Abstract. A computing policy is a sequence of rules, where each rule consists of a predicate and a decision, and where each decision is either “accept” or “reject”. A policy \( P \) is said to accept (or reject, respectively) a request if and only if the decision of the first rule in \( P \), that matches the request is “accept” (or “reject”, respectively). Examples of computing policies are firewalls, routing policies and software-defined networks in the Internet, and access control policies. It has been shown earlier that the problems of determining whether a given policy satisfies a property, such as adequacy, completeness, implication, equivalence, and redundancy, are all NP-hard. In this paper, we present efficient algorithms that use SAT solvers to determine whether any given policy satisfies a property. Experimental results show that our algorithms can determine whether a given policy with 90K rules satisfies the adequacy or completeness property in about 3 minutes. They also show that our algorithms can determine whether a given policy with 90K rules satisfies the implication, equivalence, or redundancy property in about 1 hour.

Keywords: Policies, Firewalls, Access Control, Routing Policies, SAT

1 Introduction

A computing policy is a filter that is placed at the entry point of some resource. Each request to access the resource needs to be first examined against the policy to determine whether to accept or reject the request. The decision of a policy to accept or reject a request depends on two factors:

1. The values of some attributes that are specified in the request and
2. The sequence of rules in the policy that are specified by the policy designer.

Examples of computing policies are firewalls in the Internet, routing policies and software-defined networks in the Internet, and access control policies [12]. Early methods for the logical analysis of computing policies have been reported in [17], [9], and [8]. Methods for probabilistic verification of computing policies have been reported in [4] and methods for incremental verification have been reported in [2] and [6].
A rule in a policy consists of a predicate and a decision, which is either “accept” or “reject”. To examine a request against a policy, the rules in the policy are considered one by one until the first rule, whose predicate satisfies the values of the attributes in the request, is identified. Then the decision of the identified rule, whether “accept” or “reject”, is applied to the request.

Note that there are three sets of requests that are associated with each policy $P$: (1) the set of requests that are accepted by $P$, (2) the set of requests that are rejected by $P$, and (3) the set of requests that are ignored by $P$ (i.e. neither accepted nor rejected by $P$). This third set is usually, but not always, empty.

Next, we present two policy examples $P$ and $Q$ and use these examples to specify five logical properties of policies: adequacy, completeness, implication, equivalence, and redundancy.

Let $u$ and $v$ be two attributes whose integer values are taken from the interval $[1,9]$. A policy $P$ over these two attributes can be defined as follows:

\[
\begin{align*}
((u \in [1,4]) \land (v \in [8,9])) & \rightarrow \text{reject} \\
((u \in [2,4]) \land (v \in [7,9])) & \rightarrow \text{accept} \\
((u \in [3,3]) \land (v \in [8,9])) & \rightarrow \text{accept} \\
((u \in [1,9]) \land (v \in [1,9])) & \rightarrow \text{reject}
\end{align*}
\]

Policy $P$ consists of four rules. The first rule states that each request $(u,v)$, where the value of $u$ is an integer in the interval $[1,4]$ and where the value of $v$ is an integer in the interval $[8,9]$, is to be rejected. The second rule states that each request $(u,v)$, that does not match the first rule and where the value of $u$ is an integer in the interval $[2,4]$ and where the value of $v$ is an integer in the interval $[7,9]$, is to be accepted. The set of requests that are accepted by policy $P$ is $\{(2,7), (3,7), (4,7)\}$. Notice that the fourth rule rejects all requests, so no requests are ignored.

A second policy $Q$ over attributes $u$ and $v$ can be defined as follows:

\[
\begin{align*}
((u \in [2,3]) \land (v \in [7,7])) & \rightarrow \text{accept} \\
((u \in [2,4]) \land (v \in [7,8])) & \rightarrow \text{accept} \\
((u \in [1,9]) \land (v \in [1,9])) & \rightarrow \text{reject}
\end{align*}
\]

The set of requests that are accepted by $Q$ is $\{(2,7), (3,7), (4,7), (2,8), (3,8), (4,8)\}$ and all other requests are rejected.

A policy that accepts at least one request is called *adequate*. Therefore, each of the two policies $P$ and $Q$ (defined above) is adequate.

A policy that accepts or rejects each request is called *complete*. Therefore, each of the two policies $P$ and $Q$ is complete.

On one hand, the policy pair $(P,Q)$, where each request that is accepted by $P$ is also accepted by $Q$, is called an *implication*. On the other hand, the policy pair $(Q,P)$, where $Q$ accepts at least one request that is not accepted by $P$, is not an implication.

The policy pair $(P,Q)$, where both $P$ and $Q$ are complete and where at least one of the two pairs $(P,Q)$ and $(Q,P)$ is not an implication, is not an *equivalence*. 
The third rule denoted \( rl \) in policy \( P \) is called redundant because removing this rule from \( P \) yields a new policy denoted \( (P/rl) \) such that the policy pair \( (P,(P/rl)) \) is an equivalence.

