Projective Model Counting for IP Addresses in Access Control Policies

Loris D'Antoni, Andrew Gacek, Amit Goel, Dejan Jovanović,
Rami Gökhan Kıcı, Dan Peebles, Neha Rungta, Yasmine Sharoda, Chungha Sung
Amazon Web Services
Seattle, USA
{lorisd,gacek,amgoel,dejajov,ramikici,dgp,rungta,sharoday,chunghs}@amazon.com

***Abstract*—Zelkova is an AWS service that answers questions about Identity and Access Management (IAM) access policies such as "Does this policy allow *public* access?". Zelkova formalizes IAM policies and the meaning of "public" as a logical query that can be solved using SMT solvers. Among other conditions, Zelkova defines a policy as *public* if it allows access from a number of IP addresses that exceeds a given threshold. Encoding this check so that it is supported by all SMT solvers in the Zelkova portfolio is difficult because counting and restricting the number of models are not core SMT features. We describe two SMT encodings for checking whether the number of IPs allowed by a policy exceeds a given bound. Both encodings generate an SMT formula that can be discharged with a single call to an off-the-shelf SMT solver. Our approach takes less than 3s to detect whether a policy is public for 99.999% of the evaluated policies.**

I. Introduction

Millions of customers use AWS to store their data in a variety of resources such as databases and key-value stores. These resources are secure-by-default and accessible only when the customer grants access. The customer can grant access by authoring policies in the Identity and Access Management (IAM) language. The IAM language can express properties varying from simple sharing to complex constraints with a logical combination of positive and negative operators.

AWS offers tools to help customers write and understand their policies. One of these tools is Block Public Access (BPA) [9] which protects customers from accidentally attaching "public" policies to their resources.

The central design decision in BPA is the exact definition of "public", and three factors are at play here. First, the definition must match a customer's intuitions about public access. Second, the definition must be mathematically precise so it can be checked in a provable way. Specifically, a precise mathematical definition of public access allows us to check whether a policy is public using Zelkova [6], an IAM policy analysis service based on SMT solvers. Third, the definition must require no additional information from the customer so that BPA itself can be a one-click solution.

The key idea underlying the AWS definition of public is to examine a customer policy and extract out the trusted entities—e.g. individual account IDs, networks, or users. In general, a policy should only reference a limited number of trusted entities. If there is any access granted outside this small set of trusted entities, e.g., due to misuse of wildcards—e.g., a policy that allows access from any account ID—the policy is considered to be granting public access. However, when reasoning about IP addresses, treating each IP as a separate trusted entity may make a policy look public when it really is not. A single customer may own a large collection of IP addresses, all of which are grouped together and considered trusted—e.g., the company may own the `19.0.0/8` range of IP addresses. This Classless Inter-Domain Routing (CIDR) notation represents the set of $2^{24} \approx 1.7 \times 10^6$ IP addresses between `19.0.0.0` and `19.255.255.255` inclusively. A single customer policy could reference hundreds of similarly sized CIDR blocks, which in total amounts to granting access to the entire internet. Such a policy should be considered public, so we need to update our definition of BPA to handle IP addresses. For the domain of IP addresses we have decided to draw a line at a specific number of IP addresses that can be allowed before a policy is considered public. From conversations with customers, we identified that any number of IP addresses larger than a single `/8` CIDR block—i.e., 2^{24} IP addresses—should be considered public.

To check for public access, Zelkova turns an IAM policy into a logical query that is discharged using a portfolio of SMT solvers. The original version of the BPA check [9] removes all trusted parts of the policy and then compiles the remaining parts into a single logical formula. If an SMT solver finds a model for the generated formula, the policy allows untrusted access and is marked public. To support IP addresses as trusted entities, we must precisely count how many IP addresses are allowed by the policy after the other trusted parts are removed.

Today, there are no SMT solvers that support precise model counting and that can solve this problem within a few seconds [19]. Therefore, we encode this **bounded projective IP-counting problem** into a single SMT query using arithmetic. The key idea of our encoding is to split the set of IP addresses into equivalence classes for which counting is trivial and then reduce the bounded projective IP-counting problem to an SMT query that checks if the sum of the IP counts in the "allowed" equivalence classes exceeds the given bound. We also present an encoding that eliminates the need for arithmetic by precomputing minimal sets of summands which will exceed the bound. These two encodings allow us to solve the bounded projective IP-counting problem using existing SMT solvers

https://doi.org/

with the time constraints imposed by AWS customer needs (i.e., within 3 seconds per check). We remark that both encodings use a *single* SMT query and do not rely on features such as incremental solving of multiple SMT queries (e.g., one per equivalence class), which may result in expensive enumeration and are not necessarily supported by all SMT solvers.

Contributions: This paper makes three contributions:

We formalize the bounded projective IP-counting problem (Section II) and illustrate how Block Public Access needs to solve this problem (Section III).

We design two sound and efficient SMT encodings for solving the bounded projective IP-counting problem (Section IV). The encodings compute equivalence classes of IP addresses and separately compute the sizes of each equivalence class, thus bypassing the need to reason about individual IP addresses and the need for model counting. The first encoding, which is supported by all but one of the SMT solvers in the Zelkova portfolio, requires support for arithmetic operations (e.g., summing) on top of the theories already required by Zelkova. The second uses a knapsack-based approach to identify combinations of equivalence classes that can exceed the threshold and avoids arithmetic operations, thus imposing no additional requirements on the SMT solver. This last encoding is supported by all SMT solvers in the Zelkova portfolio.

We evaluate the encodings on 700,000 policies containing IP addresses; our approaches take less than 3s (the time limit required by the target application) to detect whether a policy is public for 99.999% of the evaluated policies (Section V).

II. The Bounded Projective IP-counting Problem

In this section, we first describe the AWS policy language and its semantics, and then define the problem of checking whether the number of IP addresses for which at least one request is allowed by a given policy exceeds a given bound.

The AWS policy language is defined as serialized JSON [1]. In this paper, we describe a simplified abstract syntax of the core constructs of the language to simplify our exposition. As done in prior work [9], we model an IAM policy as a set of statements that can either allow or deny a set of requests. A request is granted when it is allowed by at least one statement and not denied by any statement. In the rest of the section, we formalize these concepts.

