

Privacy Preserving Collaborative Enforcement of Firewall Policies in Virtual Private Networks¹

Alex X. Liu

Fei Chen

Abstract—The widely deployed Virtual Private Network (VPN) technology allows roaming users to build an encrypted tunnel to a VPN server, which henceforth allows roaming users to access some resources as if that computer were residing on their home organization’s network. Although VPN technology is very useful, it imposes security threats on the remote network because its firewall does not know what traffic is flowing inside the VPN tunnel. To address this issue, we propose VGuard, a framework that allows a policy owner and a request owner to collaboratively determine whether the request satisfies the policy without the policy owner knowing the request and the request owner knowing the policy. We first present an efficient protocol, called Xhash, for oblivious comparison, which allows two parties, where each party has a number, to compare whether they have the same number, without disclosing their numbers to each other. Then, we present the VGuard framework that uses Xhash as the basic building block. The basic idea of VGuard is to first convert a firewall policy to non-overlapping numerical rules and then use Xhash to check whether a request matches a rule. Comparing with the Cross-Domain Cooperative Firewall (CDCF) framework, which represents the state-of-the-art, VGuard is not only more secure but also orders of magnitude more efficient. On real-life firewall policies, for processing packets, our experimental results show that VGuard is three to four orders of magnitude faster than CDCF.

Index Terms—Virtual Private Networks, Privacy, Network Security.

1 INTRODUCTION

1.1 Background and Motivation

VIRTUAL Private Network (VPN) is a widely deployed technology that allows roaming users to securely use a remote computer on the public Internet as if that computer were residing on their organization’s network, which henceforth allows roaming users to access some resources that are only accessible from their organization’s network. VPN works in the following manner. Suppose IBM sends a field representative to one of its customers, say Michigan State University (MSU). Assume that MSU’s IP addresses are in the range 1.1.0.0 ~ 1.1.255.255 and IBM’s IP addresses are in the range 2.2.0.0 ~ 2.2.255.255. To access resources (say a confidential customer database server with IP address 2.2.0.2) that are only accessible within IBM’s network, the IBM representative uses an MSU computer (or his laptop) with an MSU IP address (say 1.1.0.10) to establish a secure VPN tunnel to the VPN server (with IP address 2.2.0.1) in IBM’s network. Upon establishing the VPN tunnel, the IBM representative’s computer is temporarily assigned a virtual IBM IP address (say 2.2.0.25). Using the VPN tunnel, the IBM representative can access any computer on the Internet as if his computer were residing on IBM’s network with IP address 2.2.0.25. The payload of each packet inside the VPN tunnel is

another packet (to or from the newly assigned IBM IP address 2.2.0.25), which is typically encrypted. Fig. 1 illustrates an example packet that traverses from the IBM representative’s computer on MSU’s network to the customer database server in IBM’s network.

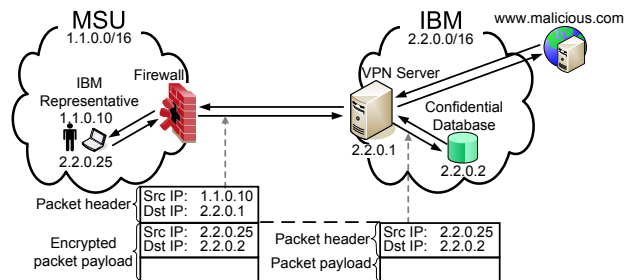


Fig. 1. A typical VPN example

While the VPN tunnel is very useful for the IBM representative, it imposes security threats on MSU’s network because MSU’s firewall does not know what traffic is flowing inside the VPN tunnel. For example, if MSU’s firewall blocks access to a remote site (say *www.malicious.com*) or disallows machines to run peer-to-peer applications due to copyright concerns, MSU’s firewall cannot enforce its policies on the IBM representative’s computer although that computer is physically on MSU’s network. Thus, the VPN tunnel opens a hole to MSU’s firewall that may allow unwanted traffic to flow in and out. Having such a hole is very dangerous because viruses or worms could flood in through it to the IBM representative’s computer first and then further spread to other computers on MSU’s network.

1.2 Technical Challenges

This problem is technically challenging. First, MSU cannot simply block VPN connections because otherwise

• Alex X. Liu and Fei Chen are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, 48824. E-mail: {alexliu, feichen}@cse.msu.edu

1. The preliminary version of this paper titled “Collaborative Enforcement of Firewall Policies in Virtual Private Networks” was published in proceedings of the Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), pages 95-104, Canada, August 2008.

the IBM representative may fail to perform his duties. Second, MSU cannot share its firewall policy with IBM. Firewall policies are typically kept confidential due to security and privacy concerns. Knowing the firewall policy of a network could allow attackers to easily spot the security holes in the policy and launch corresponding attacks. A firewall policy also reveals the IP addresses of important servers, which are usually kept confidential to reduce the chance of being attacked. Furthermore, from a firewall policy, one may derive the business relationship of the organization with their partners. Third, IBM cannot share the traffic in its VPN tunnel with MSU due to security and privacy concerns. For example, IBM may want to keep the IP address of its customer database server confidential to reduce the likelihood of being attacked. One main purpose of VPN is to achieve such confidentiality.

The fundamental problem in the above application is: *how can we collaboratively enforce firewall policies in a privacy preserving manner for VPN tunnels in an open distributed environment?* A satisfactory solution to this problem should meet the following three requirements: (1) The request owner cannot gain any more knowledge on the policy after any number of runs of the protocol than they would by brute force probing of the policy. We refer to this requirement as *policy privacy*. (2) It should be computationally infeasible for the policy owner to reveal a request. We refer to this requirement as *request privacy*. (3) The overhead of the solution should be marginal. Timely processing of every request (or packet) is critical for distributed applications. We refer to this requirement as *protocol efficiency*. Throughout this paper, we use “MSU” to represent the policy owner and “IBM” to represent the request owner.

1.3 Limitations of Prior Art

Although this is a fundamentally important problem, it is largely underinvestigated. The state-of-the-art on this problem is the seminal work in [9], where Cheng *et al.* proposed a scheme called CDCF. However, CDCF is vulnerable to selective policy updating attacks, by which the policy owner can quickly reveal the request of the other party. Furthermore, CDCF is inefficient because it uses commutative encryption functions (such as the Pohlig-Hellman Exponentiation Cipher [25] and Secure RPC Authentication (SRA) [28]), which are extremely expensive in nature, as the core cryptography primitive.

1.4 Our Solution

In this paper, we present VGuard, a secure and efficient framework for collaborative enforcement of firewall policies. In VGuard, different from CDCF, the policy owner does not know which rule matches which request; thus, it makes the selective policy updating attacks infeasible. Furthermore, unlike CDCF, VGuard obfuscates rule decisions, which prevents MSU from knowing the decision for the given packet. To make VGuard efficient, we propose a new oblivious comparison scheme, called Xhash, which uses XOR and secure hash functions. Xhash is

three orders of magnitude faster than the commutative encryption scheme used in CDCF. Moreover, VGuard uses decision diagrams to process packets, which is much faster than the linear search used in CDCF. By side by side comparison, our experimental results show that VGuard is 552 times faster than CDCF on MSU side and 5035 times faster than CDCF on IBM side.

1.5 Key Contributions

We make the following three key contributions in this paper. First, we propose Xhash, a very efficient oblivious comparison scheme that simply uses XOR and secure hash functions. Second, we propose VGuard, a privacy preserving framework for collaborative enforcement of firewall policies. Third, we implement both VGuard and CDCF and perform extensive experiments to evaluate their performance.

2 THREAT MODEL

First, we assume that the two parties of policy owner MSU and request owner IBM are semi-honest; that is, they follow the preestablished VGuard protocol, but the policy owner may attempt to reveal the request and the request owner may attempt to reveal the policy. In particular, the enforcement party IBM does enforce the decision made by MSU. The assumption that the two parties follow the VGuard protocol can be realized by the service level agreement between MSU and IBM. Furthermore, we assume that neither MSU nor IBM has the computational power to break secure hash functions such as HMAC-MD5 or HMAC-SHA1 [12], [19], [26]. Second, we assume that there exists a third party that facilitates the execution of our protocol. This third party shares a secret key with MSU. We assume that this third party follows our protocol and will collude with neither MSU nor IBM. Third, we assume that between any two of the three parties, MSU, IBM, and the third party, there exists a reliable and secure channel. These channels can be established using protocols such as SSL. Our VGuard protocol runs inside these channels. Thus, we do not consider the network level attacks on the communication channels that VGuard is built upon.

3 BACKGROUND

We first formally define the concepts of fields, packets, and firewalls. A *field* F_i is a variable of finite length (i.e., of a finite number of bits). The domain of field F_i of w bits, denoted $D(F_i)$, is $[0, 2^w - 1]$. A *packet* over the d fields F_1, \dots, F_d is a d -tuple (p_1, \dots, p_d) where each p_i ($1 \leq i \leq d$) is an element of $D(F_i)$. Firewalls usually check the following five fields: source IP address, destination IP address, source port number, destination port number, and protocol type. The lengths of these packet fields are 32, 32, 16, 16, and 8 respectively. 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 $|\Sigma| = |D(F_1)| \times \dots \times |D(F_d)|$, where $|\Sigma|$ denotes the number of elements in set Σ and $|D(F_i)|$ denotes the number of elements in set $D(F_i)$.