Five problems related to the analysis of policies are NP-hard \([7]\). These problems are determining

1. whether any given policy \( P \) is adequate;
2. whether any given policy \( P \) is complete;
3. whether any given policy pair \( (P,Q) \) is an implication;
4. whether any given policy pair \( (P,Q) \) is an equivalence;
5. and whether any given accept rule in any given policy \( P \) is redundant.

In this paper, we present five efficient algorithms to solve these five problems. All algorithms are based on the SAT approach: encode the problem into one or more propositional formulas; those formulas using a state-of-the-art SAT solver, such as Lingeling \([5]\) or Glucose \([4]\); and decode the solution back to obtain a solution for the original problem. The algorithms are as follows:

1. Algorithm 1 constructs a SAT problem that is satisfiable if and only if a given policy is adequate.
2. Algorithm 2 constructs a SAT problem that is unsatisfiable if and only if a given policy is complete.
3. Algorithm 3 constructs a SAT problem that is unsatisfiable if and only if a given policy pair is an implication.
4. Algorithm 4 constructs two SAT problems that are both unsatisfiable if and only if a given policy pair is an equivalence (checks both implications).
5. Algorithm 5 constructs three SAT problems that are all unsatisfiable if and only if a given accept rule in a given policy is redundant (checks whether the removal of the accept rule results in an equivalence).

Algorithms 2, 4, and 5 first reformulate the given policy (pair) and afterwards apply the Algorithms 1 and 3. Hence the runtime of all algorithms is dominated by the execution times of Algorithms 1 and 3. Therefore, the effectiveness of our five algorithms is directly dependent on the effectiveness of Algorithms 1 and 3.

Below, we present some experimental results where we illustrate the effectiveness of Algorithms 1 and 3 by applying these two algorithms to a large number of firewalls, which are policies with five attributes each. In these experiments, we show that the execution time of Algorithm 1 when applied to a firewall with 90K rules is less than 3 minutes. We also show that the execution time of Algorithm 3 when applied to a pair of firewalls with up to 90K rules each is less than 60 minutes. These results testify to the effectiveness of our five algorithms.

## 2 Preliminaries about Policies

In this section, we formally introduce the main concepts related to computing policies. These concepts are: Intervals, Attributes, Requests, Predicates, Decisions, Rules, and Policies.
**Intervals**: An interval is a finite and nonempty set of consecutive integers. An interval $X$ can be denoted by a pair of integers $[y, z]$, where $y$ is the smallest integer in $X$, and $z$ is the largest integer in $X$. Note that an interval $[y, y]$ has only one integer $y$. Note also that any pair $[y, z]$, where $y > z$, is not an interval.

**Attributes**: An attribute is a “variable” that has a “name” and a “value”. Throughout this paper, we assume that there are $t$ attributes whose names are $u_1, u_2, \ldots, u_t$. The value of each attribute $u_i$ is taken from an interval that is called the domain of attribute $u_i$ and is denoted $D(u_i)$.

**Requests**: A request is a tuple $(b_1, \ldots, b_t)$ of $t$ integers, where $t$ is the number of attributes and each integer $b_i$ is taken from the domain $D(u_i)$ of attribute $u_i$.

**Predicates**: A predicate is of the form $((u_1 \in X_1) \land \cdots \land (u_t \in X_t))$, where each $u_i$ is an attribute, each $X_i$ is an interval that is contained in the domain $D(u_i)$ of attribute $u_i$, and $\land$ is the logical AND or conjunction operator.

The value of each conjunct $(u_i \in X_i)$ in a predicate is true if and only if the value of attribute $u_i$ is an integer in interval $X_i$.

The value of a predicate is true if and only if the value of every conjunct $(u_i \in X_i)$ in this predicate is true.

A predicate $((u_1 \in X_1) \land \cdots \land (u_t \in X_t))$, where each interval $X_i$ is the whole domain of the corresponding attribute $u_i$, is called the ALL predicate.

A request $(b_1, \ldots, b_t)$ is said to match a predicate $((u_1 \in X_1) \land \cdots \land (u_t \in X_t))$ if and only if each integer $b_i$ in the request is an element in the corresponding interval $X_i$ in the predicate.

**Decisions**: We assume that there are two distinct decisions: “accept” and “reject”. Henceforth, we write “accept” and “reject” with quotation marks to indicate the “accept” and “reject” decisions, respectively. We also write accept and reject without quotation marks to indicate the English words accept and reject, respectively.

