer users Satisfiability Modulo Theories (SMT) based verifiers to ensure the correctness of their access control policies. One fundamental challenge in designing these verifiers is achieving high efficiency. In this paper, we identify that a significant source of inefficiency in existing access control policy verifiers is the redundant logical encoding of the verification problem. To address this, we integrate formula slicing and simplification techniques specifically tailored for access control policy verification and introduce LEOPARD, a logical encoding optimization method designed to accelerate cloud-based access control policy verification. LEOPARD introduces two novel approaches to systematically prune redundant formulas unrelated to the desired properties, leveraging both structural and semantic analysis. We constructed two high-fidelity synthetic datasets to validate our approach. Extensive evaluation results show that LEOPARD outperforms state-of-the-art SMT-based policy verifiers in terms of efficiency.**

I. INTRODUCTION

Cloud computing has become increasingly critical nowadays, with cloud providers like AWS, Microsoft Azure, Google Cloud, and Alibaba Cloud hosting vast amounts of computation and data. To secure client data and systems, cloud providers allow users to define their access control policies, also known as Access Control Lists (ACLs). Access control policies are used to precisely define who can or cannot perform specific actions, such as read, write, or modify resources.

However, due to the high complexity of the cloud environment and policy syntax, users struggle to guarantee the correctness of access control policies. First, considering the vast diversity and dynamic nature of cloud environments, policies must be tailored to complex service interactions and frequently updated. For example, in a cloud environment hosting various resources, each type of resource requires specific policies to meet its intricate safety needs, and continuous updates are essential to adapt to rapid changes in operational demands. Additionally, the intricate and precise syntax required for policies poses a significant challenge. For example, AWS provides users with a JSON format language to express access control policies, which require precise key-value structuring and a deep understanding of policy elements.

Incorrect access control policy in cloud environments can undermine security and availability, often leading to security breaches and operational failures. There have been numerous alarming instances where sensitive data has been inadvertently exposed or critical services rendered unavailable for a long time due to misconfigured access control policies on cloud platforms. For example, a misconfigured AWS policy caused McGraw Hill to expose over 100,000 students' private information and the company's source code and digital keys [1]. A similar error caused Microsoft to expose sensitive data records of numerous customers [2]. Moreover, Alibaba Cloud experienced a 3.5-hour outage [3], and reports suggested that it was caused by a misconfigured access control policy. Similarly, Salesforce experienced a 4-hour outage caused by inadvertently blocking legitimate requests [4].

Several verifiers [5], [6], [7], [8] have been proposed to help users ensure their data is well-secured and their services are reliable. These verifiers all adopt an SMT-based technique, which first encodes the verification problem into logical formulas and then utilizes SMT solvers [9], [10], [11], [12] to prove the correctness of various properties. Since writing a correct property is challenging for operators, verifiers usually provide built-in checks for common properties. A basic check is relative permissiveness analysis, which checks whether a policy is less-or-equal permissive than another. For example, given two policies P_1 and P_2, verifiers check whether all requests allowed by P_1 are also allowed by P_2. If P_2 defines a trusted zone, the check ensures P_1 does not allow any untrusted request, thus ensuring security. Conversely, if P_1 describes the trusted zone, the check ensures P_2 does not deny any legitimate request, thus ensuring availability.

As reported in [6], the policy verifier used by AWS handles a staggering one billion SMT queries each day, underscoring the massive scale of policy checks required. Thus, there is a critical need for continuous improvements in the efficiency of policy verification processes. Optimizing access control policy verification is paramount not only to effectively manage the workload, ensure scalability, and build user trust. More importantly, it guarantees real-time security protections for millions of users, thereby safeguarding the integrity and reliability of cloud services.

One challenge to efficiency arises from the flexible policy language, which allows users to define policies with complex conditions such as regular expressions. Since solving the resulting SMT formulas containing theories of regular expressions is PSPACE-complete or potentially even undecidable [13], ACL policy verification efficiency is significantly

Lizhao You and Qiao Xiang are co-corresponding authors.

979-8-3315-4940-4/25/$31.00 © 2025 IEEE

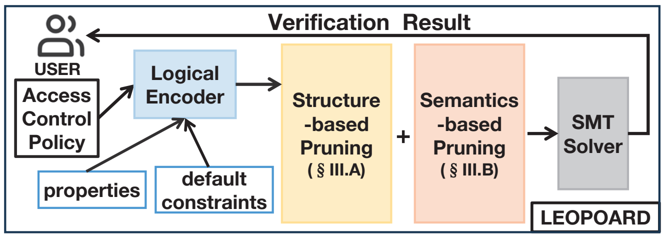


Fig. 1: The architecture and workflow of LEOPARD

limited. One possible solution is to propose heuristic methods or algorithms to optimize the efficiency of the string solver [14], [15], [16]. However, developing a new string solver is highly complex and can lead to regression issues due to trade-offs made between optimizing for specific scenarios and maintaining overall performance [17].

To achieve broad acceleration across various scenarios, we have chosen not to develop a new SMT string solver. Instead, we explore opportunities to enhance satisfiability solving by optimizing the logical encoding of access control policy verification problems. This decision stems from our observation that existing verification techniques incorporate redundant encoding mechanisms that can be optimized for greater efficiency and performance. Concretely, existing verifiers directly encode the default value and length constraints of the access control policy into the SMT formula, ignoring the structural and semantic dependencies relevant to the verification property. For example, if the policy constraints include conditions on variable p, but the verification property is unrelated to and independent of p, these conditions can be eliminated without affecting the correctness of the verification.

By specifically leveraging the domain knowledge of ACL policies, we adapt the techniques of formula slicing and simplification [18], [19], [20], [21], [22], [23] to enhance solving efficiency, which involve decomposing complex SMT formulas into more manageable parts and rewriting formulas to eliminate redundancies. This tight integration leads to significant acceleration by reducing formula redundancy, thereby enhancing the efficiency and performance of access control policy verification across various scenarios in a broadly applicable manner.

To this end, we propose LEOPARD, a **L**ogical **E**ncoding **OP**timization method to **A**cceleRate clou**D**-based access control policy verification. In particular, as depicted in Figure 1, LEOPARD introduces two pruning approaches to remove redundant formulas based on structure and semantics. The structure-based pruning approach (Section III.1) analyzes variable dependencies between formulas and removes formulas without dependency on the given property. The semantics-based pruning approach (Section III.2) eliminates formulas that cannot be satisfied according to the access control policy semantics. With these two pruning approaches, the encoded SMT formula becomes much smaller than before optimization, allowing it to be solved more efficiently while maintaining correctness.