Requests. We assume a set of variables V, which represent the possible fields in a request—e.g., `principal`, `action`, `resource`, and `sourceIP` are variables.

A request $r : V \to \texttt{val}$ is a function that maps a variable to its value (the value can be `null`). For example, the partial snippet of a request r_1 shown in Figure 1 maps the variable `principal` to the value `111122223333:user/Bob`, the variable `action` to the value `s3:ListBucket`, the variable `resource` to the value `bucket/invoices`, and the variable `sourceIP` to the value `20.121.201.3`. Each variable $v \in V$ is associated with a value of a specific type in IAM and we use $\tau(v)$ to denote it. Common value types are booleans, strings, and IP addresses. Less common types are integers and floats. Every request contains the variables `principal`, `action`, and `resource`, whereas others are optional. Requests might not contain values for all the variables, so we allow r to map variables to the special value `null`.

```
r1:(principal: 111122223333:user/Bob,
    action   : s3:ListBucket,
    resource : bucket/invoices,
    sourceIP : 20.121.201.3,
    username : Bob, ...)
```

Fig. 1: A request allowed by statement s_3 in Figure 2.

Statements. A statement s is a pair (e, Ψ) where e is either the value `allow` (we call these statements *allow-statements*) or the value `deny` (we call these statements *deny-statements*), and $\Psi : V \mapsto \texttt{pred}$ is a partial function that maps variables to predicates. For example, in the statement $s_3 = (\texttt{allow}, \Psi_3)$ in Figure 2, Ψ_3 maps the variable `action` to the predicate stating that the action of a request should start with `s3:` (i.e., the predicate is represented by the pattern `s3:`$\star$). Common predicate types are simplified regular expressions to restrict values of strings, boolean comparisons, and Classless Inter-Domain Routing (CIDR) [3] descriptions of sets of IP addresses. For example, the IP range `20.0.0.0/7` allows the $2^{32-7} = 33,554,432$ IPv4 addresses in the range `20.0.0.0` to `21.255.255.255`[1], which also includes the IP address `20.121.201.3` from the request r_1 in Figure 1.

We use $V(s)$ to denote the set of variables in the domain of Ψ—i.e., all the values for which the partial function is defined. Every statement always maps the variables `principal`, `action`, and `resource` to a predicate.

Intuitively, a statement matches a request if, for every variable appearing in the statement, the request's values are models of the corresponding predicates in the statement.

Definition (Statement-Matching Requests): *Given a request r and a statement $s = (e, \Psi)$ we say that s matches the request r if and only if, for every $v \in V(s)$, the value $r(v)$ is a model of the predicate $\Psi[v]$. We write $M(s)$ to denote the set of all requests matched by s—i.e., $M(s) = \{r \mid \bigwedge_{v \in V(s)} \Psi[v](r(v))\}$.*

The statement $s_3 = (\texttt{allow}, \Psi_3)$ in Figure 2 has five keys `principal`, `action`, `resource`, `sourceIP`, and `username` and matches the request r_1 in Figure 1.

We simplify the syntax of IAM policies and assume that each key is associated with its predicate. Our implementation maps the JSON representation of statements to this predicate format; this translation is straightforward and syntax-directed and we do not present it formally here.

Policies. A policy $P = \{s_1, s_2, ..., s_n\}$ is a set of statements and we use $AS(P)$ (resp. $DS(P)$) to denote all the allow (resp. deny) statements in P. We write $P = (AS(P), DS(P))$

[1]We note that some of the IP addresses in this range are not usable, e.g., `20.0.0.0` and `21.255.255.255`, but in this paper we assume the size of a CIDR also considers unusable IP addresses.

```
s1:(allow,  (principal: *,
             action    : s3:*,
             resource  : bucket/*,
             sourceIP  : 201.0.0.0/7,
             account   : 444455556666))
- - - - - - - - - - - - - - - - - - - - - - -
s2:(allow,  (principal: *,
             action    : s3:*,
             resource  : bucket/*,
             sourceIP  : 14.0.0.0/7,
             username  : Alice))
- - - - - - - - - - - - - - - - - - - - - - -
s3:(allow,  (principal: *,
             action    : s3:*,
             resource  : bucket/*,
             sourceIP  : 20.0.0.0/7,
             username  : Bob))
- - - - - - - - - - - - - - - - - - - - - - -
s4:(deny,   (principal: *,
             action    : s3:*,
             resource  : bucket/*,
             sourceIP  : 14.0.0.0/8))
```

Fig. 2: An IAM policy with three allow statements s_1, s_2, s_3 and one deny statement s_4. The predicates in green describe IP ranges, and the predicates `14.0.0.0/7` and `14.0.0.0/8` describe overlapping sets of IP addresses.

to directly denote these two sets and a and d instead of s to denote an allow or deny statement respectively.

Definition (Granted Requests): *A policy P grants a request r if and only if there exists an allow statement that matches the request, i.e., $\exists a \in AS(P).\ r \in M(a)$, and there does not exist a deny statement that matches the request, i.e., $\forall d \in DS(P).\ r \notin M(d)$. We write $Granted(P)$ to denote the set of all requests granted by the policy P, which can be defined as follows.*

$$Granted(P) = \Big(\bigcup_{a \in AS(P)} M(a)\Big) \setminus \Big(\bigcup_{d \in DS(P)} M(d)\Big)$$

The policy depicted in Figure 2 has three allow statements and one deny statement and allows the request r_1 shown in Figure 1, which only matches statement s_3.

Problem Definition. Given a policy P we define the set of IPs for which at least one request is granted by P as the set $IPSet(P) = \{r(\texttt{sourceIP}) \mid r \in Granted(P)\}$. We are now ready to define the problem solved in this paper.

Definition (Bounded Projective IP-Counting Problem): *The bounded projective IP-counting problem is to determine if the number of IP addresses from which at least one request is allowed by a policy P exceeds a given IP-count threshold τ, which formally can be stated as $|IPSet(P)| > \tau$.*

III. Illustrative Example

We illustrate our approach for solving the IP-count bounding problem using the example policy presented in Figure 2. This policy is for an Simple Storage Service (S3) bucket [2], an object storage service, and it consists of four statements s_1, s_2, s_3 and s_4.

