Firewall Policy Queries

Alex X. Liu, *Member*, *IEEE*, and Mohamed G. Gouda, *Member*, *IEEE*

Abstract—Firewalls are crucial elements in network security, and have been widely deployed in most businesses and institutions for securing private networks. The function of a firewall is to examine each incoming and outgoing packet and decide whether to accept or to discard the packet based on its policy. Due to the lack of tools for analyzing firewall policies, most firewalls on the Internet have been plagued with policy errors. A firewall policy error either creates security holes that will allow malicious traffic to sneak into a private network or blocks legitimate traffic and disrupts normal business processes, which in turn could lead to irreparable, if not tragic, consequences. Because a firewall may have a large number of rules and the rules often conflict, understanding and analyzing the function of a firewall has been known to be notoriously difficult. An effective way to assist firewall administrators to understand and analyze the function of their firewalls is by issuing queries. An example of a firewall query is "Which computers in the private network can receive packets from a known malicious host in the outside Internet?" Two problems need to be solved in order to make firewall queries practically useful: how to describe a firewall query and how to process a firewall query. In this paper, we first introduce a simple and effective SQL-like query language, called the Structured Firewall Query Language (SFQL), for describing firewall queries. Second, we give a theorem, called the Firewall Query Theorem, as the foundation for developing firewall query processing algorithms. Third, we present an efficient firewall query processing algorithm, which uses decision diagrams as its core data structure. Fourth, we propose methods for optimizing firewall query results. Finally, we present methods for performing the union, intersect, and minus operations on firewall query results. Our experimental results show that our firewall query processing algorithm is very efficient: it takes less than 10 milliseconds to process a query over a firewall that has up to 10,000 rules.

Index Terms—Network security, firewall queries, firewall testing, firewall correctness.

1 Introduction

Serving as the first line of defense against malicious attacks and unauthorized traffic, firewalls are crucial elements in securing the private networks of most businesses, institutions, and even home networks. A firewall is placed at the point of entry between a private network and the outside Internet so that all incoming and outgoing packets have to pass through it. A packet can be viewed as a tuple with a finite number of fields; examples of these fields are source/destination IP address, source/destination port number, and protocol type. A firewall maps each incoming and outgoing packet to a decision according to its policy (i.e., configuration). A firewall policy defines which packets are legitimate and which are illegitimate by a sequence of rules. Each rule in a firewall policy is of the form

$$\langle predicate \rangle \rightarrow \langle decision \rangle.$$

The $\langle predicate \rangle$ in a rule is a Boolean expression over some packet fields and the network interface card (NIC) on which a packet arrives.[1] For the sake of brevity, we assume that each packet has a field that contains the identification of the network interface on which a packet arrives. The $\langle decision \rangle$ of a rule can be *accept*, or *discard*, or a combination of these decisions with other options such as the logging option. For simplicity, we assume that the $\langle decision \rangle$ in a rule is either *accept* or *discard*.

A packet *matches* a rule if and only if the packet satisfies the predicate of the rule. The predicate of the last rule in a firewall is usually a tautology to ensure that every packet has at least one matching rule in the firewall. The rules in a firewall often conflict. Two rules in a firewall *conflict* if and only if they overlap and they have different decisions. Two rules in a firewall *overlap* if and only if there is at least one packet that can match both rules. Due to conflicts among rules, a packet may match more than one rule in a firewall, and the rules that a packet matches may have different decisions. To resolve conflicts among rules, for each incoming or outgoing packet, a firewall maps it to the decision of the first (i.e., highest priority) rule that the packet matches. Note that two overlapping rules with different decisions syntactically conflict but semantically do not conflict because of the first-match semantics. In this paper, the definition of "conflict" among firewall rules is based on the syntax of rules.

- *A.X. Liu is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824. E-mail: alexliu@cse.msu.edu.*
- *M.G. Gouda is with the Department of Computer Sciences, The University of Texas at Austin, 1 University Station (C0500), Austin, TX 78712-0233. E-mail: gouda@cs.utexas.edu.*

Manuscript received 1 Apr. 2008; revised 6 Oct. 2008; accepted 8 Dec. 2008; published online 22 Dec. 2008.
Recommended for acceptance by M. Singhal.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-04-0124.
Digital Object Identifier no. 10.1109/TPDS.2008.263.

1. Note that most firewall vendors (such as Cisco [10]) allow administrators to specify NIC information in rules. For firewall products (such as Check Point firewalls [9]) that do not provide this functionality, we can use the Rule Assignment and Direction Setting (RADIS) algorithm in [5], [6] to automatically assign the NIC information to each rule. Wool gave a comprehensive discussion of the NIC issue in firewall rules in [44].

The function (i.e., behavior) of a firewall is specified in its policy, which consists of a sequence of rules. The policy of a firewall is the most important component in achieving the security and functionality of the firewall [41]. However, most firewalls on the Internet are poorly configured, as witnessed by the success of worms[2] and viruses like Blaster [8] and Sapphire [11], which could be easily blocked by a well-configured firewall [43]. It has been observed that most firewall security breaches are caused by configuration errors [7]. An error in a firewall policy means that some illegitimate packets are identified as being legitimate, or some legitimate packets are identified as being illegitimate. Such a policy error either creates security holes that will allow malicious traffic to sneak into a private network or blocks legitimate traffic and disrupts normal business processes, which in turn could lead to irreparable, if not tragic, consequences. Clearly, a firewall policy should be well understood and analyzed before being deployed.

However, due to the large number of rules in a firewall and the large number of conflicts among rules, understanding and analyzing the function of a firewall has been known to be notoriously difficult [34], [35]. The implication of any rule in a firewall cannot be understood without examining all the rules listed above that rule. There are other factors that contribute to the difficulties in understanding and analyzing firewalls. For example, a corporate firewall often consists of rules that are written by different administrators at different times and for different reasons. It is difficult for new firewall administrators to understand the implication of each rule that they have not written.

An effective way to assist administrators to understand and analyze firewalls is by issuing firewall queries. Firewall queries are questions concerning the function of a firewall. An example firewall query is "Which computers in the private network can receive BOOTP[3] packets from the outside Internet?" Figuring out answers to such firewall queries is of tremendous help for a firewall administrator to understand and analyze the function of the firewall. For example, assuming the specification of a firewall requires that all computers in the outside Internet, except a known malicious host, are able to send e-mails to the mail server in the private network; a firewall administrator can test whether the firewall satisfies this requirement by issuing a firewall query "Which computers in the outside Internet cannot send e-mails to the mail server in the private network?" If the answer to this query contains exactly the known malicious host, then the firewall administrator is assured that the firewall does satisfy this requirement. Otherwise, the firewall administrator knows that the firewall fails to satisfy this requirement and needs to be reconfigured.

Firewall queries are also useful in a variety of other scenarios, such as firewall maintenance and firewall debugging. For a firewall administrator, checking whether a firewall satisfies certain conditions is a part of daily maintenance activity. For example, if the administrator detects that a computer in a private network is under attack, the firewall administrator can issue queries to check which other computers in the private network are also vulnerable to the same type of attacks. In the process of designing a firewall, the designer can issue some firewall queries to detect design errors by checking whether the answers to the queries are consistent with the firewall specification.

To make firewall queries practically useful, two problems need to be solved: how to describe a firewall query and how to process a firewall query. The second problem is especially difficult. Recall that the rules in a firewall are sensitive to the rule order and the rules often conflict. The naive solution is to enumerate every packet specified by a query and check the decision for each packet. Clearly, this solution is infeasible. For example, to process the query "which computers in the outside Internet cannot send any packet to the private network," this naive solution needs to enumerate 2^{88} possible packets and check the decision of the firewall for each packet, which is infeasible. Note that firewall queries are inherently different from relational database queries. In a relational database, each field of a tuple has a fixed value. Although each rule in a firewall seems analogous to a tuple in a database, each field of a rule is a range, not a fixed value.

