AccessRefinery: Fast Mining Concise Access Control Intents on Public Cloud

NING KANG, Xi'an Jiaotong University, China
PENG ZHANG*, Xi'an Jiaotong University, China
JIANYUAN ZHANG, Xi'an Jiaotong University, China
HAO LI, Xi'an Jiaotong University, China
DAN WANG, Xi'an Jiaotong University, China
ZHENRONG GU, Xi'an Jiaotong University, China
WEIBO LIN, Huawei Cloud, China
SHIBIAO JIANG, Huawei Cloud, China
ZHU HE, Huawei Cloud, China
XU DU, Huawei Cloud, China
LONGFEI CHEN, Huawei Cloud, China
JUN LI, Huawei Cloud, China
XIAOHONG GUAN, Xi'an Jiaotong University, China

Modern cloud applications heavily rely on Identity and Access Management (IAM) services to enforce flexible access control over their data. However, the flexibility comes at a cost: *IAM policies* are often complex and prone to misconfigurations, leading to risks of data exposure. There is an increasing need to mine a compact set of intents that describe what the policies collectively try to achieve, thereby enabling operators to better understand their policies. However, existing tools on mining access control intent have two limitations: (1) the mining process is *slow* and even times out on some complex policies; (2) the mined intents are *excessive* in number and thus still hard to understand. To overcome these, this paper presents *AccessRefinery*, which can speed up the mining process while reducing the number of intents. The key idea for the speedup is to reduce the redundancy of the multi-round SMT solving, by preprocessing the constraints into bit-vector constraints. For intent reduction, *AccessRefinery* computes a compact set of intents that can cover the mined intents, by solving a *min-set-cover* problem. Experiments based on real and synthetic datasets show that *AccessRefinery* achieves a ∼10–100× speedup in intent mining, and reduces the number of intents by up to ∼10×.

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

Additional Key Words and Phrases: Cloud computing, access control, intent mining, SMT

*Corresponding author.

Authors' Contact Information: Ning Kang, Xi'an Jiaotong University, Xi'an, Shaanxi, China, kangning@stu.xjtu.edu.cn; Peng Zhang, Xi'an Jiaotong University, Xi'an, Shaanxi, China, p-zhang@xjtu.edu.cn; Jianyuan Zhang, Xi'an Jiaotong University, Xi'an, Shaanxi, China, 2244313675@stu.xjtu.edu.cn; Hao Li, Xi'an Jiaotong University, Xi'an, Shaanxi, China, hao.li@xjtu.edu.cn; Dan Wang, Xi'an Jiaotong University, Xi'an, Shaanxi, China, dan-wang@stu.xjtu.edu.cn; Zhenrong Gu, Xi'an Jiaotong University, Xi'an, Shaanxi, China, zrgu@stu.xjtu.edu.cn; Weibo Lin, Huawei Cloud, Xi'an, Shaanxi, China, linweibo@huawei.com; Shibiao Jiang, Huawei Cloud, Xi'an, Shaanxi, China, jiangshibiao1@huawei.com; Zhu He, Huawei Cloud, Xi'an, Shaanxi, China, hezhu4@huawei.com; Xu Du, Huawei Cloud, Xi'an, Shaanxi, China, duxu1@huawei.com; Longfei Chen, Huawei Cloud, Xi'an, Shaanxi, China, chenlongfei@huawei.com; Jun Li, Huawei Cloud, Xi'an, Shaanxi, China, alanlee@huawei.com; Xiaohong Guan, Xi'an Jiaotong University, Xi'an, Shaanxi, China, xhguan@xjtu.edu.cn.

ACM XXXX-XXXX/2026/2-ART
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Ning Kang, Peng Zhang, Jianyuan Zhang, Hao Li, Dan Wang, Zhenrong Gu, Weibo Lin, Shibiao Jiang, Zhu He, Xu Du, Longfei Chen, Jun Li, and Xiaohong Guan. 2026. AccessRefinery: Fast Mining Concise Access Control Intents on Public Cloud. 1, 1 (February 2026), 23 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software applications are increasingly deployed in the cloud [4, 5, 16, 17, 31, 41]. To secure these applications, major cloud providers, including Amazon Web Services (AWS) [5], Microsoft Azure [31], and Google Cloud Platform (GCP)[25] offer Identity and Access Management (IAM) systems. These systems follow a shared responsibility model: customers secure their services by writing *access control policies* (or *IAM policies*) while the cloud provider's engine evaluates requests against policies to determine whether access should be authorized [18, 41].

To enable fine-grained access control, *IAM policies* support flexible features including wildcard matching (e.g., regular expressions) and Boolean logic (e.g., conjunctions, disjunctions, and negations) [4, 5, 16–18, 31, 41]. However, such flexibility also makes *IAM policies* complicated and error-prone. Misconfigured *IAM policies* can accidentally allow unintended access, exposing millions of customers' data to the public [37, 40, 44, 46].

A promising way to handle the complexity of *IAM policies* is to mine a concise summary of *intents*, i.e., a compact set of declarative statements that describe *who* can perform *which* actions on *what* sources (see §2.2). By reviewing such intents, users can more confidently determine whether the policies correctly grant the intended access [4], thereby detecting potential misconfigurations. Users can also better understand their policies and refine their designs [16]. Moreover, mining access control intents is crucial to software engineering. First, many software systems today are deployed to the cloud, where access is safeguarded by access control policies. Mining the intents of these policies can formally validate that access is granted as intended. Second, it is closely related to specification mining, which aims to infer invariants or specifications of given programs [19, 22].

The following shows the desired goals for intent mining (the first three are from [4]).

- **Soundness.** All requests allowed by the policy should be included in the intents.
- **Precision.** Any request not allowed by the policy should be excluded as much as possible.
- **Conciseness**. The number of intents should be minimal so that they are concise and easy to understand.
- **Speed**. The mining process should finish within a reasonable time (e.g., on a level of seconds), so as to be responsive to frequent user requests.

Among these goals, soundness is a hard requirement, since we do not want to overlook any (unintended) allowed access. Moreover, precision and conciseness are soft requirements and they may conflict with each other. For example, consider two intents: (`Action`: {`list`, `create`}, `Principal`: {`user1`}), and (`Action`: {`list`}, `Principal`: {`user1`, `user2`}). Merging them into ({`list`, `create`}, {`user1`, `user2`}) sacrifices precision, while keeping them separate sacrifices conciseness.

While the state-of-the-art mining tool *Access Analyzer* (commercially deployed) [4] proposed by AWS meets the first two goals, it still has some limitations in the conciseness and speed.

Limitation 1: Speed. *Access Analyzer* encodes both *IAM policies* and each candidate intent (a hypothesized intent to be checked) as a set of Satisfiability Modulo Theories (SMT) constraints, which are solved using off-the-shelf SMT solvers (also widely applied to analyze *IAM policies* [4, 5, 7, 8, 16–18, 31, 41]). If the constraints are satisfiable, the candidate intent is confirmed as an intent. Since each intent is a combination of values defined over multiple and possibly different domains (e.g., strings, bit-vectors, integers), the SMT solving process needs to reason about a product space of values of each domain, which can be quite large. Moreover, consider a policy with n keys and m distinct values for these keys. The number of candidate intents can be up to $n \times m$. Validation of each intent requires one round of SMT solving. To reduce the number of rounds for SMT solving, *Access Analyzer* uses a stratified approach: start from a most coarse-grained intent (all wildcards), and gradually divide it into fine-grained ones, if it can be covered by any of them (see §2.3 for a detailed discussion). Although this method avoids enumeration, the number of solving rounds is still large. As we will show in Table 2, the number of SMT solving can reach $O(10^3)$ for a policy with 5 keys and 15 statements. Considering each round of SMT solving costs $O(10)$ to $O(1000)$ms, the mining process can take tens of seconds or several minutes, and even times out after 1 hour (see Fig. 13).

Limitation 2: Conciseness. The number of intents returned by *Access Analyzer* can still be too large for users to understand. To make the intent concise, *Access Analyzer* has applied some simple reduction rules. For example, suppose there are two intents (`Action` : `list`, `IpAddress` : `112.0.0.0/24`) and (`list`, `112.0.0.0/25`), then (`list`, `112.0.0.0/25`) is removed since it is a subset of (`list`, `112.0.0.0/24`). Such a simple reduction rule does not fully account for the redundancy among the intents. Suppose there are three intents (`Action` : {`list, create`}, `IpAddress` : `112.0.0.0/25`), (`list`, `IpAddress` : `112.0.0.0/24`), and (`create`, `IpAddress` : `112.0.0.0/24`). The first intent is jointly covered by the second and third intents, and therefore should be eliminated. As we will show, without considering such redundancy, we may end up with 100 intents for an *IAM policy* with just 10 statements, where the minimum number of intents is actually 10 (see Fig. 12).

To overcome the above two limitations, we propose *AccessRefinery*, a new intent mining tool for *IAM policies*, which is faster, and more importantly guarantees the conciseness of intents. Specifically, we make the following contributions:

(1) We propose a method to speed up the mining process by preprocessing SMT constraints. we observe that even though the mining process involves multi-round SMT solving, the possible values of variables are known in advance. We leverage this observation by partitioning each domain into a set of *equivalence classes* (ECs), each of which is represented as an integer, such that the combinations of values from multi-domain values can be represented as simple bit-vectors. As a result, the original SMT problem can be reduced to an SAT problem, which can be solved faster without invoking theory solvers, e.g., string solver, bit-vector solver, integer solver. Moreover, the multiple rounds of SMT solving share intermediate results that can be reused for speedup. Crucially, the preprocessing is per mining process: it needs to be performed once and can be shared by multiple rounds of SMT solving. We realize the above idea with the *Multi-Theory Constraint Preprocessor (MCP)*, which can uniformly handle multiple domain types of *IAM policies* including regular expressions, prefixes, ranges, and enumerations.

(2) We propose a method to efficiently reduce the number of intents by solving a *min-set-cover* problem. Recall that the soundness goal requires that the space of all requests specified by intents should cover the space of requests allowed by the policy. Therefore, computing the minimum number of intents can be formulated as a *min-set-cover* problem [29], i.e., finding the minimum number of intents that can cover the space of requests allowed by the policy. Note here, the space of requests is actually a product space for multiple domains, where a domain can be infinite, e.g., `Resource: "dept*/user1.txt"`. Thanks to the preprocessing, we have reduced each domain into a finite set of ECs. Here, each EC represents a group of requests that exhibit the same behavior under both the policy and the intents; that is, requests within an EC are either all matched by the same set of intents and policy, or none are. We partition the space of the policy and intents into ECs (unlike MCP, this partitioning occurs across multiple domains), enabling both the policy and intents to be represented as finite sets of ECs. With this representation, the *min-set-cover* problem is transformed into one over a finite set; that is, the objective becomes selecting the smallest number of intents whose union covers all ECs allowed by the policy. However, computing these ECs requires iteratively solving SMT problems, which is quite slow (see §3.1). Inspired by [49, 51], we choose to use *Binary Decision Diagrams* (BDDs) [2], which allow incremental computations of ECs.