In summary, this paper makes the following contributions:

- We identify that the inefficiency of existing access control policy verifiers stems from redundant logical encoding.
- We propose LEOPARD, a generic logical encoding optimization approach that prunes redundant formulas based on structure and semantics.
- Extensive evaluations demonstrate that LEOPARD surpasses current state-of-the-art SMT-based policy verifiers in efficiency.

```
{(Allow
  Principal :   "*",
  Action    :   "s3:GetObject",
  Resource  :   "arn:aws:s3:::myexamplebucket/*",
  Condition:    ""),
 (Deny
  Principal :   "*",
  Action    :   "s3:GetObject",
  Resource  :   "arn:aws:s3:::myexamplebucket/*",
  Condition:    (StringNotLike, UserId, EXAMPLEID:*))}
```

(a) Policy X

```
{(Allow
  Principal :   "*",
  Action    :   "*",
  Resource  :   "arn:aws:s3:::myexamplebucket/*",
  Condition:    "")
```

(b) Policy Y

Fig. 2: Example ACL policies in AWS

II. Background and Motivation

In this section, we first outline cloud-based access control policies and the method to encode policies and properties into an SMT formula. Next, we provide an example that illustrates the limitations of existing verifiers and demonstrates how LEOPARD enhances verification efficiency by integrating formula slicing and simplification techniques with access control policy verification.

A. Cloud-based Access Control Policies

Given that cloud providers typically use a similar access control policy structure, we will use the AWS policy as a representative example to illustrate the concept. An AWS policy comprises one or more policy statements. Each of these statements encapsulates five critical elements: (**Effect, Principal, Action, Resource, Condition**). The **Effect** element declares if access is *Allow* or *Deny*, thus controlling the permission status. The **Principal** element identifies the specific entity to which the statement is applicable. The **Action** element outlines the exact operations that the principal is permitted or forbidden to execute. The **Resource** element pinpoints the AWS resources upon which the actions are to be conducted. Lastly, the **Condition** element specifies the circumstances under which the policy statement is considered valid or applicable. Combined with all statements in a policy, an access request is allowed by the policy only if it matches an *Allow* statement and isn't blocked by any *Deny* statements.

In Figure 2, policies X and Y provide examples of how AWS policies can be utilized to manage access permissions for requests. Policy X contains two statements: the first is an $Allow$ statement that grants all principals permission to perform the "s3:GetObject" action on resources within the example bucket. This statement is unconditional, implying that it applies broadly without any restrictions. The second statement, however, is a $Deny$ statement that specifically overrides the first by prohibiting the "s3:GetObject" action if the request originates from a user ID not matching "EXAMPLEID:*". This setup ensures that while the bucket is generally accessible, access is explicitly blocked for user IDs that do not conform to the specified pattern, enhancing security by restricting unwanted access.

Conversely, Policy Y is far more permissive, containing only one statement, which is an $Allow$ statement without any conditions. It permits any principal to perform any action on the example bucket. This means there are no restrictions whatsoever on actions that principals can take, making it significantly less secure compared to Policy X, as it does not discriminate among users or actions. This example highlights the flexibility and granularity that AWS policies offer, allowing administrators to tailor access precisely according to security requirements and operational intents.

Having explored how access control policies are structured through specific examples, let's now pivot to understand the **default constraints** that underpin these policies. In access control policies, such constraints are integral to reinforcing security standards. For instance, while "s3:GetObject" represents a valid action within these policies, an invented action like "s3:LEOPARD" would not be recognized, ensuring that only authorized operations can be carried out. Moreover, AWS sets limits on the length of policy elements, with the principal's identity being required to fall between 1 and 63 characters. These built-in safeguards serve to prevent unauthorized actions and avoid the creation of policies with overly complicated or invalid entries. Through these default constraints, the access control policies facilitate adherence to security best practices, providing a consistent and streamlined framework for effectively managing access control.

B. Logical Encoding

ZELKOVA [5] was the first verifier specifically designed for access control policies, pioneering the use of logical encoding for policy verification. Subsequent works [7], [8] adopted similar encoding approaches. This section presents the formulation of access control policy verification as an SMT problem. The logical encoding is similar to QUACKY [8], which builds on ZELKOVA and extends it by incorporating the encoding of default constraints, which is crucial for verification accuracy.

Logical encoding for policy. According to the policy semantics, policy $[P]$ and statement $[S]$ can be encoded as:

$$[P] = \left(\bigvee_{S \in Allow} [S] \right) \wedge \neg \left(\bigvee_{S \in Deny} [S] \right), \tag{1}$$

$$\boldsymbol{a} = \text{"}\boldsymbol{s3}\text{:}\,\boldsymbol{GetObject}\text{"} \wedge \text{"}\mathbf{arn{:}\,aws{:}\,s3{:}::myexamplebucket/}\text{"}\ \boldsymbol{prefixOf}\ \boldsymbol{r} \wedge$$
$$\neg(\boldsymbol{a} = \text{"}\boldsymbol{s3}\text{:}\,\boldsymbol{GetObject}\text{"} \wedge \text{"}\mathbf{arn{:}\,aws{:}\,s3{:}::myexamplebucket/}\text{"}\ \boldsymbol{prefixOf}\ \boldsymbol{r} \wedge$$
$$\boldsymbol{UserIdExists} \wedge \neg(\text{"}\boldsymbol{EXAMPLEID}\text{:}\text{"}\ \boldsymbol{prefixOf}\ \boldsymbol{UserId}))$$

(a) SMT encoding of policy X

$$\text{"}\mathbf{arn{:}\,aws{:}\,s3{:}::myexamplebucket/}\text{"}\ \boldsymbol{prefixOf}\ \boldsymbol{r}$$

(b) SMT encoding of policy Y

Fig. 3: SMT encoding of the example ACL policies in Figure 2

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

Here, same as the access control policy semantics, $[P]$ is defined as allowing access if any conditions in the $Allow$ statements are met and no conditions in the $Deny$ statements apply, effectively capturing the intersection of allowances minus any denials. Statement $[S]$ defines the conditions under which the access request is matched. It stipulates that an access request can be matched if any principal p, action a, or resource r matches its defined criteria $S(P)$, $S(A)$, $S(R)$, and all conditions in $S(C)$ are concurrently fulfilled.

Figure 3 offers a demonstrative depiction of how the example policies are encoded into the SMT formula. The Boolean variable $UserIdExists$ encodes whether $UserId$ exists in the request context, which is important for preserving the semantic integrity of the policy's conditions. This example highlights the inherent flexibility of access control policies, emphasizing the need for versatile logical encoding to capture this complexity, and also bringing challenges to the access control policy verification task.

Logical encoding for properties. After encoding the policy, we can analyze various properties based on it. One fundamental property is relative permissiveness analysis, which compares the restrictiveness of two policies by evaluating whether one policy allows fewer requests than another. This property is commonly used due to its simplicity, alleviating the burden of writing complex properties. Cloud providers or users only need to define a trusted zone using the policy language. With such a trusted zone, security and availability can be easily checked using relative permissiveness analysis, ensuring compliance without the need to craft complex properties from scratch. For example, consider an operator who defines a new policy and wants to check its security. The operator can use relative permissiveness analysis to determine whether this new policy is more restrictive than the trusted zone policy, ensuring that all requests allowed by the new policy are also allowed in the trusted zone. Similarly, the operator can check availability by verifying that the new policy is more permissive than the trusted zone.