**Rules**: A rule (in a policy) is defined as a pair, one predicate and one decision, written as follows:

$$(\text{predicate}) \rightarrow (\text{decision})$$

A rule whose decision is “accept” is called an accept rule, and a rule whose decision is “reject” is called a reject rule. An accept rule whose predicate is the ALL predicate is called an accept-ALL rule, and a reject rule whose predicate is the ALL predicate is called the reject-ALL predicate.

A request is said to match a rule if and only if the request matches the predicate of the rule. (Note that each request matches every ALL rule.)

**Policies**: A policy is a (possibly empty) sequence of rules. A policy $P$ is said to accept (or reject, respectively) a request $rq$ if and only if $P$ has an accept (or reject, respectively) rule $r$ such that request $rq$ matches rule $r$ and does not match any rule that precedes rule $r$ in policy $P$. 

3 Preliminaries about Boolean Satisfiability

To make it easier to follow the presentation, some background concepts about the Boolean Satisfiability (SAT) problem are discussed below.

For a Boolean variable $x$, there are two literals, the positive literal, denoted by $x$, and the negative literal, denoted by $\neg x$. A clause is a finite disjunction of literals and a conjunctive normal form (CNF) formula is a finite conjunction of clauses. A truth assignment for a CNF formula $C$ is a function $\tau$ that maps literals $l$ in a CNF formula $F$ to $\{t, f\}$. If $\tau(l) = v$, then $\tau(\neg l) = \neg v$, where $\neg t = f$ and $\neg f = t$. Furthermore:

- A clause $C$ is satisfied by assignment $\tau$ if $\tau(l) = t$ for some $l \in C$.
- A clause $C$ is falsified by assignment $\tau$ if $\tau(l) = f$ for all $l \in C$.
- A formula $F$ is satisfied by assignment $\tau$ if all $C \in F$ are satisfied by $\tau$.
- A formula $F$ is falsified by assignment $\tau$ if some $C \in F$ is falsified by $\tau$.

A CNF formula with no satisfying assignments is called unsatisfiable.

4 The Problem of Policy Adequacy

The problem of policy adequacy is to design an algorithm that takes as input any given policy $P$ and determines whether there is a request that is accepted by $P$.

In this section, we present an algorithm, named Algorithm 1, that encodes the adequacy property of any given policy into a SAT problem and applies an off-the-shelf SAT solver to solve the problem. An alternative SAT solving approach was presented in [18], where the authors focus on policies where the rules are defined in terms of “buckets” rather than “intervals”. An example of a rule that is defined in terms of two buckets $B_1$ and $B_2$ is as follows:

$$((u \in B_1) \land (v \in B_2)) \rightarrow \langle \text{decision} \rangle$$

Each of the two buckets is a set of integers. For example, bucket $B_1$ can be the set $\{1, 2\}$ which can be also represented as $\{4, 6, 12, 14\}$. Also, bucket $B_2$ can be the set $\{1\}$ which can be also represented as $\{3\}$. In this case, one rule that is defined in terms of these two buckets can be replaced by the following sequence of eight rules defined in terms of intervals:

$$((u \in [4, 4]) \land (v \in [1, 1])) \rightarrow \langle \text{decision} \rangle$$
$$((u \in [4, 4]) \land (v \in [3, 3])) \rightarrow \langle \text{decision} \rangle$$
$$((u \in [6, 6]) \land (v \in [1, 1])) \rightarrow \langle \text{decision} \rangle$$
$$((u \in [6, 6]) \land (v \in [3, 3])) \rightarrow \langle \text{decision} \rangle$$
$$((u \in [12, 12]) \land (v \in [1, 1])) \rightarrow \langle \text{decision} \rangle$$
$$((u \in [12, 12]) \land (v \in [3, 3])) \rightarrow \langle \text{decision} \rangle$$
$$((u \in [14, 14]) \land (v \in [1, 1])) \rightarrow \langle \text{decision} \rangle$$
$$((u \in [14, 14]) \land (v \in [3, 3])) \rightarrow \langle \text{decision} \rangle$$
Similarly, any rule that is defined in terms of intervals can be replaced by a sequence of rules that are defined in terms of buckets. In Section 9 below, we show that for an important class of policies, called firewalls, any firewall $P$ with 1000 rules defined in terms of intervals can be replaced by an equivalent firewall $Q$ with about 200,000 rules defined in terms of buckets. Therefore, to determine whether a given firewall $P$ (whose rules are defined in terms of intervals) is adequate, it is better to apply Algorithm 1 to firewall $P$ rather than apply the algorithm in [18] to the equivalent firewall $Q$ (whose rules are defined in terms of buckets). Next, we define the four steps of Algorithm 1.

In the first step of Algorithm 1, the given policy $P$ is encoded as a Boolean formula $FP$ such that a request $rq$ is accepted by $P$ if and only if $rq$ makes the value of $FP$ true.