(3) We implemented a new mining tool named *AccessRefinery*, and used both real and synthetic datasets to show its effectiveness (Both the tool and the synthetic dataset are open-sourced). Compared with existing methods based on SMT solvers, our method speeds up the mining process by ~10× to 100× on synthetic policies, and by ~10× on real policies. Moreover, our method reduces the number of intents by ~10×. Furthermore, we design MCP as a data structure that we believe will also benefit other studies.

$$
\begin{aligned}
Policy &\rightarrow [Statement] \\
Statement &\rightarrow (Effect, Principal, Action, Resource, \\
&\qquad Condition?) \\
Effect &\rightarrow \texttt{Allow} \mid \texttt{Deny} \\
Principal &\rightarrow \texttt{Principal : } [string] \\
Action &\rightarrow \texttt{Action : } [string] \\
Resource &\rightarrow \texttt{Resource : } [string] \\
Condition &\rightarrow \texttt{Condition : } [Operator] \\
Operator &\rightarrow (OpName, KeyName, [Value]) \\
OpName &\rightarrow \texttt{StringEquals} \mid \texttt{StringLike} \mid \texttt{IpAddress} \mid \ldots \\
KeyName &\rightarrow sourceVpc \mid sourceIP \mid prefix \\
Value &\rightarrow string \mid bitvector \mid range \mid enum \mid \ldots
\end{aligned}
$$

Fig. 1. Simplified abstract syntax for the AWS policy language, where "?" denotes optional elements and "[]" denotes a list of valued elements.

2 Motivation

2.1 Background on IAM Policy

IAM policies are typically *attribute-based* (e.g., AWS) or *role-based* (e.g., Azure), differing in syntax but semantically equivalent [18]. This paper focuses on the AWS attribute-based paradigm [5].

Fig. 2 shows the simple syntax for the AWS policy language as defined in [5, 8]. A policy is a list of statements. Each statement is a quintuple: ($Effect$, *Principal*, *Action*, *Resource*, *Condition*). A request is allowed if at least one `Allow` statement matches the request, and none of the `Deny` statements match the request.

```
Effect        : "Allow"    // Statement1
Principal     : "*"
Resource      : "dept*/user1.txt",
                "dept1/user*.txt"
IpAddress     : "112.0.0.0/24", "113.0.0.0/24"
--------------------------------------------
Effect        : "Deny"    // Statement2
Principal     : "*"
NotResource   : "dept*/user1.txt"
IpAddress     : "112.0.0.0/24"
--------------------------------------------
Effect        : "Deny"    // Statement3
Principal     : "*"
NotResource   : "dept1/user*.txt"
IpAddress     : "113.0.0.0/24"
```

Fig. 2. An example of an *IAM policy* with a simplified format. For clarity, we omit some necessary keys, such as `Action`, and omit the condition key for `IpAddress`.

Specifically, $Effect$ specifies whether the request is allowed or denied. *Action* and *Resource* define the permitted or prohibited actions and cloud resources (e.g., S3 buckets, IAM roles). *Principal* indicates the entities performing the actions. *Condition* imposes additional constraints (e.g., IP address or time). A *Condition* is defined as a list of expressions. Each expression applies an operator ($OpName$) to one or more values ($Value$) for a condition key ($KeyName$), where the types of values include strings, bit-vectors, ranges, enumerations, etc. The operator determines both the type of the values and the semantics of the comparison. For example, `StringEquals` checks whether two strings are identical, whereas `StringLike` checks whether a string matches a pattern including wildcards. Moreover, the `IpAddress` checks whether an IP address (i.e., a bit vector) is within an IP prefix.

To achieve flexible and complex access control, *IAM policies* rely on two mechanisms. We take Fig. 2 as an example.

(*i*) *Wildcard semantics.* String operators provide pattern matching, where `*` matches any number of characters, and `?` matches a single character (equivalent to ".*" and "." in regular expressions, respectively). For example, `dept1/user*.txt` matches any file in the `dept1` directory whose name starts with `user` and ends with `.txt`, such as `user123.txt`. `IpAddress` also supports pattern matching. It represents a range of IP addresses using CIDR notation, where a prefix in the format "A/B" specifies all IP addresses that share the first B bits with IP address A. For example, `112.0.0.0/24` restricts the range from `112.0.0.0` to `112.0.0.255`.

(*ii*) *Boolean semantics.* With one domain (key), different values together represent an OR relationship. Between different keys, the relationship is AND. NOT can be indicated by adding `Not` before the domain, such as `NotResource`.

In the example of Fig. 2, the semantics of `Statement2` are as follows: any request whose `Resource` does **not** match `"dept*/user1.txt"` and whose `IpAddress` is within `"112.0.0.0/24"` will be denied. For example, the following request is denied by `Statement2`: the file path `"dept1/user2.txt"` does not match pattern `"dept*/user1.txt"`, and IP address `"112.0.0.32"` falls within IP prefix `"112.0.0.0/24"`.

```
Principal : "user1"   Resource : "dept1/user2.txt"   IpAddress : "112.0.0.32"
```

The semantics of the policy are as follows: a request is allowed if it matches `Statement1` but does not match `Statement2` or `Statement3`; otherwise, it is denied. Thus, the following request is allowed by the policy in Fig. 2 because it satisfies this condition.

```
Principal : "user1"   Resource : "dept1/user1.txt"   IpAddress : "112.0.0.32"
```

This example illustrates that determining which requests are allowed and which are denied by a policy can be challenging. In practice, real-world policies are even more complex, making it crucial to identify the requests that a policy permits.

2.2 Access Control Model

Our access control model is identical to the model of AWS Access Analyzer[4]. The difference is that we define stricter conciseness of the intents, as defined in §4.2.

Requests. Let $K = \{k_1, ..., k_n\}$ be a set of keys. Let $V_k = \{v_1, ...\}$ be a (possibly infinite) set of values for key k. A *request* r is a mapping from keys k to values in V_k. For example, $r = \{$ `Principal` $\mapsto$ `"user1"`, `Resource` $\mapsto$ `"dept1/user2.txt"`, `IpAddress` $\mapsto$ `"112.0.0.32"`$\}$.

Policies. A *policy* is a predicate on requests $\mathbb{P} : r \to \mathit{Bool}$. The allowed requests of a policy $\mathbb{P}$ is defined as:

$$\sigma(\mathbb{P}) = \{r \mid \mathbb{P}(r) = \mathit{true}\} \tag{1}$$

Labels. A *label* l_k is a set of values for a key k, expressed in a human-readable format. All labels l_k for a key k form the universal set L_k. For example, for the key `IpAddress`, the universal set of labels is $L_{\texttt{IpAddress}} = \{\texttt{0.0.0.0/24}, \texttt{112.0.0.0/24}, \texttt{113.0.0.0/24}\}$. Similarly, for the key `Resource`, the universal set of labels is $L_{\texttt{Resource}} = \{$`"*"`, `"dept1/user*.txt"`, `"dept*/user1.txt"`$\}$.

Intents. A intent $\mathbb{I}$ is a map from keys K to labels L. The allowed request of an intent $\mathbb{I}$ is the set of requests where each key k is mapped to a value in the label $\mathbb{I}(k)$:

$$\sigma(\mathbb{I}) = \{r \mid \forall k.r(k) \in \mathbb{I}(k)\} \tag{2}$$

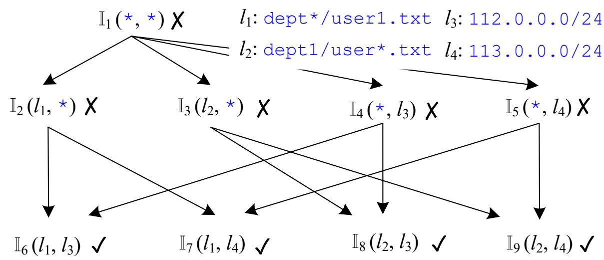


Fig. 3. The processes of *Access Analyzer* for Fig. 2. For simplicity, we only consider two keys, i.e., Resource and IpAddress.

Let $\mathbb{S}$ be a set of intents, i.e., $\mathbb{S} = \{\mathbb{I}_1, \mathbb{I}_2, \ldots, \mathbb{I}_n\}$. Then, the allowed requests of $\mathbb{S}$ is defined as:

$$\sigma(\mathbb{S}) = \bigcup_{\mathbb{I} \in \mathbb{S}} \sigma(\mathbb{I}) \tag{3}$$

For example, the policy in Fig. 2 has the following two intents:

```
Principal : "*"  Resource : "dept*/user1.txt" IpAddress : "112.0.0.0/24" //Intent1
Principal : "*"  Resource : "dept1/user*.txt" IpAddress : "113.0.0.0/24" //Intent2
```

As we can see, the two intents only have positive (allow) declarations, without any negative (deny) declarations. This is to ensure the intents are easier to understand. It is easy to see this set of intents is *sound*, as it includes all requests permitted by the policy. It is also *precise*, containing minimal requests outside the policy (in this example, none). Furthermore, this set of intents is *concise*, since removing any intent results in incomplete coverage of the policy. To formalize these goals, we introduce a set of formal properties (see §4.2).

2.3 Limitations of Existing Approaches

Existing methods for analyzing *IAM policies*, including intent mining, are based on SMT (Satisfiability Modulo Theories). These methods encode both policies and intents [4, 5, 8, 16, 17, 31, 41] as SMT constraints. Typically, a policy is modeled as a constraint $\mathbb{P} = \bigvee_{S \in Allow} S \wedge \neg \bigvee_{S \in Deny} S$, where each statement S is encoded as an SMT constraint.

To the best of our knowledge, the *Access Analyzer* deployed by AWS is the only state-of-the-art intent mining tool documented in the literature[4]. Some tools, such as AWS Zelkova[5] and Microsoft Ambit[31], only determine the implication relationships between two policies. However, even though *Access Analyzer* can ensure the soundness and precision requirements for intent mining, it still cannot meet the other two requirements, i.e., speed and conciseness.

