Policy Expressions and the Bottom-Up Design of Computing Policies

Rezwana Reaz[1], H. B. Acharya[1], Ehab S. Elmallah[2],
Jorge A. Cobb[3], and Mohamed G. Gouda[1]

[1] University of Texas at Austin, USA
[2] University of Alberta, Canada
[3] University of Texas at Dallas, USA
{rezwana,acharya,gouda}@cs.utexas.edu
elmallah@ualberta.ca,cobb@utdallas.edu

Abstract. A 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 iff 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. In this paper, we present a generalization of policies called policy expressions. A policy expression is specified using one or more policies and the three operators: "not", "and", and "or". We show that policy expressions can be utilized to support bottom-up methods for designing policies. We also show that each policy expression can be represented by a set of special types of policies, called slices. Finally, we present several algorithms that use the slice representation of given policy expressions to verify whether the given policy expressions satisfy logical properties such as adequacy, implication, and equivalence.

Keywords: Policies, Firewalls, Access Control, Routing Policies

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 [14], [7], 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 introduce the concept of "policy expressions", the subject matter of the current paper.

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:

$$\big((u \in [1,4]) \wedge (v \in [8,9])\big) \rightarrow \text{reject}$$
$$\big((u \in [2,4]) \wedge (v \in [7,9])\big) \rightarrow \text{accept}$$
$$\big((u \in [1,9]) \wedge (v \in [1,9])\big) \rightarrow \text{reject}$$

Policy P consists of three 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 third rule states that each request (u, v) that does not match the first two rules is to be rejected. Thus, the set of requests that are accepted by policy P is $\{(2, 7), (3, 7), (4, 7)\}$. Notice that because the third rule rejects all requests that do not match the first two rules, we conclude policy P ignores no requests.

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

$$\big((u \in [2,3]) \wedge (v \in [7,7])\big) \rightarrow \text{accept}$$
$$\big((u \in [2,4]) \wedge (v \in [7,8])\big) \rightarrow \text{accept}$$
$$\big((u \in [1,9]) \wedge (v \in [1,9])\big) \rightarrow \text{reject}$$

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.

Now assume that we need to use the two given policies P and Q to design a policy expression (P or Q). This policy expression accepts every request that is accepted by policy P or accepted by policy Q.

In this paper, we show that every policy expression that is specified using one or more policies and the three operators "not", "and", and "or" can be represented by a set $\{S_1, S_2, \cdots, S_k\}$ of a special class of policies called slices such that the following condition holds. A request is accepted by a policy expression iff this request is accepted by at least one slice in the set of slices that represents the policy expression.

As an example, let P and Q refer to the two policies defined above. As discussed in Algorithm 4 below, the policy expression (P or Q) can be represented by the set of three slices $\{S_1, S_2, S_3\}$:

Slice S_1 is defined as follows:

$$\big((u \in [1,4]) \wedge (v \in [8,9])\big) \rightarrow \text{reject}$$
$$\big((u \in [2,4]) \wedge (v \in [7,9])\big) \rightarrow \text{accept}$$

Slice S_2 is defined as follows:

$$\big((u \in [2,3]) \wedge (v \in [7,7])\big) \rightarrow \text{accept}$$

Slice S_3 is defined as follows:

$$\big((u \in [2,4]) \wedge (v \in [7,8])\big) \rightarrow \text{accept}$$

(Notice that, as discussed below, each slice is a policy that consists of zero or more reject rules followed by exactly one accept rule.)