Formula $FP$ is defined as follows:

$$FP = (acp(1) \lor \cdots \lor acp(nap)) \land$$

$$arp(1) \land \cdots \land arp(nap) \land$$

$$rrp(1) \land \cdots \land rrp(nrp) \land$$

$$LP$$

where $nap$ is the number of accept rules in policy $P$, $nrp$ is the number of reject rules in policy $P$. Each $acp(i)$, where $i \in \{1, \ldots, nap\}$, is a Boolean variable denoting that the $i$-th accept rule is matched. Each $arp(i)$, where $i \in \{1, \ldots, nap\}$, is a predicate whose value is true if and only if $acp(i)$ is false or request $rq$ matched the $i$-th accept rule in policy $P$. Each $rrp(j)$, where $j \in \{1, \ldots, nrp\}$, is a predicate whose value is true if and only if the $j$-th reject rule in policy $P$ is preceded by some $i$-th accept rule where $acp(i)$ is true or request $rq$ does not match the $j$-th reject rule in policy $P$. Predicate $LP$ is discussed below.

In the second step of Algorithm 1, we introduce into formula $FP$ Boolean variables that we will use in the third step of the algorithm to encode the predicates $arp(i)$ and $rrp(j)$ in $FP$.

For each interval $[y, z]$ of an attribute $u$, that occurs in any rule in policy $P$, introduce into $FP$ two Boolean variables named

$$leq(u, y - 1)\quad leq(u, z)$$

Thus, the total number of introduced “leq” variables is at most $2nt$, where $n$ is the number of rules in policy $P$ and $t$ is the number of attributes in $P$.

The leq variables are inspired by the order encoding [15], a popular and successful method to translate constraint satisfaction problems into SAT. Given a (non-Boolean) variable with a domain, the order encoding uses for each value in that domain a Boolean variable denoting that variable assignment is less or equal than that value. In contrast, the conventional (or direct) encoding [16] uses Boolean variables denoting whether a variables has assigned an exact value. As we will discuss below, we will only introduce two Boolean (leq) variables for each interval of each attribute.
The value of variable leq \((u, y - 1)\) is true if and only if the value of attribute \(u\) in request \(rq\) is less than or equal \(y - 1\). Similarly, the value of variable leq \((u, z)\) is true if and only if the value of attribute \(u\) in request \(rq\) is less than or equal \(z\).

Predicate \(LP\) in formula \(FP\) describes some expected restrictions on the values of the “leq” Boolean variables introduced into \(FP\). For example if two Boolean variables leq \((u, 5)\) and leq \((u, 11)\) are introduced into \(FP\), then predicate \(LP\) should enforce \((\neg\text{leq}\ (u, 5) \rightarrow \text{leq}\ (u, 11))\) or, equivalently, imply the clause \((\neg\text{leq}\ (u, 5) \lor \text{leq}\ (u, 11))\).

In the third step of Algorithm 1, we use the introduced “leq” Boolean variables to encode the predicates arp \((i)\) and rrp \((j)\) as follows.

Let the \(i\)-th accept rule in policy \(P\) be of the form:

\[
\begin{aligned}
&u_1 \in [y_1, z_1] \land \cdots \land u_t \in [y_t, z_t] \rightarrow \text{accept}
\end{aligned}
\]

In this case, predicate arp \((i)\) can be encoded as follows:

\[
\begin{aligned}
&\neg\text{acp}\ (i) \lor \neg\text{leq}\ (u_1, y_1 - 1) \land \neg\text{acp}\ (i) \lor \text{leq}\ (u_1, z_1) \land \\
&\cdots \land \\
&\neg\text{acp}\ (i) \lor \neg\text{leq}\ (u_t, y_t - 1) \land \neg\text{acp}\ (i) \lor \text{leq}\ (u_t, z_t)
\end{aligned}
\]

Let the \(j\)-the reject rule in policy \(P\) be of the form:

\[
\begin{aligned}
&u_1 \in [y_1, z_1] \land \cdots \land u_t \in [y_t, z_t] \rightarrow \text{reject}
\end{aligned}
\]

and assume that there are \(k\) accept rules that precede the \(j\)-th reject rule in \(P\).

In this case, predicate rrp \((j)\) can be encoded as follows:

\[
\begin{aligned}
&\neg\text{acp}\ (1) \lor \neg\text{acp}\ (2) \lor \cdots \lor \neg\text{acp}\ (k) \lor \\
&\text{leq}\ (u_1, y_1 - 1) \lor \neg\text{leq}\ (u_1, z_1) \lor \\
&\cdots \lor \\
&\text{leq}\ (u_t, y_t - 1) \lor \neg\text{leq}\ (u_t, z_t)
\end{aligned}
\]

In the fourth step of Algorithm 1, a SAT solver is applied to formula \(FP\) to determine whether \(FP\) is satisfiable. If the SAT solver determines that \(FP\) is satisfiable, we conclude that the given policy \(P\) accepts at least one request. On the other hand, if the SAT solver determines that \(FP\) is unsatisfiable, we conclude that \(P\) accepts no request. We give an example Example 1 in Appendix to show how to apply Algorithm 1 to determine whether a given policy is adequate.