Limitation in speed due to the redundancy among multi-round SMT solving. Mining intents [4] invokes the SMT solver independently for each intent, resulting in a large number of solver invocations. For example, *Access Analyzer* adopts a stratified refinement process based on predefined *labels*, as illustrated in Fig. 3. Starting from an over-permissive intent $\mathbb{I}_1(*, *)$ (ensuring *soundness*), it iteratively checks the following constraint[1]. Here, Child($\mathbb{I}$) (defined in Formula 8) denotes the set of intents that are maximally covered by the current intent (e.g., $\mathbb{I}_2, \mathbb{I}_3, \mathbb{I}_4, \mathbb{I}_5$).

$$\mathbb{P} \wedge \left(\mathbb{I}_i \wedge \neg \left(\bigvee_{\mathbb{I}_j \in \mathrm{Child}(\mathbb{I}_i)} \mathbb{I}_j\right)\right) \tag{4}$$

[1]To enable efficient solving, [4] uses a simplified but equivalent version of this constraint. For clarity, we present the original form.

If the constraint is satisfiable, the intent is included in the final set of intents, indicating that it captures a unique part of the policy (ensuring *precision*); otherwise, its child intents are added to the worklist, and the above process repeats until the worklist is empty. As shown in Fig. 3, 9 intents are checked. Moreover, even a policy with only 15 statements can trigger $O(10^3)$ SMT solver (§6.6) invocations.

Meanwhile, SMT solving per query is expensive due to the iterative process between theory solvers and the SAT solver. As shown in Fig. 4, after the SMT constraint is converted into conjunctive normal form (CNF) [45] and theory-specific parts are replaced with Boolean variables [10], a new constraint Φ is produced. An SAT solver searches for a satisfying assignment M for Φ, which is then iteratively checked by theory solvers. If a theory solver check fails (T-UNSAT), the solver backtracks and tries a new assignment. This loop continues until M is T-SAT or Φ is UNSAT[23]. Thus, each round of SMT solving is expensive, ranging from 28ms to 4545ms (see § 6.6), causing the mining process to take over an hour (§6.3).

However, checking whether a policy and an intent overlap requires full SMT solving, involving multiple theories such as string theory, bit-vector theory, and so on. SMT constraints cannot be simplified through syntax-level transformations.

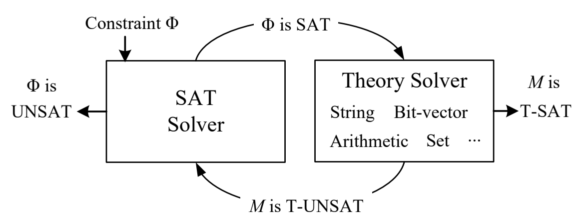
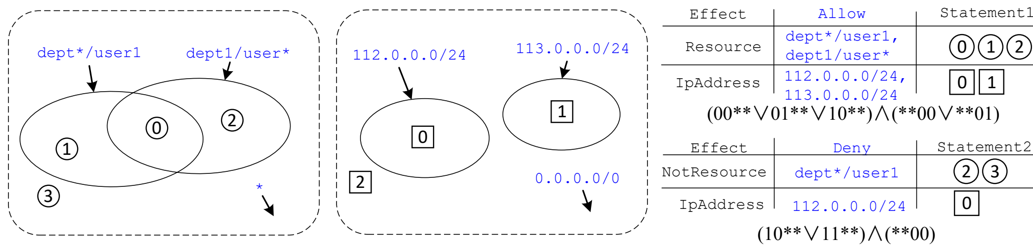


Fig. 4. The basic idea of DPLL(T).

Limitation in conciseness due to not considering cross-intent relations. Fig. 3 shows that four intents, $\mathbb{I}_6, \mathbb{I}_7, \mathbb{I}_8, \mathbb{I}_9$, are generated by *Access Analyzer*, as their corresponding constraints (Formula 4) are satisfiable. However, §2.1 demonstrates that only two intents, $\mathbb{I}_6$ and $\mathbb{I}_9$, are sufficient. This redundancy arises because the four intents partially overlap with one another but are not fully contained within each other. However, these intents still contain redundancy, i.e., $(\mathbb{I}_6 \cup \mathbb{I}_7 \cup \mathbb{I}_8 \cup \mathbb{I}_9) \cap \mathbb{P} \subseteq (\mathbb{I}_6 \cup \mathbb{I}_9) \cap \mathbb{P}$, where $\mathbb{P}$ (or $\mathbb{I}$) denotes the set of permitted requests. Meanwhile, *Access Analyzer* does not detect such relationships. As a result, a policy with only 10 statements can yield results containing more than 100 intents, which is 10× larger than the original policy (§6.2).

However, selecting a minimal subset of intents from the candidate set requires multi-round SMT solving again. Given a candidate set of intents $\mathbb{S}$, the search space grows exponentially as $O(2^{|\mathbb{S}|})$. Each enumeration requires the SMT solver to check whether the selected intents together cover the policy.

Existing works hard to address the limitations. Quacky [17] employs the automata-based solver ABC [3] for model counting, i.e., enumerating the number of policy-allowed requests, but it is less efficient than SMT solvers for single-round satisfiability checks. Similarly, [13] applies the equivalence class approach to count the number of IPs permitted by a policy, but it only supports only IP and lacks regular expression semantics. In addition, Margrave [21] leverages BDDs [2] to check implication between two policies under XACML (rather than IAM) semantics, supporting only exact string matching. Other tools [5, 8, 31, 41] similarly focus on analyzing a single policy or checking implication between two policies. However, all of these approaches do not support accelerating multi-round solving, and thus are difficult to apply to intent mining in IAM policies.



(a) ECs of `Resource` domain (b) ECs of `IpAddress` domain (c) Examples of bit-vectors encoding

Fig. 5. The example of Fig. 2. For clarity, we only consider two domains `Resource`, `IpAddress`.

3 Overview

In this section, we present two key insights: (1) accelerating intent mining by pre-extracting theory redundancy across multi-round SMT solving, and (2) selecting minimal intents by reducing the problem to a *min-set-cover* over finite sets. We then describe the overall workflow.

3.1 Key Insights

Insight 1: The redundancy for theory solvers can be pre-extracted, thereby avoiding theory solving in multi-round SMT solving. For intent mining, we observe that even though it involves multi-round SMT solving, the possible values of variables are known in advance. Taking Fig. 2 as an example, the value of `Resource` is selected from the set $L_{\texttt{Resource}}$ = {`*`, `dept1/user.txt`, `dept*/user1.txt`}. In another case, the value of `IpAddress` comes from the set $L_{\texttt{IpAddress}}$ = {`0.0.0.0/24`, `112.0.0.0/24`, `113.0.0.0/24`}. The difference in multi-round SMT solving lies in how these possible values are combined.

Since all possible values of all constraints are known in advance, this enables us to partition these possible values (i.e., *labels*) into disjoint spaces, i.e., *equivalence classes (ECs)* [49], where two values in the same EC yield the same satisfiability result for the theory solver. Returning to the example in Fig. 2, the *label* set $L_{\texttt{Resource}}$ can be partitioned into four ECs ⓪, ①, ②, ③, as shown in Fig. 5 (a). The *label* set $L_{\texttt{IpAddress}}$ can be partitioned into three ECs $\boxed{0}$, $\boxed{1}$, $\boxed{2}$ as shown in Fig. 5 (b).

Then, we can maintain a mapping $\mathcal{M}$ from each *label* to its corresponding ECs, e.g.,

- `dept*/user1` $\mapsto$ { ⓪, ① }
- `dept1/user*` $\mapsto$ { ⓪, ② }
- `*` $\mapsto$ { ⓪, ①, ②, ③ }

Based on the mapping, we can quickly obtain single-theory constraints over bit-vectors that are equivalent to multi-theory SMT constraints. We use `Statement1` and `Statement2` in Fig. 2 as illustrative examples. We first obtain their ECs from the mapping. As shown in Fig. 5(c), the ECs of `Statement1` in the `Resource` domain are $\mathcal{M}$(`dept*/user1`) ∪ $\mathcal{M}$(`dept1/user*`) = { ⓪, ①, ② } (recall that, for one key, different values together represent an OR relationship). Similarly, the ECs in the `IpAddress` domain are $\mathcal{M}$(`112.0.0.0/24`) ∪ $\mathcal{M}$(`113.0.0.0/24`) = { $\boxed{0}$, $\boxed{1}$ }. Thus, `Statement1` can be represented as the Cartesian product { ⓪, ①, ② } × { $\boxed{0}$, $\boxed{1}$ } (recall that, for different keys, the relationship is AND). We then encode it as a bit-vector, as shown on the left of Fig. 5(c). There is a special case in the figure where $\mathcal{M}$(¬`dept*/user1`) = {⓪, ①, ②, ③} - {⓪, ①} = {②, ③} (recall that, NOT can be indicated by adding `Not` before the domain). After encoding each statement, the policy $\mathbb{P}$ can be expressed as $\mathbb{P} = \bigvee_{S \in Allow} S \wedge \neg \bigvee_{S \in Deny} S$. In this way, both the policy and the intents are encoded using only bit-vector constraints.

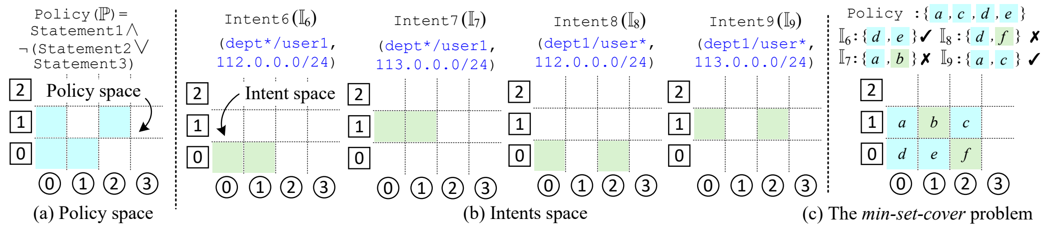


Fig. 6. An example of encoding the selection of minimal intents as *min-set-cover problem.*

Note the pre-computing of equivalence classes is performed only once, and the results are stored in the mapping. In this way, we extract the redundancy among multiple rounds of invoking theory solvers, as shown in Fig. 4. We will show our approach speeds up intent mining by one to two orders of magnitude compared with SMT solvers (§6.3).

Insight 2: Reducing intents requires covering all allowed requests of the policy, and can then be formulated as a min-set-cover problem. We observe that to guarantee *soundness*, the space of requests specified by the intents must fully cover the space of requests allowed by the policy. Therefore, the problem of minimizing the number of intents can be naturally formulated as a *min-set-cover* problem: *selecting the minimal intents from candidate intents that still cover all requests of the policy*, which ensures no valid requests are omitted.