The first statement s_1 grants access for anyone from account `444455556666` to perform S3 actions on any S3 object in the bucket `bucket`. The second statement s_2 grants access for a user named `Alice` to perform S3 actions on any S3 object in the bucket `bucket`. Similarly, the third statement s_3 grants access for a user named `Bob` to perform S3 actions on the same objects. Because users work in different companies, the statements allow requests from different ranges of IP addresses for each user (denoted in green).

The fourth statement s_4 is a deny-statement that removes access for any requests coming from an IP in the range `14.0.0.0/8`. This IP range is a subset of the IP range `14.0.0.0/7` allowed by statement s_2.

The question we are interested in answering is whether the policy in Figure 2 allows *public* access to the S3 bucket. As discussed in Section I, some AWS customers want to consider a resource to be publicly accessible if the number of IP addresses from which one can issue an allowed request exceeds a given threshold. However, not all IPs should contribute to the total count. In our example, the statement s_1 only allows requests from a specific account ID. Because account IDs are assigned by AWS (unlike usernames), this statement is already associated with what in Section I we called a trusted entity and it is irrelevant how many IP addresses it allows access from. The existing work on AWS public access [9] can detect such trusted entities and remove this statement from the policy before we need to reason about IP addresses.

Once statement s_1 has been removed, we are ready to count how many IP addresses the other statements allow requests from. In this section, we assume that the threshold is $\tau = 2^{24} = 16,777,216$ IP addresses—i.e., the size of a single `/8` CIDR block, which is the largest block size owned by a single entity. In general, the threshold can be set to any value.

Checking whether the number of IP addresses exceeds the threshold requires *counting* how many IP addresses one can issue an allowed request from, a problem that on the surface requires going beyond the capability of SMT solvers, the current tool of choice for reasoning about public access in IAM policies [9], [6]. The encodings proposed in this paper provide a way of checking if the number of allowed IP addresses exceeds the threshold τ using traditional SMT solvers (i.e., without counting models).

We discuss what are the key insights of the encoding.

IP Equivalence Classes: The SMT encoding used by Zelkova to describe what requests each statement allows (or denies) is a conjunction of monadic predicates[2] where each predicate describes what values a request can contain for each specific variable, and particularly for source IPs. For example, the statement s_2 is translated by Zelkova into the following SMT formula φ_{s_2} involving the theory of Strings (e.g., $L(R)$ denotes the language of a regular expression R),

[2]We use the term monadic for predicates that involve one variable—e.g., the predicate $x > 0$ is monadic, whereas the predicate $x > y$ is polyadic.

and bit-vectors (e.g., $in\_ip\_range(\texttt{sourceIP}, \texttt{I})$ denotes a predicate for checking if the value of variable sourceIP is in the set of bit-vectors encoding IPs belonging to the CIDR block I):

$$\begin{gathered}\texttt{principal} \in L(*) \wedge \texttt{action} \in L(\texttt{s3:*}) \wedge \\ \texttt{resource} \in L(\texttt{bucket/*}) \wedge \\ in\_ip\_range(\texttt{sourceIP}, \texttt{20.0.0.0/7}) \wedge \\ \texttt{username} = \text{“}\texttt{Alice}\text{”}\end{gathered} \tag{1}$$

The final result describing whether a request is allowed by the policy is expressed by the formula $\varphi_P(\bar{x}, \texttt{sourceIP}) = (\varphi_{s_1} \vee \varphi_{s_2} \vee \varphi_{s_3}) \wedge \neg\varphi_{s_4}$, that is, a request is allowed by the policy if it is allowed by an allow statement and not denied by any deny statement. The notation $\varphi_P(\bar{x}, \texttt{sourceIP})$ separates the variable sourceIP from all other free variables—i.e., $\bar{x}$. Our goal is to check whether $\#SAT(\exists\bar{x}.\ \varphi_P(\bar{x}, \texttt{sourceIP})) > 2^{24}$. Because the source IP predicates are all monadic (i.e., they do not interact with other variables other than sourceIP), the predicate $\varphi_P(\bar{x}, \texttt{sourceIP})$ can be expressed as a Boolean combination of predicates of the following form (where i denotes the ith statement):

$$\psi_i^1(\bar{x}) \wedge \psi_i^2(\texttt{sourceIP})$$

Therefore, our first key idea is that we can take all satisfiable Boolean combinations of the predicates $\psi_i^2(\texttt{sourceIP})$—i.e., all the predicates of the form $in\_ip\_range(\texttt{sourceIP}, \texttt{range})$—to obtain IP predicates describing equivalence classes of IP addresses—i.e., if a request is allowed (resp. denied) with an IP address, replacing that address with another one in the same class will still make the request allowed (resp. denied).

We show in Section IV-B how to compute equivalence, but in our example, after removing s_1, we have 3 equivalence classes (other combinations are unsatisfiable and we simplify each predicate $in\_ip\_range(\texttt{sourceIP}, \texttt{range})$ as simply its range range):

- $e_1 = \texttt{14.0.0.0/8}$ is the result of the Boolean combination $\texttt{14.0.0.0/7} \wedge \neg\texttt{20.0.0.0/7} \wedge \texttt{14.0.0.0/8}$;
- $e_2 = \texttt{20.0.0.0/7}$ is the result of the Boolean combination $\neg\texttt{14.0.0.0/7} \wedge \texttt{20.0.0.0/7} \wedge \neg\texttt{14.0.0.0/8}$;
- $e_3 = \texttt{15.0.0.0/8}$ is the result of the Boolean combination $\texttt{14.0.0.0/7} \wedge \neg\texttt{20.0.0.0/7} \wedge \neg\texttt{14.0.0.0/8}$.

In our example, each predicate is defined using a single CIDR block, but in general, a predicate can be described as a union of CIDR blocks; we support this more general form in our approach presented in Section IV.

Counting IPs without Model Counting: Once we have computed equivalence classes, it is trivial to compute how many IP addresses each class contains using the definition of a CIDR block.

- $|e_1| = |\texttt{14.0.0.0/8}| = 2^{24} = 16,777,216$;
- $|e_2| = |\texttt{20.0.0.0/7}| = 2^{25} = 33,554,432$; and
- $|e_3| = |\texttt{15.0.0.0/8}| = 2^{24} = 16,777,216$.