Formally, given two policies P_1 and P_2, the relative permissiveness analysis checks whether P_1 is less-or-equal permissive than P_2. The corresponding SMT formula is $[P_1] \wedge \neg[P_2]$, meaning that a request is allowed by P_1 but denied by P_2. If the solver finds this formula satisfiable, there is a counterexample where P_1 is more permissive than P_2 in at least one case. If the solver returns unsatisfiable, it confirms that P_1 is less or equally permissive compared to P_2 in all scenarios. Considering the SMT encoding of the example policy shown in Figure 3, we can verify the relative permissiveness of policies X and Y by putting the formula $[X] \wedge \neg[Y]$ into the SMT solver. According to the semantics above, any access allowed by Policy X is also permitted by Policy Y. Therefore, the SMT solver would return unsatisfiable, confirming that Policy X is more restrictive than Policy Y. In this case, if Policy Y is considered the trusted zone, this check guarantees the security of Policy X. If Policy X is considered the trusted zone, then this check guarantees the availability of Policy Y.

Logical encoding for default constraints. The default constraints of access control policies ensure that only recognized actions and resources are valid, and set length limits on principal identities to uphold security and simplify access management. The default constraints of the action range and string value length can be encoded as $[D_A]$ and $[D_S]$, respectively:

$$[D_A] = \bigvee_{(T_A, T_R) \in ActionTable} \left(\left(\bigvee_{v \in T_A} a = v \right) \wedge \left(\bigvee_{v \in T_R} r = v \right) \right), \tag{3}$$

$$[D_S] = \bigwedge_{(t, min, max) \in StringTable} (p.type = t \Rightarrow min \leq |p| \leq max) \tag{4}$$

Here, $ActionTable$ defines all legitimate actions and the corresponding legitimate resources. (T_A, T_R) is an item in $ActionTable$ and indicates that the action in T_A can only access the resource in T_R. In combination, $[D_A]$ states that a pair of action a and resource r is valid only if a and r respectively match any of the values defined in any item in $ActionTable$. For the default constraints on string value length, $StringTable$ defines all limited string types and their corresponding length range. $[D_S]$ states that a principal p is valid only if p and its length match any item defined in $StringTable$. The default constraints $[D]$ are the conjunction of $[D_A]$ and $[D_S]$.

When combined with the default constraints, the property of relative permissiveness is encoded into $[P_1] \wedge \neg[P_2] \wedge [D]$. It is essential to incorporate default constraints in the verification process of access control policies, as they crucially impact the accuracy of the verification outcomes. For example, consider the scenario where the solver finds $[P_1] \wedge \neg[P_2]$ to be satisfiable, indicating that there are requests permitted by $[P_1]$ but not by $[P_2]$. If all these requests are invalid under the default constraint $[D]$, then solving $[P_1] \wedge \neg[P_2] \wedge [D]$ will yield unsatisfiable, highlighting a discrepancy from the results of $[P_1] \wedge \neg[P_2]$ alone. Since operators typically focus on legitimate requests, ignoring encoding default constraints can lead to misidentifying an accurately defined policy as flawed, affecting practical policy enforcement and compliance assessments.

$$\begin{gathered}
(\boldsymbol{a} = \text{"}\mathbf{s3}{:}\,\boldsymbol{PutbBucketAcl}\text{"} \vee \boldsymbol{a} = \text{"}\mathbf{s3}{:}\,\boldsymbol{PutBucketCors}\text{"} \vee \\
\boldsymbol{a} = \text{"}\mathbf{s3}{:}\,\boldsymbol{PutBucketLogging}\text{"} \vee \boldsymbol{a} = \text{"}\mathbf{s3}{:}\,\boldsymbol{PutBucketNotification}\text{"} \vee \\
\boldsymbol{a} = \text{"}\mathbf{s3}{:}\,\boldsymbol{GetBucketPolicy}\text{"} \vee \boldsymbol{a} = \text{"}\mathbf{s3}{:}\,\boldsymbol{GetBucketPolicyStatus}\text{"} \vee \\
\boldsymbol{a} = \text{"}\mathbf{s3}{:}\,\boldsymbol{CreateBucket}\text{"} \vee \boldsymbol{a} = \text{"}\mathbf{s3}{:}\,\boldsymbol{DeleteBucket}\text{"} \;\wedge \\
\text{"}\mathbf{arn{:}\,aws{:}\,s3{:}::\,listbucket/}\text{"}\;\; \boldsymbol{prefixOf}\;\; \boldsymbol{r})
\end{gathered}$$

(a) Example encoding snippets of D_A

$$\begin{gathered}
\boldsymbol{p.type} = \text{"}\boldsymbol{account}\text{"} \Rightarrow (\mathbf{2} < |\boldsymbol{p}| < \mathbf{64}) \wedge \\
\boldsymbol{p.type} = \text{"}\boldsymbol{role}\text{"} \Rightarrow (\mathbf{2} < |\boldsymbol{p}| < \mathbf{1225}) \wedge \\
\boldsymbol{p.type} = \text{"}\boldsymbol{service}\text{"} \Rightarrow (\mathbf{0} < |\boldsymbol{p}| < \mathbf{6145}) \wedge \\
\boldsymbol{p.type} = \text{"}\boldsymbol{resource}\text{"} \Rightarrow (\mathbf{0} < |\boldsymbol{p}| < \mathbf{6145})
\end{gathered}$$

(b) Example encoding snippets of D_S

Fig. 4: Example encoding snippets of default constraints D

C. A Motivating Example

Here, we introduce a motivating example to illustrate the limitations of current verification tools and demonstrate how LEOPARD optimizes the process by integrating formula slicing and simplification techniques.

Existing verifiers commonly verify the relative permissiveness between two policies, with the encoded SMT formula being $[P_1] \wedge \neg[P_2] \wedge [D]$. We identified significant redundancies in these formulas, which can affect verification efficiency. Specifically, verifiers encode all default constraints into the SMT formula without considering their relation to the properties. We aim to optimize the encoding and reduce the SMT solver's search space by pruning irrelevant parts of the default constraint, thus accelerating the verification process.

Figure 4 shows an example of SMT encoding snippets of default constraints, highlighting significant redundancy even in this small portion. Using the example policies in Figure 2, we will demonstrate opportunities for encoding optimization in verifying relative permissiveness.

First, let's investigate structural formula redundancy for potential encoding optimizations. Given that the property formula $[X] \wedge \neg[Y]$ does not include any principal variable p, encoding default constraints related only to p is redundant. For example, $[D_S]$ pertains only to the principal variable and is irrelevant to $[X] \wedge \neg[Y]$, so there is no need to model any part of $[D_S]$. This pruning can significantly reduce the formula size to be verified. To remove all structural redundancies, we use formula slicing, a technique that trims away irrelevant parts of the formula, efficiently excluding all structurally unrelated components and optimizing the verification process.