Similarly, as discussed below, the policy expression (P and Q) accepts any request r iff both polices P and Q accept r. As discussed in Algorithm 3 below, the policy expression (P and Q) can be represented by the set of two slices $\{S_4, S_5\}$:

Slice S_4 is defined as follows:

$$\big((u \in [1,4]) \wedge (v \in [8,9])\big) \rightarrow \text{reject}$$
$$\big((u \in [2,3]) \wedge (v \in [7,7])\big) \rightarrow \text{accept}$$

Slice S_5 is defined as follows:

$$\big((u \in [1,4]) \wedge (v \in [8,9])\big) \rightarrow \text{reject}$$
$$\big((u \in [2,4]) \wedge (v \in [7,8])\big) \rightarrow \text{accept}$$

This paper suggests a novel bottom-up design method that can be followed by a designer in designing a computing policy. This design method proceeds as follows. First, the designer designs several simple elementary policies. Second, the designer combines these elementary policies using the three operators "not", "and", and "or" into a single policy expression PE. Finally, the designer uses the algorithms in Section 5 below to verify that designed policy expression PE satisfies desired adequacy, implication, and equivalence properties.

As an example, a designer can start by designing two policies P and Q, then use these two policies to design the policy expression (P and not(Q)). This policy expression accepts every request that is accepted by policy P and rejected by policy Q. Then the designer can use Algorithm 8 in Section 5 below to prove that this policy expression implies both policy P and policy not(Q).

The rest of this paper is organized as follows. In Section 2, we present our formal definition of policies. Then in Section 3, we present our formal definition of policy expressions and discuss three theorems that state fundamental properties

of policy expressions. In Section 4, we introduce the concept of a base of a policy expression as a set of slices that satisfies the following condition. For every request r, the policy expression accepts r iff at least one slice in the base of the policy expression accepts r. Also in Section 4, we present algorithms for constructing a base for every policy expression. In Section 5, we show that the bases of given policy expressions can be used to determine whether the given policy expressions satisfy some logical properties such as adequacy, implication, and equivalence. Finally, we discuss related work in Section 6, and present our concluding remarks in Section 7.

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, Policies, and Complete Policies.

2.1 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.

2.2 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, ..., and 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 $D(u_i)$.

2.3 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. We adopt R to denote the set of all requests. Notice that set R is finite.

2.4 Predicates

A predicate is of the form $((u_1 \in X_1) \wedge \cdots \wedge (u_t \in X_t))$, 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 $\wedge$ is the logical AND or conjunction operator.

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

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

A predicate $((u_1 \in X_1) \wedge \cdots \wedge (u_t \in X_t))$, 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 $((u_1 \in X_1) \wedge \cdots \wedge (u_t \in X_t))$ iff each integer b_i in the request is an element in the corresponding interval X_i in the predicate.

2.5 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.

2.6 Rules

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

$$\langle \text{predicate} \rangle \rightarrow \langle \text{decision} \rangle$$

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 iff the request matches the predicate of the rule. (Note that each request matches every ALL rule.)

2.7 Policies

A policy is a (possibly empty) sequence of rules. A policy P is said to accept (or reject, respectively) a request rq iff 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.

2.8 Complete Policies

A policy P is complete iff every request is either accepted by P or rejected by P.

Let P be a policy. We adopt the notation not(P) to denote the policy that is obtained from policy P by (1) replacing each "accept" decision in P by a "reject" decision in not(P) and (2) replacing each "reject" decision in P by an "accept" decision in not(P).

Note that a policy P is complete iff the policy not(P) is complete.

3 Definition of Policy Expressions

In this section, we present a generalization of policies called policy expressions. Informally, a policy expression is specified using one or more policies and three operators: "not", "and", and "or". Each one of these operators can be applied to one or two policy expressions to produce a policy expression.

Formally, a ⟨policy expression PE⟩ is defined recursively as one of the following four options:

A complete policy P
A complete policy not(P)
⟨policy expression PE_1⟩ and ⟨policy expression PE_2⟩
⟨policy expression PE_1⟩ or ⟨policy expression PE_2⟩

An example of a policy expression is as follows:

(P and not(Q)) or (not(P) and Q)

In this example, P and Q are complete policies.