With this information available, we can now write an SMT formula that checks whether the sum of allowed IPs exceeds the threshold. (We write $\varphi_P(\bar{x}, \texttt{ip})$ to denote the result of substituting the variable sourceIP with the constant ip in the formula φ_P).

$$\begin{gathered}(\text{if } \exists\bar{x}.\ \varphi_P(\bar{x}, \texttt{14.0.0.0}) \text{ then } 2^{24} \text{ else } 0) + \\ (\text{if } \exists\bar{x}.\ \varphi_P(\bar{x}, \texttt{20.0.0.0}) \text{ then } 2^{25} \text{ else } 0) + \\ (\text{if } \exists\bar{x}.\ \varphi_P(\bar{x}, \texttt{15.0.0.0}) \text{ then } 2^{24} \text{ else } 0) > 2^{24}\end{gathered} \tag{2}$$

Intuitively, each row uses the formula φ_P to check if the policy allows a representative IP address from each equivalence class, in which case it contributes the size of that class to the counter. In this case, the policy is public because it allows access from $2^{24} + 2^{25} > 2^{24}$ IP addresses.

The constraint in (2) requires arithmetic to describe whether the number of IP addresses exceeds the threshold. Because some SMT solvers do not support both the theories of strings and arithmetic at the same time [21], in Section IV we also introduce a version of the encoding that pre-computes what combinations of equivalence classes can exceed that given threshold and generates a formula that does not involve arithmetic. In our example, the minimal combinations of equivalence classes that exceed the threshold are $|e_1| + |e_3|$ and $|e_2|$, therefore one can write an SMT formula that does not use arithmetic and that checks whether the sum of allowed IPs exceeds the threshold as follows:

$$\begin{gathered}(\varphi_P(\bar{x}, \texttt{14.0.0.0}) \wedge \varphi_P(\bar{x}, \texttt{15.0.0.0})) \vee \\ \varphi_P(\bar{x}, \texttt{20.0.0.0})\end{gathered} \tag{3}$$

In this case, the formula is satisfied by making the second disjunct true, thus denoting that the 2^{25} IP addresses allowed by the equivalence class e_2 exceed the threshold 2^{24}.

IV. Counting IPs without Model Counting

Before presenting our technique for solving the bounded projective IP-counting problem presented in Section II, we recall that our goal is to devise an SMT-based approach for solving the problem that does not rely on model counting.

At the high-level, given a policy P and a threshold τ our main approach proceeds in two steps:

1) We compute a set of equivalence classes of IP addresses such that all the IP addresses appearing in the same equivalence class e are treated the same way by the policy P (Section IV-B)—i.e., if a request r with an IP address in e is granted (resp. not granted) by the policy P, the request obtained by replacing the IP address with any member of the equivalence class e is still granted (resp. not granted).
2) Once the equivalence classes are computed, we can separately compute the size of each equivalence class (i.e., the number of IP addresses in it), and rewrite the SMT formula encoding the policy semantics to remove any mention of IP addresses and instead directly reason about the size of each equivalence class (Section IV-C).

Step 2 in the algorithm above requires arithmetic operations, and for some solvers, specifically NFA2SAT [21], this theory is not supported in combination with the many theories (e.g., strings) required to model IAM policies. To address this

limitation, we introduce a new encoding that avoids arithmetic and instead uses a knapsack-based approach to compute a new formula that identifies combinations of equivalence classes that can lead to exceeding the threshold. The formula is entirely expressible in propositional logic (Section IV-D).

Before presenting our approaches, we distill the essence of the problem solved by our approach to a purely logical formalization that is agnostic from the specific problem of counting IPs (Section IV-A).

A. Bounded Projective Counting Problem

Prior work on verifying policies in the IAM language [9], [6] has shown how, given a policy P, one can create a formula $\varphi_P(x_1, \ldots, x_n)$ that is satisfied *exactly* by all the granted requests in the set $Granted(P)$. Specifically, each variable x_i corresponds to a variable in the set V, and a satisfying assignment $c_1, \ldots, c_n$ corresponds to the request mapping each variable to the corresponding value—i.e., $[x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]$.

Thanks to the above formalization if we consider `sourceIP` to denote the variable denoting a source IP address in a request, by appropriately massaging the formula φ, we can express the bounded projective IP-counting problem defined in Section II as a formula of the following form

$$\#SAT(\exists \bar{x}.\ \varphi_P(\bar{x}, \texttt{sourceIP})) > \tau \tag{4}$$

Here $\#SAT(\cdot)$ is the function denoting the number of satisfying assignments to a formula. Because `sourceIP` is the only non-quantified (i.e., free) variable, the set of satisfying assignments to the formula $\exists \bar{x}.\ \varphi_P(\bar{x}, \texttt{sourceIP})$ corresponds exactly to the set $IPSet(P)$. Thus, Equation (4) correctly captures the bounded projective IP-counting problem.

With this observation, we can focus the rest of the section on the following generalized version of the counting problem.

Definition (Bounded Projective Counting Problem): *We say that a formula $\varphi(x, y)$ exceeds a y-count threshold τ if the following is true:*

$$\#SAT(\exists x.\ \varphi(x, y)) > \tau \tag{5}$$

In the rest of the section, we show how one can avoid solving the hard $\#SAT(\cdot)$ problem over the quantified formula $\exists x.\ \varphi(x, y)$ by instead solving an easier satisfiability problem over formulas involving only y.

B. Computing Equivalence Classes

Given a formula of the form $\exists x.\ \varphi(x, y)$, the first step of our algorithm is to compute equivalence classes for the variable y for the following equivalence relation, which captures that two values of y are equivalent if they behave the same for every possible value of x. Because computing maximal equivalence classes is in general unnecessary and in fact something we want to avoid (as we will see later), we instead define valid partitions of the domain into equivalent elements.