Next, we show how to prune formula redundancy based on semantics. For the formula $([X] \wedge \neg[Y] \wedge [D])$ to be satisfiable, all variables must meet the conditions specified in $[X]$. For example, as shown in Figure 3, a = "s3:GetObject" is part of the conjunction in $[X]$. This means a must be assigned "s3:GetObject"; otherwise, the conjunction would be false. Based on this, within the action range constraint T_A in D_A, any v that is not "s3:GetObject" can be disregarded. Additionally, if no v in T_A matches "s3:GetObject", the entire constraint (T_A, T_R) becomes unsatisfiable, as the disjunction over T_A would fail to find a valid action, making the conjunction false. Therefore, it can be directly pruned. To remove semantic redundancies, we combine formula simplification with the specific semantics of access control policies, enabling encoding optimization.

Moreover, if all constraints in D_A are pruned, the property can be directly concluded as correct without invoking the SMT solver. This is because D_A becomes false when all its constraints are removed, leaving no valid actions to satisfy the disjunction. Consequently, the solving formula $[X] \wedge \neg [Y] \wedge [D]$ becomes unsatisfiable. Removing these redundant constraints results in a simplified formula, reducing complexity and enabling the SMT solver to solve it more efficiently, thereby improving verification performance.

III. LEOPARD

In this section, we introduce LEOPARD, a logical encoding optimization method to accelerate cloud-based access control policy verification. LEOPARD offers two novel methods to systematically eliminate redundant formulas unrelated to the desired properties from the logical encoding, based on structural and semantic analysis, respectively.

A. Structure-based Pruning

The fundamental idea of structure-based pruning is to thoroughly analyze the dependencies between variables within the given formulas and systematically eliminate those formulas that do not involve any variables relevant to the specific property being verified. By focusing solely on the essential components that influence the verification outcome, this approach significantly reduces the complexity of the formulas, enhancing the efficiency of the verification process.

The structure-based pruning technique is inspired by the concept of formula slicing [19], [24], [18], which breaks down a large formula into smaller, more manageable subformulas by analyzing dependency relations, thereby improving SMT solver efficiency. However, structure-based pruning differs from formula slicing techniques in two key ways. First, structure-based pruning specifically targets formulas directly related to the property being verified, enhancing efficiency by focusing only on relevant components. Second, unlike formula slicing techniques such as axiom selection, structure-based pruning maintains correctness by ensuring that only irrelevant formulas are removed, thereby preserving the integrity of the verification process.

Algorithm 1 shows the basic algorithm of structure-based pruning. Given a set of default constraint formulas D and the property formula $Prop$, this algorithm systematically eliminates redundant formulas from D by analyzing the dependencies between variables. The goal is to produce a pruned set of formulas, D_p, which only contains those formulas relevant to the property being verified. From a holistic perspective, the pruning algorithm operates similarly to Breadth-First Search (BFS). This similarity is evident in its iterative approach, where it explores and processes formulas with at least one variable related to the given property, ensuring that only necessary components are retained for efficient verification.

Algorithm 1 StrucutreBasedPruning(D, Prop)

```
Input: Formulas $D$, $Prop$
Output: Pruned formulas $D_p$
1:  Initialize a queue $Q$
2:  $Q$.enqueue($Prop$)
3:  $D_p$ = $\{\emptyset\}$
4:  while NotEmpty($Q$) do
5:      $f$ = $Q$.dequeue()
6:      for formula $d$ in $D$ do
7:          if $d$ and $f$ share a variable and $d \notin D_p$ then
8:              $Q$.enqueue($d$)
9:              $D_p$ = $D_p \cup d$
10: return $D_p$
```

The algorithm starts by initializing a queue and enqueuing the property formula $Prop$. Then, it enters a loop where it dequeues a formula f from the queue and examines each formula d in D. If f and d share a variable and d is not already included in D_p, the algorithm recognizes d as relevant to the property. Consequently, d is added to the queue to further explore related formulas and is also included in D_p. This iterative process continues until the queue is empty, ensuring all relevant formulas are identified and retained. As a result, the algorithm effectively prunes redundant formulas that do not impact the verification of the given property, significantly reducing the complexity of the formula set and enhancing the efficiency of the verification process.

Theorem III.1: Structure-based pruning does not compromise the accuracy of the solving result. Specifically, solving $D \wedge \neg Prop$ and $D_p \wedge \neg Prop$ returns the same result.

Proof: Assume D is satisfiable, if D is not satisfiable, verifying the property by solving $D \wedge \neg Prop$ would be meaningless. Let D_r denote the redundant formulas such that D can be expressed as $D_r \wedge D_p$. Therefore, we can rewrite $D \wedge \neg Prop$ as $D_r \wedge D_p \wedge \neg Prop$. Given that D is satisfiable, D_r must also be satisfiable, meaning there exists an assignment that makes D_r true. Furthermore, considering the independence between $D_p \wedge \neg Prop$ and D_r, the truth value of $D_p \wedge \neg Prop$ does not depend on the values assigned to the variables in D_r. Therefore, the truth value of D_p and $Prop$ remains constant regardless of the values of variables in D_r. As a consequence, $D_r \wedge D_p \wedge \neg Prop$ will be true if and only if $D_p \wedge \neg Prop$ is true. This demonstrates that solving $D \wedge \neg Prop$ and solving $D_p \wedge \neg Prop$ yield the same result, thus validating that structure-based pruning does not affect the accuracy of the solving result. ■

Additionally, while structure-based pruning ensures correctness by removing irrelevant formulas, future research could explore more aggressive pruning using domain knowledge, similar to axiom selection [24], [18]. To maintain correctness after aggressive pruning, an adaptive verification framework can be used. If the pruned formula is unsatisfiable, the original formula must also be unsatisfiable due to the conjunctive nature of the formulas. Otherwise, correctness is tested by

Algorithm 2 SemanticsBasedPruning(D, Prop)

```
Input: Formulas $D_A$, $Prop$
Output: pruned formulas $D_p$
1: $D_p = \{\emptyset\}$
2: for formula $d$ in $D_A$ do
3:     $intersect = d.actions \cap Prop.allowedActions$
4:     if $intersect \neq \emptyset$ then
5:         $d.actions = intersect$
6:         $D_p = D_p \cup d$
7: return $D_p$
```

checking the original formula's satisfiability with the pruned formula's satisfiable assignment. If this test fails, additional formulas are adaptively selected for re-verification. This approach balances aggressive pruning with accuracy, enabling more efficient and reliable verification processes.

B. Semantics-based Pruning

The key idea behind semantics-based pruning is to leverage the semantics of the access control policies to eliminate formulas that cannot satisfy the property. This approach ensures that the verification process focuses only on the formulas that can satisfy the property, thereby streamlining the overall procedure. By reducing the complexity of the solving formula, semantics-based pruning enhances the efficiency of the verification process, making it more manageable and effective.