The complexity of Algorithm 1 to determine whether a given policy \(P\) is adequate is measured by the number of Boolean variables introduced into formula \(FP\) in Algorithm 1. Note that if the given policy \(P\) has \(n\) rules and \(t\) attributes, then formula \(FP\) in Algorithm 1 has \(O(nt)\) Boolean variables and the complexity of Algorithm 1 does not depend on the range of values of the different attributes in policy \(P\).

One of our goals was to make the encoding as compact as possible, since this is generally believed to result in the best performance. Our encoding is
more compact compared to alternative translations and we will illustrate that using a complexity analysis on the number of clauses. For each accept rule, we use $t$ clauses, while only a single clause is used for each reject rule. The number of clause of the $LP$ part depends on the overlap of the interval bounds among the rules, but is at most $(nap + nrp)t$. Hence our encoding uses at most $1 + nap \cdot t + nrp + (nap + nrp)t$ clauses. So on average, we use at most $2t$ clauses per rule.

Formulating the problem as an SMT instance, could result in much a larger representations. For example, bit-blasting the SMT formulation of the problem will increase the number of clauses with approximately $\log(m)$ with $m$ denoting the maximum element in the intervals, because a constant $c$ encoded using $\log(c)$ clauses. Since it is common to use 16-bit integers in intervals, bit-blasting will result is a representation that is about 16 times larger. Without bit-blasting the size could become even larger. Using difference logic, could learn $O(m^2t)$ relations between the interval variables (i.e., potentially also all the transitive ones).

Notice that the encoding for accept rules and reject rules are different. For accept rules, we want to know whether it is matched, while for reject rules, we want to know whether it is not matched. Any automated translation of the problem would generate both directions for both types of rules. Using only one direction reduces the encoding by a factor of two. This technique is similar as the Plaisted-Greenbaum encoding of digital circuits into SAT [13].

Throughout the paper we evaluate the effectiveness of our approach on firewalls, which is a special case of a policy with five attributes. All attributes have domain $[0, 65535]$ (16 bits) and for each rule the intervals of the attributes are picked randomly\footnote{Our benchmarks and tools are available at \url{http://www.cs.utexas.edu/~marijn/firewall}}. The state-of-the-art SAT solver \texttt{Glucose} version 3.0 [4] was used to solve the generated formulas. All the experiments were run on an Intel Xeon X5440 (quad-core) machine with 2.83GHz of processor speed and 32GB RAM using operating system is Ubuntu 14.04.4 LTS.

Figure 1 shows the relationship between the number of rules in a given firewall $P$ and the execution time of Algorithm 1 when this algorithm is applied to firewall $P$ to determine whether $P$ is adequate. From Figure 1, the execution time of Algorithm 1 is less than 3 minutes when the given firewall $P$ has up to 90,000 rules.

\section{The Problem of Policy Completeness}

The problem of policy completeness is to design an algorithm that takes as input any given policy $P$ and determines whether there is a request that is ignored by $P$.

To solve the policy completeness problem, we present in this section Algorithm 2 that takes as input any given policy $P$ and computes another policy $Q$ such that there is a request that is ignored by $P$ if and only if there is a
request that is accepted by $Q$. Recall that, Algorithm 1 in the previous section can be invoked to determine whether there is a request that is accepted by $Q$. If Algorithm 1 determines that there is a request that is accepted by $Q$, then we conclude that there is a request that is ignored by $P$. Otherwise, we conclude that there is no request that is ignored by $P$.

In Algorithm 2, a new policy $Q$ is obtained from the given policy $P$ in two steps. First, every “accept” decision in every rule in policy $P$, is replaced by a “reject” decision. Second, an accept-All rule is added at the end of policy $P$ yielding policy $Q$. It is straightforward to show that there is a request that is ignored by the given policy $P$ if and only if there is a request that is accepted by the obtained policy $Q$. We give an example Example 2 in Appendix to show how to apply Algorithm 2 to determine whether a given policy is complete.