Associated with each policy expression PE is a request set RS defined as follows:

- If PE is a complete policy P,
  then RS is the set of all requests accepted by P
- If PE is a complete policy not(P),
  then RS is the set of all requests accepted by not(P)
- If PE is a policy expression (PE_1 and PE_2),
  then RS is the intersection of two request sets RS_1 and RS_2 where RS_1 is the request set associated with PE_1 and RS_2 is the request set associated with PE_2
- If PE is a policy expression (PE_1 or PE_2),
  then RS is the union of two request sets RS_1 and RS_2 where RS_1 is the request set associated with PE_1 and RS_2 is the request set associated with PE_2

As an example, the request set associated with the policy expression (P and not(Q)) is the intersection of the two request sets RS_1 and RS_2, where RS_1 is the set of all requests accepted by policy P and RS_2 is the set of all requests accepted by policy not(Q).

Two policy expressions PE_1 and PE_2 are said to be *equivalent* iff the two request sets associated with PE_1 and PE_2 are identical.

For example, the policy expression (P and not(Q)) and the policy expression (not(Q) and P) are equivalent.

Let PE be a policy expression. We adopt the notation not(PE) to denote the policy expression that is recursively obtained from PE as follows:

- If PE is a complete policy P,
  then not(PE) denotes the policy expression not(P)
- If PE is a complete policy not(P),
  then not(PE) denotes the policy expression P

- If PE is a policy expression (PE_1 and PE_2),
  then not(PE) denotes the policy expression (not(PE_1) or not(PE_2))
- If PE is a policy expression (PE_1 or PE_2),
  then not(PE) denotes the policy expression (not(PE_1) and (PE_2))

As an example, not$\big((P$ and not$(Q))$ or $($not(P) and $Q)\big)$ denotes the policy expression $\big((\text{not}(P)$ or $Q)$ and $(P$ or not$(Q))\big)$.

The following three theorems state fundamental properties of policy expressions.

Theorem 1. *For every policy expression PE,* (1) *the request set associated with the policy expression* (PE and not(PE)) *is the empty set, and* (2) *the request set associated with the policy expression* (PE or not(PE)) *is the set R of all requests.*

Proof. We prove Part 1 of this theorem. (A proof of Part 2 is similar to our proof of Part 1.) Our proof of Part 1 makes use of the following definition of the "rank" of a policy expression PE.

The rank k of a policy expression PE is a non-negative integer defined recursively as follows:

- If PE is a complete policy P or is a complete policy not(P), then $k = 0$
- If PE is of the form (PE_1 and PE_2) or is of the form (PE_1 or PE_2),
  then $k = (1 + \max(k_1, k_2))$, where k_1 is the rank of PE_1 and k_2 is the rank of PE_2

Our proof of Part1 is by induction on the rank k of the policy expression PE. This induction proof consists of two parts: a base case and an induction step.

Base Case: We prove that if the rank of PE is 0 then the request set associated with (PE and not(PE)) is empty. Because the rank of PE is 0 then PE is either of the form P or of the form not(P). If PE is of the form P, then we need to prove that the request set associated with (P and not(P)) is empty, which is true for any policy P. If PE is of the form not(P), then we need to prove that the request set associated with $\big($not(P) and not(not(P))$\big)$ is empty, which is also true for any policy P.

Induction Step: We assume that for every PE of rank k or less, the request set associated with (PE and not(PE)) is empty and the request set associated with (PE or not(PE)) is the set R of all requests. We then use this assumption (often called the Induction Hypothesis) to prove that for every PE of rank $(k+1)$, the request set associated with (PE and not(PE)) is empty.

Let PE be any policy expression of rank $(k + 1)$. In this case, PE is either of the form (PE_1 and PE_2) or of the form (PE_1 or PE_2), where the rank k_1 of PE_1 is k or less and the rank k_2 of PE_2 is k or less. In the rest of this proof, we focus on the case where PE is of the form (PE_1 and PE_2). (The proof for the case where PE is of the form (PE_1 or PE_2) is similar.)