A 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 , and is specified as $F_1 \in S_1 \wedge \dots \wedge F_d \in S_d$ where each S_i is a subset of $D(F_i)$ and is specified as either a prefix or a range. A *prefix* $\{0,1\}^k\{*\}^{w-k}$ with k leading 0s or 1s for a packet field of length w denotes the range $[\{0,1\}^k\{0\}^{w-k}, \{0,1\}^k\{1\}^{w-k}]$. For example, prefix 01^{**} denotes the range $[0100, 0111]$. A rule $F_1 \in S_1 \wedge \dots \wedge F_d \in S_d \rightarrow \langle decision \rangle$ is a *prefix rule* if and only if each S_i is represented as a prefix. In firewall rules, source IP addresses, destination IP addresses, and protocol types are typically specified as prefixes, and source ports and destination ports are typically specified as ranges. A packet (p_1, \dots, p_d) matches a predicate $F_1 \in S_1 \wedge \dots \wedge F_d \in S_d$ and the corresponding rule if and only if the condition $p_1 \in S_1 \wedge \dots \wedge p_d \in S_d$ holds. For firewalls, the typical decisions include permit, deny, permit with logging, and deny with logging. A rule $F_1 \in S_1 \wedge \dots \wedge F_d \in S_d \rightarrow \langle decision \rangle$ is called a singleton rule if and only if $|S_i| = 1$ for every i .

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. To ensure that a sequence of rules is complete and thus is a firewall, the predicate of the last rule is usually specified as $F_1 \in D(F_1) \wedge \dots \wedge F_d \in D(F_d)$. A *firewall* is a sequence of rules that is complete. Two rules in a firewall may overlap; that is, there exists at least one packet that matches both rules. Furthermore, two rules in a firewall may conflict; that is, the two rules not only overlap but also have different decisions. Firewalls typically resolve conflicts by employing 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 the firewall. Table 1 shows an example firewall. The format of these rules is based upon the format used in Cisco Access Control Lists.

Rule	Src. IP	Dest. IP	Src. Port	Dest. Port	Prot.	Action
r_1	1.2.*.*	192.168.0.1	*	25	TCP	accept
r_2	*	*	*	*	*	discard

TABLE 1
An example firewall

4 OBLIVIOUS COMPARISON

In this section, we consider the following *oblivious comparison* problem. Suppose we have two parties, denoted MSU and IBM, where MSU has a private number N_1 and IBM has a private number N_2 . MSU wants to compare whether $N_1 = N_2$; however, neither MSU nor IBM wants to disclose its number to others. If $N_1 \neq N_2$, no party should learn the value of the other party. This is a technically challenging problem because MSU needs to have some information about N_2 to enable the comparison; yet, such information about N_2 should not allow MSU to reveal the value of N_2 . Next, we introduce the concept of oblivious comparison functions and an oblivious comparison protocol based on such functions.

Oblivious Comparison Functions: Two functions f_1 and f_2 are called a pair of oblivious comparison functions if and only if they satisfy the following four properties:

- 1) Secrecy: Neither $f_1(x, K)$ nor $f_2(x, K)$ reveals the values of x and K .
- 2) Nondeducibility: Given x and $f_2(x, K)$, it is computationally infeasible to compute K .
- 3) Commutativity: For any x, K_1, K_2 , we have $f_2(f_1(x, K_1), K_2) = f_2(f_1(x, K_2), K_1)$.
- 4) Distinguishability: For any x, y , and K , if $x \neq y$, then we have $f_1(x, K) \neq f_1(y, K)$ and $f_2(x, K) \neq f_2(y, K)$.

Here f_1 is called the *inner oblivious comparison function* and f_2 is called the *outer oblivious comparison function*. We discuss the construction of f_1 and f_2 later.

Oblivious Comparison Protocol: Assuming that we have a pair of oblivious comparison functions f_1 and f_2 , MSU and IBM can achieve oblivious comparison in the following three steps. Assume that MSU has a secret key K_1 and IBM has a secret key K_2 . First, MSU computes $f_1(N_1, K_1)$ and sends the result to IBM. Because of the secrecy property of f_1 , IBM cannot reveal the values of N_1 and K_1 . Second, after receiving $f_1(N_1, K_1)$ from MSU, IBM computes $f_2(f_1(N_1, K_1), K_2)$ and sends the result to MSU. Because of the nondeducibility property of f_2 , MSU cannot compute the value of IBM's secret key K_2 . Third, IBM computes $f_1(N_2, K_2)$ and sends the result to MSU. Because of the secrecy property of f_1 , from $f_1(N_2, K_2)$, MSU cannot reveal the values of N_2 and K_2 . After receiving $f_1(N_2, K_2)$ from IBM, MSU computes $f_2(f_1(N_2, K_2), K_1)$ and compares the result with $f_2(f_1(N_1, K_1), K_2)$, which was received from IBM in the second step. Because of the commutativity and distinguishability properties of f_1 and f_2 , $N_1 = N_2$ if and only if $f_2(f_1(N_1, K_1), K_2) = f_2(f_1(N_2, K_2), K_1)$. Fig. 2 shows the oblivious comparison protocol.

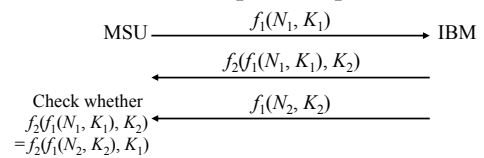


Fig. 2. The oblivious comparison protocol

The Xhash Protocol: We propose a simple and efficient protocol, called Xhash, to achieve oblivious comparison. Xhash works as follows. First, MSU sends $N_1 \oplus K_1$ to IBM. Then, IBM computes $HMAC_k(N_1 \oplus K_1 \oplus K_2)$ and sends the result to MSU. Second, IBM sends $N_2 \oplus K_2$ to MSU. Third, MSU computes $HMAC_k(N_2 \oplus K_2 \oplus K_1)$ and compares it with $HMAC_k(N_1 \oplus K_1 \oplus K_2)$, which was received from IBM. Finally, the condition $N_1 = N_2$ holds if and only if $HMAC_k(N_2 \oplus K_2 \oplus K_1) = HMAC_k(N_1 \oplus K_1 \oplus K_2)$. Fig. 3 illustrates the Xhash protocol.

The above function HMAC is a keyed-Hash Message Authentication Code, such as HMAC-MD5 or HMAC-SHA1, which satisfies the one-wayness property (i.e., given $HMAC_k(x)$, it is computationally infeasible to compute x and k) and the collision resistance property (i.e., it is computationally infeasible to find two distinct

numbers x and y such that $HMAC_k(x) = HMAC_k(y)$. Note that the key k is shared between MSU and IBM. Although hash collisions for HMAC do exist in theory, the probability of collision is negligibly small in practice. Furthermore, by properly choosing the shared key k , we can safely assume that HMAC has no collision.

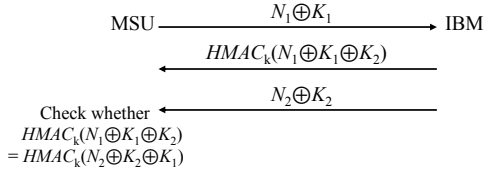


Fig. 3. The Xhash protocol

To prevent brute force attacks, we need to choose key K to be sufficiently long. In our implementation, we choose K to be 128 bits. Note that in our framework x is at most 38 bits. To meet the length of K such that x can be XORed with K , we first use a pseudo random generation function R to generate $x_1 = R(x)$. Second, we apply R to x_1 to generate $x_2 = R(x_1)$. Repeat this process until we can concatenate x, x_1, x_2, \dots to form a bit string that meets the length of K . Extra bits in the concatenation beyond the length of K are discarded.

The correctness of Xhash follows from the commutative property of XOR operation (i.e., $x \oplus K_1 \oplus K_2 = x \oplus K_2 \oplus K_1$) and the one-wayness and collision resistance properties of HMAC functions.

Nondeducibility Property of f_1 : Note that if f_1 does not satisfy the nondeducibility property, when $N_1 = N_2$, MSU is able to compute K_2 because MSU knows both $f_1(N_2, K_2)$ and N_2 . This is fine if MSU and IBM only want to compare two numbers where K_2 will be used only once. However, in VGuard, MSU and IBM need to compare MSU's firewall with all the packets in the VPN tunnel rather than comparing two numbers. IBM will apply f_1 to all the packets with its key K_2 . In this case, as long as MSU reveals K_2 , it can compute the plaintext of all these packets. To address this issue, we have two options. The first option is that we can introduce a third party to prevent MSU from knowing $f_1(N_2, K_2)$ such that MSU cannot reveal K_2 . The second option is that instead of introducing the third party, we find a function f_1 that satisfies the nondeducibility property. To our best knowledge, the only function that satisfies the nondeducibility property and the four properties of oblivious comparison functions is the commutative encryption function such as the Pohlig-Hellman Exponentiation Cipher [25]. A commutative encryption function satisfies the following four properties, where $(x)_K$ denotes the encryption of x using key K : (1) Given x and $(x)_K$, it is computationally infeasible to compute the value of K . (2) Given x, K_1 , and K_2 , we have $((x)_{K_1})_{K_2} = ((x)_{K_2})_{K_1}$. (3) Given x, y , and K , if $x \neq y$, we have $(x)_K \neq (y)_K$. (4) Given K , $(x)_K$ can be decrypted in polynomial time. However, such commutative encryption functions are computationally too expensive. Thus, we choose the first option in our VGuard framework. We defer the discussion of preventing MSU from knowing K_2 in Section 6.