In this paper, we present solutions to both problems. First, we introduce a simple and effective SQL-like query language, called the Structured Firewall Query Language (SFQL), for describing firewall queries. This language uses queries of the form "$select \dots from \dots where \dots$." Second, we present a theorem, called the Firewall Query Theorem, as the foundation for developing firewall query processing algorithms. Third, we present an efficient query processing algorithm that uses firewall decision diagrams as its core data structure. For a given firewall of a sequence of rules, we first construct an equivalent firewall decision diagram using a construction algorithm. Then, the firewall decision diagram is used as the core data structure of this query processing algorithm for answering each firewall query. Fourth, we propose methods for optimizing firewall query results. Finally, we present methods for performing the union, intersect, and minus operations on firewall query results. Our experimental results show that our firewall query processing algorithm is very efficient: It takes less than 10 milliseconds to process a query over a firewall that has up to 10,000 rules.

The rest of the paper proceeds as follows: We first review related work in Section 2. We then formally define our problem and notation in Section 3. In Section 4, we present the actual syntax of the structured firewall query language and show how to use this language to describe firewall queries. The theory foundation and algorithms on firewall query processing are presented in Section 5. In Section 6, we present methods for optimizing firewall query results. In Section 7, we present methods for performing the union, intersect, and minus operations on firewall query results. In Section 8, we show experimental results. Finally, we give concluding remarks in Section 9.

2. Note that not all worms can be blocked by only examining packet headers.

3. The Bootp protocol is used by workstations and other devices to obtain IP addresses and other information about the network configuration of a private network. Since there is no need to offer the service outside a private network, and it may offer useful information to hackers, usually Bootp packets are blocked from entering a private network.

2 Related Work

There is little work that has been done on firewall queries. In the seminal work [34], [35], [42], a query-based firewall analysis system named *Fang* was presented. In Fang, a firewall query is described by a triple (a set of source addresses, a set of destination addresses, a set of services), where each service is a tuple (protocol type, source port number, and destination port number). The meaning of such a query is, "which IP addresses in the set of source addresses can use which services in the set of services to which IP addresses in the set of destination addresses." We make three contributions in comparison with Fang. First, in processing a query on a firewall, our query processing algorithm is much more efficient than Fang. In Fang, a query is processed by comparing the query with every rule in a firewall in a linear fashion. In contrast, we first convert a firewall to a tree representation and then process queries on the tree, which is much more efficient. Second, our system can describe and process firewall queries over discard traffic, while Fang only supports queries over accept traffic. Third, we formulate firewall queries using an SQL-like language.

Some firewall analysis methods have been proposed in [4], [14], [15], [19], [25], [26], [29], [30], [31], [36]. In [29], Liu presented algorithms for performing the change impact analysis of firewall policies. In [30], Liu presented an algorithm for verifying firewall policies. The verification of distributed firewalls is studied in [19]. In [31], Liu and Gouda studied the redundancy issues in firewall policies and gave an algorithm for removing all the redundant rules in a firewall policy. In [26], some ad hoc "what if" questions that are similar to firewall queries were discussed. However, no algorithm was presented for processing the proposed "what if" questions. In [15], expert systems were proposed to analyze firewall rules. Clearly, building an expert system just for analyzing a firewall is overwrought and impractical. Detecting potential firewall policy errors by conflict detection was discussed in [4], [14], [25], [36]. Similar to conflict detection, some anomalies are defined and techniques for detecting anomalies are presented in [2], [47]. Examining each conflict or anomaly is helpful in reducing potential firewall policy errors; however, the number of conflicts or anomalies in a firewall is typically large, and manual checking of each conflict or anomaly is unreliable because the meaning of each rule depends on the current order of the rules in the firewall, which may be incorrect.

Some firewall design methods have been proposed in [5], [20], [21], [22], [24], [32]. These works aim at creating firewall rules, while we aim at analyzing firewall rules. Gouda and Liu proposed to use decision diagrams for designing firewalls in [20], [22]. In [32], Liu and Gouda applied the technique of design diversity to firewall design. Gouda and Liu also proposed a model for specifying stateful firewall policies [21]. Guttman proposed a Lisp-like language for specifying high-level packet filtering policies in [24]. Bartal et al. proposed a UML-like language for specifying global filtering policies in [5].

Design of high-performance ATM firewalls was discussed in [45], [46] with focus on firewall architectures. Firewall vulnerabilities were discussed and classified in [45], [46]. However, the focus of [17], [28] are the vulnerabilities of the packet filtering software and the supporting hardware part of a firewall, not the policy of a firewall.

There are some tools currently available for network vulnerability testing, such as Satan [16], [18] and Nessus [38]. These vulnerability testing tools scan a private network based on the current publicly known attacks, rather than the requirement specification of a firewall. Although these tools can possibly catch errors that allow illegitimate access to the private network, they cannot find the errors that disable legitimate communication between the private network and the outside Internet. Firewall policy testing was studied in [27].

3 Formal Definitions

We now formally define the concepts of fields, packets, firewalls, and the Firewall Compression Problem. A *field* F_i is a variable whose domain, denoted $D(F_i)$, is a finite interval of nonnegative integers. For example, the domain of the source address field in an IP packet is $[0, 2^{32} - 1]$. A *packet* over the d fields $F_1, \dots, F_d$ is a d-tuple $(p_1, \dots, p_d)$, where each p_i ($1 \le i \le d$) is an element in $D(F_i)$. We use Σ to denote the set of all packets over fields $F_1, \dots, F_d$. It follows that Σ is a finite set and $|\Sigma| = |D(F_1)| \times \cdots \times |D(F_d)|$, where $|\Sigma|$ denotes the number of elements in set Σ and $|D(F_i)|$ denotes the number of elements in set $D(F_i)$ for each i.

A firewall rule has the form $\langle predicate \rangle \rightarrow \langle decision \rangle$. A $\langle predicate \rangle$ defines a set of packets over the fields F_1 through F_d specified by the predicate $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$, where each S_i is one nonempty interval that is a subset of $D(F_i)$. A packet $(p_1, \dots, p_d)$ *matches* a predicate $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$ and the corresponding rule, if and only if the condition $p_1 \in S_1 \wedge \cdots \wedge p_d \in S_d$ holds. We use α to denote the set of possible values that $\langle decision \rangle$ can be. Typical elements of α include accept, discard, accept with logging, and discard with logging. For any i, if $S_i = D(F_i)$, we often use the keyword *all* to denote S_i.

Some existing firewall products, such as Linux's ipchains [1], represent source and destination IP addresses as prefixes in their rules. An example of a prefix is 192.168.0.0/16 or 192.168.$*$.$*$, both of which represent the set of IP addresses in the range from 192.168.0.0 to 192.168.255.255. Essentially, each prefix represents one integer interval (as we can treat an IP address as a 32-bit integer). In this paper, we uniformly represent firewall rules using intervals.

A *firewall* f over the d fields $F_1, \dots, F_d$ is a sequence of firewall rules. The size of f, denoted $|f|$, is the number of rules in f. A sequence of rules $\langle r_1, \dots, r_n \rangle$ is *complete* if and only if for any packet p, there is at least one rule in the sequence that p matches. A sequence of rules needs to be complete for it to serve as a firewall. To ensure that a firewall is complete, the predicate of the last rule in a firewall is usually specified as $F_1 \in D(F_1) \wedge \cdots F_d \in \wedge D(F_d)$, which every packet matches.

Fig. 1 shows an example of a simple firewall. In this example, we assume that each packet has only two fields: S (source address) and D (destination address), and both

$$
\begin{array}{lllll}
r_1: & S \in [4,7] & \wedge & D \in [6,8] & \rightarrow \mathit{accept} \\
r_2: & S \in [3,8] & \wedge & D \in [2,9] & \rightarrow \mathit{discard} \\
r_3: & S \in [1,10] & \wedge & D \in [1,10] & \rightarrow \mathit{accept}
\end{array}
$$

Fig. 1. Example firewall f_1.

fields have the same domain $[1, 10]$. Let f_1 be the name of this firewall.

Two rules in a firewall may overlap; that is, a single packet may match both rules. For example, rule r_1 and r_2 in the firewall in Fig. 1 overlap. Furthermore, two rules in a firewall may conflict; that is, the two rules not only overlap but also have different decisions. For example, rule r_1 and r_2 in the firewall in Fig. 1 not only overlap but also conflict. To resolve such conflicts, firewalls typically employ a first-match resolution strategy where the decision for a packet p is the decision of the first (i.e., highest priority) rule that p matches in f. The decision that firewall f makes for packet p is denoted $f(p)$.