Semantics-based pruning is inspired by the concept of formula simplification [20], [23], [22], [21], which involves refining complex formulas into simpler, logically equivalent forms by eliminating redundancies. Semantics-based pruning extends formula simplification techniques to the domain of ACL policy verification by leveraging specific policy semantics, ensuring that formulas that cannot satisfy the property being verified are removed, thereby making it more effective for ACL policy verification.

Algorithm 2 shows the procedure for semantics-based pruning. Given a set of action constraint formulas D_A and the property formula $Prop$, this algorithm systematically identifies and eliminates those formulas from D_A that are deemed irrelevant by leveraging the semantics of the access control policies. The overarching goal is to produce a pruned set of formulas, denoted as D_p, which exclusively contains those formulas that semantically satisfy the property being verified. The algorithm first identifies the set of requests allowed by the property. Then, it determines the actions permitted by these allowed requests. Finally, it prunes the formulas in D_A that do not fall within the range of these actions, thereby ensuring that D_p only contains formulas that can potentially satisfy the property.

Concretely, the algorithm iterates through each formula in D_A. For each formula, it checks whether there are actions defined in the formula that are also allowed by the property. If such actions exist, the formula is considered relevant and is added to the result D_p after being updated to include only the allowed actions. Otherwise, the formula is pruned due to its semantic unsatisfiability. This ensures that only formulas defining actions consistent with the property's semantics are retained, effectively removing those that do not contribute to the property's satisfiability. Additionally, considering the solving formula $D_A \wedge D_S \wedge \neg$Prop, if all formulas in D_A are pruned, D_A effectively becomes false. This allows the entire formula to be directly implied as false without invoking an SMT solver, thereby saving significant computational resources.

Theorem III.2: Semantic-based pruning does not compromise the accuracy of the solving result. Specifically, solving $D_A \wedge D_S \wedge \neg Prop$ and $D_p \wedge D_S \wedge \neg Prop$ return the same result.

Proof: Since the formulas in D_A are in disjunction, we can denote D_A as $D_p \vee D_r$, where D_r denotes the redundant formulas. Each formula in D_r is unsatisfiable because there is a conflict in the value of the action variable between the formula and the property $Prop$. Therefore, D_r is unsatisfiable and can be replaced with false, effectively reducing D_A to D_p. Consequently, D_A and D_p are equivalent, as the presence of false in a disjunction does not affect the truth value of the other disjuncts. This implies that the satisfiability of $D_A \wedge D_S \wedge \neg Prop$ is the same as the satisfiability of $D_p \wedge D_S \wedge \neg Prop$, because the unsatisfiable parts of D_A do not contribute to the overall satisfiability. Thus, by pruning the redundant formulas, we streamline the verification process while preserving the logical equivalence necessary for accurate verification.

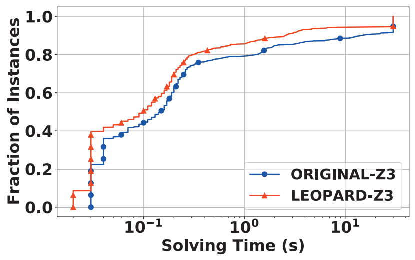
■

To maintain efficiency, semantics-based pruning currently omits consideration of deny statements and the intricate semantics associated with complex regular expressions when computing the allowed actions. This approach involves a tradeoff between pruning complexity and efficiency. Incorporating the semantics of deny statements and complex regular expressions would require additional pruning time but could enhance pruning effectiveness. However, focusing on simple cases has already achieved considerable acceleration, rendering the current approach effective. This tradeoff highlights a promising direction for future research: integrating more complex regular expression handling and deny statement considerations to further optimize pruning efficiency.

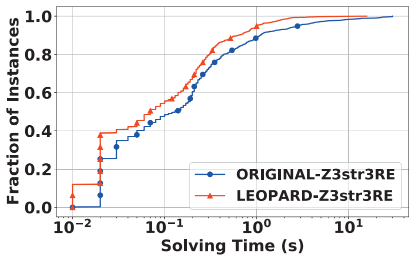
Additionally, semantics-based pruning, while primarily focused on allowed actions, can be extended to other variables such as principal and resource. This extension would allow for further simplification beyond the current scope. Similar to the earlier tradeoffs, incorporating these additional variables would require extra pruning time but offers the potential for a more thorough and effective verification process. Thus, future research could explore these extensions to enhance the overall effectiveness of access control policy verification.

IV. EVALUATION

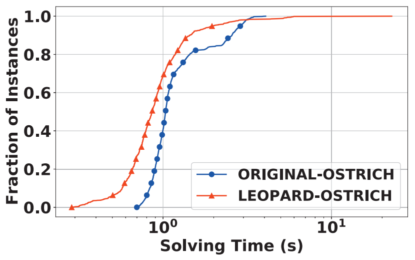
In this section, we conduct a comprehensive set of experiments to demonstrate the effectiveness of LEOPARD. We implement LEOPARD in Java and evaluate its efficiency by comparing the verification time of the original formulas with that of the pruned formulas. All experiments are run on a



(a) Z3



(b) Z3str3RE



(c) OSTRICH

Fig. 5: CDF of solving times for binary analysis of original and pruned SMT instances in Dataset1 across three solvers

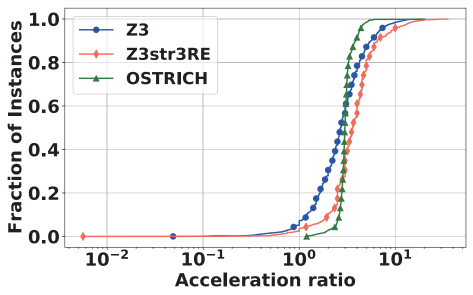


Fig. 6: The ratio of solving time for binary analysis of the original and pruned instances on Dataset1 for all solvers

TABLE I: The Number of the Original and Pruned Instances on Datasets.

Dataset	#Instance		
	ORIGINAL	LEOPARD	RATIO(%)
Dataset1	2792	1267	45%
Dataset2	3018	1836	61%

Linux server equipped with an Intel Xeon Silver 4210R CPU clocked at 2.40 GHz and 128 GB of memory.

A. Experiment Setting

1) Policy Datasets: We use two synthesized datasets to evaluate LEOPARD. Dataset1 is the AWS policy dataset synthesized from the public dataset provided by QUACKY [8], while Dataset2 is a proprietary dataset from a large cloud provider. Both datasets are encoded using the AWS policy language, and different cloud providers employ distinct default constraints due to variations in the default policy behavior. We group similar policies by their structure and semantics, then analyze their relative permissiveness by generating SMT instances for every policy pair within each group. Each instance includes the logical encoding of both policies, property, and default constraints. Dataset1 and Dataset2, comprise 2,792 and 3,018 instances respectively (Table I).

2) Setup: We evaluate the performance of LEOPARD by comparing SMT solving time before and after applying LEOPARD for formula pruning.