Let RS_1 be the request set associated with PE_1, and RS_2 be the request set associated with PE_2. Thus, the request set associated with PE is the intersection of the two sets RS_1 and RS_2.

The induction step proceeds as follows:

(1) From the induction hypothesis, the request set associated with (PE_1 and not(PE_1)) is empty
(2) From the induction hypothesis, the request set associated with (PE_1 or not(PE_1)) is the set R of all requests
(3) From (1) and (2), the request set associated with not(PE_1) is $(R - RS_1)$
(4) Applying steps (1), (2), and (3) to PE_2, the request set associated with not(PE_2) is $(R - RS_2)$
(5) From the fact that PE is (PE_1 and PE_2), not(PE) is $\big(\text{not}(PE_1)$ or not(PE_2)$\big)$
(6) From (3), (4), and (5), the request set associated with not(PE) is $\big((R-RS_1) \bigcup (R - RS_2)\big)$
(7) From the fact that PE is (PE_1 and PE_2), the request set associated with PE is $(RS_1 \bigcap RS_2)$
(8) From (6) and (7), the request set associated with (PE and not(PE)) is $\big((RS_1 \bigcap RS_2) \bigcap ((R - RS_1) \bigcup (R - RS_2))\big)$
(9) From (8), the request set associated with (PE and not(PE)) is $\big((RS_1 \bigcap RS_2) \bigcap (R-(RS_1 \bigcap RS_2))\big)$
(10) From (9), the request set associated with (PE and not(PE)) is empty.

Theorem 2. *For every policy expression PE, the request set associated with the policy expression not(PE) is $(R - RS)$, where R is the set of all requests, RS is the request set associated with PE, and "–" is the set difference operator.*

Proof. Let NS denote the request set associated with not(PE). Thus, the request set associated with the policy expression (PE and not(PE)) is $(RS \bigcap NS)$, and the request set associated with the policy expression (PE or not(PE)) is $(RS \bigcup NS)$. Hence, from Theorem 1, the set $(RS \bigcap NS)$ is empty and the set $(RS \bigcup NS)$ is the set R of all requests. Therefore, set NS is $(R - RS)$.

A policy expression PE is said to be *complete* iff for every request r either PE accepts r or PE rejects r.

Theorem 3. *Every policy expression is complete.*

Proof. Proof by contradiction: Assume that there is a policy expression PE that is not complete. Thus, there is a request r such that PE neither accepts r nor rejects r. Hence, from Theorem 2, request r is neither in the request set RS associated with PE nor in the request set $(R - RS)$ associated with not(PE). Therefore, request r is not in the union of the two sets RS and $(R - RS)$, which constitutes the set R of all requests. This contradicts the fact that r is a request in the set R of all requests.

4 Bases of Policy Expressions

In this section, we introduce the concept of "a base of a policy expression PE" as a set SS of slices that satisfies the following condition. For every request r, the policy expression PE accepts r iff at least one slice in the base SS accepts r. We start this section by introducing the concept of "a slice".

A *slice* is a policy that consists of zero or more reject rules followed by exactly one accept rule.

Let SS be a set of slices and let PE be a policy expression. Set SS is said to be a base of the policy expression PE iff the following condition holds. Each request that is accepted by at least one slice in set SS is in the request set associated with the policy expression PE, and vice versa.

The following five algorithms can be applied to any policy expression PE to construct a slice set SS that is a base of PE.