We can think of a firewall f as defining a many-to-one mapping function from Σ to α. Two firewalls f_1 and f_2 are *equivalent*, denoted $f_1 \equiv f_2$, if and only if they define the same mapping function from Σ to α; that is, for any packet $p \in \Sigma$, we have $f_1(p) = f_2(p)$. For any firewall f, we use $\{f\}$ to denote the set of firewalls that are semantically equivalent to f.

4 Structured Firewall Query Language

In this section, we present the syntax of our firewall query language and show how to use this language to describe firewall queries.

4.1 Query Language

A *query*, denoted Q, in our Structured Firewall Query Language (SFQL) is of the following format:

select F_i
from f
where $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \wedge (\textbf{decision} = \langle dec \rangle)$

where F_i is one of the fields $F_1, \dots, F_d$, f is a firewall, each S_j is a nonempty subset of the domain $D(F_j)$ of field F_j, and $\langle dec \rangle$ is either *accept* or *discard*.

The result of query Q, denoted $Q.result$, is the following set:

$$
\{p_i | (p_1, \dots, p_d) \text{ is a packet in } \Sigma, \text{and}, \\
(p_1 \in S_1) \wedge \cdots \wedge (p_d \in S_d) \wedge (f.(p_1, \dots, p_d) = \langle dec \rangle)\}.
$$

Recall that Σ denotes the set of all packets, and $f.(p_1, \dots, p_d)$ denotes the decision to which firewall f maps the packet $(p_1, \dots, p_d)$.

We can get the above set by first finding all the packets $(p_1, \dots, p_d)$ in Σ such that the following condition holds,

$$
(p_1 \in S_1) \wedge \cdots \wedge (p_d \in S_d) \wedge (f((p_1, \dots, p_d)) = \langle dec \rangle),
$$

then projecting all these packets to the field F_i.

For example, a question to the firewall in Fig. 1, "Which computers whose addresses are in the set $[4, 8]$ can send packets to the computer whose address is 6?," can be formulated as the following query using SFQL:

select S
from f_1
where $(S \in [4,8]) \wedge (D \in \{6\}) \wedge (\textbf{decision} = \mathit{accept})$

The result of this query is $\{4, 5, 6, 7\}$.

As another example, a question to the firewall in Fig. 1, "Which computers cannot send packets to the computer whose address is 6?," can be formulated as the following query using SFQL:

select S
from f_1
where $(S \in \mathit{all}) \wedge (D \in \{6\}) \wedge (\textbf{decision} = \mathit{discard})$

The result of this query is $\{3, 8\}$.

4.2 Firewall Query Examples

Next, we give some example firewall queries using SFQL. Let f be the name of the firewall that resides on the gateway router in Fig. 2. This gateway router has two interfaces: interface 0, which connects the gateway router to the outside Internet, and interface 1, which connects the gateway router to the inside local network. Attached to the private network are a mail server, and two hosts, host 1 and host 2. In these examples, we assume each packet has the following five fields: I (Interface), S (Source IP), D (Destination IP), N (Destination Port), and P (Protocol Type).

Question 1:
Which computers on the private network protected by the firewall f can receive BOOTP[4] packets from the outside Internet?
Query Q_1:
select D
from f
where $(I \in \{0\}) \wedge (S \in \mathit{all}) \wedge (D \in \mathit{all})$
$\wedge (N \in \{67, 68\}) \wedge (P \in \{\mathit{udp}\})$
$\wedge (\textbf{decision} = \mathit{accept})$
Answer to question 1 is $Q_1.result$.

Question 2:
Which ports on the mail server protected by the firewall f are accessible through f?
Query Q_2:
select N
from f
where $(I \in \{0, 1\}) \wedge (S \in \mathit{all}) \wedge (D \in \{\mathit{Mail}$
$\mathit{Server}\} \wedge (N \in \mathit{all}) \wedge (P \in \mathit{all})$
$\wedge (\textbf{decision} = \mathit{accept})$
Answer to question 2 is $Q_2.result$.

4. Bootp packets are UDP packets and use port number 67 or 68.

Question 3:
Which computers in the outside Internet cannot send SMTP[5] packets to the mail server protected by the firewall f?
Query Q_3:
select S
from f
where $(I \in \{0\}) \wedge (S \in all) \wedge (D \in \{Mail\ Server\}) \wedge (N \in \{25\}) \wedge (P \in \{tcp\})$
$\wedge$(**decision** $= discard$)
Answer to question 3 is $Q_3.result$.

Question 4:
Which computers in the outside Internet cannot send any packet to the private network protected by the firewall f?
Query Q_4:
select S
from f
where $(I \in \{0\}) \wedge (S \in all) \wedge (D \in all)$
$\wedge (N \in all) \wedge (P \in all)$
$\wedge$(**decision** $= accept$)
Answer to question 4 is $T - Q_4.result$, where T is the set of all IP addresses outside of the private network

Question 5:
Which computers in the outside Internet can send SMTP packets to both host 1 and host 2 in the private network protected by the firewall f?
Query Q_{5a}:
select S
from f
where $(I \in \{0\}) \wedge (S \in all) \wedge (D \in \{Host\ 1\})$
$\wedge (N \in \{25\}) \wedge (P \in \{tcp\})$
$\wedge$(**decision** $= accept$)
Query Q_{5b}:
select S
from f
where $(I \in \{0\}) \wedge (S \in all) \wedge (D \in \{Host\ 2\})$
$\wedge (N \in \{25\}) \wedge (P \in \{tcp\})$
$\wedge$(**decision** $= accept$)
Answer to question 5 is $Q_{5a}.result \cap Q_{5b}.result$.

5 Firewall Query Processing

5.1 Theory Foundation

In this section, we discuss how to process a firewall query for *consistent firewalls*. A firewall is *consistent* if and only if no two rules in the firewall conflict. A firewall is *inconsistent* if and only if there are at least two rules in the firewall that conflict. Note that a firewall with conflicting rules is syntactically inconsistent, but semantically consistent because of the first-match semantics. In this paper, our definitions of "consistent firewalls" and "inconsistent firewalls" are based on the syntax of firewalls.

5. SMTP stands for Simple Mail Transfer Protocol. SMTP packets are TCP packets and use port number 25.

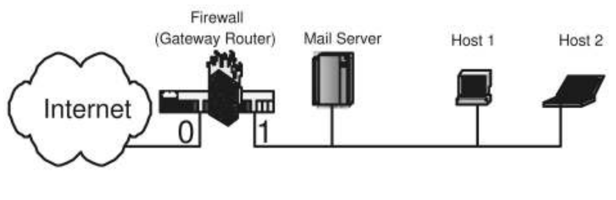

Fig. 2. Firewall f.

$$\begin{array}{lll}
r'_1 : S \in [4,7] & \wedge\ D \in [6,8] & \to a \\
r'_2 : S \in [4,7] & \wedge\ D \in [2,5] \cup [9,9] & \to d \\
r'_3 : S \in [4,7] & \wedge\ D \in [1,1] \cup [10,10] & \to a \\
r'_4 : S \in [3,3] \cup [8,8] & \wedge\ D \in [2,9] & \to d \\
r'_5 : S \in [3,3] \cup [8,8] & \wedge\ D \in [1,1] \cup [10,10] & \to a \\
r'_6 : S \in [1,2] \cup [9,10] & \wedge\ D \in [1,10] & \to a
\end{array}$$

Fig. 3. Consistent firewall f_2.

Recall that two rules in a firewall conflict if and only if they have different decisions and there is at least one packet that matches both rules. For example, the first two rules in the firewall in Fig. 1, namely r_1 and r_2, conflict. Note that for any two rules in a consistent firewall, if they overlap, i.e., there is at least one packet that can match both rules, they have the same decision. So, given a packet and a consistent firewall, all the rules in the firewall that the packet matches have the same decision. Fig. 1 shows an example of an inconsistent firewall, and Fig. 3 shows an example of a consistent firewall. Note that these two firewalls are equivalent. In these two firewall examples, we assume that each packet only has two fields: S (source address) and D (destination address), and both fields have the same domain $[1, 10]$.