Definition (y-equivalence, y-partition): *Given a formula $\varphi(x, y)$ say that two constants c_1 and c_2 are y-equivalent iff*

$$\forall x.\ \varphi(x, c_1) \iff \varphi(x, c_2).$$

We say a partition $\Pi = \{e_1, \ldots, e_j\}$ forms a y-partition of the domain $Dom(y)$ with respect to y-equivalence iff (i) Π is a valid partition of $Dom(y)$ (i.e., the union of all e_i is $Dom(y)$, and all elements of Π are disjoint), and (ii) for every class $e_i \in \Pi$, all elements of e_i are y-equivalent.

If the variable y only appears within monadic predicates in the formula $\varphi(x, y)$ (which is the case for the problem of IP-count bounding), we can always compute a y-partition of $Dom(y)$ by computing the set of minimal satisfiable Boolean combinations of all the monadic predicates over y, also called minterms [16].

For example, if the only predicates involving y in the formula $\varphi(x, y)$ are the monadic predicates $\psi_1(y)$ and $\psi_2(y)$, a valid y-partition can be computed as the set

$$\Pi = \{\psi_1 \wedge \psi_2, \neg\psi_1 \wedge \psi_2, \psi_1 \wedge \neg\psi_2, \neg\psi_1 \wedge \neg\psi_2\}$$

If any of the predicates in Π is unsatisfiable, they can be discarded before continuing to the next steps. In the worst case, the y-partition can contain exponentially many classes in the size of the formula $\varphi(x, y)$, but in practice this is rarely the case.

The appealing aspect of computing y-partitions in the aforementioned way is that one *does not* need to reason about satisfiability of the whole formula $\varphi(x, y)$ and instead only needs to check satisfiability of Boolean combinations of predicates involving y, which in our application domain, counting IPs, is a very friendly theory to work with, as we illustrate next.

`sourceIP`-equivalence: For IP addresses, each monadic predicate appearing in an IAM statement is a union of CIDR blocks $c_1 \cup \ldots \cup c_n$ (in our running example, each union only contain one CIDR block). We note that two CIDR blocks can be disjoint, or one can be a subset of the other; other logical relations are not possible—e.g., partial overlap. We can therefore assume that $c_1 \cup \ldots \cup c_n$ contains all disjoint CIDR blocks (ones that are subsets of others can be removed).

After this pre-processing, by collecting positive and negative terms, any satisfiable Boolean combination of unions of CIDR blocks can be written in the following form:

$$(\psi_1 \cap \ldots \cap \psi_j) \setminus (\psi_{j+1} \cup \ldots \cup \psi_k).$$

The right-hand side of the $\setminus$ is itself a union of CIDR blocks.

We next show that the left-hand side can also be rewritten as a union of CIDR blocks and that the $\setminus$ of two unions of CIDR blocks can also be translated to a union of CIDR blocks.

Given two unions of CIDR blocks $C_1 \cup \ldots \cup C_n$ and $D_1 \cup \ldots \cup D_m$, their *intersection* can be defined as the union of CIDR blocks $\cup_{i \le n, j \le m} C_i \cap D_j$, where the intersection of two CIDR blocks is defined as:

1) $C \cap D = \emptyset$ if C is disjoint from D;
2) `IP1/M1` $\cap$ `IP2/M2` $=$ `IP2/M2` $\cap$ `IP1/M1` $=$ `IP1/M1` if `IP1/M1` is a subset of `IP2/M2`—i.e., `M1` $\geq$ `M2` and the first `M2` bits of `IP1` and `IP2` are the same.

Given two unions of CIDR blocks $C_1 \cup \ldots \cup C_n$ and $D_1 \cup \ldots \cup D_m$, their *difference* can be defined as $\cup_{i \le n}(\cap_{j \le m} C_i \setminus D_j)$ where

1) $C \setminus D = \emptyset$ if $C \subseteq D$;
2) $C \setminus D = C$ if $C \cap D = \emptyset$;
3) if IP1/M1 $\supset$ IP2/M2—i.e., M1 $<$ M2 and the first M1 bits of IP1 and IP2 are the same—we recursively split the CIDR of IP1/M1 into two longer CIDR blocks (the one obtained by choosing the (M1+1)-th bit to be 0 or 1, respectively) and recursively subtract IP2/M2 from them—i.e., IP1/M1 $\cap$ IP2/M2 = (IP1[1..M1]0/M1+1 $\setminus$ IP2/M2) $\cup$ (IP1[1..M1]1/M1+1 $\setminus$ IP2/M2).

Because of the 0-1 splitting in case 3, the above algorithm guarantees that CIDR blocks appearing in the final union are all disjoint.

Example (From Section III): *In the example in Section III, the class* e_3 = 15.0.0.0/8 *is the result of* 14.0.0.0/7 $\setminus$ (20.0.0.0/7 $\cup$ 14.0.0.0/8). *First, we rewrite the formula as* (14.0.0.0/7 $\setminus$ 20.0.0.0/7) $\cap$ (14.0.0.0/7 $\setminus$ 14.0.0.0/8)) *following the definition of* $\setminus$ *on unions of CIDR blocks. The first conjunct is* 14.0.0.0/7 *following case 2 of the definition of* $\setminus$ *on CIDR blocks, whereas the second conjunct is rewritten as* (14.0.0.0/8 $\setminus$ 14.0.0.0/8) $\cup$ (15.0.0.0/8 $\setminus$ 14.0.0.0/8)) *following case 3. Now, the first disjunct rewrites to the empty set (case 1), and the second disjunct rewrites to* 15.0.0.0/8 *(case 2). Finally,* e_3 = 14.0.0.0/7 $\cap$ 15.0.0.0/8 = 15.0.0.0/8.

C. Arithmetic Approach

The arithmetic approach formalizes the IP counting problem as a summation problem. Recall that our goal is to asses whether the formula in Equation (5) is true. One way to encode this problem as an SMT formula involving arithmetic operations for counting is to rewrite the formula as follows:

$$(\Sigma_{c \in Dom(y)} \text{ if } \exists x.\ \varphi(x, c) \text{ then } 1 \text{ else } 0) > \tau \qquad (6)$$

By skolemizing the existentially quantified variable x, we can simplify the formula as follows:

$$(\Sigma_{c \in Dom(y)} \text{ if } \varphi(x_c, c) \text{ then } 1 \text{ else } 0) > \tau \qquad (7)$$

The encoding in Equation (6) is expressible as an SMT formula whenever $Dom(y)$, the domain of y, is finite. However, if the domain of y is large (which is the case when y represents IP addresses) solving Equation (6) will either require a very large SMT formula or iterating over many possible smaller formulas (one per IP address).