Some classic works [9, 26, 48] address the *min-set-cover* problem over finite sets. However, both policy and intents can potentially span infinite spaces (e.g., `dept*/user1.txt` matches infinitely requests). Therefore, we employ equivalence class (EC) techniques to transform the *min-set-cover* problem over infinite sets into an equivalent problem over finite sets, thereby enabling the use of existing algorithms for the *min-set-cover* problem. We take the policy in Fig. 2 and the candidate intents $\mathbb{I}_6$, $\mathbb{I}_7$, $\mathbb{I}_8$, and $\mathbb{I}_9$ in Fig. 3 as the example. The policy space and intents space (both infinite) are shown in Fig. 6(a) and (b). The request space, over both the policy space and the intents space, is partitioned into six disjoint spaces (ECs), i.e., $\{a, b, c, d, e, f\}$, as shown in Fig. 6(c). Under this partitioning, $\mathbb{I}_6 = \{d, e\}$, $\mathbb{I}_7 = \{a, b\}$, $\mathbb{I}_8 = \{d, f\}$, and $\mathbb{I}_9 = \{a, c\}$, while $\mathbb{P}$ corresponds to $\{a, c, d, e\}$. Therefore, the problem reduces to selecting intents that cover a, c, d, e in $\mathbb{P}$. As a result, $\mathbb{I}_6$ and $\mathbb{I}_9$ are selected.

However, the existing tools compute *equivalence classes* (ECs) primarily using an iterative method [33, 34, 49, 51]. For example, we consider $\mathbb{P}$ and $\mathbb{I}_6$, resulting in three ECs: $\neg\mathbb{I}_6 \wedge \mathbb{P}, \mathbb{I}_6 \wedge \neg\mathbb{P}, \mathbb{I}_6 \wedge \mathbb{P}$. Among these, $\mathbb{I}_6 \wedge \neg\mathbb{P}$ is empty. Next, we consider $\mathbb{I}_7$ and compute intersections with the remaining ECs: $\neg\mathbb{I}_7 \wedge (\neg\mathbb{I}_6 \wedge \mathbb{P}), \mathbb{I}_7 \wedge (\neg\mathbb{I}_6 \wedge \mathbb{P}), \ldots$. We find that even after converting the SMT constraints of policy and intent into bit-vector constraints and employing an SAT solver to check the satisfiability of each constraint, the search space remains large due to the involvement of up to 100 intents.

Therefore, we adopt *Binary Decision Diagrams* (BDDs) to compactly represent bit-vectors instead of using the SAT solver. The insight is that BDDs allow more efficient re-use of intermediate results for incremental computation. BDD is a directed acyclic graph (DAG) with terminal nodes (true and false) and variable nodes. Each variable node assigns a Boolean variable x, with two outgoing edges: a solid line and a dashed line, representing true and false assignments, respectively. Each path from the root to the "true" terminal node represents a truth assignment of the constraint. For clarity, we illustrate the benefits of BDDs with a simple example. Consider two formulas: (1) `11**∧1*0*` and (2) `11**∧1*0*∧*100`. After computing formula (1) with BDDs, we obtain a BDD representing `110*`. When computing formula (2), we can reuse the existing BDD and incrementally compute `110*∧*100`. In contrast, although SAT solvers support push/pop operations, they still need to consider the full formula (2) and only leverage learned clauses to accelerate the search, which is generally less efficient than BDDs. We now consider the scenario of computing ECs. Fig. 7 shows the BDDs of $\mathbb{P}$, $\mathbb{I}_6$ and $\neg\mathbb{I}_6 \wedge \mathbb{P}$ respectively. When computing $\mathbb{I}_7 \wedge (\neg\mathbb{I}_6 \wedge \mathbb{P})$, BDD avoids recomputing $\neg\mathbb{I}_6 \wedge \mathbb{P}$. Instead, BDD compactly represents the solution space of $\neg\mathbb{I}_6 \wedge \mathbb{P}$ so that $\mathbb{I}_7$ operates directly on it.

We will show that our approach reduces the number of intents by one order of magnitude (§6.2) and speeds up intent reduction by one to four orders of magnitude compared with SMT solvers (§6.3). Moreover, in our approach, using BDD to represent bit-vectors is one to two orders of magnitude faster than using SAT solver (§6.5).

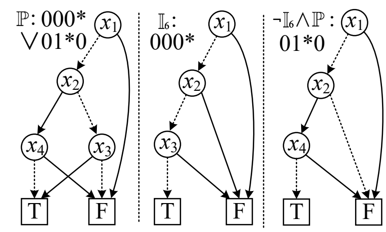


Fig. 7. The *binary decision diagram (BDD)* for $\mathbb{P}$, $\mathbb{I}_6$ and $\neg\mathbb{I}_6 \wedge \mathbb{P}$, respectively.

3.2 Workflow of AccessRefinery

AccessRefinery mines the intents of an *IAM policy*, as shown in Fig. 8. It takes the *IAM policy* as input and produces the corresponding set of intents. To achieve this, *AccessRefinery* consists of three major modules: the *Intent Miner*, the *Intent Reducer*, and the *Multi-theory Constraint Preprocessor* (MCP). Among them, MCP decouples the underlying Boolean operations from the higher-level application logic, which abstracts the low-level operations and allows the upper-layer algorithms to focus solely on their logic. *AccessRefinery* is fully automatic, requiring no operator intervention, and its workflow is as follows:

AccessRefinery first initializes MCP, which (1) automatically collects all variables according to their type, e.g., regular expressions and IP prefixes; (2) maps them to the corresponding operable data structures, e.g., automata and BDDs; (3) maps each operable data structure (e.g., automata and BDDs) to a unified superclass based on language features, e.g., polymorphism in Java; and (4) consistently applies the EC partitioning algorithm.

Then, *Intent Miner* generates a set of raw (non-reduced) intents based on the input policy. It adopts the existing stratified framework [4], but instead of directly using SMT solvers, it relies on MCP for faster multi-round SMT solving.

Finally, *Intent Reducer* generates a minimum set of intents from the set of raw intents, such that they can still cover all raw intents. *Intent Reducer* achieves this by modeling a min-set-cover problem, which MCP transforms into a min-set-cover problem over a finite set, and solving it with an Integer Linear Programming (ILP) solver [24].

We highlight that MCP differs from existing approaches by employing a two-level encoding scheme. First, MCP encodes each domain label using a data structure appropriate for its type: strings are represented using regular expressions, prefixes using BDDs, ranges using red-black trees, and enumerated variables using sets (first level). MCP then partitions equivalence classes and encodes them into BitVectors. These BitVectors are subsequently represented as BDDs (second level), and all SMT solving is performed as Boolean operations on this second-level BDD.

In contrast, prior work [13] directly partitions equivalence classes for IP addresses and then performs single-domain operations on the set of equivalence classes for IP prefixes. Margrave [21] maps strings directly to BDDs by encoding each character as a separate BDD variable; the entire string is represented as the conjunction of these variables. All subsequent operations are performed directly on this BDD, which supports only exact string matching and does not support regular expressions or multi-round SMT solving for acceleration.

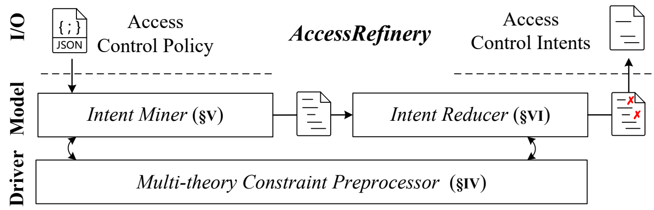


Fig. 8. The workflow of *AccessRefinery*.

4 Design Details

4.1 Multi-Theory Constraint Preprocessor (MCP)

Given the fundamental role of Multi-Theory Constraint Preprocessor (MCP) in both intent mining and reduction, we first introduce the design of MCP.

The APIs. MCP hides the details of calculating ECs for each domain and encoding them as BDDs, allowing upper-layer applications to focus on their own logic. For example, Fig. 9 illustrates the use of MCP to solve $\neg\mathbb{I}_6 \wedge \mathbb{I}_7$ without exposing the underlying details. The APIs of MCP are explained as follows:

- `AddValue(Domain, Type, Value)` defines a variable in `Domain` with the specified `Type` and `Value`. For IAM policies, supported types include `REGEXP` (e.g., the `Resource` key), `PREFIX` (e.g., the `IpAddress` key), `RANGE` (e.g., the `DateGreaterThan` key), and `ENUM` (e.g., the `Action` key).
- `PartitionECs()` partitions all values into ECs for each domain and maps each *label* to its corresponding ECs. This function is called after all domain values have been declared.
- `GetValue(Domain, Value)` returns the BDD representation of a `Value` in a `Domain`. SMT solving over *labels* is equivalently converted into BDD operations.

Definition of Equivalence Classes. A *label* is denoted as $l = \mathit{false}$ if it is a contradiction, e.g., $l =$ `112.0.0.0/24` $\wedge$ `113.0.0.0/24`, and as $l = \mathit{true}$ if it is a tautology, e.g., $l =$ `0.0.0.0/0`.

Following [49], we define ECs as follows:

Definition 1. *Given a set of labels $\mathcal{L}$, a set of equivalence classes $\mathcal{C} = \{c_1, \dots, c_N\}$ satisfies:*

i) $c_i \neq \mathit{false}, \forall i \in \{1, \dots, N\}$,
ii) $\bigvee_{i=1}^{N} c_i = \mathit{true}$,
iii) $c_i \wedge c_j = \mathit{false},\ \mathit{if}\ i \neq j$,
iv) Each $l \in \mathcal{L}, l \neq \mathit{false}$*, can be expressed as the disjunction of a subset of equivalence classes:*

$$l = \bigvee_{i \in \mathcal{M}(l)} c_i, \quad \mathcal{M}(l) \subseteq \{1, \dots, N\},$$

v) N *is the minimal number that the above conditions hold.*

The above definition induces a mapping $\mathcal{M} : \mathcal{L} \to 2^{\{1,\dots,N\}}$ from *label* to its corresponding ECs. For example, $\mathcal{M}($`dept*/user1`$) = \{①, ②\}$. This decomposition is *unique* for any $l \in \mathcal{L}$ [49]. Crucially, logical operations over *labels* are reduced to efficient set operations: $\mathcal{M}(l_1 \wedge l_2) = \mathcal{M}(l_1) \cap \mathcal{M}(l_2)$, $\mathcal{M}(l_1 \vee l_2) = \mathcal{M}(l_1) \cup \mathcal{M}(l_2)$[49].