Our interest in consistent firewalls is twofold. First, as discussed in Section 5.3, each inconsistent firewall can be converted to an equivalent consistent firewall. Second, as shown in the following theorem, it is easier to process queries for consistent firewalls than for inconsistent ones.

Theorem 1. (Firewall Query Theorem). *Let Q be a query of the following form:*
select F_i
from f
where $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \wedge (\textbf{decision} = \langle dec \rangle)$
Also let f be a consistent firewall that consists of n rules $r_1, \ldots, r_n$, where each rule r_j is of the form $(F_1 \in S'_1) \wedge \cdots \wedge (F_d \in S'_d) \to \langle dec' \rangle$. Then,

$$Q.result = \bigcup_{j=1}^{n} Q.r_j,$$

where each $Q.r_j$ is defined using rule r_j as follows:

$$Q.r_j = \begin{cases} S_i \cap S'_i, & \text{if } (S_1 \cap S'_1 \neq \emptyset) \wedge \cdots \wedge (S_d \cap S'_d \neq \emptyset) \\ & \wedge (\langle dec \rangle = \langle dec' \rangle), \\ \emptyset, & \text{otherwise.} \end{cases}$$

Proof. First, we prove $\bigcup_{j=1}^{n} Q.r_j \subseteq Q.result$ by proving $Q.r_j \subseteq Q.result$ for each j. Let r_j be $(F_1 \in S'_1) \wedge \cdots \wedge (F_d \in S'_d) \rightarrow \langle dec' \rangle$. If $(S_1 \cap S'_1 \neq \emptyset) \wedge \cdots \wedge (S_d \cap S'_d \neq \emptyset) \wedge (\langle dec \rangle = \langle dec' \rangle)$ does not hold, then $Q.r_j = \emptyset$ and clearly $Q.r_j \subseteq Q.result$. If $(S_1 \cap S'_1 \neq \emptyset) \wedge \cdots \wedge (S_d \cap S'_d \neq \emptyset) \wedge (\langle dec \rangle = \langle dec' \rangle)$ does hold, then $Q.r_j = S_i \cap S'_i$. Because of rule r_j, for any packet $(p_1, \ldots, p_d)$ that $(p_1 \in S_1 \cap S'_1) \wedge \cdots \wedge (p_d \in S_d \cap S'_d)$, we have $f.(p_1, \ldots, p_d) = \langle dec' \rangle$. Since $\langle dec \rangle = \langle dec' \rangle$, the following set

$$\{p_i \mid (p_1, \ldots, p_d) \textit{ is a packet in } \Sigma, \textit{ and } (p_1 \in S_1 \cap S'_1) \wedge \cdots \wedge (p_d \in S_d \cap S'_d)\}$$

is a subset of $Q.result$. Because each S_i or S'_i is a nonempty set, we have

$$\begin{aligned} Q.r_j &= S_i \cap S'_i \\ &= \{p_i \mid (p_1, \ldots, p_d) \textit{ is a packet in } \Sigma, \textit{ and}, \\ &\quad (p_1 \in S_1 \cap S'_1) \wedge \cdots \wedge (p_d \in S_d \cap S'_d)\}. \end{aligned}$$

So, $Q.r_j \subseteq Q.result$.

Second, we prove $Q.result \subseteq \bigcup_{j=1}^{n} Q.r_j$. Consider a p_i in $Q.result$. By the definition of $Q.result$, there is at least one packet $(p_1, \ldots, p_d)$ such that the condition $(p_1 \in S_1) \wedge \cdots \wedge (p_d \in S_d) \wedge (f.(p_1, \ldots, p_d) = \langle dec \rangle)$ holds. Let rule r, $(F_1 \in S'_1) \wedge \cdots \wedge (F_d \in S'_d) \rightarrow \langle dec' \rangle$, be a rule that $(p_1, \ldots, p_d)$ matches in f. Because all the rules in the consistent firewall f that $(p_1, \ldots, p_d)$ matches have the same decision, we have $f.(p_1, \ldots, p_d) = \langle dec' \rangle$. So $\langle dec \rangle = \langle dec' \rangle$. Because each p_i is an element of $S_i \cap S'_i$, we have $(S_1 \cap S'_1 \neq \emptyset) \wedge \cdots \wedge (S_d \cap S'_d \neq \emptyset) \wedge (\langle dec \rangle = \langle dec' \rangle)$. So $Q.r_j = S_i \cap S'_i$. Therefore, $p_i \in Q.r_j$. So $Q.result \subseteq \bigcup_{j=1}^{n} Q.r_j$. □

5.2 Rule-Based Firewall Query Processing

The Firewall Query Theorem implies a simple query processing algorithm: given a consistent firewall f that consists of n rules $r_1, \ldots, r_n$ and a query Q, compute $Q.r_j$ for each rule r_j, then $\bigcup_{j=1}^{n} Q.r_j$ is the result of query Q. We call this algorithm the *rule-based firewall query processing algorithm*. Algorithm 1 shows the pseudocode of this algorithm.

Algorithm 1. Rule-Based Firewall Query Processing Algorithm

```
Input: 1 A consistent firewall $f$ that consists of $n$ rules:
         $r_1, \ldots, r_n$.
       2 A query $Q$: select $F_i$ from $f$ where
         $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \wedge (\textbf{decision} = \langle dec \rangle)$
Output: Result of query $Q$.
1 $Q.result := \emptyset$;
2 for $j := 1$ to $n$ do
3     /*Let $r_j = (F_1 \in S'_1) \wedge \cdots \wedge (F_d \in S'_d) \rightarrow \langle dec' \rangle$*/
4     if $(S_1 \cap S'_1 \neq \emptyset) \wedge \cdots \wedge (S_d \cap S'_d \neq \emptyset) \wedge$
      $(\langle dec \rangle = \langle dec' \rangle)$ then
5         $Q.result := Q.result \cup (S_i \cap S'_i)$ then
6 return $Q.result$;
```

5.3 FDD-Based Firewall Query Processing Algorithm

Observe that multiple rules in a consistent firewall may share the same prefix. For example, in the consistent firewall in Fig. 3, the first three rules, namely r'_1, r'_2, and r'_3 share the same prefix $S \in [4, 7]$. Thus, if we apply the query processing algorithm in Algorithm 1 to answer a query, for instance, whose "where clause" contains the conjunct $S \in \{3\}$, over the firewall in Fig. 3, then the algorithm will repeat three times the calculation of $\{3\} \cap [4, 7]$. Clearly, if we reduce the number of these repeated calculations, the efficiency of the firewall query processing algorithm can be greatly improved.

Next, we present a more efficient firewall query processing algorithm that has no repeated calculations and can be applied to both consistent and inconsistent firewalls. The basic idea of this query processing algorithm is as follows: First, we convert the firewall (whether consistent or inconsistent) that we want to query to an equivalent firewall decision diagram. Second, because the resulting firewall decision diagram is a consistent and compact representation of the original firewall, it is used as the core data structure for query processing. We call this algorithm the *FDD-based firewall query processing algorithm*.

Here, we give a brief introduction to firewall decision diagrams [20]. A similar data structure was used by Rovniagin and Wool in [40] and by Dobkin and Lipton in [12].

Definition 1. (Firewall Decision Diagram). *A Firewall Decision Diagram (FDD) with a decision set DS and over fields $F_1, \ldots, F_d$ is an acyclic and directed graph that has the following five properties:*

1. *There is exactly one node in f that has no incoming edges. This node is called the root of f. The nodes in f that have no outgoing edges are called terminal nodes of f.*
2. *Each node v has a label, denoted $F(v)$, such that*

$$F(v) \in \begin{cases} \{F_1, \ldots, F_d\}, & \text{if } v \text{ is a nonterminal node,} \\ DS, & \text{if } v \text{ is a terminal node.} \end{cases}$$

3. *Each edge e in f has a label, denoted $I(e)$, such that if e is an outgoing edge of node v, then $I(e)$ is a nonempty subset of $D(F(v))$.*
4. *A directed path in f from the root to a terminal node is called a decision path of f. No two nodes on a decision path have the same label.*
5. *The set of all outgoing edges of a node v in f, denoted $E(v)$, satisfies the following two conditions:*
   a. *Consistency: $I(e) \cap I(e') = \emptyset$ for any two distinct edges e and e' in $E(v)$,*
   b. *Completeness: $\bigcup_{e \in E(v)} I(e) = D(F(v))$.*

We define a *full-length ordered FDD* as an FDD where in each decision path all fields appear exactly once and in the same order. For ease of presentation, as the rest of this paper only concerns full-length ordered FDDs, we use the term "FDD" to mean "full-length ordered FDD" if not otherwise specified.