5 BOOTSTRAPPING PROTOCOL

In the bootstrapping protocol, MSU first converts its firewall policy to a set of non-overlapping prefix rules. Second, MSU converts each prefix to a number. Third, MSU applies an XOR operation to every number using its secret key K_1 . Finally, MSU sends the anonymized policy to IBM. IBM then applies XOR and HMAC operations to every number in the received policy using its secret key K_2 , obfuscates the decision of each rule, and shuffle the resulting rules. To complete the process, IBM sends the resulting policy back to MSU. Fig. 4 illustrates the bootstrapping protocol.

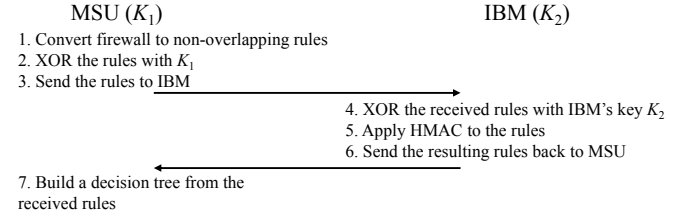


Fig. 4. The bootstrapping protocol

Converting a firewall policy to a set of non-overlapping prefix rules consists of four steps: FDD construction, range conversion, prefix numericalization, and rule generation.

5.1 FDD Construction

In this step, MSU converts its firewall policy to an equivalent *Firewall Decision Diagram* [15]. A *Firewall Decision Diagram* (FDD) with a decision set DS and over fields F_1, \dots, F_d is an acyclic and directed graph that has the following five properties: (1) There is exactly one node that has no incoming edges. This node is called the *root*. The nodes that have no outgoing edges are called *terminal* nodes. (2) Each node v has a label, denoted $F(v)$. If v is a nonterminal node, then $F(v) \in \{F_1, \dots, F_d\}$. If v is a terminal node, then $F(v) \in DS$. (3) Each edge $e:u \rightarrow v$ is labeled with a nonempty set of integers, denoted $I(e)$, where $I(e)$ is a subset of the domain of u 's label (i.e., $I(e) \subseteq D(F(u))$). (4) A directed path from the root to a terminal node is called a *decision path*. No two nodes on a decision path have the same label. (5) The set of all outgoing edges of a node v , 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*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$. Fig. 5(a) shows an example firewall policy over two fields F_1 and F_2 , where the domain of each field is $[0, 15]$. The FDD that is semantically equivalent to this firewall policy is shown in Fig. 5(b). Note that in labeling terminal nodes, we use "a" as a shorthand for "accept" (i.e., "permit") and "d" as a shorthand for "discard" (i.e., "deny"). The algorithm for converting a firewall to an FDD is in [21].

5.2 Range Conversion

For every edge e in the FDD, MSU converts its label $I(e)$ to the minimum set of prefixes whose union is equal to $I(e)$. As one prefix can be converted to one range, a range

may be converted to multiple prefixes. In converting a range to prefixes, we want to find the minimum set of prefixes such that the union of the prefixes is equal to the range. For example, given range $[0001, 1110]$, the corresponding minimum set of prefixes would be $0001, 001*, 01**, 10**, 110*, 1110$. The minimum number of prefixes for representing an integer interval $[a, b]$, where a and b are two numbers of w bits, is at most $2w-2$ [16]. We call such FDDs, where each edge is labeled by a set of prefixes, *prefix FDDs*. Fig. 5(c) shows the prefix FDD converted from the FDD in Fig. 5(b).

5.3 Prefix Numericalization

In this step, MSU converts each prefix in the FDD to a concrete number. This process is called *prefix numericalization*. A prefix numericalization function f needs to satisfy the following two properties: (1) for any prefix \mathcal{P} , $f(\mathcal{P})$ is a binary string; (2) for any two prefixes \mathcal{P}_1 and \mathcal{P}_2 , $f(\mathcal{P}_1) = f(\mathcal{P}_2)$ if and only if $\mathcal{P}_1 = \mathcal{P}_2$. There are many ways to do prefix numericalization. We use the prefix numericalization scheme used in [6]. Given a prefix $b_1b_2 \dots b_k * \dots *$ of w bits, we first insert 1 after b_k . The bit 1 represents a separator between $b_1b_2 \dots b_k$ and $* \dots *$. Second, we replace every $*$ by 0. Note that if there is no $*$ in a prefix, we add 1 at the end of this prefix. For example, $101*$ is converted to 10110 . After prefix numericalization, the FDD in Fig. 5(c) becomes the one in Fig. 5(d).

5.4 Applying XOR by MSU

After prefix numericalization, MSU applies XOR to every number in the numericalized FDD using its secret key K_1 . Fig. 5(e) shows the numericalized and XORed FDD. Then, MSU generates non-overlapping rules from the numericalized and XORed FDD. From each decision path in the FDD, MSU generates a set of non-overlapping rules. For example, from the left-most decision path in Fig. 5(e), MSU generates the following four non-overlapping rules:

$$\begin{aligned} F_1 \in 01100 \oplus K_1 \wedge F_2 \in 00100 \oplus K_1 &\rightarrow a, \\ F_1 \in 01100 \oplus K_1 \wedge F_2 \in 01010 \oplus K_1 &\rightarrow a, \\ F_1 \in 10100 \oplus K_1 \wedge F_2 \in 00100 \oplus K_1 &\rightarrow a, \\ F_1 \in 10100 \oplus K_1 \wedge F_2 \in 01010 \oplus K_1 &\rightarrow a, \end{aligned}$$

Fig. 5(f) shows the disjoint rules generated from the FDD in Fig. 5(e).

After non-overlapping rules are generated, MSU sends the resulting policy to IBM. If MSU needs to prevent IBM from knowing the number of non-overlapping prefix rules that MSU's firewall is converted to, MSU can randomly insert some dummy rules formulated by out-of-range dummy numbers and random decisions into the set of non-overlapping numerical rules before applying XOR. An out-of-range dummy number is a number that corresponds to no prefix. Thus, no packet will match a dummy rule that consists of at least one out-of-range dummy number. According to our prefix numericalization scheme, there is only one dummy number in which every bit is 0. To create more dummy numbers, we

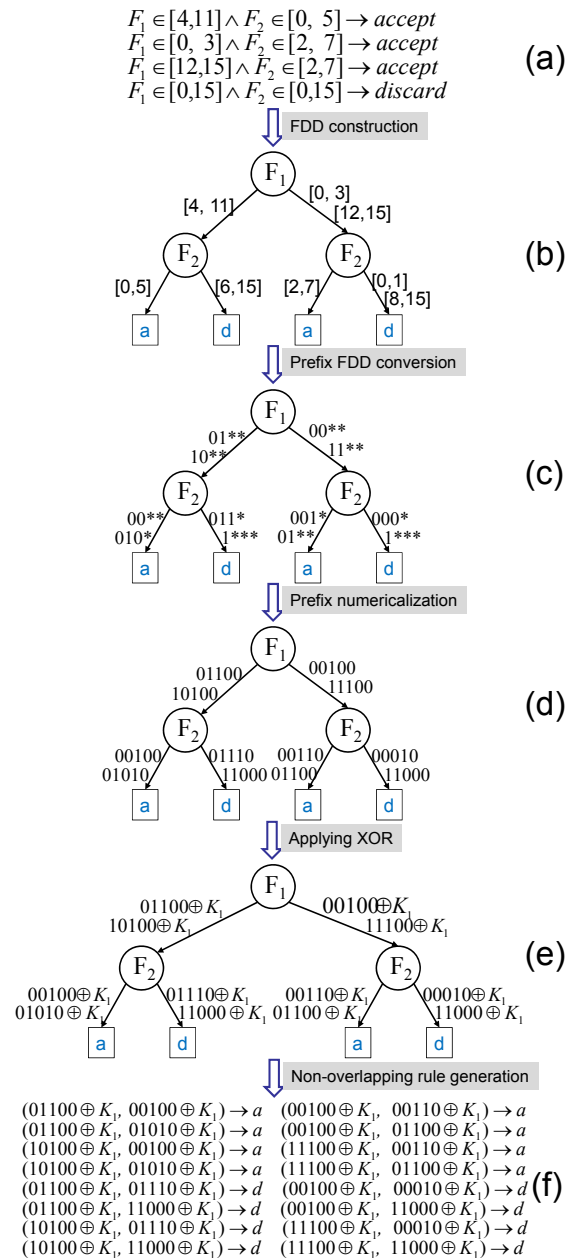


Fig. 5. Example of bootstrapping at MSU

can simply add extra bits. Note that IBM knowing the number of converted rules is not much a concern. As we will show in the experimental results, the number of non-overlapping prefix rules that a firewall is converted to far exceeds the number of original rules.