We call this approach the Arithmetic Approach (AA). Our Arithmetic Approach sidesteps this problem thanks to the previously computed equivalence classes, which we call $EC_y(\varphi(x, y)))$. For every equivalence class e in the set $EC_y(\varphi(x, y))$, we use the symbol rep_e to denote a representative value of y from that class. Equation (7) can then be optimized as the following formula:

$$(\Sigma_{e \in EC_y(\varphi(x,y))} \text{ if } \varphi(x_e, rep_e) \text{ then } |e| \text{ else } 0) > \tau \qquad (8)$$

If we consider the example formula in Equation (1), and the equivalence class e_1 = 14.0.0.0/8 the formula $\varphi(x_{e_1}, 14.0.0.0)$ can be obtained by replacing the variable x with x_{e_1} and the variable sourceIP with the concrete IP 14.0.0.0. The encoding in Equation (8) is expressible as an SMT formula whenever the size $|e|$ of an equivalence class e is computable.

Because our y-partition algorithm computes equivalence class that are expressed as monadic predicates $\psi(y)$, all one needs to generate the formula in Equation (8) is a technique for counting the number of models for the theory of y, a trivial problem for predicates involving IPs.

Theorem (Soundness of Arithmetic Approach): *A formula* $\varphi(x, y)$ *exceeds a* y*-count threshold* τ *iff Equation* (8) *holds.*

Proof. We know that for any two elements c_1, c_2 in the same y-equivalence class $e \in EC_y(\varphi(x, y))$, the following holds $\forall x.\ \varphi(x, c_1) \iff \varphi(x, c_2)$. Thus, $\exists x.\ \varphi(x, rep_e)$ holds iff $\exists x.\ \varphi(x, c)$ holds for every $c \in e$. Therefore, the formula if $\exists x.\ \varphi(x, rep_e)$ then $|e|$ else 0 correctly computes the size of the equivalence class e. □

Note that if the formula $\varphi(x, y)$ lies in a theory $\mathcal{T}$, the constraints in Equation (8) are in the theory $\mathcal{T}$ + QFLIA.

Counting IPs: In particular, when y represents IP addresses, $|e|$ can be computed efficiently. If e is represented by a set of disjoint CIDR blocks—which the Boolean operations defined in Section IV-B guarantee—then $|e|$ is the sum of the size of each CIDR block. In particular, for IPv4, the size of a CIDR IP/M is 2^{32-M}—e.g., $|14.0.0.0/8| = 2^{24}$.

D. Arithmetic-free Approach

The arithmetic approach discussed in Section IV-C requires adding an arithmetic theory on top of the theory $\mathcal{T}$ needed to reason about the formula $\varphi(x, y)$. In some cases, an SMT solver might support the theory $\mathcal{T}$, but not the combined theory, e.g., $\mathcal{T}$ + QFLIA. For example, the NFA2SAT solver [21] is a powerful solver used by Zelkova [6] to prove properties of policies, and relies on SAT solving to reason about strings and does not support arithmetic. Since industrial applications rely on portfolio solving to provide performance and robustness, an ideal solution to the bounded projective IP-counting problem should work with all possible available solvers.

In this section, we describe an approach for solving the y-count bounding problem entirely within the theory $\mathcal{T}$—i.e., without the need for an arithmetic theory for counting. We call this approach the Arithmetic-free Approach (AFA).

At a high level, the AFA proceeds in the following steps:

1) First, it uses a dynamic programming algorithm (a variant of knapsack) to compute *all minimal combinations of equivalence classes* $\mathcal{C}$ that can cause the threshold τ to be exceeded.
2) Then, it creates a new constraint $\psi_{\mathcal{C}}$ that is satisfied exactly by combinations of equivalence classes that exceed the threshold τ.

Computing Minimal Possible Violations: We have shown in Section IV-C how we can compute the size $|e|$ of every equivalence class e. First, we define the combinations of equivalence classes that are minimal possible violations of the given threshold. Given a set A of equivalence classes, we write $Weight(A)$ to denote the sum of the sizes of the equivalence classes in A—i.e., $Weight(A) = \sum_{e \in A} |e|$.

Definition (Minimal Possible Violation): *Given a set of equivalence classes EC, we say that a subset $A \subseteq EC$ forms a* possible violation *of the threshold τ iff the sum of the sizes of each class exceeds the threshold—i.e., $Weight(A) > \tau$.*

Furthermore, A is a minimal possible violation *iff no strict subset of A is a possible violation—i.e., $\neg \exists e \in A.\ Weight(A) \geq Weight(A \setminus \{e\}) > \tau$.*

In the worst case, if we have n equivalence classes, it is possible to have $2^{O(n)}$ minimal possible violations. For example, if we have a threshold of $n/2$ and each class has weight 1, there are approximately $\binom{n}{n/2}$ minimal possible violations—i.e., all ways to pick $n/2$ classes from the set. In practice, the number of minimal possible violations is often much smaller as illustrated by the following example.

Example (Minimal IP Violations): *We discussed in Section III how the example in Figure 2 leads to three equivalence classes, `14.0.0.0/8`, `20.0.0.0/7`, and `15.0.0.0/8`.*

The minimal possible violations are obtained by the sets $\{$`14.0.0.0/8`, `15.0.0.0/8`$\}$ and $\{$`20.0.0.0/7`$\}$.

The problem of computing the set of all minimal possible violations can be solved using a variant of the knapsack dynamic programming algorithm. Intuitively, starting from an empty set, one can build incrementally larger subsets, by adding additional equivalence classes as long as the unused equivalence classes can still be used to cross the threshold. By considering the equivalence classes ordered by their size, this process ensures that we can stop as soon as we cross the threshold, resulting in a minimal possible violation.

Minimal Satisfiable Violations: We now assume we have computed the set $MPV = \{A_1, \ldots, A_m\}$ of all minimal possible violations. The last step is to find one that is an actual satisfiable violation—i.e., a set A_i such that each class $e \in A$ makes the formula $\exists x.\ \varphi(x, rep_e)$ true.