Computation of Equivalence Classes. After collecting all values for each domain, we calculate the EC and mapping $\mathcal{M}$ as follows:

Step (1) Domain-Specific Label Representation. Partitioning ECs requires Boolean operations, e.g., conjunction ($\wedge$) and negation ($\neg$). However, *labels* cannot directly perform this operation. For example, `113.0.0.0/24` $\wedge\neg$ `112.0.0.0/24` cannot be represented as a single CIDR block. Therefore, we map each *label* to a corresponding data structure that supports equivalent Boolean operations.

```
/* Declare the domain values. */
MCP mcp = new MCPFactory();
mcp.addValue("Res", REGEXP, "dept*/user1.txt");
mcp.addValue("Res", REGEXP, "dept1/user*.txt");
mcp.addValue("IP",  PREFIX, "112.0.0.0/24");
mcp.addValue("IP",  PREFIX, "113.0.0.0/24");
/* Extract theory redundancy.*/
mcp.partitionECs();
/* Multi-round SMT solving. */
BDD res1  = mcp.getValue("Res", "dept*/user1.txt");
BDD res2  = mcp.getValue("Res", "dept1/user*.txt");
BDD ip1 = mcp.getValue("IP", "112.0.0.0/24");
BDD ip2 = mcp.getValue("IP", "113.0.0.0/24");
BDD s1 = (res1.or(res2)).and(ip1.or(ip2));
BDD s2 = res1.not().and(ip1);
BDD s3 = res2.not().and(ip2);
BDD policy = s1.diff(s2).diff(s3);
BDD intent6 = res1.and(ip1);
if (policy.and(intent6.not()).sat() == true) {...}
```

Fig. 9. An example usage of MCP.

- `Regexp` *labels* are mapped to automatons [27]. An automaton is a finite-state machine capable of recognizing patterns described by regular expressions. For example, `dept*/user1.txt` $\vee$ `dept*/user2.txt` is represented as an automaton that accepts both patterns.
- `BitVec` *labels* are mapped to nodes in a Binary Decision Diagram (BDD) [2], a data structure used to represent Boolean functions. For example, `112.0.0.0/24` $\wedge\neg$ `112.0.0.128/25` results in a BDD representing the IP range `112.0.0.0/25`.
- `Range` *labels* are mapped to nodes in a balanced binary search tree (BST) [28]. For example, `[9, 18)` $\vee$ `[19, 24)` results in two disjoint intervals, represented as two nodes in the tree.
- `Enum` *labels* are mapped to sets, i.e., collections of unique elements. For example, `ListBucket` $\vee$ `CreateBucket` results in a set with two elements.

Step (2) Domain-Independent ECs Partitioning. For a given set of *labels* $\mathcal{L}$, Algorithm 1 generates the set of ECs $\mathcal{C}$ and the mapping $\mathcal{M}$. Specifically, Line 2 initializes the current set of ECs with the *true label*. In Line 3, it iterates over each *label* $l \in \mathcal{L}$. For each *label* l and each current EC $c \in \mathcal{C}$, it computes the conjunctions $l \wedge c$ and $l \wedge \neg c$ (Line 6). Only non-empty results are included in the current EC set. Line 7 updates $\mathcal{C}$ with the new ECs for the next iteration. Lines 8–10 construct the mapping $\mathcal{M}$ by associating each *label* l with the set of ECs. Finally, $\mathcal{C}$ and $\mathcal{M}$ are returned.

Many prior approaches [13, 49, 51] employ a similar approach to partition equivalence classes (ECs). However, they are limited to EC partitioning over IP prefixes. Our approach differs from previous work in that we use labels representing abstract semantics. By mapping variables of different types to corresponding operable data structures, we then perform EC partitioning on a unified abstract representation.

Label Encoding. After calculating the mappings from *label* to ECs, we encode SMT constraints over *labels* as follows:

Step (1) Encoding EC as Bit-Vectors. Suppose there are m domains, and the k-th domain contains N_k ECs. We adopt binary encoding, where the total number of bits required is $\sum_{k=1}^{m}\lceil\log_2(N_k)\rceil$. Each domain is assigned a unique range $\texttt{Position}(k) = \left[\sum_{j=1}^{k-1}\lceil\log_2(N_j)\rceil + 1,\ \sum_{j=1}^{k}\lceil\log_2(N_j)\rceil\right]$. We next define $\mathcal{V}(i,k)$ as a bit-vector encoding the i-th EC in the k-th domain. Let $\texttt{binary}(a,\ b)$

Algorithm 1: Compute ECs for a single domain

```
Input  : 𝓛: the set of labels.
Output : 𝓒: the set of equivalence classes for 𝓛.    𝓜: the mapping from label to its ECs.
Function ComputeSECs(𝓛):
    𝓒 ← {true}
    foreach l ∈ 𝓛 do
        𝓒′ ← {}
        foreach c ∈ 𝓒 do
            𝓒′ ← 𝓒′ ∪ {l ∧ c, l ∧ ¬c}          ▷ calculating equivalence classes
        𝓒 ← 𝓒′
    foreach l ∈ 𝓛, c ∈ 𝓒 do
        if l ∧ c ≠ false then
            𝓜(l) ← 𝓜(l) ∪ {id(c)}      ▷ calculating the mapping from label to its ECs
    return 𝓒, 𝓜
```

denote the bit-vector representing the range from integer a to b. For example, $\texttt{binary}(0, 2) = \texttt{0*} \vee \texttt{10}$. Then, $\mathcal{V}(i, k)$ assigns $\texttt{binary}(i, i)$ to the bit range specified by $\texttt{Position}(k)$. For all other domains $j \neq k$, the bits in their respective ranges $\texttt{Position}(j)$ are set to $\texttt{binary}(0, N_j - 1)$, where N_j is the number of ECs in domain j.

Step (2) Encoding labels as BDDs. Each *label* l belongs to a specific domain k and has corresponding ECs $\mathcal{M}_k(l)$. Let $\mathcal{B}(v)$ return the BDD for the bit-vector v. Then, the BDD encoding of *label* l is $\mathcal{F}(l) = \bigvee_{i \in \mathcal{M}_k(l)} \mathcal{B}(\mathcal{V}(i, k))$.

THEOREM 1. *Let l_1 and l_2 be labels, and let $\circ \in \{\wedge, \vee, \setminus, \Rightarrow, \Leftrightarrow\}$ be a Boolean operation. Then, $\mathcal{F}(l_1 \circ l_2) = \mathcal{F}(l_1) \circ \mathcal{F}(l_2)$ and $\mathcal{F}(\neg l_1) = \neg \mathcal{F}(l_1)$ both hold.*

PROOF SKETCH. Mapping each domain label to its corresponding operable data structure preserves the equivalence of Boolean operations, as established in [2, 32, 38, 43] for `Regexp`, `BitVec`, `Range`, and `Enum`, respectively. Furthermore, encoding equivalence classes (ECs) as bitvectors also preserves the equivalence, since each domain is assigned a non-overlapping segment of the bitvector, ensuring a one-to-one correspondence between the ECs and their bitvector representations. □

4.2 Intent Miner

In this section, we show the goals that high-quality intents should satisfy, and then describe how to generate such intents.

Goals. To satisfy soundness, precision, and conciseness, we define the following properties, inspired by [4]. Recall that $\sigma(\mathbb{P}), \sigma(\mathbb{I}), \sigma(\mathbb{S})$ denote the set of allowed requests of the policy, intent, and set of intents, respectively.

- **Soundness (Coverage)**: The set of intents must cover all allowed requests:

$$\sigma(\mathbb{P}) \subseteq \sigma(\mathbb{S}) \tag{5}$$

- **Precision (Irreducibility)**: Each intent captures a unique part of the policy and cannot be further refined:

$$\forall \mathbb{I} \in \mathbb{S},\ \exists r \in \sigma(\mathbb{I}) \cap \sigma(\mathbb{P}),\ \forall \sigma(\mathbb{I}') \subseteq \sigma(\mathbb{I}),\ r \notin \sigma(\mathbb{I}') \tag{6}$$

- **Conciseness (Minimality)**: No intent is redundant with respect to the others:

$$\forall \mathbb{I} \in \mathbb{S}, \sigma(\mathbb{I}) \cap \sigma(\mathbb{P}) \not\subseteq \bigcup_{\mathbb{I}' \in \mathbb{S}, \mathbb{I} \neq \mathbb{I}'} \sigma(\mathbb{I}') \cap \sigma(\mathbb{P}) \tag{7}$$

We highlight that our definition of *minimality* is stricter than in prior work [4], which only analyzes the mutual inclusion relationship between two intents.

Intent Mining. *Intent Miner* adopts a stratified analysis approach to ensure both *coverage* and *irreducibility*. While this approach is inspired by *Access Analyzer* [4], we replace its underlying SMT solver with our customized MCP engine. Specifically, *Intent Miner* initializes a *worklist* set containing a single element, i.e., the universe intent, and an empty *result* set. We then iteratively process the *worklist*: for each current intent, we remove it from the *worklist* and enumerate all its *maximal subsets* via Child($\mathbb{I}$). These maximal subsets are defined as intents included by the current intent and not implied by any other intent, where $\mathbb{I}' \prec \mathbb{I}$ means $\sigma(\mathbb{I}') \subseteq \sigma(\mathbb{I})$.

$$\mathrm{Child}(\mathbb{I}) \triangleq \{\mathbb{I}' \mid \mathbb{I}' \prec \mathbb{I} \wedge \forall \mathbb{I}''. \neg(\mathbb{I}' \prec \mathbb{I}'' \prec \mathbb{I})\} \tag{8}$$

Next, *Intent Miner* checks Formula 4. If Formula 4 is satisfiable, we add it to the *result*. If the formula is unsatisfiable, we add all proper subsets of the current intent to the *worklist*. This process repeats until the *worklist* becomes empty. According to [4], the result satisfies both *coverage* and *irreducibility*.

4.3 Intent Reducer

In this section, we first formulate the task of selecting a minimal subset of intents as a *min-set-cover* problem. We then describe how this problem is reduced to a finite instance using ECs, and finally solved using an Integer Linear Programming (ILP) solver[24].

Problem Formulation. Let the set of candidate intents be $\mathbb{S} = \{\mathbb{I}_1, \mathbb{I}_2, \ldots, \mathbb{I}_n\}$, generated by *Intent Miner*. These candidate intents satisfy both *coverage* and *irreducibility* [4].