First, we evaluate the efficiency improvement of solving time for binary analysis, which aims to obtain a binary yes/no answer for a given SMT instance, adopted by verifiers such as ZELKOVA [5], [6] and Block Public Access [7]. We do not directly compare with ZELKOVA and Block Public Access because they are closed-source and omit encoding

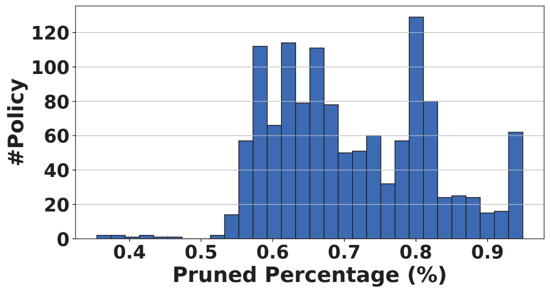


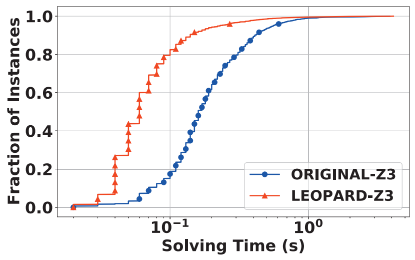
Fig. 7: The distribution of the percentage of atomic formulas pruned by LEOPARD across all pruned instances in Dataset1

default constraints. However, we believe that LEOPARD could accelerate solving times if these techniques included default constraints. We carefully selected three SMT solvers, Z3 [9], Z3str3RE [11], and OSTRICH [12] in our experiments, for their advanced capabilities and performance. CVC4 [25] and CVC5 [10] are not utilized due to their incapability of handling extended regular expressions. Z3str3 [26] and Z3str4 [27] were excluded because their solving times are longer compared to Z3. However, it is worth noting that LEOPARD can still accelerate solving times by reducing irrelevant formulas when using Z3str3 and Z3str4. A 30-second timeout is set for the solving process, consistent with ZELKOVA, reflecting a user-acceptable solving time.

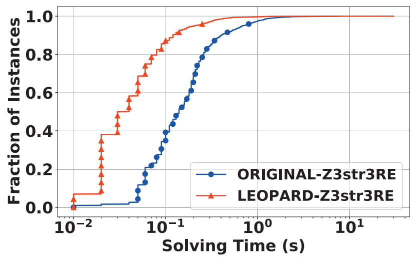
Second, we evaluate the efficiency improvements for quantitative analysis, which aims to obtain the count of solutions for a given SMT instance, as adopted by QUACKY [8]. Like QUACKY, we use the model counting constraint solver ABC [28], [29], [30] to achieve this goal. Additionally, we enhance Z3 [9] to iteratively find different satisfiable solutions, setting a limit of 100 solutions to avoid excessive computation time. We only use DATASET1, provided by QUACKY, to evaluate the efficiency for simplicity. However, similar efficiency gains can also be observed with DATASET2. A 600-second timeout is set for the solving process.

B. Experiment Result

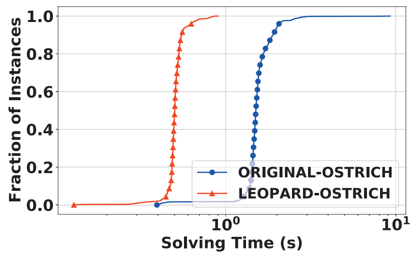
Table I shows the number of instances pruned by LEOPARD. A significant proportion of the instances can be effectively pruned. Specifically, within Dataset1, 45% of the instances were pruned, while within Dataset2, the pruning ratio was even higher at 61%. This demonstrates the substantial impact of LEOPARD in reducing the formula size of instances, thereby enhancing overall efficiency.



(a) Z3



(b) Z3str3RE



(c) OSTRICH

Fig. 8: CDF of solving times for binary analysis of original and pruned SMT instances in Dataset2 across three solvers

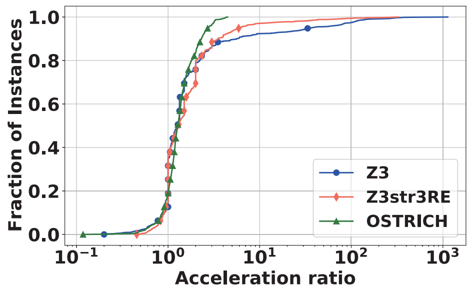


Fig. 9: The ratio of solving time for binary analysis of the original and pruned instances on Dataset2 for all solvers

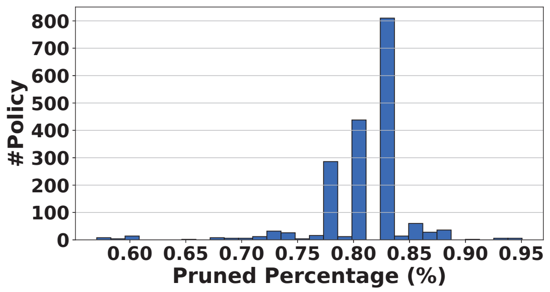


Fig. 10: The distribution of the percentage of atomic formulas pruned by LEOPARD across all pruned instances in Dataset2

Effectiveness for Binary Analysis on Dataset1. Figure 5 shows the Cumulative Distribution Function (CDF) of the solving time for binary analysis of the SMT instances from Dataset1 that can be pruned by LEOPARD, comparing the performance before and after pruning using the three solvers. It is evident that LEOPARD significantly improves solving efficiency. Specifically, the CDF curves for the instances pruned by LEOPARD (indicated in orange) consistently lie to the left of the curves for the original instances (indicated in blue) across all three solvers, which signifies a reduction in solving time, demonstrating the effectiveness of LEOPARD in pruning irrelevant parts of the instances and accelerating the solving process.

Figure 6 illustrates the ratio of the solving time before and after pruning. The CDF curves show that for the majority of instances, pruning by LEOPARD results in a significant reduction in solving time, as indicated by acceleration ratios greater than 1. The steep rise in the CDF curves for each solver demonstrates that a large fraction of instances benefit from substantial speed-ups due to pruning, reaffirming the effectiveness of LEOPARD in enhancing solver performance.

Figure 7 presents the distribution of the percentage of atomic formulas pruned by LEOPARD across all pruned instances in Dataset1. Specifically, a formula is atomic if it does not contain any nested formulas. For instance, in Figure 4, $a = "s3 : PutBucketAcl"$ is an atomic formula. The histogram in Figure 7 reveals that a considerable portion of instances have a high pruning rate, with the majority concentrated between 55% and 95%, indicating the widespread effectiveness of LEOPARD in reducing the complexity of the input formulas.

Building upon the above data, we observe substantial efficiency improvements with LEOPARD. Notably, Z3str3RE, the fastest solver for verifying Dataset1, shows an average speedup of 2.93× with LEOPARD. Additionally, LEOPARD yields average speedups of 1.69× for Z3 and 1.26× for OSTRICH. Most instances accelerate significantly after pruning, with LEOPARD achieving up to 35.20× acceleration on a single instance. However, a few instances exhibit longer solving times post-pruning. This counterintuitive observation may be related to the specific implementation details and heuristics of the various solvers. In some cases, the removal of certain constraints might inadvertently create a more complex search space, increasing the solving time. Overall, the results demonstrate the remarkable potential of LEOPARD to effectively enhance solver performance.