5.5 Applying XOR and HMAC by IBM

Upon receiving a sequence of non-overlapping numerical rules from MSU, IBM further applies XOR and HMAC to every number in the received policy using its secret key K_2 . To destroy the correspondence between the rules after applying XOR and HMAC and the rules received from MSU, IBM randomly shuffles the resulting rules after applying XOR and HMAC. To prevent MSU from knowing the decision of IBM's packet, IBM obfuscates the decision of each rule by mapping each decision to another distinct decision. More formally, the decision

obfuscation is a one-to-one mapping function f from the set of all decisions to the same set of all decisions. IBM stores the mapping function f in its decision obfuscation table and replaces the decision of each rule in r_i , say d_i , by $f(d_i)$. To prevent MSU from statistically discovering the obfuscation mapping function f , for any decision d_i , IBM needs to ensure that the number of rules that have decision d_i is the same. This can be easily achieved by adding dummy rules. Due to the rule shuffling and decision obfuscation, MSU cannot correlate the received rules with the original rules, and also cannot identify the decision of each rule. Fig. 6(b) shows the rules after IBM applies XOR and HMAC, and Fig. 6(c) shows the rules after IBM shuffles rules and obfuscates decisions. The obfuscation mapping function is shown in Fig. 6(d). Note that in these figures h denotes the HMAC function. Finally, IBM sends the resulting rules to MSU.

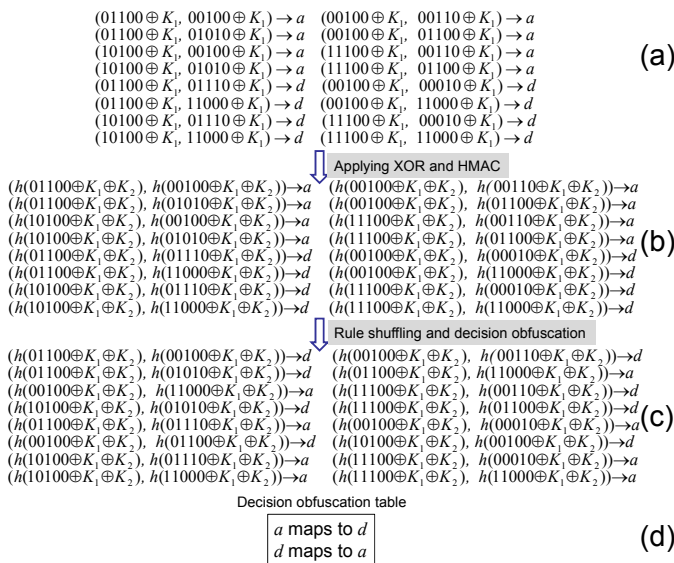


Fig. 6. Example of bootstrapping at IBM

6 FILTERING PROTOCOL

In the filtering protocol, each time IBM receives a packet that originated from or was sent to its representative, IBM first converts the packet to prefixes and then further converts each prefix to a number using the same prefix numericalization scheme. Then, IBM XORs every number in the packet with its secret key K_2 , then sends the resulting packet to the third party. The third party further applies XOR and HMAC to the received packet with the secret key K_1 . Note that the third party and MSU share key K_1 . Then, the third party sends the resulting packet to MSU. MSU then searches the obfuscated decision for the packet using the received firewall policy from IBM in the bootstrapping protocol. Finally, MSU sends the obfuscated decision to IBM and IBM finds the original decision using its decision obfuscation table. Fig. 7 shows the filtering protocol.

6.1 Address Translation

When the IBM VPN server sends (or receives) a packet on behalf of its representative in MSU, the source (or

destination) IP address of the packet is an IBM IP address that the IBM VPN server assigned to the IBM representative's computer in MSU. To inquiry the decision for this packet from MSU, IBM needs to replace the source (or destination) IP address in the packet by IBM representative's MSU IP address. Otherwise, it is likely that MSU firewall policy blocks all incoming packets that are not sent to MSU and all outgoing packets that are not originated from MSU. Take the example in Fig. 1, the packet that IBM should ask MSU for a decision has a source IP 1.1.0.10 and a destination IP 2.2.0.2.

6.2 Prefix Membership Verification

We first define two concepts: k -prefix and prefix family. We call the prefix $\{0, 1\}^k \{*\}^{w-k}$ with k leading 0s and 1s followed by $w - k$ *s a k -prefix. If a value x matches a k -prefix, the first k bits of x and the k -prefix are the same. For example, if $x \in 01**$ (i.e., $x \in [0100, 0111]$), then the first two bits of x must be 01. Given a binary number $b_1b_2 \dots b_w$ of w bits, the prefix family of this number is the set of $w + 1$ prefixes $\{b_1b_2 \dots b_w, b_1b_2 \dots b_{w-1}*, \dots, b_1* \dots *, ** \dots *\}$, where the i -th prefix is $b_1b_2 \dots b_{w-i+1} * \dots *$. We use $PF(x)$ to represent the prefix family of x . For example, $PF(0101) = \{0101, 010*, 01**, 0***, ** \dots *\}$. Based on the above definitions, it is easy to draw the following conclusion: given a number x and a prefix \mathcal{P} , $x \in \mathcal{P}$ if and only if $\mathcal{P} \in PF(x)$.

6.3 Packet Preprocessing by IBM

For each of the d fields of a packet, IBM first generates its prefix family. Second, IBM converts each prefix to a number using the same prefix numericalization scheme in the bootstrapping protocol. Third, IBM applies XOR to each number using its secret key K_2 . Last, IBM sends a sequence of d sets of numbers, which corresponds to the d fields of the packet, to the third party. For example, given a packet (0101, 0011) as shown in Fig. 8(a), the prefix family of each field is shown in Fig. 8(b). The result of prefix numericalization is shown in Fig. 8(c). The final two sequences of numbers are shown in Fig. 8(d).

6.4 Packet Preprocessing by The Third Party

Upon receiving the packet as d sequences of numbers from IBM, the third party further applies XOR using key K_1 and HMAC to each number and then sends the resulting packet to MSU. Here we choose the third party, instead of MSU, to apply XOR and HMAC for the purpose of preventing MSU from knowing the IBM's XOR results (e.g., Fig. 8(d)) before applying HMAC. Otherwise, MSU may break IBM's secret key K_2 and further reveal packet headers. If MSU knows IBM's XOR results, to break K_2 , MSU first stores its rules before anonymization in the bootstrapping protocol (e.g. the rules generated from Fig. 5(d)). Let $\langle r_1, \dots, r_n \rangle$ denote these rules, where each rule r_j ($1 \leq j \leq n$) is in the form $\langle m_1^j, \dots, m_d^j \rangle \rightarrow \langle dec^j \rangle$. In the filtering protocol, when MSU finds that a packet $\langle p_1, p_2, \dots, p_d \rangle$ matches

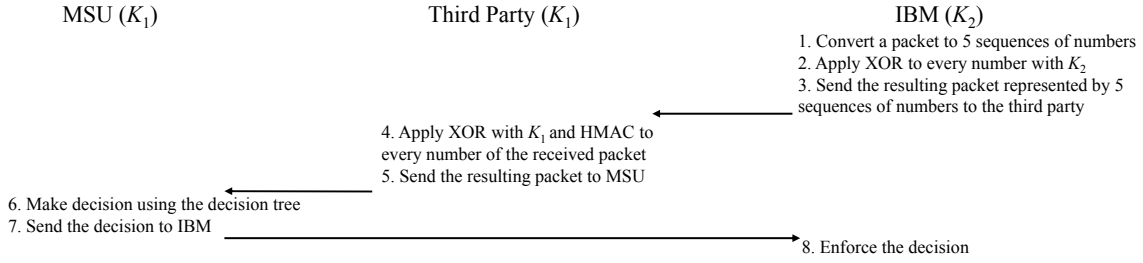


Fig. 7. The filtering protocol

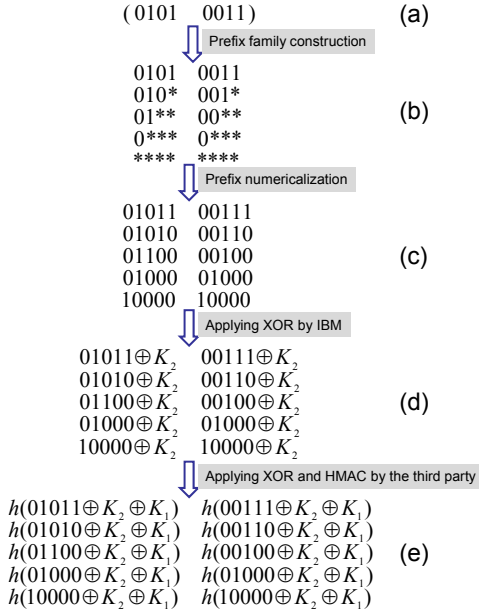


Fig. 8. Example of packet preprocessing

a rule, according to the property of prefix membership verification, for each $1 \leq i \leq d$, there must be a number n_i in $PF(p_i)$ that is equal to one number in the set $\{m_1^i, \dots, m_n^i\}$. Third, for each $1 \leq i \leq d$, MSU XORs $n_i \oplus K_2$ received from IBM with every number in the set $\{m_1^i, \dots, m_n^i\}$. Because one of the numbers in $\{m_1^i, \dots, m_n^i\}$ is equal to n_i , then the resulting set, denoted S_i , must contains K_2 . For example, if $m_1^i = n_i$, then $m_1^i \oplus n_i \oplus K_2 = K_2$. Thus, for each packet P , we can compute a set $S(P) = S_1 \cap \dots \cap S_d$, which contains K_2 . When MSU receives a large set of packets P_1, \dots, P_g , the set $S(P_1) \cap \dots \cap S(P_g)$ may only contains K_2 . After finding K_2 , MSU can reveal packet headers by applying XOR to every number of packets received from IBM using K_2 . However, in VGuard, using the third party to apply XOR and HMAC to packets eliminates this possibility.