Algorithm 1
Input: A complete policy P
Output: A slice set SS that is a base of P
Steps: For each accept rule ar in P, construct a slice sl in SS as follows. All the reject rules that precede rule ar in P are added to slice sl. Then rule ar is added at the end of slice sl.
Time Complexity: The time complexity of Algorithm 1 is of $\mathcal{O}(n^2)$ where n is the number of rules in the input policy P.
End

Algorithm 2
Input: A complete policy not(P)
Output: A slice set SS that is a base of not(P)
Steps: For each accept rule ar in not(P), construct a slice sl in SS as follows. All the reject rules that precede rule ar in not(P) are added to slice sl. Then rule ar is added at the end of slice sl.
Time Complexity: The time complexity of Algorithm 2 is of $\mathcal{O}(n^2)$ where n is the number of rules in the input policy not(P).
End

Algorithm 3
Input: A policy expression PE of the form (PE_1 and PE_2)
A slice set SS_1 that is a base of PE_1
A slice set SS_2 that is a base of PE_2
Output: A slice set SS that is a base of PE
Steps: For every slice sl_1 in SS_1 and every slice sl_2 in SS_2, construct a slice sl in SS as follows:

1. The reject rules of slice sl is constructed by merging the reject rules of sl_1 with the reject rules of sl_2 in any order

2. The accept rule of slice sl is constructed by taking the intersection of the predicates of the two accept rules of slices sl_1 and sl_2. If this intersection is empty, then discard slice sl from the base SS of the policy expression PE.

Time Complexity: The time complexity of Algorithm 3 is of $\mathcal{O}((m_1 \times m_2) \times (n_1 + n_2))$ where m_1 is the number of slices in SS_1, m_2 is the number of slices in SS_2, n_1 is the number of rules in the largest slice in SS_1, and n_2 is the number of rules in the largest slice in SS_2.
End

Algorithm 4
Input: A policy expression PE of the form (PE_1 or PE_2)
A slice set SS_1 that is a base of PE_1
A slice set SS_2 that is a base of PE_2
Output: A slice set SS that is a base of PE
Steps: The slice set SS is constructed as the union of the two slice sets SS_1 and SS_2.
Time Complexity: The time complexity of Algorithm 4 is of $\mathcal{O}((m_1 \times n_1) + (m_2 \times n_2))$ where m_1 is the number of slices in SS_1, m_2 is the number of slices in SS_2, n_1 is the number of rules in the largest slice in SS_1, and n_2 is the number of rules in the largest slice in SS_2.
End

Algorithm 5
Input: A policy expression PE
Output: A slice set SS that is a base of PE
Steps: SS is constructed by recursively applying the following four steps:

1. If PE is a complete policy P then use Algorithm 1 to construct SS as a base of P
2. If PE is a complete policy not(P) then use Algorithm 2 to construct SS as a base of not(P)
3. If PE is (PE_1 and PE_2) and SS_1 is a base of PE_1 and SS_2 is a base of PE_2 then use Algorithm 3 to construct SS as a base of PE from the two slice sets SS_1 and SS_2
4. If PE is (PE_1 or PE_2) and SS_1 is a base of PE_1 and SS_2 is a base of PE_2 then use Algorithm 4 to construct SS as a base of policy expression PE from the two slice sets SS_1 and SS_2

Time Complexity: The time complexity of Algorithm 5 depends on the number and type of operators in the input policy expression PE.
End

5 Analysis of Policy Expressions

In this section, we discuss several properties of policy expressions (namely adequacy, implication, and equivalence) and present algorithms that can be used

to determine whether any given policy expressions satisfy these properties.

Algorithm 6
Input: A policy expression PE and a request r
Output: A determination of whether PE accepts r
Steps: Construct a base SS of the policy expression PE using Algorithm 5. If any of the slices in the base SS accepts request r, then PE accepts r. Otherwise, PE does not accept request r.
Time Complexity: Let T denote the time complexity of Algorithm 5 when applied to the input policy expression to construct its base SS. Also let m be the number of slices in SS and let n be the number of rules in the largest slice SS. Therefore, the time complexity of Algorithm 6 is of $\mathcal{O}(T + (m \times n))$.
End

A policy expression PE is said to be *adequate* iff PE accepts at least one request. The following algorithm can be used to determine whether any given policy expression is adequate.