Effectiveness for Binary Analysis on Dataset2. We plot similar figures for Dataset2 in Figure 8, Figure 9, and Fig 10. Figure 8 shows that CDF curves for the pruned SMT instances (indicated in orange) consistently lie to the left of the original instances (indicated in blue), indicating reduced solving times. Figure 9 demonstrates significant reductions for the majority of instances, as indicated by acceleration ratios greater than 1. Furthermore, Figure 10 highlights a significant portion of instances with pruning rates between 75% and 85%, emphasizing LEOPARD's effectiveness in simplifying input formulas.

Notably, LEOPARD demonstrates a substantial speedup of up to 1131.21× on a single instance within Dataset2. The average speedups are also impressive: 2.77× with Z3str3RE, 2.16× with Z3, and 3.08× with OSTRICH. These results highlight LEOPARD's capability to be effectively utilized by different cloud providers with distinct default constraints.

While the overall trends are similar to those observed with Dataset1, the specific speedup ratios vary slightly, reflecting the unique characteristics of each dataset. For instance, Dataset2 shows higher variability in speedup ratios, which

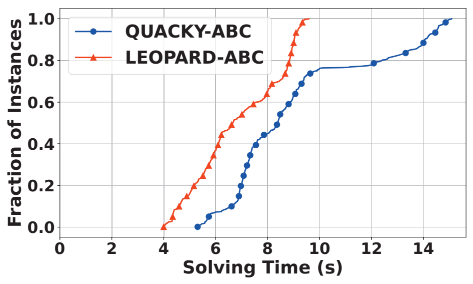


Fig. 11: CDF of solving times for quantitative analysis of original and pruned SMT instances in Dataset1 using ABC

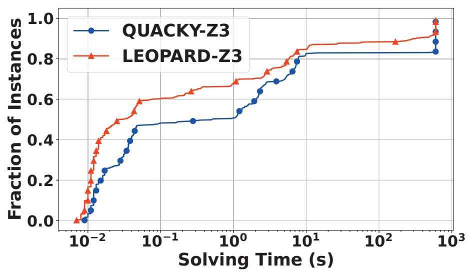


Fig. 12: CDF of solving times for quantitative analysis of original and pruned SMT instances in Dataset1 using Z3

may be due to differences in the constraint structures and complexities of the instances. Despite these differences, the consistent improvements across both datasets underscore the robustness and versatility of LEOPARD in enhancing solver performance through effective formula reduction.

Effectiveness for Quantitative Analysis on Dataset 1. Figure 11 illustrates the CDF of the satisfiable solution counting time using ABC for SMT instances from Dataset1 that can be pruned by LEOPARD. This comparison with the state-of-the-art tool QUACKY [8] shows that, with LEOPARD's formula pruning, the process of counting satisfiable solutions is more efficient, achieving an average speedup of 1.33× and a maximum speedup of 1.96×. Furthermore, Figure 12 demonstrates that, after pruning, counting 100 different satisfiable solutions using Z3 is also accelerated, with an average speedup of 15.45× and up to 680.27×. Overall, the results indicate that LEOPARD achieves better performance than state-of-the-art tools in analyzing the quantitative properties of access control policies.

Evaluation of pruning overhead. A common concern is whether the pruning process of LEOPARD introduces significant overhead, potentially offsetting the benefits of improved solving times. To address this, we measured the pruning time, finding it averaged 25.97 ms per instance in Dataset 1 and 19.60 ms in Dataset 2. While these times may seem notable, especially with some SMT instances solved within 100 ms, the benefits still outweigh these costs. In numerous instances, the solving times improve by orders of magnitude, such as reducing from seconds to milliseconds. In these cases, the pruning overhead becomes negligible compared to the substantial performance gains. Moreover, the pruning process can be optimized further by implementing it in a faster language than Java. The consistent speedups across various instances and solvers highlight LEOPARD's efficiency in enhancing solver performance. Thus, despite some overhead, its impact on overall verification is minor given the significant improvements and the potential for further optimization.

V. RELATED WORK

Cloud-based access control policy verification. The first access control policy verifier, ZELKOVA [5], [6], introduced a generic encoding framework for policy verification. Block Public Access [7] enhanced ZELKOVA by computing smaller policy formulas through semantics-based pruning. However, they overlooked encoding the default constraints, which can lead to incorrect answers and undermine verification accuracy. In contrast, LEOPARD employs a similar semantics-based pruning method but specifically targets default constraints and introduces an additional structure-based pruning approach to further enhance efficiency. QUACKY [8] aims to quantify the permissiveness of policies rather than providing a binary answer. Our evaluation results demonstrate that LEOPARD outperforms QUACKY by effectively pruning redundant formulas to accelerate the analysis process.

Cloud-based access control policy repair. A quantitative symbolic analysis approach [31] has been proposed to assist users in automatically repairing erroneous policies. The key concept is to iteratively refine the permissiveness of the faulty policy, gradually bringing it in line with the desired specifications. Since this approach employs the same logical encoding as QUACKY, LEOPARD can be utilized to accelerate the repair process.

SMT formula slicing. Formula slicing decomposes a large formula into smaller, manageable sub-formulas based on dependency relations, improving SMT solver efficiency. Tools like Klee divide constraint sets into disjoint subsets based on symbolic variable relationships, facilitating targeted analysis and testing [19]. Axiom selection selects relevant formulas from large theories to accelerate solving, though it risks pruning essential formulas [18], [24], [32]. The structure-based pruning of LEOPARD is inspired by these approaches, but it differs by preserving only the formulas related to the property while ensuring correctness.

SMT formula simplification. Formula simplification is a critical part of SMT solvers, which simplifies the SMT problem before solving it. For instance, solvers like Z3 [9], [20], CVC5 [10], MathSAT [33], and Yices [34] employ various techniques such as term rewriting, constant propagation, and domain-specific simplifications to reduce formula complexity and improve solving efficiency. The biggest challenge of formula simplification is discovering the simplification rules, due to the difficulty of devising effective simplifications and ensuring they do not make the problem harder for the solver. Recently, approaches have leveraged machine learning techniques or syntax-guided synthesis (SyGuS) to discover effective simplification strategies [35], [22]. Additionally, compiler optimization techniques have been adapted for formula simplification [23]. Differing from previous methods, the semantic-based pruning of LEOPARD introduces a novel approach to formula simplification based on ACL policy semantics, focusing on domain-specific optimizations relevant to access control policies. Future improvements could involve incorporating richer and more complex semantics, as well as integrating machine learning and SyGuS techniques to further refine and enhance this simplification method.