6.5 Packet Processing by MSU

Upon receiving the packet from the third party, MSU searches the obfuscated decision for the packet using the resulting firewall rules from the bootstrapping protocol. Recall that each rule is represented as d numbers and an obfuscated decision. A packet (p_1, \dots, p_d) matches a rule $(m_1, \dots, m_d) \rightarrow \langle \text{obfuscated decision} \rangle$ if and only if the condition $m_1 \in PF(p_1) \wedge \dots \wedge m_d \in PF(p_d)$ holds. Therefore, MSU can use linear search to find the first rule that the packet matches. Then, MSU sends the

obfuscated decision to IBM and IBM finds the original decision using its decision obfuscation table. Because all the firewall rules resulted from the bootstrapping protocol are non-overlapping, there exists one and only one rule that the packet matches. For example, given the resulting firewall rules in Fig. 6(c) and the preprocessed packet in Fig. 8(e), the only rule that matches the packet is $(h(01100 \oplus K_2 \oplus K_1), h(00100 \oplus K_2 \oplus K_1)) \rightarrow d$.

To improve search efficiency, MSU can use the following two techniques: decision tree and hash table. First, MSU converts the non-overlapping rules resulted from the bootstrapping protocol to an equivalent decision tree. For example, Fig. 9 shows the decision tree constructed from the firewall in Fig. 6(c). Thus, MSU can search the decision for a packet using the decision tree. Second, for the basic operation of testing $m_i \in PF(p_i)$, MSU builds one hash table for each $PF(p_i)$ and then tests whether m_i is in the hash table that constructed from $PF(p_i)$.

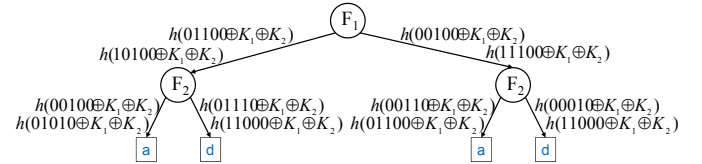


Fig. 9. Decision tree constructed from Fig. 6(c)

7 VGUARD FOR DEEP PACKET INSPECTION

With the growing need to filter malicious packets, advanced firewalls, as well as intrusion detection/prevention systems such as Snort [27], Bro [24], 3Com's TippingPoint X505 [2], and a variety of Cisco Systems [1], examine not only packet headers but also packet payload by checking whether its payload contains some predefined strings in a signature database. More formally, given a string $a_1 a_2 \dots a_n$ and a packet payload $s_1 s_2 \dots s_m$ where each a_i ($1 \leq i \leq n$) and s_j ($1 \leq j \leq m$) are characters, we want to check whether the string $s_1 s_2 \dots s_m$ contains the sub-string $s_{k+1} s_{k+2} \dots s_{k+n}$ that is the same as the string $a_1 a_2 \dots a_n$. If so, the packet payload $s_1 s_2 \dots s_m$ matches the string $a_1 a_2 \dots a_n$.

We can adapt our VGuard framework to deal with the cases where MSU's firewall performs deep packet inspection. The basic idea is that MSU and IBM apply Xhash protocol to each character of every string in the signature database and each character of the packet payload, and check whether the resulting packet payload contains the resulting string.

7.1 The Bootstrapping Protocol

In the bootstrapping protocol, MSU first applies XOR to every character of the strings in its signature database using its secret key K_1 , and then sends the resulting strings to IBM. To prevent IBM from knowing the number of strings in its signature database, MSU adds some random strings and XORs them with K_1 . Upon receiving the anonymized strings from MSU, IBM further applies XOR and HMAC operations to each character using its secret key K_2 . To prevent MSU from identifying the original string that a packet matches by comparing the number of characters in each resulting string with that in each original string, IBM adds some dummy strings and XORs them with its secret key K_2 . Then, IBM obfuscates the decision associated with each string and shuffles the strings. At last, IBM sends the resulting strings back to MSU. Note that all the random strings added by MSU and the dummy strings added by IBM should have the default action, which is “permit”. Fig. 10 shows the bootstrapping protocol for deep packet inspection. Suppose an intrusion detection system has n rules and the i -th ($1 \leq i \leq n$) rule has c_i characters. In the bootstrapping protocol, the computation overhead of MSU and IBM is $O(\sum_{i=1}^n c_i)$, and the communication overhead between MSU and IBM is also $O(\sum_{i=1}^n c_i)$.

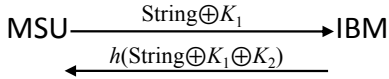


Fig. 10. Bootstrapping for deep packet inspection

Considering three strings “ eb , ebf , ecg ” in Fig. 11(a), Fig. 11(b) shows the anonymized string s after MSU applies XOR to these strings. Fig. 11(c) shows the resulting strings after MSU adds the random string r_1r_2 , where r_1 and r_2 denote two random characters. Fig. 11(d) shows the resulting strings after IBM adds the dummy string d_1d_2 , where d_1 and d_2 denote two random characters. Fig. 11(e) shows the strings after IBM applies XOR and HMAC, and Fig. 11(f) shows the strings after IBM shuffles rules and obfuscates decisions.

As the dummy strings that IBM generated are unlikely to match any packet, MSU may identify them and then delete them. To prevent MSU from identifying such strings, IBM can generate fake packets that match the dummy strings and periodically send them to MSU.

7.2 The Filtering Protocol

In the filtering protocol, each time IBM receives a packet originated from or sent to its representative, IBM first applies XOR to every character in the packet payload using K_2 and sends the resulting packet to the third party, which further applies XOR and HMAC to the packet payload using key K_1 and then sends the resulting packet to MSU. String matching algorithms have been investigated for many years and several famous algorithms have been proposed, such as Aho-Corasick algorithm [4] and Commentz-Walter algorithm [10]. MSU can use these algorithms to search the obfuscated decision

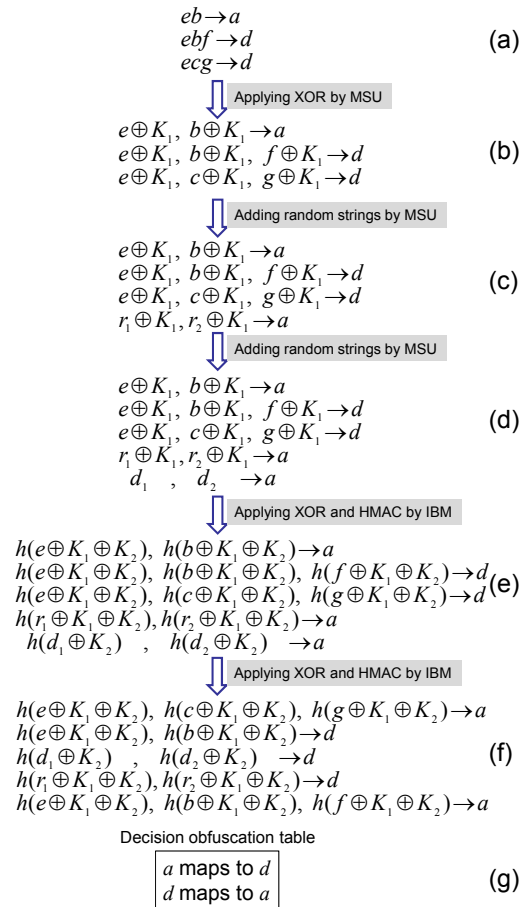


Fig. 11. Example of string processing

for the packet based on the received strings from IBM in the bootstrapping protocol. Finally, MSU sends the obfuscated decision to IBM and IBM finds the original decision using its decision obfuscation table.

For example, given a packet that contains strings “ $ebkf$ ” as shown in Fig. 12(a), the packet payload after IBM applies XOR is in Fig. 12(b). Fig. 12(c) shows the result payload after the third party applies XOR and HMAC. For the resulting strings in Fig. 11(f), the only string in the signature database that matches the packet payload is $h(e \oplus K_1 \oplus K_2), h(b \oplus K_1 \oplus K_2) \rightarrow d$.

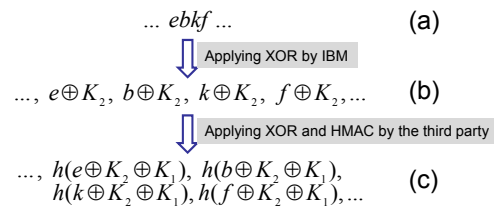


Fig. 12. Example of packet payload processing

8 DISCUSSION

8.1 Firewall Updates

When MSU updates its firewall policy, MSU and IBM need to run the bootstrapping protocol again. To prevent IBM from identifying the unchanged rules, in each run of the bootstrapping protocol, MSU and IBM should change their secret keys K_1 and K_2 . Thus, the rules that

IBM receives in each run of the bootstrapping protocol are totally different. Note that MSU and IBM do not need to run the bootstrapping protocol as long as MSU's firewall remains the same.