To encode this problem as a constraint, we introduce for each class e, a new variable v_e to model whether the class e corresponds to a positive class (i.e., one that makes the formula $\exists x.\ \varphi(x, rep_e)$ true) or a negative class (i.e., one that makes the formula $\exists x.\ \varphi(x, rep_e)$ false). After replacing the existentially quantified variable x with x_e, we get the constraint:

$$v_e \Leftrightarrow \varphi(x_e, rep_e) \tag{9}$$

The y-count bounding problem can then be solved by checking satisfiability of the following formula, which simply looks for a minimal possible violation $A \in MPV$ consisting only of positive classes.

$$\bigvee_{A \in MPV} \bigwedge_{e \in A} v_e \tag{10}$$

For our example in Section III, we obtain Equation (3).

	Arithmetic		Arithmetic-free	
	# fastest	% fastest	# fastest	% fastest
cvc4	2,012	0.3%	403	0.1%
cvc5	196,431	28.0%	92,412	13.2%
trivial	500,927	71.6%	500,996	71.5%
z3	622	0.1%	241	0.1%
nfa2sat	N/A	N/A	105,948	15.1%
timeout (3s)	13	0.0%	7	0.0%

TABLE I: The numbers of problems solved by every solver in the Zelkova portfolio.

Theorem (Soundness of Arithmetic-free Approach): *A formula $\varphi(x, y)$ exceeds a y-count threshold τ iff the conjunction of the constraints in Equations* (9) *and* (10) *is satisfiable.*

Proof. In Equation (9), variable y_e can only be true if every $\exists x.\ \varphi(x, rep_e)$ holds. From the definition of y-equivalence, we have that $\exists x.\ \varphi(x, rep_e)$ holds iff $\exists x.\ \varphi(x, c)$ holds for every $c \in e$. Therefore Equation (10) is true iff and only if there exists a combination of minimal possible violations that is actually satisfiable. The definition of MPV and minimal satisfiable violation ensures that if any violation exists—i.e., there exists a set of equivalence classes that exceeds the threshold—there also exists a minimal satisfiable version of it that is considered in Equation (10). □

If $\varphi(x, y)$ lies in a theory $\mathcal{T}$, assuming a decision procedure for counting models over each equivalence class e, the constraints in Equations (9) and (10) are in the theory $\mathcal{T}$.

V. Implementation and Evaluation

The Block Public Access (BPA) feature detects if a bucket is publicly accessible (`Public`) or not (`Not Public`), and is integrated in many AWS services. Some services use BPA as a preventative control that prevents attaching any policy that is detected to be public to an AWS resource. BPA is an essential guard rail to ensure data is not exposed to broad access. Other detective services, like Config, Macie, Guard Duty and Security Hub, reports to customers which resources have public policies attached, without preventing any access.

Implementation: BPA is built on top of Zelkova's encoding of IAM policies and runs on the portfolio of solvers supported by Zelkova. Zelkova runs on AWS Lambda, a serverless computing platform that runs applications without users needing to provision or manage servers. Zelkova currently uses the solvers CVC4, CVC5, Z3, and NFA2SAT [21] as part of its portfolio. The arithmetic-free approach is supported by all solvers, whereas the arithmetic approach produces an encoding that is not supported by NFA2SAT [21]. Zelkova invokes all supported solvers in parallel and returns the results as soon as one of the solvers provides the answer.

Evaluation: We evaluate the performance of our encodings on 700K randomly chosen policies that contain IP addresses and set a timeout of 3 seconds. The BPA checker [9] performs a pre-processing step that simplifies statements that do not allow any access to untrusted entities. This step removes a

large number of statements, thus leaving us with 225,215 policies to still analyze with our technique. We report a timeout if none of the solvers terminates within 3s, the timeout used in production for BPA checks. We run our experiments on an x86_64 cloud desktop running Amazon Linux version 2 with 96 CPUs and 382GB memory.

Table I shows how many times (# fastest) and for what percentage of the benchmarks (% fastest) each solver was the fastest. The arithmetic approach described (Section IV-C) times out on 13 policies whereas the arithmetic-free approach (Section IV-D) times out on 7 policies. The arithmetic-free approach could solve 6/13 problems the arithmetic approach timed out on, whereas the arithmetic approach couldn't solve any of the policies the arithmetic-free approach timed out on.

The average running time of the arithmetic approach is 8ms, with 50% of the policies terminating within 3ms, 90% of the policies terminating within 21ms, and 99.99% of the policies terminating within 850ms. The average running time of the arithmetic-free approach is 8ms, with 50% of the policies terminating within 3ms, 90% of the policies terminating within 21ms, and 99.99% of the policies terminating within 709ms.

The arithmetic-free approach is on average 0.09 times slower (geomean) than arithmetic approach. However, it times out on 6 fewer policies.

The arithmetic-free approach enables using the NFA2SAT solver [21], and for 105,948 queries (15.13% of our dataset) the NFA2SAT was faster than any other solver. Table I presents the number of problems solved by each solver in the Zelkova portfolio for both techniques.

To summarize, both solving approaches are effective for the BPA application and the arithmetic-free approach can solve more queries, but is slightly (0.09 times) slower.

We further analyze the experiments. The formula size ranges from 2K to 1,095K bytes (avg. 17K), the number of equivalence classes ranges from 1 to 9 (avg. 1.1), the time taken to compute equivalence classes ranges from 1ms to 254ms (avg. 2ms), and the SMT solver takes 1ms to 1,559ms (avg. 8ms). The time taken to compute the MPV sets in the arithmetic-free approach ranges from 1ms to 50ms (avg. 1ms). The additional data indicates that the SMT solvers takes most of the time needed to check BPA for IPs.

VI. Related Work

Previous work on Block Public Access [9] did not address the issue of counting IP addresses; this new use case emerged afterwards through conversations with customers. The previous work was designed for cases where there were a relatively small number of trusted values drawn from an overwhelmingly larger universe of possibilities. The difference in size between the trusted values and the possible universe meant that no model counting was necessary; any set of trusted values was small enough. For IP addresses, one needs to consider large sets of trusted values drawn from the limited universe of IP addresses. Here one must count models to precisely capture the boundary between public and non-public access.