However, the number of intents is often large, motivating the need to select a smaller subset. Removing intents may lead to a loss of *coverage*, as some requests allowed by the policy might no longer be covered. Fortunately, *irreducibility* is preserved under subset selection:

Lemma 1. *If $\mathbb{S}$ is irreducible, then any non-empty subset $\mathbb{S}' \subseteq \mathbb{S}$ is also irreducible.*

Proof Sketch. We prove it by contradiction. Assume that a non-empty subset $\mathbb{S}' \subseteq \mathbb{S}$ is *reducible*. Then, (1) $\exists \mathbb{I}' \in \mathbb{S}'$ such that $\forall r \in \sigma(\mathbb{I}') \cap \sigma(\mathbb{P})$, $\exists \sigma(\mathbb{I})'' \subseteq \sigma(\mathbb{I})'$, $r \in \sigma(\mathbb{I}'')$). However, since $\mathbb{S}$ is *irreducible* and $\mathbb{I}' \in \mathbb{S}$ (because $\mathbb{S}' \subseteq \mathbb{S}$), the *irreducibility* condition demands that (2) $\exists r \in \sigma(\mathbb{I}') \cap \sigma(\mathbb{P})$ such that $\forall \sigma(\mathbb{I})'' \subseteq \sigma(\mathbb{I})'$, $r \notin \sigma(\mathbb{I}'')$. This leads to a contradiction between (1) and (2). Hence, all non-empty subsets of $\mathbb{S}$ must be *irreducible*. □

We now formally state our goal:

Problem 1. *Given $\mathbb{S} = \{\mathbb{I}_1, \ldots, \mathbb{I}_n\}$ and a target $\mathbb{P}$, find a minimal $\mathbb{S}' \subseteq \mathbb{S}$ such that $\sigma(\mathbb{P}) \subseteq \sigma(\mathbb{S}')$ and $|\mathbb{S}'|$ is minimized.*

Problem Solving. We solve the above problem as follows.

Step (1) Converting problem on finite sets. To handle potentially infinite policies and intents, we define a *label* set $\mathcal{L} = \{\mathbb{P}, \mathbb{I}_1, \ldots, \mathbb{I}_n\}$, encode each element in $\mathcal{L}$ using MCP, and partition the request space (i.e., the policy and intents) into ECs using Algorithm 1. Recall that $\mathcal{M}(\mathbb{P})$ and $\mathcal{M}(\mathbb{I}_i)$ denote the set of ECs of policy $\mathbb{P}$ and intent $\mathbb{I}_i$, respectively. Thus, Problem 1 is reduced to the following finite variant:

Problem 2. *Given* $\{\mathcal{M}(\mathbb{I}_1), \ldots, \mathcal{M}(\mathbb{I}_n)\}$ *and* $\mathcal{M}(\mathbb{P})$*, find a minimal subset* $\mathbb{S}$ *such that* $\mathcal{M}(\mathbb{P}) \subseteq \mathcal{M}(\mathbb{S})$.

Step (2) Solving problem via ILP Solver. In our datasets, since each policy and intent maps to only a small number of ECs, we can solve Problem 2 exactly using ILP solver[24]. Let $x_i \in \{0, 1\}$ indicate whether intent $\mathbb{I}_i$ is selected. The ILP formulation for Problem 2 is:

$$\text{minimize} \quad \sum_{i=1}^{n} x_i \tag{9}$$

$$\text{subject to} \quad \left(\sum_{i \,:\, c_j \in \mathcal{M}(\mathbb{I}_i)} x_i \right) \geq 1, \quad \forall c_j \in \mathcal{M}(\mathbb{P}) \tag{10}$$

$$x_i \in \{0, 1\}, \quad \forall i = 1, \ldots, n \tag{11}$$

5 Experiment Setup

Implementation. We implement *AccessRefinery* with ~5k lines of Java code, which includes three major components: *Intent Miner*, *Intent Reducer*, and *Multi-theory Constraint Preprocessor (MCP)*. For the MCP, we implement two methods to solve the bit-vector constraints, one based on an SAT solver, i.e., MiniSAT [15], and one based on a BDD solver, i.e., JavaBDD [47], and refer to them as *AccessRefinery*(MiniSAT) and *AccessRefinery*(JavaBDD), respectively.

Baselines. To compare with *Intent Miner*, we re-implement *Access Analyzer* proposed by AWS, with approximately 5k lines of Java code. AWS *Access Analyzer* [4] is not open-sourced, we therefore reproduce it for a consistent environment setup. To the best of our knowledge, *Access Analyzer* [4] is the only state-of-the-art intent-mining tool. *Access Analyzer* is built on Zelkova [5], which relies on an SMT solver. We also reproduced Zelkova and integrated it into *Access Analyzer*. To enable a more comprehensive comparison, we replace the underlying SMT solver with CVC5 and Z3, respectively. The original paper uses Z3 [11] as the SMT solver, and is efficient for the bit-vector theory [42]. CVC5 [6], another SMT solver, generally performs better on string theory [36]. We refer to these two implementations as *Access Analyzer* (Z3) and *Access Analyzer* (CVC5). Moreover, Quacky [17] uses the ABC solver [3] as its underlying engine, which is primarily designed for model counting. The performance of ABC for satisfiability solving is generally slower than that of CVC5 and Z3. Therefore, we do not present results for *Access Analyzer* using ABC as its underlying solver. Additionally, AWS provides an online Command Line Interface (CLI) [1] for *Access Analyzer*, which we use to validate the correctness of our re-implementations.

To compare with our *Intent Reducer*, we also implemented two baselines. Both baselines follow the *Intent Reducer* algorithm. The only difference is that they use an SMT solver (Z3 or CVC5) to partition the ECs of the policy and intents. We refer to them as *Baseline* (Z3) and *Baseline* (CVC5).

Datasets. We conduct experiments on both real-world and synthetic datasets.

i). Real-world datasets include 506 policies collected over one year from a customer of a large cloud provider (anonymity requirements). Each policy has 1 to 12 statements, each with 5 to 8 keys and 5 to 20 distinct values.

ii) **Synthetic datasets 1** for the correctness experiment. To fully test correctness, these datasets cover all possible relationships between values: $v_1 = v_2$ (equal), $v_1 \subseteq v_2$ or $v_2 \subseteq v_1$ (overriding), $v_1 \cap v_2 \neq \emptyset$[2] (overlapping), and $v_1 \cap v_2 = \emptyset$ (disjoint). Specifically, we synthesized *policies* containing two statements, each involving two keys, as two keys suffice to test all value and statement relationships. Since many value combinations are redundant, we selected 12 representative datasets

[2]For simplicity, this notation indicates intersection, i.e., $v_1 \cap v_2 \neq \emptyset$, $v_1/v_2 \neq \emptyset$, and $v_2/v_1 \neq \emptyset$.

(shown in Table 1) that cover all possible relationships. We also include value complements, denoted by $\overline{v_4}$, such as `NotResource`.

Table 1. Categories of Policies for Correctness Testing

Intent Category	Relation Category	Logical Conditions
Allow & Allow	Equal	$v_1 = v_3,\ v_2 = v_4$
$\text{Allow}_1[k_1{=}v_1, k_2{=}v_2]$,	Overriding	$v_1 = v_3,\ v_2 \subseteq v_4$
$\text{Allow}_2[k_1{=}v_3, k_2{=}v_4]$	Overlapping1	$v_1 \cap v_3 \neq \emptyset,\ v_4 \subseteq v_2$
	Overlapping2	$v_1 \cap v_3 \neq \emptyset,\ v_2 \cap v_4 \neq \emptyset$
	Disjoint	$v_1 \cap v_3 = \emptyset,\ v_2 \cap v_4 \neq \emptyset$
Allow & Deny	Equal	$v_1 = v_3,\ v_2 = v_4$
$\text{Allow}_1[k_1{=}v_1, k_2{=}v_2]$,	Overrding1	$v_1 = v_3,\ v_2 \subseteq v_4$
$\text{Deny}_1[k_1{=}v_3, k_2{=}v_4]$	Overrding2	$v_1 = v_3,\ v_4 \subseteq v_2$
	Disjoint	$v_1 \cap v_3 = \emptyset,\ v_2 \cap v_4 \neq \emptyset$
Allow & Deny (Comp.)	Overlapping1	$v_1 \cap v_3 \neq \emptyset,\ v_4 \subseteq v_2$
$\text{Allow}_1[k_1{=}v_1, k_2{=}v_2]$,	Overlapping2	$v_1 \cap v_3 \neq \emptyset,\ v_2 \subseteq v_4$
$\text{Deny}_1[k_1{=}v_3, k_2{=}\overline{v_4}]$	Disjoint	$v_1 \cap v_3 = \emptyset, v_2 \cap v_4 \neq \emptyset$

iii) **Synthetic datasets 2** for the scalability experiment. To avoid performance bias, these datasets are designed to (1) cover all possible relationships between values, (2) use keys consistent with those in real-world datasets, and (3) vary the number of statements to assess scalability. Specifically, we generated datasets with 5 and 6 keys, based on our real-world datasets and prior work [16]. In the 5-Key datasets, three keys (`Principal`, `Action`, and `Resource`) have equal values; one key (`SourceArn`) has pairwise *disjoint* values; and another key (`PrincipalArn`) has pairwise *overlapping* values. In the 6-Key datasets, we add an additional key (`IpAddress`) with pairwise *overriding* values. For both settings, we vary the number of `Allow` statements from 1 to 15.

All experiments run on a Linux server with two 16-core Intel Xeon Gold 6226R CPUs @2.9GHz and 256G memory. Additionally, the timeout for each experiment is set to 1 hour.

6 Experiment Results

In the following, we evaluate *AccessRefinery* to answer six research questions:

6.1 Is the re-implementation of *Access Analyzer valid*, and is *AccessRefinery* (including MCP, *Intent Miner* and *Intent Reducer*) correct?

6.1.1 Validity of re-implementation of Access Analyzer. We treat the AWS Access Analyzer via the CLI API as the ground truth. We verified the validity of our reproduced version from two aspects.

Execution Time. The trend of execution time for our reproduced version and the commercial version is consistent for both the 5-Key and 6-Key datasets, as shown in Fig. 10. For the 6-Key dataset with 13 to 15 statements, both versions time out (>1 hour). Although the reproduced and commercial versions do not maintain a perfectly linear relationship, this is expected because the underlying SMT solver uses heuristic search algorithms.

Result Correctness. The reproduced version produces intents identical to those of the commercial version as shown in the right of Fig. 12.

6.1.2 Correctness of AccessRefinery. We validate the correctness of AccessRefinery as the following.

Correctness of MCP. To ensure that the satisfiability results produced by MCP and SMT solver are equivalent, we conducted a series of basic Boolean operations tests (available in our source code), as follows: (1) Nested expressions: $(A \wedge B) \vee (\neg A \wedge C)$; (2) Implication chains: $(A \rightarrow B) \wedge (B \rightarrow$

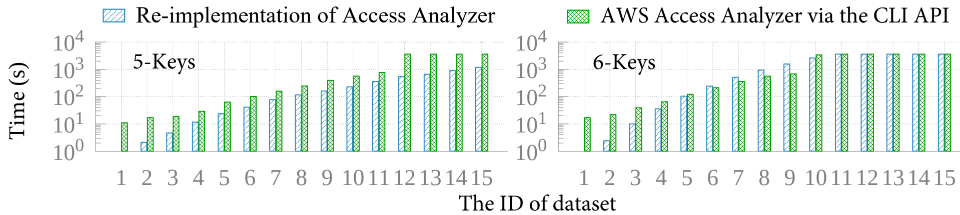


Fig. 10. The total time of mining intents by using re-implementation of *Access Analyzer* and AWS *Access Analyzer* via the CLI API for synthetic datasets.