VI. Conclusion

In this paper, we introduced LEOPARD, a logical encoding optimization method designed to accelerate cloud-based access control policy verification by pruning redundant formulas. By leveraging both structure-based and semantics-based pruning techniques, LEOPARD effectively eliminates irrelevant parts of the logical encoding, significantly enhancing verification efficiency. Extensive evaluations on high-fidelity synthetic datasets demonstrated that LEOPARD outperforms state-of-the-art SMT-based policy verifiers in terms of efficiency.

Acknowledgments

We are extremely grateful to the anonymous IWQoS 2025 reviewers for their constructive and insightful feedback. This work is supported in part by the National Key R&D Program of China under Grant 2024YFB2906900, NSFC Grants #62172345 and #62302409, and NSF-Fujian-China under Grant 2022J01004, as well as by the Fundamental Research Funds for the Central Universities of China under Grant 20720240049.

References

[1] J. L. Hardcastle, "Mcgraw hill's s3 buckets exposed 100,000 students' grades and personal info," www.theregister.com, 12 2022. [Online]. Available: https://www.theregister.com/2022/12/20/mcgraw_hills_s3_buckets_exposed/

[2] S. Wadhwani, "Misconfigured azure blob storage exposed the data of 65k companies and 548k users," www.spiceworks.com, 10 2022. [Online]. Available: https://www.spiceworks.com/it-security/cloud-security/news/microsoft-azure-cloud-misconfiguration/

[3] K. Loomes, "Alibaba cloud outage sees apps taken offline," W.Media, 11 2023. [Online]. Available: https://w.media/alibaba-cloud-outage-sees-apps-taken-offline/

[4] R. Chirgwin, "Salesforce cloud outage caused by security change," ITnews, 09 2023. [Online]. Available: https://www.itnews.com.au/news/salesforce-cloud-outage-caused-by-security-change-600643

[5] J. Backes, P. Bolignano *et al.*, "Semantic-based automated reasoning for aws access policies using smt," in *FMCAD*. Austin, TX, USA: IEEE, 2018, pp. 1–9.

[6] N. Rungta, "A billion smt queries a day (invited paper)," in *CAV*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 3–18.

[7] M. Bouchet *et al.*, "Block public access: Trust safety verification of access control policies," in *FSE*. New York, NY, USA: ACM, 2020, p. 281–291.

[8] W. Eiers *et al.*, "Quantifying permissiveness of access control policies," in *ICSE*. New York, NY, USA: ACM, 2022, p. 1805–1817.

[9] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*. Berlin, Heidelberg: Springer, 2008, pp. 337–340.

[10] H. Barbosa, C. Barrett *et al.*, "cvc5: A versatile and industrial-strength smt solver," in *TACAS*. Cham: Springer, 2022, pp. 415–442.

[11] M. Berzish, M. Kulczynski *et al.*, "An smt solver for regular expressions and linear arithmetic over string length," in *CAV*. Cham: Springer, 2021, pp. 289–312.

[12] T. Chen *et al.*, "Decision procedures for path feasibility of string-manipulating programs with complex operations," *Proc. ACM Program. Lang*, vol. 3, no. POPL, pp. 1–30, 2019.

[13] R. Amadini, "A survey on string constraint solving," *ACM Comput. Surv.*, vol. 55, no. 1, nov 2021.

[14] A. Nötzli, A. Reynolds, H. Barbosa, C. Barrett, and C. Tinelli, "Even faster conflicts and lazier reductions for string solvers," in *CAV*. Cham: Springer, 2022, pp. 205–226.

[15] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, R. Majumdar, and D. Nowotka, "Solving string constraints using sat," in *CAV*. Cham: Springer, 2023, pp. 187–208.

[16] Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč, "Solving string constraints with lengths by stabilization," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, oct 2023.

[17] Y. Zhang, X. Xie, Y. Li, Y. Lin, S. Chen, Y. Liu, and X. Li, "Demystifying performance regressions in string solvers," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 947–961, 2023.

[18] K. Hoder *et al.*, "Sine qua non for large theory reasoning," in *CADE*. Berlin, Heidelberg: Springer, 2011, pp. 299–314.

[19] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*. USA: USENIX, 2008, p. 209–224.

[20] N. Bjørner and K. Fazekas, "On incremental pre-processing for smt," in *CADE*. Cham: Springer, 2023, pp. 41–60.

[21] K. Hoder, Z. Khasidashvili, K. Korovin, and A. Voronkov, "Preprocessing techniques for first-order clausification," in *FMCAD*, 2012, pp. 44–51.

[22] A. Nötzli, A. Reynolds, H. Barbosa, A. Niemetz, M. Preiner, C. Barrett, and C. Tinelli, "Syntax-guided rewrite rule enumeration for smt solvers," in *SAT*. Cham: Springer, 2019, pp. 279–297.

[23] B. Mikek and Q. Zhang, "Speeding up smt solving via compiler optimization," in *ESEC/FSE*, ser. ESEC/FSE 2023. New York, NY, USA: ACM, 2023, p. 1177–1189. [Online]. Available: https://doi.org/10.1145/3611643.3616357

[24] G. Sutcliffe and Y. Puzis, "Srass - a semantic relevance axiom selection system," in *CADE*. Berlin, Heidelberg: Springer, 2007, pp. 295–310.

[25] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "cvc4," in *CAV*. Springer, 2011, pp. 171–177.

[26] M. Berzish *et al.*, "Z3str3: A string solver with theory-aware heuristics," in *FMCAD*. Austin, Texas: FMCAD Inc, 2017, p. 55–59.

[27] F. Mora *et al.*, "Z3str4: A multi-armed string solver," in *FM*. Cham: Springer, 2021, pp. 389–406.

[28] A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *CAV*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer, 2015, pp. 255–272.

[29] W. Eiers, S. Saha, T. Brennan, and T. Bultan, "Subformula caching for model counting and quantitative program analysis," in *ASE*, ser. ASE '19. IEEE Press, 2020, p. 453–464. [Online]. Available: https://doi.org/10.1109/ASE.2019.00050

[30] A. Aydin, W. Eiers, L. Bang, T. Brennan, M. Gavrilov, T. Bultan, and F. Yu, "Parameterized model counting for string and numeric constraints," in *ESEC/FSE*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, p. 400–410. [Online]. Available: https://doi.org/10.1145/3236024.3236064

[31] W. Eiers *et al.*, "Quantitative policy repair for access control on the cloud," in *ISSTA*. New York, NY, USA: ACM, 2023, p. 564–575.

[32] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," *SIGPLAN Not.*, vol. 37, no. 1, p. 58–70, jan 2002. [Online]. Available: https://doi.org/10.1145/565816.503279

[33] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *TACAS*. Springer, 2013, pp. 93–107.

[34] B. Dutertre, "Yices 2.2," in *CAV*. Cham: Springer, 2014, pp. 737–744.

[35] R. Singh and A. Solar-Lezama, "Swapper: A framework for automatic generation of formula simplifiers based on conditional rewrite rules," in *FMCAD*, 2016, pp. 185–192.