Quacky [17] can quantify the permissions provided by IAM policies. It uses model counting to count how many requests of size up to a certain bound a policy can match. While Quacky solves a different problem, their methodology could be in principle adapted to count how many IPs one can issues requests from. As we have argued, model counting is a feature supported by very few solvers; Quacky uses one solver called ABC [5], which only supports strings and integer constraints (and thus limited sets of policies). Furthermore, model counting is generally expensive: for simple EC2 policies consisting of often just one statement, Quacky incurs an average running time of more than 100s, a time that is not acceptable for customers using BPA. We attempted using the solver `abc` used by Quacky [17] to quantify permissions of access control policies, but `abc` did not support the constraints generated by Zelkova, specifically the theory of bit vectors.

To put other work in context, we will consider them through the lens of the requirements of Zelkova. Zelkova is a live AWS service handling customer access policies and supporting many security use cases where soundness is paramount.

Model Counting in SAT: Model counting and model enumeration [19] are established research areas in the SAT community, with a wide range of application in domains that require quantitative analysis—e.g., probabilistic inference [13]. Approaches from SAT that directly enumerate and count Boolean-level models have straightforward translations to the SMT level [20], and are available in some SMT solvers [15]. In our domain, it is insufficient to enumerate the Boolean structure without accounting for the complex SMT formulas involving, e.g., strings, generated by Zelkova to model IAM policies. Enumerating all models is also infeasible due to the size of the solution space—i.e., a typical threshold for the IP-count bounding problem is 2^{24}!

Approximate Model Counting: To avoid enumeration of all models in SAT, approximate model counting [12] relies uses universal hashing to provide provable approximations. Beyond the initial theoretical results [23], this approach yielded highly scalable tools for SAT problems [11], [18] with extensions to extended to some SMT theories, such as bit-vectors [10] and linear arithmetic [14]. These results do not work in the presence of string constraints, which are ubiquitous when modeling IAM policies. Most importantly, our application domain—i.e., counting IPs—requires exact results and cannot rely on probabilistic approximations.

Model Counting in SMT: Precise model counting at the SMT level has mostly focused on individual theories such as integers [8] and restricted theory of strings [22]. Our domain requires reasoning about many string operations, often combined with the theories of arithmetic over integers. The most relevant work that attempts to cover these theories translates string and integer constraints into automata representations that facilitates counting of feasible solutions [4], [5]. Because this line of work relies on transforming constraints to automata, it is limited to string constraints with integer bounds and thus cannot handle the full multi-sorted constraints required by Zelkova.

Summary: In contrast to related work, our approach encodes the counting constraints into a single SMT query that relies on standard SMT-LIB [7] language and theories. The use of standard SMT language allows one to rely on battle-tested general-purpose SMT solvers that match Zelkova's portfolio approach and goals.

VII. Conclusion

This paper defined the IP-count bounding problem as the problem of checking whether the number of IPs from which an IAM policy allows requests exceeds a given bound. The bounded projective IP-count problem is formalized logically as the *bounded projective counting problem* where the goal is to check whether a formula $\#SAT(\exists x.\ \varphi(x,y)) > \tau$ is true.

We presented two SMT encodings of the bounded projective counting problem that avoid the need to solve a model counting problem—i.e., the $\#SAT(\cdot)$ primitive—for which no performant solver support exists. Our encodings are general: if the variable y only appears within monadic predicates in the formula $\varphi(x,y)$ and one has access to a model counter for the theory of y (but one for the theory of x is not required!), our encodings generate SMT formulas that are true iff the bounded projective counting problem admits a solution.

The generality of our encoding opens opportunities to solve other bounding problems for IAM policies, but also in other domains, e.g., if constraints denote valid tuples in a table and one wants to bound the number satisfying assignments for the values of a numerical column.

References

[1] Amazon Web Services. Amazon IAM, Apr 2024. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html.

[2] Amazon Web Services. Amazon S3, Apr 2024. URL: https://aws.amazon.com/s3/.

[3] Amazon Web Services. What is CIDR?, Apr 2024. URL: https://aws.amazon.com/what-is/cidr/.

[4] A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*, pages 255–272. Springer, 2015.

[5] A. Aydin, W. Eiers, L. Bang, T. Brennan, M. Gavrilov, T. Bultan, and F. Yu. Parameterized model counting for string and numeric constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 400–410, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3236024.3236064.

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

[7] C. Barrett, A. Stump, C. Tinelli, et al. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[8] A. I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4):769–779, 1994.

[9] M. Bouchet, B. Cook, B. Cutler, A. Druzkina, A. Gacek, L. Hadarean, R. Jhala, B. Marshall, D. Peebles, N. Rungta, et al. Block public access: trust safety verification of access control policies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 281–291, 2020.

[10] S. Chakraborty, K. Meel, R. Mistry, and M. Vardi. Approximate probabilistic inference via word-level counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[11] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings 19*, pages 200–216. Springer, 2013.

[12] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Approximate model counting. In *Handbook of Satisfiability*, pages 1015–1045. IOS Press, 2021.

[13] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6):772–799, 2008. URL: https://www.sciencedirect.com/science/article/pii/S0004370207001889, doi:10.1016/j.artint.2007.11.002.

[14] D. Chistikov, R. Dimitrova, and R. Majumdar. Approximate counting in SMT and value estimation for probabilistic programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, pages 320–334. Springer, 2015.

[15] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[16] L. D'Antoni and M. Veanes. Automata modulo theories. *Commun. ACM*, 64(5):86–95, 2021. doi:10.1145/3419404.

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

[18] S. Ermon, C. Gomes, A. Sabharwal, and B. Selman. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *International Conference on Machine Learning*, pages 334–342. PMLR, 2013.

[19] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of satisfiability*, pages 993–1014. IOS press, 2021.

[20] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*, pages 424–437. Springer, 2006.

[21] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, R. Majumdar, and D. Nowotka. Solving string constraints using SAT. In C. Enea and A. Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 187–208. Springer, 2023. doi:10.1007/978-3-031-37703-7_9.

[22] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 565–576, 2014.

[23] L. Stockmeyer. The complexity of approximate counting. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 118–126, 1983.