Algorithm 7
Input: A policy expression PE
Output: A determination of whether PE accepts a request.
Steps: Construct a base SS of the policy expression PE using Algorithm 5. For each slice in the constructed base SS, determine whether this slice accepts a request using the PSP method described in [2] and [13]. If one or more slices in SS accepts a request, then PE accepts a request. Otherwise, PE does not accept any request.
Time Complexity: Let T denote the time complexity of Algorithm 5 when applied to the input policy expression to construct its base SS. Also let m be the number of slices in the constructed base SS and n be the number of rules in the largest slice in SS. As discussed in [2] and [13], the time complexity of using the PSP method to determine whether a slice of n rules and t attributes accepts a request is of $\mathcal{O}(n^t)$. Therefore, the time complexity of Algorithm 7 is of $\mathcal{O}(T + (m \times (n^t)))$.
End

A policy expression PE_1 is said to *imply* a policy expression PE_2 iff the request set associated with the policy expression (PE_1 and not(PE_2)) is empty.

Theorem 4. *PE_1 implies PE_2 iff the request set RS_1 associated with PE_1 is a subset of the request set RS_2 associated with PE_2.*

Proof. Proof of the If-Part: Assume that PE_1 implies PE_2. Thus, the request set associated with the policy expression (PE_1 and not(PE_2)) is empty. From Theorem 2, the request set associated with not(PE_2) is the set $(R - RS_2)$, where R is the set of all requests. Therefore, the set $\left(RS_1 \bigcap (R - RS_2)\right)$ is empty and RS_1 is a subset of RS_2.

Proof of the Only-If-Part: Assume that the request set RS_1 associated with PE_1 is a subset of the request set RS_2 associated with PE_2. Thus, the set $\big(RS_1 \bigcap (R - RS_2)\big)$, where R is the set of all requests, is empty. From Theorem 2, the request set associated with not(PE_2) is the set $(R - RS_2)$. Therefore, the request set associated with the policy expression (PE_1 and not(PE_2)) is empty and PE_1 implies PE_2.

Algorithm 8
Input: Two policy expressions PE_1 and PE_2
Output: A determination of whether PE_1 implies PE_2
Steps: First, construct a policy expression PE from the policy expression (PE_1 and not(PE_2)) by pushing the "not" (which is applied to PE_2) deeper into PE_2 until every "not" is applied to a policy. Second, use Algorithm 7 to determine whether the constructed policy expression PE accepts a request. From the definition of "implies", if PE accepts no request then PE_1 implies PE_2. Otherwise, PE_1 does not imply PE_2.
Time Complexity: The time complexity of Algorithm 8 is of $\mathcal{O}(T + (m \times (n^t)))$, where T is the time complexity for constructing the policy expression PE and its base SS, m is the number of slices in the constructed base SS, n is number of rules in the largest slice in SS, and t is the number of attributes in each slice in SS.
End

Theorem 5. *Two policy expressions PE_1 and PE_2 are equivalent iff PE_1 implies PE_2 and PE_2 implies PE_1.*

Proof. Proof of the If-Part: Assume that PE_1 and PE_2 are equivalent. Thus, the request set RS_1 associated with PE_1 and the request set RS_2 associated with PE_2 are identical. Therefore, RS_1 is a subset of RS_2 and RS_2 is a subset of RS_1. From Theorem 2, PE_1 implies PE_2 and PE_2 implies PE_1.

Proof of the Only-If-Part: Assume that PE_1 implies PE_2 and PE_2 implies PE_1. Thus, from Theorem 2, RS_1 is a subset of RS_2 and RS_2 is a subset of RS_1. Therefore, the request set RS_1 associated with PE_1 and the request set RS_2 associated with PE_2 are identical and the two policy expressions PE_1 and PE_2 are equivalent.