Fig. 4 shows an example FDD named f_3. In this example, we assume that each packet has only two fields: S (source address) and D (destination address), and both fields have the same domain $[1, 10]$. In the rest of this paper, including this example, we use "a" as a shorthand for *accept* and "d"

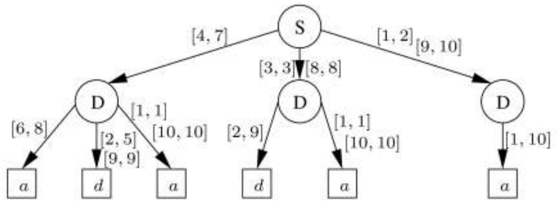

Fig. 4. Firewall decision diagram f_3.

as a shorthand for *discard*. Note that this FDD represents the firewall in Fig. 1 and also the firewall in Fig. 3.

A decision path in an FDD f is represented by $(v_1 e_1 \cdots v_k e_k v_{k+1})$ where v_1 is the root, v_{k+1} is a terminal node, and each e_i is a directed edge from node v_i to node v_{i+1}. A decision path $(v_1 e_1 \cdots v_k e_k v_{k+1})$ in an FDD defines the following rule:

$$F_1 \in S_1 \wedge \cdots \wedge F_n \in S_n \rightarrow F(v_{k+1}),$$

where

$$S_i = \begin{cases} I(e_j), & \text{if there is a node } v_j \text{ in the decision} \\ & \text{path that is labeled with field } F_i, \\ D(F_i), & \text{if no node in the decision path is} \\ & \text{labeled with field } F_i. \end{cases}$$

For an FDD f, we use S_f to denote the set of all the rules defined by all the decision paths of f. For any packet p, there is one and only one rule in S_f that p matches because of the consistency and completeness properties; therefore, f maps p to the decision of the only rule that p matches. Considering the FDD f_3 in Fig. 4, Fig. 3 shows all the six rules in S_{f_3}.

Given an FDD f, any sequence of rules that consists of all the rules in S_f is equivalent to f. The order of the rules in such a firewall is immaterial because the rules in S_f are nonoverlapping.

Given a sequence of rules, we can construct an equivalent FDD using the FDD construction algorithm in [32]. For example, the FDD generated from the firewall in Fig. 1 is shown in Fig. 4.

The algorithm for converting an inconsistent firewall f to a consistent firewall consists of two steps. First, convert f to an equivalent FDD f' by the construction algorithm in this section. Second, generate a rule for each decision path in f', i.e., obtain $S_{f'}$; then any sequence of rules that consists of all the rules in $S_{f'}$ is a consistent firewall that is equivalent to f.

The pseudocode of the FDD-based firewall query processing algorithm is shown in Algorithm 5.3. This algorithm has two inputs, an FDD and a query described by SFQL. Note that $Q.result$ is a global variable. Also note that our assumption of using full-length ordered FDDs is only for simplifying the presentation of this paper. In practice, the FDDs used for processing firewall queries do not need to be full-length or ordered. Our FDD-based firewall query processing algorithm can be easily adapted for processing queries on FDDs that are not full-length or not ordered.

This FDD-based firewall query processing algorithm works as follows. Suppose the two inputs of this algorithm are an FDD f and a query Q:

select F_i
from f
where $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \wedge (\textbf{decision} = \langle dec \rangle)$.

The algorithm starts by traversing the FDD from its root. Let F_j be the label of the root. For each outgoing edge e of the root, we compute $I(e) \cap S_j$. If $I(e) \cap S_j = \emptyset$, we skip edge e and do not traverse the subgraph that e points to. If $I(e) \cap S_j \neq \emptyset$, then we continue to traverse the subgraph that e points to in a similar fashion. Whenever a terminal node is encountered, we compare the label of the terminal node and $\langle dec \rangle$. If they are the same, assuming the rule defined by the decision path containing the terminal node is $(F_1 \in S'_1) \wedge \cdots \wedge (F_d \in S'_d) \rightarrow \langle dec' \rangle$, then add $S_i \cap S'_i$ to $Q.result$. In this pseudocode and the rest of this paper, we use $e.t$ to denote the (target) node that the edge e points to.

Algorithm 2. FDD-based Firewall Query Processing Algorithm

```
Input: (1) An FDD $f$.
       (2) A query $Q$: select $F_i$ from $f$ where
           $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \wedge (\textbf{decision} = \langle dec \rangle)$
Output: Result of query $Q$.
1 $Q.result = \emptyset$
2 CHECK($f.root$,
  $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \wedge (\textbf{decision} = \langle dec \rangle)$);
3 return $Q.result$;
4 CHECK($v, (F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \wedge$
   $(\textbf{decision} = \langle dec \rangle)$)
5 if ($v$ is a terminal node) and ($F(v) = \langle dec \rangle$) then
6     Let $(F_1 \in S'_1) \wedge \cdots \wedge (F_d \in S'_d) \rightarrow \langle dec' \rangle$ be the rule
      segment defined by the decision path containing
      node $v$;
7     $Q.result := Q.result \cup (S_i \cap S'_i)$;
8 If ($v$ is a nonterminal node)
9     /*Let $F_j$ be the label of $v$*/
10    For each edge $e$ in $E(v)$ do
11        If $I(e) \cap S_j \neq \emptyset$ then
12            CHECK($e.t, (F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \wedge$
                  $(\textbf{decision} = \langle dec \rangle)$).
```

5.4 Efficient FDD Reduction Using Hashing

To further improve the efficiency of the FDD-based firewall query processing algorithm, after we convert a firewall to an equivalent FDD, we need to reduce the size of the FDD. A full-length ordered FDD is *reduced* if and only if no two nodes are isomorphic and no two nodes have more than one edge between them. Two nodes v and v' in an FDD are *isomorphic* if and only if v and v' satisfy one of the following two conditions: 1) both v and v' are terminal nodes with identical labels; 2) both v and v' are nonterminal nodes and there is a one-to-one correspondence between the outgoing edges of v and the outgoing edges of v' such that every pair of corresponding edges have identical labels and they both point to the same node. Fig. 5 shows an FDD before reduction and Fig. 6 shows the corresponding FDD after reduction.

A brute force deep comparison algorithm for FDD reduction was proposed in [22]. In this paper, we use a more efficient FDD reduction algorithm that processes the

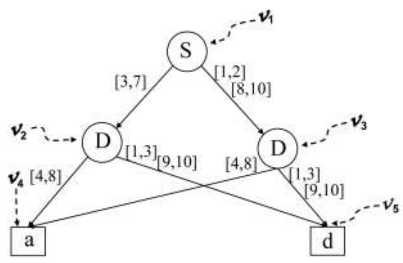

Fig. 5. A full-length ordered FDD before reduction.

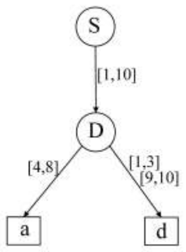

Fig. 6. A full length ordered after reduction.

nodes level by level from the terminal nodes to the root node using signatures to speed up comparisons [33].

Starting from the bottom level, at each level, we compute a signature for each node at that level. For a terminal node v, set vs signature to be its label. For a nonterminal node v, suppose v has k children $v_1, v_2, \ldots, v_k$, in increasing order of signature ($Sig(v_i) < Sig(v_{i+1})$ for $1 \leq i \leq k-1$), and the edge between v and its child v_i is labeled with E_i, a sequence of nonoverlapping prefixes in increasing order. Set the signature of node v as follows:

$$Sig(v) = h(Sig(v_1), E_1, \ldots, Sig(v_k), E_k),$$

where h is a one-way and collision resistant hash function such as MD5 [39] and SHA-1 [13]. For any such hash function h, given two different inputs x and y, the probability of $h(x) = h(y)$ is extremely small.