C); (3) Distributive law: $A \wedge (B \vee C)$ vs $(A \wedge B) \vee (A \wedge C)$; (4) De Morgan's laws: $\neg(A \wedge B)$ vs $\neg A \vee \neg B$; (5) Tautology and contradiction: $A \vee \neg A$ (tautology) and $A \wedge \neg A$ (contradiction); (6) Ternary logic / cyclic implications: $(A \rightarrow B) \wedge (B \rightarrow C) \wedge (C \rightarrow A)$, verified against $A \leftrightarrow B \leftrightarrow C$; (7) Bi-implication: $(A \wedge B) \leftrightarrow (C \vee \neg A)$; (8) All variables false: $\neg A \wedge \neg B \wedge \neg C$; (9) All variables true: $A \wedge B \wedge C$; (10) Complex combination: $(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee \neg C)$. All tests successfully verified the expected satisfiability behaviors, confirming that `MCP` correctly computes Boolean satisfiability.

Correctness of Intent Miner. We compared the intents produced by *AccessRefinery* (without intent reduction), our re-implementation of *Access Analyzer*, and the AWS *Access Analyzer* via the CLI API. On synthetic datasets, all three produce the same set of intents. On real-world datasets, due to privacy constraints, we only compare *AccessRefinery* with our re-implementation of *Access Analyzer*, and they also produce the same set of intents. Fig. 11 shows the number of intents returned by these tools, which further confirms the correctness of MCP.

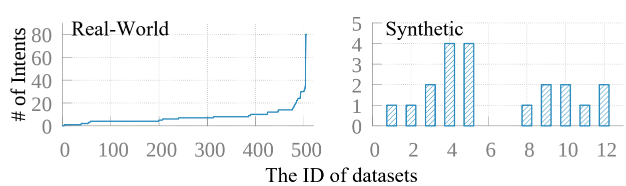


Fig. 11. The number of intents generated by *AccessRefinery*, our re-implementation of *Access Analyzer*, and AWS *Access Analyzer* via the CLI API for *mining intents*.

Correctness of Intent Reducer. The oracle may use an SMT solver to enumerate all intent sets covering the original policy and select the minimal one. However, since this is slow and even time out (>24 h), we instead use an SMT solver to check that *Intent Reducer* satisfies (for real-world and synthetic datasets 1 & 2): (1) The reduced intents fully cover the policy; (2) Removing any intent from the reduced intents causes the remaining intents to no longer cover the policy.

Answer to RQ1: The re-implementation of *Access Analyzer* shows consistent execution time trends and identical intents compared to the AWS *Access Analyzer* via the CLI API. MCP shows same satisfiability with SMT solver. *Intent Miner* produces the same intents as *Access Analyzer*. *Intent Reducer* produces the minimal intents that still cover the original policy.

6.2 Can *AccessRefinery* reduce the number of intents?

Fig. 12 shows that, for real-world datasets, the number of intents can reach up to 80, while for synthetic datasets, it can reach up to 225 intents. These correspond to the original policies containing 5 and 15 statements, respectively. For both real-world and synthetic datasets, *AccessRefinery* reduces the number of intents by up to one order of magnitude. In both types of datasets, the reduction becomes more significant as the number of `Allow` statements increases.

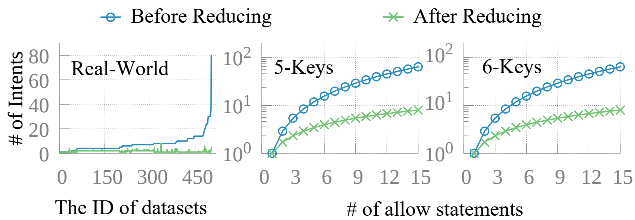


Fig. 12. The number of intents generated before *reducing intents* and after *reducing intents*. Note that the x-axis for the synthetic datasets represents the number of allowed statements.

> **Answer to RQ2**: *AccessRefinery* can reduce the number of intents by up to one order of magnitude for both real and synthetic datasets.

6.3 Can *AccessRefinery* speed up intent mining and reduction by using MCP?

For the *intent mining*, Fig. 13 shows that *AccessRefinery* is faster than *Access Analyzer* (Z3) by one to two orders of magnitude, and faster than *Access Analyzer* (CVC5) by one to three orders of magnitude on both the 5-Key and 6-Key datasets. Although *Access Analyzer* (CVC5) is initially faster than *Access Analyzer* (Z3), its performance degrades as the number of allowed statements increases. This is because Z3 handles formulas involving set intersections and bit-vector[42] theory more efficiently on our datasets.

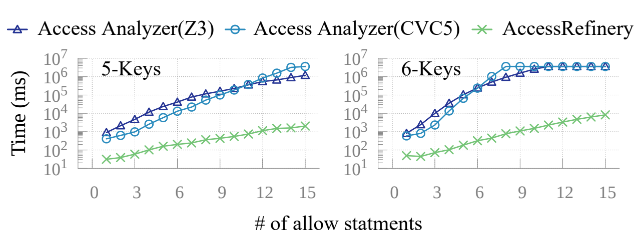


Fig. 13. The total time of mining intents for synthetic datasets.

For the *intent reduction*, Fig. 14 shows that *AccessRefinery* outperforms both *Baseline*(Z3) and *Baseline*(CVC5) by one to four orders of magnitude on the 5-Key and 6-Key datasets. Note that *Baseline*(CVC5) times out with 12 statements on the 5-Key dataset and 8 statements on the 6-Key dataset, while *AccessRefinery* can still reduce intents within ~10s even with 15 statements.

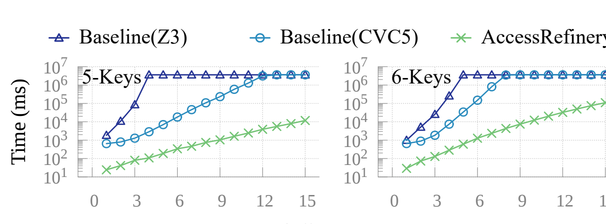


Fig. 14. The total time of reducing intents for synthetic datasets.

> **Answer to RQ3:** For the *intent mining*, *AccessRefinery* achieves a speedup of one to two orders of magnitude, and for the *intent reduction*, it achieves a speedup of one to four orders of magnitude.

6.4 How does *AccessRefinery* perform on real-world datasets?

Since we have previously analyzed the time for individual policies, we now present the cumulative time to better evaluate how *AccessRefinery* handles a large number of policies in real-world scenarios. Fig. 15 compares the cumulative time for both the *intent mining* and *intent reduction* stages. Using Z3, *Access Analyzer* (Z3) and *Baseline* (Z3) require approximately 5983.6s and 2109.1s, respectively, to process 506 policies. With CVC5, which is relatively more efficient at handling string constraints, *Access Analyzer* (CVC5) and *Baseline* (CVC5) still require around 912.3s and 3401.1s, respectively. In contrast, *AccessRefinery* completes both stages in just 64.2 and 20.2s, respectively. We clearly observe performance spikes in *Access Analyzer* (Z3) and *Baseline* (Z3) during both stages. This is primarily due to the presence of many website-related values, such as patterns like "www.example.*", which are expensive to process.

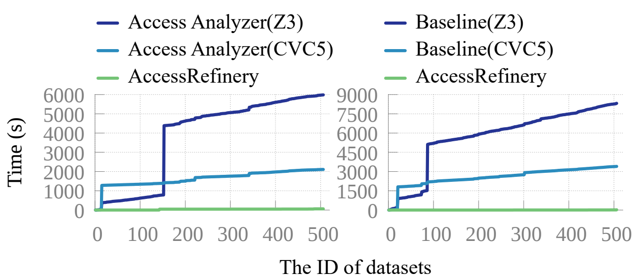


Fig. 15. The cumulative time of the *intent mining* stage and the *intent reduction* stage for real-world datasets. The runtime of *AccessRefinery* is not visible in the figure as it is minimal: 64.2s and 20.2s, respectively.

> **Answer to RQ4:** For the real-world dataset of 506 policies, *AccessRefinery* achieves substantial speedups over the baselines, being up to 15× faster in the *intent mining* and up to 170× faster in the *intent reduction*.

6.5 Is SAT or BDD better for intent mining and reduction?

For *intent mining*, using JavaBDD is 1–6× faster than using MiniSAT [3] (For clarity, the figure is omitted.). This improvement mainly comes from solving Formula (4). For *intent reduction*, using JavaBDD allows our method two orders of magnitude faster than using MiniSAT, as shown in Fig. 16. Moreover, when processing 14 statements under the 5-Keys and 6-Keys settings, the MiniSAT-based version times out, while the JavaBDD-based version finishes in 100s.

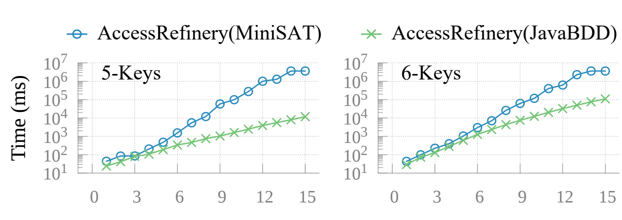


Fig. 16. The total time of *reducing intents* for synthetic datasets.

[3] Incrementally computing ECs requires clause deletion, which is not efficiently supported by existing approaches. Maintaining derivation information for learned clauses [30, 35] or using assumption variables [14, 20] both incur non-negligible overhead. Therefore, we do not use an incremental setting in the MiniSAT solver.

Answer to RQ5: Our method based on BDD is orders of magnitude faster than that based on SAT for both intent mining and reduction.

6.6 How does *AccessRefinery* accelerate single-round solving in multi-round SMT solving compared to SMT solvers?

Due to the limited scalability of SMT solvers, we applied a minor optimization to reduce the number of solving rounds. In the 5-Key dataset, three of the keys share identical value domains; therefore, we assigned specific values to these domains for the root intent in *Intent Miner*, instead of using wildcards (*). It was not applied to *AccessRefinery*, which already achieves high efficiency without additional optimization.

Table 2. Average Time of Single-Round Boolean Solving (Z3, CVC5, MCP) and Preprocessing Time of MCP on the 5-Key Dataset.