Algorithm 9
Input: Two policy expressions PE_1 and PE_2
Output: A determination of whether PE_1 and PE_2 are equivalent
Steps: Use Algorithm 8 twice to determine: (1) whether PE_1 implies PE_2 and (2) whether PE_2 implies PE_1. From Theorem 5, if PE_1 implies PE_2 and PE_2 implies PE_1, then PE_1 and PE_2 are equivalent. Otherwise, also from Theorem 5, PE_1 and PE_2 are not equivalent.
Time Complexity: The time complexity of Algorithm 9 is twice the time complexity of Algorithm 8.
End

6 Related Work

As mentioned earlier, this paper suggests the following bottom-up design method that can be followed by a designer in designing a desired computing policy. First, the designer designs several simple elementary policies. Second, the designer combines these elementary policies using the three operators "not", "and", and "or" into a single policy expression PE that specifies the desired policy. Third, the designer uses Algorithm 5 to construct a base for the policy expression PE. Fourth, the designer uses the constructed base and Algorithms 7, 8, and 9 to verify that the policy expression PE satisfies desired adequacy, implication, and equivalence properties.

Other methods that can be used in designing policies are reported in [5], [11], [3], and [13]. These design methods, along with the bottom-up in the current paper can constitute a library of policy design methods. When designing a policy, it is up to the designer to decide which design method in this library will the designer follow to generate the desired policy.

The method for designing policies in [5] consists of two steps. In the first step, the designer designs the desired policy using a large conflict-free decision diagram instead of a compact sequence of often conflicting rules. In the second step, the designer uses several algorithms to convert the large decision diagram into a compact, yet functionally equivalent, sequence of rules. This design method can be referred to as "simplifying policies by introducing conflicts".

The method for designing policies in [11] consists of three steps. In the first step, the same specification of the desired policy is given to multiple teams who proceed independently to design different versions of the policy. In the second step, the resulting multiple versions of the policy are compared with one another to detect all functional discrepancies between them. In the third step, all discrepancies between the multiple policy versions are resolved, and a final policy that is agreed upon by all teams is generated. This design method can be referred to as "diverse policy design".

The method for designing policies in [3] consists of three steps. In the first step, the set of all expected requests is partitioned into non-overlapping subsets $S_1, S_2, \cdots, S_k$. In the second step, for each subset S_i (obtained in the first step), design a policy P_i that accepts some of the requests in the subset S_i. In the third step, identify policies $P_1, P_2, \cdots, P_k$ generated in the second step as the desired policy. This design methods can be referred to as "divide-and-conquer".

The method for designing policies in [13] consists of k steps. In the first step, the designer starts with a simple policy P_1 that accepts more requests than the designer wishes. In the second step, the designer designs a second policy P_2 such that if any request is accepted by P_2 then the same request is also accepted by P_1. (In other words, P_2 implies P_1.) This process is repeated k times until the designer reaches a policy P_k that accepts those requests and only those requests that the designer wishes to be accepted. This design method can be referred to as "step-wise refinement".

7 Concluding Remarks

The main contribution in this paper is to present a generalization of policies called policy expressions. Each policy expression is specified using one or more policies and the three operators "not", "and", and "or". We showed that each policy expression can be represented by a set of slices called a base of the policy expression. We also showed that the bases of given policy expressions can be used to determine whether the given policy expressions satisfy some desired properties of adequacy, implication, and equivalence. Finally, we showed that policy expressions can be utilized to support bottom-up methods for designing policies.

The authors in [10, 9] investigated a novel representation of policies as finite automata rather than as sequences of rules. They show later in [8], how to use the automata representation of a given policy to determine whether the given policy satisfies some desired properties of adequacy, implication, and equivalence. The question of whether a policy expression can be represented as a finite automaton rather than as a set of slices remains open.

It has been shown in [4] that the problems of determining whether given policies satisfy some desired properties of adequacy, implication, and equivalence are all NP-hard. From this fact and the fact that each (complete) policy is also a policy expression, it follows that the problems of determining whether given policy expressions satisfy some desired properties of adequacy, implication, and equivalence are also NP-hard. Indeed, the time complexities of Algorithms 7, 8, and 9 that can be used to determine whether given policy expressions satisfy some desired properties of adequacy, implication, and equivalence are all exponential.