After we have assigned signatures to all nodes at a given level, we search for isomorphic subgraphs as follows: For every pair of nodes v_i and v_j ($1 \leq i \neq j \leq k$) at this level, if $Sig(v_i) \neq Sig(v_j)$, then we can conclude that v_i and v_j are not isomorphic; otherwise, we explicitly determine if v_i and v_j are isomorphic. If v_i and v_j are isomorphic, we delete node v_j and its outgoing edges, and redirect all the edges that point to v_j to point to v_i. Further, we eliminate double edges between node v_i and its parents.

For example, the signatures of the non-root nodes in Fig. 5 are computed as follows:

$$\begin{aligned}
&Sig(v_4) = a,\\
&Sig(v_5) = d,\\
&Sig(v_2) = h(Sig(v_4), [4,8], Sig(v_5), [1,3], [9,10]),\\
&Sig(v_3) = h(Sig(v_4), [4,8], Sig(v_5), [1,3], [9,10]).
\end{aligned}$$

Note that we can perform further optimization of removing nonterminal nodes that have only one outgoing edge in an FDD, which is similar to the path compression technique for binary tries [37].

5.5 Complexity Analysis of Firewall Query Processing Algorithms

Next, we analyze the complexity of rule-based and FDD-based firewall query processing algorithms, which shows that the FDD-based algorithm is much more efficient.

5.5.1 *Complexity of Rule-Based Firewall Query Processing Algorithm*

Given a firewall, which may be consistent or inconsistent, we first need to convert it to an equivalent consistent firewall using the algorithm in [32]. Given a firewall with n rules where each rule examines d packet fields, its equivalent consistent firewall will have $O(n^d)$ rules. As the rule-based firewall query algorithm linearly scans every rule and performs d comparison for each rule, its complexity is $O(n^{d+1})$.

5.5.2 *Complexity of FDD-Based Firewall Query Processing Algorithm*

As every nonterminal node in a reduced FDD cannot have more than $2n-1$ outgoing edges, finding the right outgoing edge to traverse takes $O(\log^n)$ time using binary search. Let k be the total number of paths that a query overlaps on an FDD, the processing time for the query is $O(kd\log^n)$. Note that k is typically small.

6 Firewall Query Postprocessing

To keep our presentation simple, we have described a somewhat watered-down version of the firewall query language where the "select" clause in a query has only one field. In fact, the "select" clause in a query can be extended to have more than one field. The results in this paper, e.g., the Firewall Query Theorem and the two firewall query processing algorithms, can all be extended in a straightforward manner to accommodate the extended "select" clauses.

However, when the "select" clause in a query has more than one field, the query result may contain many disjoint multidimensional predicates. For example, consider the following query on the firewall in Fig. 1, the FDD of which is shown in Fig. 4.

select S, D
from f_1
where$(S \in all) \wedge (D \in [2,5]) \wedge ($**decision** $= discard)$

Running the FDD-based firewall query processing algorithm, the result contains the following two predicates:

$$S \in [4,7] \wedge D \in [2,5],$$

$$S \in ([3,3] \vee [8,8]) \wedge D \in [2,5].$$

To make the query result easier for firewall administrators to read, we next present an algorithm to minimize the number of predicates generated from the firewall query engine. This algorithm consists of three steps. In the first step, we treat every predicate as a firewall rule and convert

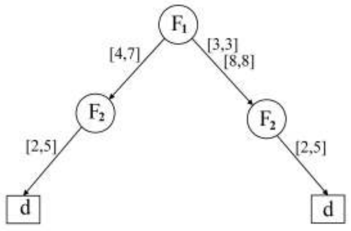

Fig. 7. A partial FDD before reduction.

these nonoverlapping rules with the same decisions to an equivalent partial FDD. A *partial FDD* is a diagram that has all the properties of an FDD except the completeness property. For example, we treat the above two predicates as the following two rules where the decision of each rule is the decision in the "where" clause

$$S \in [4,7] \wedge D \in [2,5] \rightarrow d,$$

$$S \in ([3,3] \vee [8,8]) \wedge D \in [2,5] \rightarrow d.$$

Fig. 7 shows the converted partial FDD from these two rules.

In the second step, we run the FDD reduction algorithm on the partial FDD. Essentially, this step combines some predicates together. Fig. 8 shows the reduced partial FDD.

In the third step, we generate predicates from the reduced partial FDD. The predicate that is generated from the reduced partial FDD in Fig. 8 is

$$S \in [3,8] \wedge D \in [2,5].$$

Alternately, we can simply present the reduced partial FDD as the query result.

In some cases, minimizing the number of predicates generated from the firewall query engine may not be the best way to present query results to firewall administrators. It is certainly worth investigating better ways to present query results to administrators. We leave a comprehensive study of this issue to future work.

7 Firewall Query Algebra

Similar to SQL, a complex firewall query can be formulated by the union, intersect, or minus of multiple queries. In this section, we present algorithms for processing such complex firewall queries.

7.1 Union

Performing the union of two firewall query results is simple: Combine the two sets of predicates into one set and then run the firewall query postprocessing algorithm to minimize the number of predicates.

7.2 Intersect

The intersect of two firewall query results A_1 and A_2 can be done by simply intersecting every predicate in A_1 and every predicate in A_2. More formally, $S_1 \cap S_2 = \{\mathcal{P}_1 \cap \mathcal{P}_2 | \mathcal{P}_1 \in A_1, \mathcal{P}_2 \in A_2\}$. Given two predicates $\mathcal{P}_1 = F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$ and $\mathcal{P}_2 = F_1 \in S_1' \wedge \cdots \wedge F_d \in S_d'$, $\mathcal{P}_1 \cap \mathcal{P}_2 = F_1 \in (S_1 \cap S_1') \wedge \cdots \wedge F_d \in (S_d \cap S_d')$. For example, the intersect of two predicates $S \in [3,8] \wedge D \in [2,5]$ and

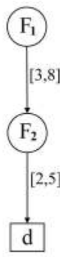

Fig. 8. A reduced partial FDD.

$S \in [6,9] \wedge D \in [4,7]$ is $S \in [6,8] \wedge D \in [4,5]$. As the predicates in any query result are nonoverlapping, for any predicate in A_1, it overlaps at most one predicate in A_2.

7.3 Minus

Computing the minus of one firewall query result from another one is more involved. Given two firewall query results A_1 and A_2, we compute $A_1 - A_2$ as follows. In the first step, we build a partial FDD from the rules formed by the predicates in A_2. In the second step, for each predicate $\mathcal{P}$ in A_1, we append a rule r formed by $\mathcal{P}$ to the partial FDD such that the resulting partial FDD is equivalent to the rule sequence that is formed by all the rules formed by the predicates in A_2 followed by the rule formed by $\mathcal{P}$. After appending r, all the new paths generated from r constitute $r - A_2$.

Next, we consider how to append rule r to this partial FDD. Suppose the partial FDD constructed from A_2 has a root v with label F_1, and v has k outgoing edges $e_1, \cdots, e_k$. Let r be the rule $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$, which is formed by a predicate in A_1.

First, we examine whether we need to add another outgoing edge to v. If $S_1 - (I(e_1) \cup \cdots \cup I(e_k)) \neq \emptyset$, we need to add a new outgoing edge with label $S_1 - (I(e_1) \cup \cdots \cup I(e_k))$ to v, because any packet whose F_1 field is an element of $S_1 - (I(e_1) \cdots \cup I(e_k))$ does not match any of the first i rules, but does match r provided that the packet satisfies $(F_2 \in S_2) \wedge \cdots \wedge (F_d \in S_d)$. We then build a decision path from $(F_2 \in S_2) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$, and make the new edge of the node v point to the first node of this decision path.