8.2 Decision Caching

A packet contains a header (with fields of source and destination IP addresses, source and destination port numbers, and protocol type) and a payload. For different packets with the same header, IBM only needs to ask MSU once and then cache the packet header along with its decision. Whenever the IBM representative builds a connection through the VPN tunnel, IBM first checks whether its cache has an entry that corresponds to the connection. If yes, then IBM executes that decision; if no, IBM asks MSU for the decision using the filtering protocol and then adds an entry into its cache. Because IBM may have multiple collaborators, IBM should record the name of its collaborator (which is "MSU" in this case) to every entry in the cache. Thus, IBM only needs to maintain one cache. When MSU updates its firewall policy and reruns the bootstrapping protocol, IBM needs to delete all the entries that corresponds to MSU. IBM can implement caches efficiently using hash tables or by counting Bloom filters [13].

8.3 Decision Obfuscation vs. Decision Encryption

In the bootstrapping protocol, to prevent MSU from knowing the decision of IBM's packet, we have two choices, decision obfuscation and decision encryption. A decision obfuscation function is a one-to-one mapping function from the set of all decisions to the same set of all decisions. The idea of decision obfuscation is to apply such a function to the decisions of the rules sent from IBM to MSU in the bootstrapping protocol. The idea of decision encryption is to concatenate each decision dec with a number i and then encrypt the decision $dec|i$ using another secret key K , i.e., $(dec|i)_K$.

Although decision encryption obfuscates the decisions well, it can be exploited. Using decision encryption, IBM can add more information to the decision and then encrypt it. In particular, if IBM attaches the identification number of a rule to each decision, IBM can identify the exact rule that matches the packet. This would help IBM compute MSU's secret key K_1 and further reveal the exact context of the rule. More formally, IBM can store the anonymized rules received from MSU in the bootstrapping protocol. These anonymized rules are in the form $(m_1 \oplus K_1, m_2 \oplus K_1, \dots, m_5 \oplus K_1) \rightarrow \langle dec \rangle$ (where each m_i is a number). If IBM can identify the one rule that matches the packet (p_1, p_2, \dots, p_5) , IBM can generate prefix family for each attribute of the packet $PF(p_i)$ ($1 \leq i \leq 5$). Due to the property of prefix membership verification, there should be one number n_i in each $PF(p_i)$ that is equal to m_i . Therefore, IBM can compute MSU's secret key K_1 by applying XOR to the number $m_i \oplus K_1$ using n_i . If $m_i = n_i$, then $m_i \oplus K_1 \oplus n_i = K_1$. After finding K_1 , IBM can reveal the

exact context of non-overlapping rules in MSU firewall policy by applying XOR to every number of anonymized rules using K_1 . Because of the above security problem, we use the decision obfuscation technique.

8.4 Special Treatment of IP Addresses

As we discussed previously, for every packet from (or to) the IBM representative, IBM needs to translate the source (or destination) IP from an address in IBM's domain to the address in MSU's domain. Thus, for each packet that IBM asks MSU for a decision, either the source or the destination IP of the packet is the IP that MSU assigns to the IBM representative. This IP address is known to MSU, MSU can use it to compute the secret key K_2 of IBM, which henceforth violates the packet privacy. Note that $x \oplus K_2 \oplus x = K_2$.

To prevent this type of attacks, we modify VGuard as follows. First, MSU chooses five secret keys K_1, K_2, \dots, K_5 that correspond to the five packet fields (source IP, destination IP, source port, destination port, and protocol type), and similarly IBM chooses five secret keys K'_1, K'_2, \dots, K'_5 as well. Second, in the bootstrapping protocol, for each non-overlapping rule $(m_1, m_2, \dots, m_5) \rightarrow \langle dec \rangle$ (where each m_i is a number), MSU applies XOR to each m_i using key K_i . Thus, the rule becomes $(m_1 \oplus K_1, m_2 \oplus K_2, \dots, m_5 \oplus K_5) \rightarrow \langle dec \rangle$. Then, MSU sends these rules to IBM. For each rule $(m_1 \oplus K_1, m_2 \oplus K_2, m_3 \oplus K_3, m_4 \oplus K_4, m_5 \oplus K_5) \rightarrow \langle dec \rangle$ that IBM receives from MSU, assuming that m_1 corresponds to the field of source IP and m_2 corresponds to the field of destination IP, IBM creates the two rules:

$$\begin{aligned} & (m_1 \oplus K_1, HMAC_k(m_2 \oplus K_2 \oplus K'_2), HMAC_k(m_3 \oplus K_3 \oplus K'_3), \\ & \quad HMAC_k(m_4 \oplus K_4 \oplus K'_4), HMAC_k(m_5 \oplus K_5 \oplus K'_5)) \rightarrow \langle dec \rangle \\ & (HMAC_k(m_1 \oplus K_1 \oplus K'_1), m_2 \oplus K_2, HMAC_k(m_3 \oplus K_3 \oplus K'_3), \\ & \quad HMAC_k(m_4 \oplus K_4 \oplus K'_4), HMAC_k(m_5 \oplus K_5 \oplus K'_5)) \rightarrow \langle dec \rangle \end{aligned}$$

Basically, IBM keeps the source IP field unchanged in the first rule and keeps the destination IP field unchanged in the second rule. At last, IBM sends two sets of rules back to MSU, where in one set the source IP is unchanged and in the other set the destination IP is unchanged. Third, in the filtering protocol, when IBM receives a packet, without loss of generality assuming that the packet is originated from its representative, IBM applies XOR to four fields of the packet (destination IP, source port, destination port, and protocol type) using its four corresponding keys K'_2, K'_3, K'_4 , and K'_5 . In other words, when the source IP of the packet is the MSU IP address, IBM does not apply XOR to that field. When the resulting packet $(p_1, p_2 \oplus K'_2, p_3 \oplus K'_3, p_4 \oplus K'_4, p_5 \oplus K'_5)$ is sent to the third party, IBM asks the third party to apply only XOR to p_1 using key K_1 and process $p_2 \oplus K'_2, p_3 \oplus K'_3, p_4 \oplus K'_4, p_5 \oplus K'_5$ as usual using keys K_2, K_3, K_4 and K_5 respectively. Then, the third party sends the resulting packet to MSU. Finally, MSU searches the decision for the packet in the rule set where the source IP field was not processed by IBM. Thus, leaving the source (or destination) IP field unprocessed by IBM, MSU cannot break any secret key of IBM.

8.5 Securing Keys of MSU

In our VGuard, IBM only knows $m_i \oplus K_i$ ($1 \leq i \leq 5$) but doesn't know m_i . However, if MSU's firewall policy has popular values in a field, say the j -th field ($1 \leq j \leq 5$), IBM may reveal K_j by XORing these popular values with $m_j \oplus K_j$ of all non-overlapping rules received from MSU. To prevent such attack, when MSU's firewall has some popular values, MSU can XOR each popular value using a different key in the bootstrapping protocol. The reason is that for the j -th field, although IBM can obtain a set of values through the above attack, it cannot distinguish the keys of each popular value from that set.

Similarly, for deep packet inspection, if MSU has some popular strings in its signature database, IBM may reveal MSU's secret key. The strings that MSU needs to keep confidential are the ones that are specific to MSU needs. However, if some strings are popular among signature databases, it is unlikely that MSU needs to keep them confidential. Thus, to prevent such attack, MSU can simply tell IBM these popular strings and ask IBM to check whether a packet payload matches these strings.

8.6 Stateful Firewalls

So far we have assumed that firewalls are stateless. A stateful firewall is a firewall that keeps track of the state of network connections across the firewall. When a stateful firewall receives a packet, it first checks its connection table to see whether the packet belongs to an ongoing connection. If yes, the packet is permitted right away. If no, the packet needs to be checked with its stateless rules to determine whether the packet should be permitted; if the stateless rules allows the packet to be permitted, then a new connection is built and inserted into the connection table of the firewall. Such stateful firewalls typically allow inside non-server computers (i.e., the computers that are not servers) to initiate connection with outside computers but disallow outside computers to initiate connection with non-server computers. When an inside non-server computer sends a packet to an outside computer, the stateful firewall uses its stateless rules to decide whether the packet should be permitted; if yes, the firewall adds an entry in its connection table that will allow subsequent packets sent from that outside computer to the inside computer. When an outside computer sends a packet to an insider non-server computer, if there is no corresponding entry in the connection table, the firewall will use its stateless rules to make a decision, which is typically *deny*.

Our framework can be extended to handle stateful firewalls. The basic idea is to let IBM maintain a connection table for its representative. If MSU's firewall is stateful, in the extended framework, when IBM receives a packet from or to its representative, IBM first consults its connection table to see whether the packet belongs to an ongoing connection. If yes, the packet is accepted right away. If no, IBM asks MSU for the decision for this packet; if the packet is permitted, IBM adds an entry into the connection table. Note that the connection table

is different from IBM's decision cache. If MSU's firewall is stateless, for every connection, with the help of cache, IBM needs to ask MSU the decision for two packets that go in exactly the opposite direction. If MSU's firewall is stateful, with the help of the cache and the connection table, IBM only needs to ask MSU the decision for one packet, which is the one that initiates the connection.