# of allow statements	# of rounds	The time of single-round Boolean solving			The time of MCP preprocessing
		Z3	CVC5	MCP	
3	64	192.1ms	27.9ms	77.5μs	54.5ms
6	196	756.2ms	254.3ms	64.4μs	189.7ms
9	400	1517.2ms	987.0ms	104.1μs	394.3ms
12	676	3122.2ms	4979.3ms	190.2μs	1011.7ms
15	1024	4545.8ms	5543.4ms	274.5μs	1741.5ms

Table 2 shows that for the SMT solver Z3, the average solving time per round ranges from 192.1ms to 4546.8ms, while for CVC5 it ranges from 27.9ms to 5543.4ms. In contrast, *AccessRefinery* employs MCP that extracts theory redundancy in 54.5ms to 1741.5ms. After preprocessing, the average solving time per round is reduced to only 77.5-274.5μs. These results demonstrate that pre-extracting theory redundancy significantly enhances the efficiency of each solving step. Furthermore, Table 2 reveals that the number of solving rounds increases sharply with the number of policy statements, further highlighting the scalability advantage of *AccessRefinery*. Note that although our redundancy extraction is faster than CVC5 in a single round, this does not necessarily mean that we outperform CVC5. The main reason is that real-world regular expressions are generally not very complex. CVC5 may be more efficient when handling very complex string constraints.

Answer to RQ6: Through preprocessing SMT constraints, *AccessRefinery* reduces the time of single-round SMT solving from seconds to milliseconds.

7 Discussion

In this part, we discuss the limitations and scopes of Multi-Theory Constraint Preprocessor (MCP).

Limitations. MCP significantly improves performance over SMT [12] solvers, but it has two main limitations. First, MCP is mainly designed for multi-round SMT solving with predefined values. If a problem only needs to be solved once, MCP may not outperform an SMT solver. Second, the theories supported by MCP are still limited. It only supports a limited set of types, including strings, bit-vectors, ranges, and enumerations. Also, it currently does not support string concatenation, arithmetic, or quantified logic [7]. Extending MCP to support more theories and logics is a valuable direction for future work.

Scopes. Even though MCP currently targets specific theories, we tentatively think that MCP can be applied to domains beyond IAM, including: (1) Access Control Lists (ACLs) [50] that filter or redirect packets based on five tuples (source/destination port, source/destination IP, protocol number); and (2) routing policies [39] that filter or modify routes based on a set of attributes (AS path, community tags, local preference, etc.). For example, MCP supports the modeling of ACLs by representing ports as ranges, IPs as bit-vectors, and protocol numbers as enumerations. Moreover, analyzing ACLs for misconfigurations or redundancies also involves multi-round solving [50], and can potentially benefit from using MCP.

8 Conclusion

We propose *AccessRefinery* to fast mine concise intents from complex cloud *IAM policies*. Specifically, *AccessRefinery* uses *Multi-Theory Constraint Preprocessor* to accelerate multi-round SMT solving and applies *Intent Miner* to mine the raw intents and *Intent Reducer* to reduce intents from the raw intents. We evaluate *AccessRefinery* on both real-world and synthetic datasets against the state-of-the-art approach. Experimental results show that *AccessRefinery* achieves ~10-100× speedup compared to SMT-based methods, while reducing the number of mined intents ~10×.

Data Availability

The replication package is available at https://anonymous.4open.science/r/accessrefinery-fse26/, including the implementation of *AccessRefinery* and the reproduced implementation of *Access Analyzer*, and two synthetic datasets derived from real-world cases. We cannot release the real-world datasets due to customer permissions, as they contain commercially confidential information.

References

[1] Amazon Web Services. 2020. AWS IAM Access Analyzer Samples. https://github.com/aws-samples/aws-iam-access-analyzer-samples. Accessed: 2025-05-22.

[2] Henrik Reif Andersen. 1997. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen* 5 (1997).

[3] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. 2018. 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*. 400–410.

[4] John Backes, Ulises Berrueco, Tyler Bray, Daniel Brim, Byron Cook, Andrew Gacek, Ranjit Jhala, Kasper Luckow, Sean McLaughlin, Madhav Menon, et al. 2020. 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, 165–176.

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

[6] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[7] Mordechai Ben-Ari. 2012. *Mathematical logic for computer science*. Springer Science & Business Media.

[8] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Dan Peebles, Neha Rungta, et al. 2020. Block public access: trust safety verification of access control policies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 281–291.

[9] Vasek Chvatal. 1979. A greedy heuristic for the set-covering problem. *Mathematics of operations research* 4, 3 (1979), 233–235.

[10] Martin Davis and Hilary Putnam. 1960. A computing procedure for quantification theory. *Journal of the ACM (JACM)* 7, 3 (1960), 201–215.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[12] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.

[13] Loris D'Antoni, Andrew Gacek, Amit Goel, Dejan Jovanovic, Rami Gökhan Kıcı, Dan Peebles, Neha Rungta, Yasmine Sharoda, and Chungha Sung. 2024. Projective model counting for IP addresses in access control policies. In *#PLACEHOLDER_PARENT_METADATA_VALUE#*. TU Wien Academic Press, 208–216.

[14] N. Een. 2004. An Extensible SAT-solver. In *Proc Sixth International Conference on Theory and Applications of Satisfiability Testing*.

[15] Niklas Eén and Niklas Sörensson. 2003. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*. Springer, 502–518.

[16] William Eiers, Ganesh Sankaran, and Tevfik Bultan. 2023. Quantitative Policy Repair for Access Control on the Cloud. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 564–575.

[17] William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, and Tevfik Bultan. 2022. Quacky: Quantitative Access Control Permissiveness Analyzer. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[18] William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, and Tevfik Bultan. 2022. Quantifying permissiveness of access control policies. In *Proceedings of the 44th International Conference on Software Engineering*. 1805–1817.

[19] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45. doi:10.1016/j.scico.2007.01.015 Special issue on Experimental Software and Toolkits.

[20] Niklas Eén and Niklas SRensson. 2003. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science* 89, 4 (2003), 543–560.

[21] Kathi Fisler, Shriram Krishnamurthi, Laurie A. Meyer, and Michael F. Reidy. 2005. Verification and Change-Impact Analysis of Access-Control Policies. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. ACM, St. Louis, Missouri, USA, 196–205. doi:10.1145/1062455.1062481

[22] Mark Gabel and Zhendong Su. 2008. Symbolic mining of temporal specifications. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 51–60. doi:10.1145/1368088.1368096

[23] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL (T): Fast decision procedures. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*. Springer, 175–188.

[24] Ralph E Gomory. 2009. Outline of an algorithm for integer solutions to linear programs and an algorithm for the mixed integer problem. In *50 Years of integer programming 1958-2008: From the early years to the State-of-the-Art*. Springer, 77–103.

[25] Antonios Gouglidis, Anna Kagia, and Vincent C Hu. 2023. Model checking access control policies: A case study using google cloud iam. *arXiv preprint arXiv:2303.16688* (2023).

[26] Dorit S Hochbaum. 1982. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal on computing* 11, 3 (1982), 555–556.

[27] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[28] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 735–749.

[29] Richard M Karp. 2009. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*. Springer, 219–241.

[30] Joonyoung Kim, J. Whittemore, and K. Sakallah. 2000. On solving stack-based incremental satisfiability problems. In *IEEE International Conference on Computer Design: Vlsi in Computers and Processors*.

[31] Matija Kuprešanin and Pavle Subotić. 2023. Ambit: Verification of Azure RBAC. In *Proceedings of the 2023 on Cloud Computing Security Workshop*. 31–40.

[32] Charles Eric Leiserson, Ronald L Rivest, Thomas H Cormen, and Clifford Stein. 1994. *Introduction to algorithms*. Vol. 3. MIT press Cambridge, MA, USA.

[33] Zechun Li, Peng Zhang, Yichi Zhang, and Hongkun Yang. 2025. {NDD}: A Decision Diagram for Network Verification. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 237–258.

[34] Xu Liu, Peng Zhang, Hao Li, and Wenbing Sun. 2023. Modular Data Plane Verification for Compositional Networks. *Proceedings of the ACM on Networking* 1, CoNEXT3 (2023), 1–22.

[35] J. P. Marquessilva and K. A. Sakallah. 1999. GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. Comput.* 48, 5 (1999), 506–521.

[36] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Clark Barrett, and Cesare Tinelli. 2022. Even faster conflicts and lazier reductions for string solvers. In *International Conference on Computer Aided Verification*. Springer, 205–226.

[37] Port Swigger. 2021. Data leak Down Under: 50,000 gov't employee records found on open S3 bucket. https://portswigger.net/daily-swig/data-leak-down-under-50-000-govt-employee-records-found-on-open-s3-bucket.

[38] Michael O Rabin and Dana Scott. 1959. Finite automata and their decision problems. *IBM journal of research and development* 3, 2 (1959), 114–125.

[39] Yakov Rekhter, Susan Hares, and Tony Li. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271.

[40] SC Media. 2017. Another misconfigured Amazon S3 server leaks data of 50,000 Australian employees. https://www.scworld.com/news/another-misconfigured-amazon-s3-server-leaks-data-of-50000-australian-employees.

[41] Ilia Shevrin and Oded Margalit. 2023. Detecting Multi-Step IAM attacks in AWS environments via model checking. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6025–6042.

[42] Jiaxin Song, Hongfei Fu, and Charles Zhang. 2024. Equational Bit-Vector Solving via Strong Gr\" obner Bases. *arXiv preprint arXiv:2402.16314* (2024).

[43] Robert Roth Stoll. 1979. *Set theory and logic*. Courier Corporation.

[44] The Verge. 2021. Microsoft Azure Cloud Vulnerability Is the 'Worst You Can Imagine'. https://www.theverge.com/2021/8/27/22644161/microsoft-azure-database-vulnerabilty-chaosdb.

[45] Grigori S Tseitin. 1983. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970* (1983), 466–483.

[46] UpGuard. 2017. Cloud Leak: WSJ Parent Company Dow Jones Exposed Customer Data. https://www.upguard.com/breaches/cloud-leak-dow-jones.

[47] John Whaley. [n. d.]. JavaBDD: A Java library for Binary Decision Diagrams. https://javabdd.sourceforge.net/. Accessed: 2025-05-27.

[48] Laurence A Wolsey and George L Nemhauser. 1999. *Integer and combinatorial optimization*. John Wiley & Sons.

[49] Hongkun Yang and Simon S Lam. 2015. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking* 24, 2 (2015), 887–900.

[50] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, and Prasant Mohapatra. 2006. FIREMAN: a toolkit for firewall modeling and analysis. *IEEE* (2006).

[51] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. {APKeep}: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 241–255.