Second, we compare S_1 and $I(e_j)$ for each j where $1 \leq j \leq k$. This comparison leads to one of the following three cases:

1. $S_1 \cap I(e_j) = \emptyset$: In this case, we skip edge e_j because any packet whose value of field F_1 is in set $I(e_j)$ does not match r.
2. $S_1 \cap I(e_j) = I(e_j)$: In this case, for a packet whose value of field F_1 is in set $I(e_j)$, it may match one of the first i rules, and it may also match rule r. So, we append the rule $(F_2 \in S_2) \wedge \cdots \wedge (F_d \in S_d) \rightarrow \langle decision \rangle$ to the subgraph rooted at the node that e_j points to.
3. $S_1 \cap I(e_j) \neq \emptyset$ and $S_1 \cap I(e_j) \neq I(e_j)$: In this case, we split edge e into two edges: e' with label $I(e_j) - S_1$ and e'' with label $I(e_j) \cap S_1$. Then, we make two copies of the subgraph rooted at the node that e_j points to, and let e' and e'' point to one copy each.

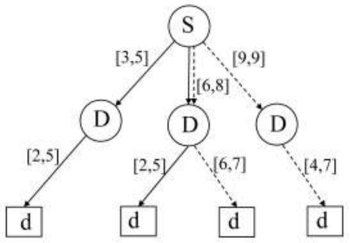

Fig. 9. A partial FDD.

We then deal with e' using the first case, and e'' using the second case.

We next show an example of computing the minus on firewall query results. Let $A_1 = \{S \in [6,9] \wedge D \in [4,7]\}$ and $A_2 = \{S \in [3,8] \wedge D \in [2,5]\}$. To compute $A_1 - A_2$, we first construct a partial FDD from A_2, which is shown in Fig. 8. Second, we append $S \in [6,9] \wedge D \in [4,7]$ to this partial FDD. The resulting FDD is shown in Fig. 9. The dashed paths represent the result of $A_1 - A_2$.

8 Experimental Results

In this section, we evaluate the efficiency of our firewall query processing algorithms by the average execution time of each algorithm versus the total number of rules in the original inconsistent firewalls. In the absence of publicly available firewalls, we create synthetic firewalls according to the characteristics of real-life packet classifiers discovered in [3], [23]. Note that a firewall is also a packet classifier. Each rule has the following five fields: interface, source IP address, destination IP address, destination port number, and protocol type. The programs are implemented in SUN Java JDK 1.4. The experiments were carried out on a SunBlade 2,000 machine running Solaris 9 with 1Ghz CPU and 1 GB of memory.

Fig. 10 shows the average execution time of the rule-based firewall query processing algorithm and the FDD-based firewall query processing algorithm versus the total number of rules in the original inconsistent firewalls. In Fig. 10, the horizontal axis indicates the total number of rules in the original inconsistent firewalls, and the vertical axis indicates the average execution time (in milliseconds) for processing a firewall query. Note that in Fig. 10, the execution time of the FDD-based firewall query processing algorithm does not include the FDD construction time because the conversion from a firewall to an equivalent FDD is performed only once for each firewall, not for each query. Similarly, the execution time of the rule-based firewall query processing algorithm does not include the time for converting an inconsistent firewall to an equivalent consistent firewall because this conversion is performed only once for each firewall, not for each query. Recall that the procedure for converting an inconsistent firewall to a consistent firewall consists of two steps: first, construct an equivalent FDD from the original inconsistent firewall; second, generate one rule for each decision path of the FDD, and any sequence of all the rules defined by the decision paths of the FDD constitutes the final consistent firewall.

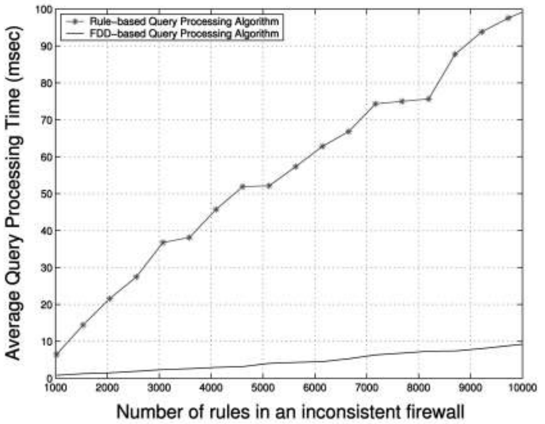

Fig. 10. Query processing time versus number of rules.

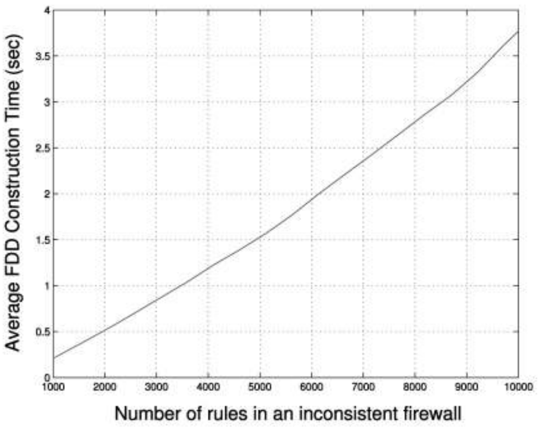

Fig. 11. FDD construction time versus number of rules.

From Fig. 11, we can see that the FDD-based firewall query processing algorithm is much more efficient than the rule-based firewall query processing algorithm. For example, for processing a query over an inconsistent firewall that has 10,000 rules, the FDD-based query processing algorithm uses about 10 milliseconds, while the rule-based query processing algorithm uses about 100 milliseconds. The experimental results in Fig. 11 confirm our analysis that the FDD-based query processing algorithm saves execution time by reducing repeated calculations.

Fig. 11 shows the average execution time for constructing an equivalent FDD from an inconsistent firewall. In Fig. 11, the horizontal axis indicates the total number of rules in the original inconsistent firewalls, and the vertical axis indicates the average execution time (in seconds) for constructing an equivalent FDD from an inconsistent firewall. From Fig. 11, we can see that the FDD construction algorithm is very efficient. It takes less than 4 seconds to construct an equivalent FDD from an inconsistent firewall that has up to 10,000 rules. Thus, if one intends to run more than 40-50 queries on the same firewall, then using the FDD-based algorithm, even including the cost of building the FDD, is more efficient than the simple linear search.

9 Concluding Remarks

We make a number of contributions in this paper. First, we introduce a simple and effective SQL-like query language, which is called the Structured Firewall Query Language, for describing firewall queries. Second, we present a theorem, the Firewall Query Theorem, as the foundation for developing firewall query processing algorithms. Third, we present an efficient query processing algorithm that uses firewall decision diagrams as its core data structure. Our experimental results show that this query processing algorithm is very efficient. Fourth, we present methods for optimizing firewall query results. At last, we present methods for performing the union, intersect, and minus operations on firewall query results.

Acknowledgments

The authors thank the anonymous reviewers for their constructive comments and suggestions on improving the presentation of this work. The work of Alex X. Liu is supported in part by the US National Science Foundation (NSF) under Grant No. CNS-0716407. The work of Mohamed G. Gouda is supported by the NSF under Grant No. 0520250.