6 The Problem of Policy Implication

A pair of policies $(P, Q)$ is called an implication if and only if the set of requests accepted by policy $P$ is a subset of the set of requests accepted by policy $Q$ \cite{14}. The problem of policy implication is to design an algorithm that takes as input any given two policies $P$ and $Q$ and determines whether the pair $(P, Q)$ is an implication. In this section, we present Algorithm 3 to solve the policy implication problem.
In the first step of Algorithm 3, we use the first three steps of Algorithm 1 to encode policy $P$ into formula $FP$, where

$$FP = (acp(1) \lor \cdots \lor acp(nap)) \land$$
$$arp(1) \land \cdots \land arp(nap) \land$$
$$rrp(1) \land \cdots \land rrp(nrp) \land$$
$$LP$$

In the second step of Algorithm 3, we construct a policy $Q'$ from policy $Q$ in the following two steps: (i) Every “reject” decision in $Q$ is replaced by an “accept” decision in $Q'$ and vice versa; and (ii) an “accept-ALL” rule is added at the end of $Q$.

In the third step of Algorithm 3, we use the first three steps of Algorithm 1 to encode policy $Q'$ into formula $FQ'$, where

$$FQ' = (acq'(1) \lor \cdots \lor acq'(naq')) \land$$
$$arq'(1) \land \cdots \land arq'(naq') \land$$
$$rrq'(1) \land \cdots \land rrq'(nrq') \land$$
$$LQ'$$

In the fourth step of Algorithm 3, we construct a formula $FPQ'$ from the two formulas $FP$ and $FQ'$ as follows:

$$FPQ' = (acp(1) \lor \cdots \lor acp(nap)) \land$$
$$arp(1) \land \cdots \land arp(nap) \land$$
$$rrp(1) \land \cdots \land rrp(nrp) \land$$
$$acq'(1) \lor \cdots \lor acq'(naq') \land$$
$$arq'(1) \land \cdots \land arq'(naq') \land$$
$$rrq'(1) \land \cdots \land rrq'(nrq') \land$$
$$LPQ'$$

where $acp(1), \ldots, acp(nap)$, $arp(1), \ldots, arp(nap)$, and $rrp(1), \ldots, rrp(nap)$ are as defined in formula $FP$ and $acq'(1), \ldots, acq'(naq')$, $arq'(1), \ldots, arq'(naq')$, and $rrq'(1), \ldots, rrq'(nrq')$ are as defined in formula $FQ'$.

Predicate $LPQ'$ in formula $FPQ'$ describes some expected restrictions on the values of the “leq” Boolean variables introduced into $FP$ and $FQ'$. For example, if two Boolean variables $leq(u, 5)$ and $leq(u, 11)$ are introduced into $FP$ and another Boolean variable $leq(u, 8)$ is introduced into $FQ'$, then predicate $LPQ'$ should imply $(\neg leq(u, 5)) \lor (leq(u, 8))$ and $(\neg leq(u, 8)) \lor (leq(u, 11))$.

In the fifth step of Algorithm 3, we use SAT to determine whether the formula $FPQ'$ is satisfiable. $FPQ'$ is satisfiable if and only if the pair $(P, Q)$ is not an implication pair.

Figure 2 shows the relationship between the number of rules in a given firewall $P$ (which is a special case of a policy with five attributes) and the execution time
Analysis of Computing Policies Using SAT Solvers

Fig. 2. Execution time (in seconds) of Algorithm 3 to determine whether a given firewall pair \((P, Q)\) is an implication, where \(Q\) is obtained from \(P\) by removing an arbitrary rule from \(P\).

of Algorithm 3 when this algorithm is applied to a pair of firewalls \(P\) and \(Q\), where \(Q\) is obtained from \(P\) by removing an arbitrary rule from \(P\), to determine whether \((P, Q)\) is an implication pair. It turns out that each of the pairs \((P, Q)\) that were used in generating Figure 2 is an implication pair, i.e., all formulas were unsatisfiable.

From Figure 2, the execution time of Algorithm 3 is less than 60 minutes when the given firewall \(P\) has up to 90K rules.

Figure 3 shows the relationship between the number of rules in a given firewall \(P\) and the execution time of Algorithm 3 when this algorithm is applied to a pair of firewalls \(P\) and \(Q\), where \(Q\) is obtained from \(P\) by removing the smallest number of rules from the beginning of \(P\) such that the pair \((P, Q)\) is not an implication pair, resulting in satisfiable formulas.

From Figure 3, the execution time of Algorithm 3 is less than 45 minutes when the given firewall \(P\) has up to 90K rules.

7 The Problem of Policy Equivalence

A pair \((P, Q)\) of two complete policies is said to be an equivalence if and only if each of the two pairs \((P, Q)\) and \((Q, P)\) is an implication [10]. In this section, we present Algorithm 4 that takes as input any given pair \((P, Q)\) of two complete policies and determines whether the given pair \((P, Q)\) is an equivalence.

In the first step of Algorithm 4, we use Algorithm 2 in Section 5 to check that indeed the given two policies \(P\) and \(Q\) are complete.