8.7 Statistical Analysis Attack and Countermeasures

MSU could launch a statistical analysis attack to reduce the number of possible rules that a packet matches. This attack works as follows. After non-overlapping rule generation, MSU calculates the frequency for each number in the rules. The frequency of each number is preserved in processing the policy by MSU and IBM in the bootstrapping protocol. MSU could exploit such frequency information to reduce the number of possible rules that an IBM packet can match. For example, considering the numericalized FDD in Fig. 13(a), the frequency of the number 01100 in the generated non-overlapping rules is 2, and none of the other numbers have the same frequency. Thus, MSU can identify which rules received from IBM correspond to the left branch of the FDD. Interestingly, MSU cannot correlate any rule received from IBM with the right most decision path of this FDD because of the use of dummy rules.

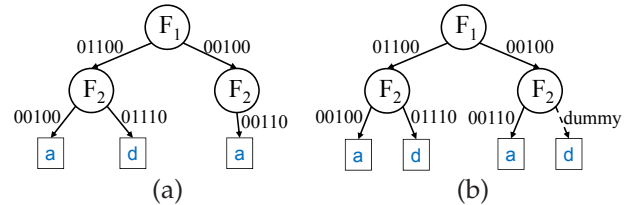


Fig. 13. An example of statistical analysis attack

The statistical analysis attack is based on the assumption that the frequency of each number remains the same before and after IBM's processing. Actually, to prevent statistical analysis attacks, IBM can also make a statistical analysis of the hashed rules and add some dummy rules to disturb the statistical properties of the rules. Taking the example numericalized FDD in Fig. 13(a), IBM can easily destroy the frequency information of the number in the FDD by adding a dummy number to the right F_2 node as shown in Fig. 13(b). Basically, IBM generates dummy rules to make all the numbers for each field have the same frequency. This will prevent MSU from launching statistical analysis attacks. Recall that dummy rules use out-of-range numbers and they cannot be matched by any packet.

8.8 Hash Collision

The chance of having hash collisions for HMAC is extremely small. However, to be on the safe side, we propose the following solution to the problem. Our solution is based on the observation that the bootstrapping protocol in our framework can detect hash collisions easily. Recall that the bootstrapping protocol converts firewall policies to non-overlapping rules. If hash collision happens, then among the rules that MSU receives

from IBM, there exist at least two rules that are exactly the same. This fact can be used by MSU to easily detect whether hash collision happens. In the case that hash collision does happen, MSU and IBM can simply rerun the bootstrapping protocol, in which they will choose different secret keys and henceforth the hash collision is most likely removed.

9 RELATED WORK

9.1 Secure Function Evaluation

Secure Function Evaluation (SFE) was first introduced by Yao with the famous “Two-Millionaire Problem” [32]. A secure function evaluation protocol enables two parties, one with input x and the other with y , to collaboratively compute a function $f(x, y)$ without disclosing one party’s input to the other. The classical solutions for SFE are Yao’s “garbled circuits” protocol [33] and Goldreich’s protocol [23]. The method provided by Yao has a computational cost of $O(2^b)$, where b is the number of bits needed to encode x and y . Later, the Secure Function Evaluation problem was generalized to the Secure Multiparty Computation (SMC) problem. Chaum *et al.* proved that any multiparty protocol problem can be solved if there is an authenticated secrecy channel between every pair of participants [7]. Zero-knowledge protocols [5], [11], [22] also aims to provide the privacy between two parties. A zero knowledge protocol is an interactive method for one party (suppose IBM) to prove to another (suppose MSU) that a statement is true without revealing anything other than the veracity of the statement. Although we could use SFE or SMC solutions to solve this problem as well as the problem of checking whether a value is in a range, the $O(2^b)$ complexity makes such solutions infeasible.

9.2 CDCF Framework

Prior work that is closest to ours is the Cross-Domain Cooperative Firewall (CDCF) framework [9]. The CDCF framework also consists of a bootstrapping protocol and a filtering protocol. In CDCF bootstrapping protocol, for each firewall rule specified as $F_1 \in S_1 \wedge \dots \wedge F_d \in S_d$ where each S_i is a range, MSU first converts each range S_i to a set of prefixes and then converts each prefix to a concrete number using their numericalization scheme. Second, MSU and IBM use a commutative encryption function (such as the Pohlig-Hellman Exponentiation Cipher [25] and Secure RPC Authentication (SRA) [28]) to encrypt each number generated from the prefixes with MSU’s secret key K_1 and IBM’s secret key K_2 , respectively. A commutative encryption function satisfies the following four properties, where K_1 and K_2 are two secret keys: (a) For any x and K , given x and $(x)_K$, it is computationally infeasible to compute the value of K . (b) For any x, K_1, K_2 , we have $((x)_{K_1})_{K_2} = ((x)_{K_2})_{K_1}$. (c) For any x, y , and K , if $x \neq y$, then we have $(x)_K \neq (y)_K$. (d) For any x and K , given K , $(x)_K$ can be decrypted in polynomial time. Finally, MSU stores the resulting firewall rules after commutative encryption. In the filtering

protocol of CDCF, for each packet p_1, p_2, \dots, p_d , IBM first computes the prefix family $PF(p_i)$ ($1 \leq i \leq d$) and then converts each prefix in $PF(p_i)$ to a number. Second, for each number y , IBM computes $(y)_{K_2}$ using commutative encryption. Third, IBM sends d sequences of numericalized and encrypted prefixes to MSU. Upon receiving the d sequences, for each number $(y)_{K_2}$, MSU computes $((y)_{K_2})_{K_1}$. Finally, based on the properties of prefix membership verification and commutative encryption, MSU checks which rule that matches the received packet and hence finds the decision for the packet.

There are five major differences between VGuard framework and CDCF framework.

(1) The computation and communication costs of CDCF is much more expensive than those of VGuard because of the following two reasons. First, to achieve oblivious comparison, VGuard uses the Xhash protocol and CDCF uses commutative encryption functions, which are computationally expensive. Second, VGuard uses firewall decision diagrams to speed up the processing of packets, while CDCF uses the straightforward sequential search.

(2) CDCF allows MSU to discover the original firewall rule that a packet matches, which jeopardizes packet privacy. This is particularly dangerous if the matching rule is a singleton rule, which will allow MSU to immediately know the corresponding value in the matching packet. In comparison, VGuard does not allow MSU to discover the original firewall rule that a packet matches. The key operation in VGuard is that it converts the original firewall rules to non-overlapping rules, which enables IBM to shuffle the rules. Thus, MSU cannot reveal the correspondence between the original rules and the received rules from IBM. Because a firewall policy follows first-match semantics, without such a conversion, IBM cannot disturb the order among rules in CDCF.

(3) CDCF allows MSU to know the decision for each IBM’s packet, while VGuard does not. Knowing both the original rules and the decision of a packet p , MSU could guess what packets that p could be.

(4) CDCF does not perform the address translation that we discussed in Section 6.1, which could render MSU’s firewall policy ineffective for IBM’s packets. However, the address translation procedure could be easily added to CDCF.

(5) CDCF is vulnerable to a type of attacks that we call *selective policy updating attacks*. Such attacks allow MSU to quickly discover the field values in a packet in the following manner. When MSU receives a double encrypted packet p , assuming p matches the prefix rule $F_1 \in S_1 \wedge \dots \wedge F_d \in S_d \rightarrow \langle decision \rangle$, MSU splits each prefix S_i into two prefixes by instantiating the first $*$ by 0 and 1 respectively, and therefore converts the rule to a maximum of 2^d rules. For example, suppose a packet p from IBM matches the prefix rule $F_1 \in 10** \wedge F_2 \in 001* \rightarrow a$. Then, MSU splits the rule into the following four rules: $F_1 \in 100* \wedge F_2 \in 0010 \rightarrow a$, $F_1 \in 101* \wedge F_2 \in 0010 \rightarrow a$, $F_1 \in 100* \wedge F_2 \in 0011 \rightarrow a$, $F_1 \in 101* \wedge F_2 \in 0011 \rightarrow a$.

Then, MSU requests to rerun the CDCF protocol due to firewall update. In the new run of CDCF, MSU replaces the above rule by the rules after splitting. (Note that d is typically 4 or 5.) After MSU receives the double encrypted rules from IBM in the new run, MSU compares p with the split rules again. One of the split rules must match p . The above process repeats with a maximum of 32 times before p matches a singleton rule at the end. To counter the selective policy updating attacks, CDCF can be fixed by updating the secret keys on both MSU and IBM sides in each run of the CDCF protocol.

(6) CDCF does not support deep packet inspection while VGuard supports it.

9.3 Secure Queries

Secure queries in outsourced database systems have been studied in prior work (e.g., [3], [17], [18]). These work aims to design a scheme for querying encrypted data in the outsourced database system where sensitive data are outsourced to an untrusted server. Later, researchers extended these work to securely process range queries in two-tiered sensor networks [8], [29], [30], [34], where storage nodes gather data from nearby sensors and answer queries from the sink. They proposed different schemes for preventing compromised storage nodes from gaining information from the sensitive data and forging query results to the sink. These schemes cannot be directly applied to the problem in this paper for two reasons. First, both outsourced database systems and two-tiered sensor networks only deal with one untrusted party, i.e., untrusted servers or untrusted storage nodes, while in this paper two parties MSU and IBM don't trust each other. Second, firewall policies are different from the data stored in database or collected by sensors in terms of both structure and semantics.