There are two main approaches to face the NP-hardness of determining whether given policy expressions satisfy some desired properties of adequacy, implication, and equivalence. The first approach is to use SAT solvers, for example as discussed in [15], to determine whether given policy expressions satisfy some desired properties of adequacy, implication, and equivalence. Note that the time complexity of using SAT solvers is polynomial in most practical situations.

The second approach is to use probabilistic algorithms, for example as discussed in [1], to determine whether given policy expressions satisfy some desired properties of adequacy, implication, and equivalence. Note that the time complexities of probabilistic algorithms are always polynomial but unfortunately these algorithms can yield wrong determinations in very rare cases.

References

1. Acharya, H.B., Gouda, M.G.: Linear-time verification of firewalls. In: Proceedings of the 17th IEEE International Conference on Network Protocols (ICNP). pp. 133–140. IEEE (2009)
2. Acharya, H.B., Gouda, M.G.: Projection and division: Linear-space verification of firewalls. In: Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS). pp. 736–743. IEEE (2010)

3. Acharya, H.B., Joshi, A., Gouda, M.G.: Firewall modules and modular firewalls. In: Proceedings of the 18th IEEE International Conference on Network Protocols (ICNP). pp. 174–182. IEEE (2010)
4. Elmallah, E.S., Gouda, M.G.: Hardness of firewall analysis. In: Proccedings of the 2nd International Conference on NETworked sYStems (NETYS), Lecture Notes in Computer Science, vol. 8593, pp. 153–168. Springer (2014)
5. Gouda, M.G., Liu, A.X.: Structured firewall design. Computer Networks 51(4), 1106–1120 (2007)
6. Hoffman, D., Yoo, K.: Blowtorch: a framework for firewall test automation. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 96–103. ACM (2005)
7. Kamara, S., Fahmy, S., Schultz, E., Kerschbaum, F., Frantzen, M.: Analysis of vulnerabilities in internet firewalls. Computers & Security 22(3), 214–232 (2003)
8. Khoumsi, A., Erradi, M., Ayache, M., Krombi, W.: An approach to resolve np-hard problems of firewalls. In: Proceedings of the 4th International Conference on NETworked sYStems (NETYS). Springer (2016)
9. Khoumsi, A., Krombi, W., Erradi, M.: A formal approach to verify completeness and detect anomalies in firewall security policies. In: Proceedings of the 7th International Symposium on Foundations and Practice of Security. pp. 221–236. Springer (2014)
10. Krombi, W., Erradi, M., Khoumsi, A.: Automata-based approach to design and analyze security policies. In: Proceedings of the 12th Annual International Conference on Privacy, Security and Trust (PST). pp. 306–313. IEEE (2014)
11. Liu, A.X., Gouda, M.G.: Diverse firewall design. IEEE Transactions on Parallel and Distributed Systems (TPDS) 19(9), 1237–1251 (2008)
12. Mayer, A., Wool, A., Ziskind, E.: Fang: A firewall analysis engine. In: Proceedings of IEEE Symposium on Security and Privacy. pp. 177–187. IEEE (2000)
13. Reaz, R., Ali, M., Gouda, M.G., Heule, M.J., Elmallah, E.S.: The implication problem of computing policies. In: Proceedings of the 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems, pp. 109–123. Springer (2015)
14. Wool, A.: A quantitative study of firewall configuration errors. Computer 37(6), 62–67 (2004)
15. Zhang, S., Mahmoud, A., Malik, S., Narain, S.: Verification and synthesis of firewalls using SAT and QBF. In: Proceedings of the 20th IEEE International Conference on Network Protocols (ICNP). pp. 1–6. IEEE (2012)