In the second step of Algorithm 4, we use Algorithm 3 in Section 6 to determine whether the two policy pairs \((P, Q)\) and \((Q, P)\) are implication pairs. The
Fig. 3. Execution time (in seconds) of Algorithm 3 to determine whether a given firewall pair \((P, Q)\) is an implication, where \(Q\) is obtained from \(P\) by removing the smallest number of rules from the beginning of \(P\) such that the pair \((P, Q)\) is not an implication.

A pair \((P, Q)\) is an equivalence if and only if both \((P, Q)\) and \((Q, P)\) are implications.

8 The Problem of Redundancy Checking

Let \(P\) be a complete policy that has an accept rule \(rl\). Let \((P/rl)\) denotes the policy that is obtained from policy \(P\) by (1) removing rule \(rl\) from \(P\) then (2) adding a reject-ALL rule at the bottom of \(P\). Rule \(rl\) is said to be redundant in policy \(P\) if and only if the pair \((P, (P/rl))\) is an equivalence [3,11]. In this section, we present Algorithm 5 that takes as input a complete policy \(P\) and an accept rule \(rl\) in \(P\) and determines whether rule \(rl\) is redundant in policy \(P\).

In the first step of Algorithm 5, we use Algorithm 2 in Section 5 to check that indeed the given policy \(P\) is complete.

In the second step of Algorithm 5, we construct policy \((P/rl)\) from policy \(P\) and use Algorithm 4 in Section 7 to determine whether the pair \((P, (P/rl))\) is an equivalence. Rule \(rl\) is redundant if and only if the pair \((P, (P/rl))\) is an equivalence.

9 Related Work

In this paper, we presented a SAT approach to solve five problems related to the logical analysis of computing policies. These five problems are: adequacy
Analysis of Computing Policies Using SAT Solvers 13

(discussed in Section 4), completeness (discussed in Section 5), implication (discussed in Section 6), equivalence (discussed in Section 7), and redundancy (discussed in Section 8).

In [18], the authors presented an alternative SAT approach to solve the same five problems for a different class of policies. Specifically, whereas our approach applies to policies where the rules are defined in terms of intervals, the approach in [18] applies to policies where the rules are defined in terms of buckets.

One may argue that any policy $P$ whose rules are defined in terms of intervals can always be replaced by an equivalent policy $Q$ whose rules are defined in terms of buckets. Hence, instead of using our SAT approach in this paper to solve the five problems for policy $P$, one can always use the SAT approach in [18] to solve the same five problems for policy $Q$.

However, as we show next, when the equivalent policy $Q$ is constructed from policy $P$, the number of rules in the resulting policy $Q$ is much larger than the number of rules in the original policy $P$. Therefore, it is much more efficient to use our SAT approach to solve the five problems for the original policy $P$ than to use the approach in [18] to solve the same five problems for the resulting policy $Q$.

Let $P$ be a firewall (with five attributes) whose rules are defined in terms of intervals and let $Q$ be an equivalent firewall (with the same five attributes) whose rules are defined in terms of buckets. To explore the relationship between the number of rules in firewall $P$ and the number of rules in the equivalent firewall $Q$, we carried out the following experiment. First, we specified ten $P$ firewalls (whose rules are defined in terms of intervals and whose numbers of rules range from 1K to 10K). Second, for each of these $P$ firewalls, we specified an equivalent $Q$ firewall (whose rules are defined in terms of buckets). Third, we plotted in Figure 4 the relationship between the numbers of rules in the $P$ firewalls against the numbers of rules in the equivalent $Q$ firewalls.

From Figure 4, we conclude that if a $P$ firewall has $n$ rules then the equivalent $Q$ firewall has about $200n$ rules. Therefore, it is much more efficient to use our SAT approach in this paper to solve the five analysis problems for a policy $P$ (whose rules are defined in terms of intervals) than to transform policy $P$ to an equivalent policy $Q$ (whose rules are defined in terms of buckets) and use the approach in [18] to solve the same five problems for policy $Q$.

10 Concluding Remarks

In this paper, we identify five properties of policies: adequacy, completeness, implication, equivalence and redundancy. We also present five algorithms that use SAT solvers to determine whether any given policy satisfies each one of these five properties: (1) Algorithm 1 determines whether a policy $P$ is adequate, (2) Algorithm 2 determines whether a policy $P$ is complete, (3) Algorithm 3 determines whether a policy pair $(P,Q)$ is an implication, (4) Algorithm 4 determines whether a policy pair $(P,Q)$ is an equivalence, and (5) Algorithm 5 determines whether a rule in a policy $P$ is redundant.
It turns out that Algorithm 2 invokes Algorithm 1 and that each of the two Algorithms 4 and 5 invokes Algorithms 1 and/or 3. Therefore, the execution time of each of our five algorithms is dominated by the execution times of Algorithms 1 and 3. We measured the execution times of Algorithms 1 and 3 when these two algorithms are applied on a large number of firewalls (which constitute a special class of policies with five attributes each). The results of our measurements are as follows. First, the execution time of Algorithm 1 is up to 3 minutes when this algorithm is applied to firewalls with up to 90K rules. Second, the execution time of Algorithm 3 is up to 60 minutes when this algorithm is applied to firewalls with up to 90K rules.