References

[1] ipchains, http://www.tldp.org/howto/ipchains-howto.html, 2009.
[2] E. Al Shaer and H. Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," *Proc. IEEE INFOCOM '04,* Mar. 2004.
[3] F. Baboescu, S. Singh, and G. Varghese, "Packet Classification for Core Routers: Is There an Alternative to CAMs?" *Proc. IEEE INFOCOM,* 2003.
[4] F. Baboescu and G. Varghese, "Fast and Scalable Conflict Detection for Packet Classifiers," *Proc. 10th IEEE Int'l Conf. Network Protocols,* 2002.
[5] Y. Bartal, A.J. Mayer, K. Nissim, and A. Wool, "Firmato: A Novel Firewall Management Toolkit," *Proc. IEEE Symp. Security and Privacy,* pp. 17-31, 1999.
[6] Y. Bartal, A.J. Mayer, K. Nissim, and A. Wool, "Firmato: A Novel Firewall Management Toolkit," *ACM Trans. Computer Systems,* vol. 22, no. 4, pp. 381-420, 2004.
[7] CERT, Test the Firewall System, http://www.cert.org/security-improvement/practices/p060.html, 2009.
[8] CERT Coordination Center, http://www.cert.org/advisories/ca-2003-20.html, Aug. 2003.
[9] CheckPoint FireWall-1, http://www.checkpoint.com/, Mar. 2005.
[10] Cisco PIX 500 Series Firewalls, http://www.cisco.com/en/us/products/hw/vpndevc/ps2030/, Nov. 2003.
[11] D. Moore et al., http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html, 2003.
[12] D. Dobkin and R.J. Lipton, "Multidimensional Searching Problems," *SIAM J. Computing,* vol. 5, no. 2, pp. 181-186.
[13] D. Eastlake and P. Jones, "Us Secure Hash Algorithm 1 (SHA-1)," *RFC 3174,* 2001.
[14] D. Eppstein and S. Muthukrishnan, "Internet Packet Filter Management and Rectangle Geometry," *Proc. Symp. Discrete Algorithms,* pp. 827-835, 2001.
[15] P. Eronen and J. Zitting, "An Expert System for Analyzing Firewall Rules," *Proc. Sixth Nordic Workshop Secure IT Systems (NordSec '01),* pp. 100-107, 2001.
[16] D. Farmer and W. Venema, *Improving the Security of Your Site by Breaking into It,* http://www.alw.nih.gov/Security/Docs/admin-guide-to-cracking.101.html, 1993.
[17] M. Frantzen, F. Kerschbaum, E. Schultz, and S. Fahmy, "A Framework for Understanding Vulnerabilities in Firewalls Using a Dataflow Model of Firewall Internals," *Computers and Security,* vol. 20, no. 3, pp. 263-270, 2001.
[18] M. Freiss, *Protecting Networks with SATAN.* O'Reilly & Assoc., Inc., 1998.
[19] M. Gouda, A.X. Liu , and M. Jafry, "Verification of Distributed Firewalls," *Proc. IEEE GLOBECOM,* 2008.
[20] M.G. Gouda and A.X. Liu, "Firewall Design: Consistency, Completeness and Compactness," *Proc. 24th IEEE Int'l Conf. Distributed Computing Systems (ICDCS '04),* pp. 320-327, 2004.
[21] M.G. Gouda and A.X. Liu, "A Model of Stateful Firewalls and its Properties," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '05),* pp. 320-327, June 2005.
[22] M.G. Gouda and A.X. Liu, "Structured Firewall Design," *Computer Networks J.,* vol. 51, no. 4 pp. 1106-1120, Mar. 2007.
[23] P. Gupta, "Algorithms for Routing Lookups and Packet Classification," PhD thesis, Stanford Univ., 2000.
[24] J.D. Guttman, "Filtering Postures: Local Enforcement for Global Policies," *Proc. IEEE Symp. Security and Privacy,* pp. 120-129, 1997.
[25] A. Hari, S. Suri, and G.M. Parulkar, "Detecting and Resolving Packet Filter Conflicts," *Proc. IEEE INFOCOM '00,* pp. 1203-1212, 2000.
[26] S. Hazelhurst, A. Attar, and R. Sinnappan, "Algorithms for Improving the Dependability of Firewall and Filter Rule Lists," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '00),* pp. 576-585, 2000.
[27] J. Hwang, T. Xie, F. Chen, and A.X. Liu, "Systematic Structural Testing of Firewall Policies," *Proc. 27th IEEE Int'l Symp. Reliable Distributed Systems (SRDS),* 2008.
[28] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, and M. Frantzen, "Analysis of Vulnerabilities in Internet Firewalls," *Computers and Security,* vol. 22, no. 3, pp. 214-232, 2003.
[29] A.X. Liu, "Change-Impact Analysis of Firewall Policies," *Proc. 12th European Symp. Research Computer Security (ESORICS '07),* pp. 155-170, Sept. 2007.
[30] A.X. Liu, "Firewall Policy Verification and Troubleshooting," *Proc. IEEE Int'l Conf. Comm. (ICC '08),* May 2008.
[31] A.X. Liu and M.G. Gouda, "Complete Redundancy Detection in Firewalls," *Proc. 19th Ann. IFIP Conf. Data and Applications Security,* pp. 196-209, Aug. 2005.
[32] A.X. Liu and M.G. Gouda, "Diverse Firewall Design," *IEEE Trans. Parallel and Distributed Systems,* to be published.
[33] A.X. Liu, C.R. Meiners, and E. Torng, "TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs," *IEEE/ACM Trans. Networking,* to be published.
[34] A. Mayer, A. Wool, and E. Ziskind, "Fang: A Firewall Analysis Engine," *Proc. IEEE Symp. Security and Privacy,* pp. 177-187, 2000.
[35] A. Mayer, A. Wool, and E. Ziskind, "Offline Firewall Analysis," *Int'l J. Information Security,* vol. 5, no. 3, pp. 125-144, 2005.
[36] J.D. Moffett and M.S. Sloman, "Policy Conflict Analysis in Distributed System Management," *J. Organizational Computing,* vol. 4, no. 1, pp. 1-22, 1994.
[37] D.R. Morrison, "Patricia Practical Algorithm to Retrieve Information Coded in Alphanumeric," *J. ACM,* vol. 15, no. 4, pp. 514-534, 1968.
[38] Nessus, http://www.nessus.org/, Mar. 2004.
[39] R. Rivest, "The MD5 Message-Digest Algorithm," *RFC 1321,* 1992.
[40] D. Rovniagin and A. Wool, "The Geometric Efficient Matching Algorithm for Firewalls," *Proc. 23rd IEEE Convention of Electrical and Electronics Eng. in Israel (IEEEI),* pp. 153-156, http://www.eng.tau.ac.il/~yash/ees2003-6.ps, 2004.
[41] A.D. Rubin, D. Geer, and M.J. Ranum, *Web Security Sourcebook,* first ed. Wiley Computer Publishing, 1997.
[42] A. Wool, "Architecting the Lumeta Firewall Analyzer," *Proc. 10th USENIX Security Symp.,* pp. 85-97, Aug. 2001.
[43] A. Wool, "A Quantitative Study of Firewall Configuration Errors," *Computer,* vol. 37, no. 6, pp. 62-67, June 2004.
[44] A. Wool, "The Use and Usability of Direction-Based Filtering in Firewalls," *Computers & Security,* vol. 23, no. 6, pp. 459-468, 2004.
[45] J. Xu and M. Singhal, "Design and Evaluation of a High-Performance ATM Firewall Switch and Its Applications," *IEEE J. Selected Areas in Comm.,* vol. 17, no. 6, pp. 1190-1200, 1999.
[46] J. Xu and M. Singhal, "Design of a High-Performance ATM Firewall," *ACM Trans. Information and System Security,* vol. 2, no. 3, pp. 269-294, 1999.
[47] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, "Fireman: A Toolkit for Firewall Modeling and Analysis," *Proc. IEEE Symp. Security and Privacy,* May 2006.

Alex X. Liu received the PhD degree in computer science from The University of Texas at Austin in 2006. He is currently an assistant professor in the Department of Computer Science and Engineering at Michigan State University. He won the 2004 IEEE & IFIP William C. Carter Award, the 2004 National Outstanding Overseas Students Award sponsored by the Ministry of Education of China, the 2005 George H. Mitchell Award for Excellence in Graduate Research in the University of Texas at Austin, and the 2005 James C. Browne Outstanding Graduate Student Fellowship in the University of Texas at Austin. His research interests include computer and network security, dependable and high-assurance computing, applied cryptography, computer networks, operating systems, and distributed computing. He is a member of the IEEE.

Mohamed G. Gouda received the BSc degree in engineering and in mathematics from Cairo University, the MA degree in mathematics from York University, and the master's and PhD degrees in computer science from the University of Waterloo. He was with the Honeywell Corporate Technology Center at Minneapolis from 1977 to 1980. In 1980, he joined The University of Texas at Austin, where he currently holds the Mike A. Myers Centennial professorship in computer sciences. He is the 1993 winner of the Kuwait Award in Basic Sciences. He won the 2001 IEEE Communication Society William R. Bennet Best Paper Award for his paper "Secure Group Communications Using Key Graphs," coauthored with C.K. Wong and S.S. Lam, and published in the *IEEE/ACM Transactions on Networking* (vol. 8, no. 1, pp. 16-30). In 2004, his paper "Diverse Firewall Design," coauthored with Alex X. Liu and published in the Proceedings of the International Conference on Dependable Systems and Networks, won the William C. Carter award. He is a member of the IEEE.