In [2], the authors argue that each policy $P$ can be partitioned into a number $k$ of sub-policies called accept slices $AS_1, AS_2, \ldots, AS_k$, such that policy $P$ is adequate if and only if at least one of the accept slices $AS_i$ is adequate. Therefore to determine whether $P$ is adequate it is faster to apply Algorithm 1 on all the accept slices $AS_1, AS_2, \ldots, AS_k$ in parallel rather than apply Algorithm 1 on policy $P$. The parallel applications of Algorithm 1 on the accept slices come to halt when either one of the following two conclusions is reached: one of the accept slices is shown to be adequate or all accept slices are shown to be not-adequate.

References

Appendix

Example 1. We discuss how to apply Algorithm 1 to the following policy $P_{\text{example}}$ to determine whether $P_{\text{example}}$ is adequate.

$((u \in [3, 5]) \land (v \in [4, 4])) \rightarrow \text{accept}$

$((u \in [2, 4]) \land (v \in [4, 4])) \rightarrow \text{reject}$

$((u \in [2, 5]) \land (v \in [4, 4])) \rightarrow \text{accept}$

In the first step of Algorithm 1, we encode $P_{\text{example}}$ as the following formula $FP_{\text{example}}$:

$$FP_{\text{example}} = (\text{acp} (1) \lor \text{acp} (2)) \land$$

$$\text{arp} (1) \land \text{arp} (2) \land$$

$$\text{rrp} (1) \land LP_{\text{example}}$$

In the second step of Algorithm 1, we introduce the following six Boolean variables:

leq $(u, 2)$ leq $(u, 5)$ leq $(u, 1)$ leq $(u, 4)$

leq $(v, 3)$ leq $(v, 4)$

In the third step of Algorithm 1, we use the “leq” variables to encode the arp $(i)$ and rrp $(j)$ predicates and to define predicate $LP$ as follows:

$$\text{arp} (1) = (\neg \text{acp} (1) \lor \neg \text{leq} (u, 2)) \land$$

$$(\neg \text{acp} (1) \lor \text{leq} (u, 5)) \land$$

$$(\neg \text{acp} (1) \lor \neg \text{leq} (v, 3)) \land$$

$$(\neg \text{acp} (1) \lor \text{leq} (v, 4))$$

$$\text{arp} (2) = (\neg \text{acp} (2) \lor \neg \text{leq} (u, 1)) \land$$

$$(\neg \text{acp} (2) \lor \text{leq} (u, 5)) \land$$

$$(\neg \text{acp} (2) \lor \neg \text{leq} (v, 3)) \land$$

$$(\neg \text{acp} (2) \lor \text{leq} (v, 4))$$

$$\text{rrp} (1) = (\text{acp} (1) \lor \text{leq} (u, 1) \lor \neg \text{leq} (u, 4) \lor$$

$$\text{leq} (v, 3) \lor \neg \text{leq} (v, 4))$$

$$LP_{\text{example}} = (\neg \text{leq} (u, 1) \lor \text{leq} (u, 2)) \land$$

$$(\neg \text{leq} (u, 2) \lor \text{leq} (u, 4)) \land$$

$$(\neg \text{leq} (u, 4) \lor \text{leq} (u, 5)) \land$$

$$(\neg \text{leq} (v, 3) \lor \text{leq} (v, 4))$$

In the fourth step of Algorithm 1, we can use any SAT solver to determine whether the above formula $FP_{\text{example}}$ is satisfiable. The above policy $P_{\text{example}}$ is adequate if and only if formula $FP_{\text{example}}$ is satisfiable.
Example 2. Consider policy $P_{\text{example}}$ of the prior example and assume that the domain of the first attribute is $[2, 5]$ and the domain of the second attribute is $[3, 4]$. In order to check whether $P_{\text{example}}$ is complete, we transform it into policy $Q_{\text{example}}$ as described in Section 5:

\[
\begin{align*}
((u \in [3, 5]) \land (v \in [3, 4])) &\rightarrow \text{reject} \\
((u \in [2, 4]) \land (v \in [4, 4])) &\rightarrow \text{reject} \\
((u \in [2, 5]) \land (v \in [4, 4])) &\rightarrow \text{reject} \\
((u \in [2, 5]) \land (v \in [3, 4])) &\rightarrow \text{accept}
\end{align*}
\]

Solving $Q_{\text{example}}$ using Algorithm 1 shows that this policy is adequate. Consequently, $P_{\text{example}}$ is not complete.