Some prior work (e.g., [14], [31]) has investigated keyword searching on encrypted data, where a keyword is a word of a natural language and data are text.

10 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our schemes on both real-life and synthetic firewall policies. In particular, we implemented our schemes without and with adding dummy rules. For ease presentation, we use *VGuard* and *VGuard+* to denote our schemes without and with adding dummy rules, respectively. Then, we compared VGuard, VGuard+, and CDCF, side by side. We implemented VGuard, VGuard+, and CDCF using Java 1.6.3. Our experiments were carried out on a desktop PC running Windows XP SP2 with 3G memory and dual 3.4 GHz Intel Pentium processors. On real-life firewall policies, for processing packets, our experimental results show that VGuard is 552 times faster than CDCF on MSU side and 5035 times faster than CDCF on IBM side; VGuard+ is 544 times faster than CDCF on MSU side and 5021 times faster than CDCF on IBM side. On synthetic firewall policies, for processing packets, our experimental results show that VGuard is 252 times faster than CDCF on MSU side and 5529 times faster

than CDCF on IBM side; VGuard+ is 248 times faster than CDCF on MSU side and 5513 times faster than CDCF on IBM side.

10.1 Efficiency on Real-life Firewall Policies

We conducted experiments on 16 real-life firewall policies that we collected from a variety of sources. Each firewall examines five packet fields of source IP, destination IP, source port, destination port, and protocol type. The number of rules ranges from dozens to hundreds. We measured the computational cost of the two parties MSU and IBM for both bootstrapping and filtering protocols. We also measured the communication overhead for both bootstrapping and filtering protocols. For fair comparison, in implementing CDCF, we used the same parameters as in [9], i.e., we used the Pohlig-Hellman algorithm [25] with a 1024-bit prime modulus and 160-bit encryption keys. In implementing VGuard and VGuard+, we chose the HMAC-MD5 hash function with 128-bit keys.

Fig. 14 shows the communication overhead in the bootstrapping protocol for VGuard, VGuard+, and CDCF. We observe that the communication overhead in the bootstrapping protocol of VGuard is similar as that of CDCF for all firewalls. We have two explanations. First, CDCF needs to

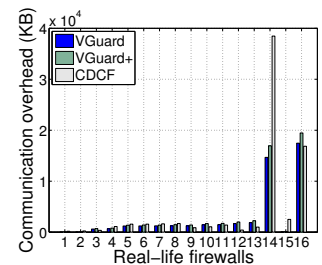


Fig. 14. Communication overhead in the bootstrapping protocol

convert the range to prefix numbers for every field in the firewall rules. Recall that the minimum number of prefixes for representing an integer $[a, b]$, where a and b are two numbers of w bits, is at most $2w-2$. For example, the minimum number of prefixes for representing the range of IP address that has 32 bits is at most 62. Thus, the number of prefix rules converted from a firewall rule can be at most $62 \times 62 \times 30 \times 30 \times 14 = 48434400$. Second, every prefix number will be encrypted to be 1024 bits in CDCF instead of 128 bits in VGuard. Therefore, even though the ratio between the number of non-overlapping firewall rules and the number of original firewall rules is large (i.e., in the 16 real-life firewall policies, the average ratio between the number of non-overlapping firewall rules and the number of original firewall rules is 887), the communication overhead in the bootstrapping protocol of VGuard is still similar as that of CDCF. We also observe that on average, the communication overhead in the bootstrapping protocol of VGuard+ is 15% higher than that of VGuard because of adding dummy rules. In the filtering protocol, the only difference between CDCF and VGuard is that the encryption method is different. CDCF applies double encryption to every prefix number, and VGuard applies Xhash to every prefix number. Therefore, in the filtering protocol, the overhead of CDCF is $1024/128=8$ times higher than that of VGuard. Similarly, in the filtering

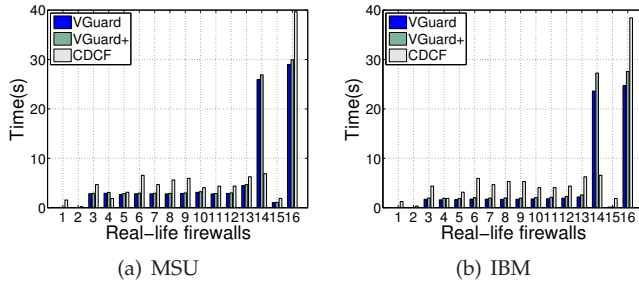


Fig. 15. The bootstrapping time on real-life firewalls

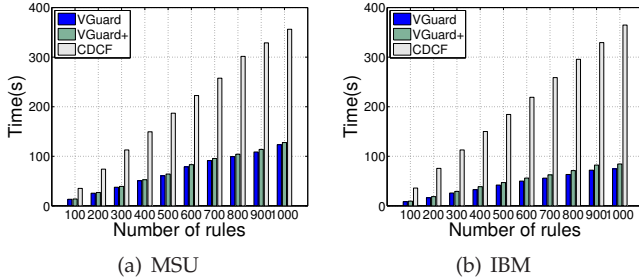


Fig. 17. The bootstrapping time on synthetic firewalls

protocol, the overhead of CDCF is 8 times higher than that of VGuard+.

Figures 15(a) and 15(b) show the computational cost of MSU and IBM respectively in the bootstrapping protocol for VGuard, VGuard+, and CDCF. We observe that the bootstrapping costs of VGuard and VGuard+ are lower than that of CDCF for most firewalls. Although the Xhash scheme is three orders of magnitude faster than the commutative encryption scheme, the bootstrapping costs of VGuard and VGuard+ are not three orders of magnitude lower than CDCF because VGuard and VGuard+ convert the given firewall policy to non-overlapping prefix rules, which result in a significant expansion. Note that the bootstrapping protocol only needs to run once between MSU and IBM unless MSU updates its firewall policy. The performance of the bootstrapping protocol is less critical than that of the filtering protocol.

Figures 16(a) and 16(b) show the computational cost of MSU and IBM respectively in the filtering protocol for VGuard, VGuard+, and CDCF. Note that the vertical axis of these two figures are in a logarithmic scale. We observe that the filtering costs of VGuard and VGuard+ are significantly lower than that of CDCF. On average, VGuard is 552 times faster than CDCF on MSU side and 5035 times faster than CDCF on IBM side; VGuard+ is 544 times faster than CDCF on MSU side and 5021 times faster than CDCF on IBM side. Note that the packet processing time for CDCF on both MSU and IBM side in these two figures seems constant, instead of increasing as the number of rules increases. This is because in processing each packet on both MSU and IBM side, for CDCF, the encryption time, which is roughly constant for each packet, is about 20 times more than the time for performing a linear search in the firewall.

10.2 Efficiency on Synthetic Firewall Policies

Firewall policies are considered confidential due to security concerns. It is difficult to get a large number of

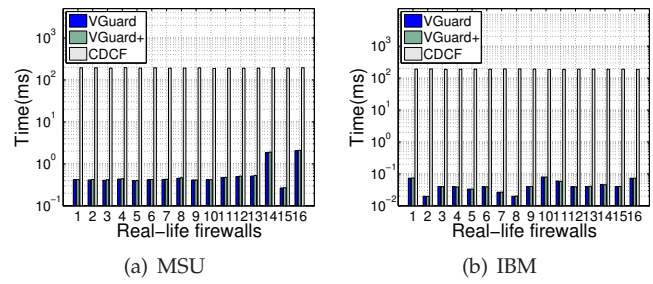


Fig. 16. The filtering time on real-life firewalls

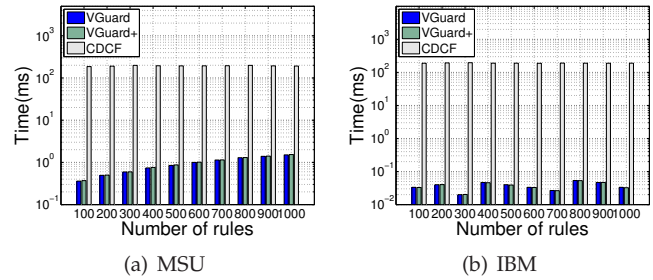


Fig. 18. The filtering time on synthetic firewalls

real-life firewall policies to experiment with. To further evaluate the performance of VGuard and VGuard+ in comparison with CDCF, we generated a large number of synthetic firewall policies and conducted experiments on them. Every predicate of a rule in our synthetic firewall has five fields: source IP address, destination IP address, source port number, destination port number, and protocol type. We first randomly generated a list of values for each field. For IP addresses, we generated a random class C address then generated single IP addresses within the class C addresses; for ports we generated a random range; for protocols, we choose either TCP, UDP, or ICMP. Every field also has the "*" value included in the list. We then generated a list of predicates by taking the cross product of these five lists and randomly selected from the cross product until we reached our desired classifier size by including a final default predicate. Finally, we randomly assigned one of two decisions, accept or discard, to each predicate to make a complete rule. We generated firewall policies with the number of rules ranging from 100 to 1000, where for each number we generated ten synthetic firewall policies.

Figures 17(a) and 17(b) show the computational cost of MSU and IBM in the bootstrapping protocol for VGuard, VGuard+, and CDCF. Figures 18(a) and 18(b) show the computational cost of MSU and IBM respectively in the filtering protocol for VGuard, VGuard+, and CDCF. On average, for these synthetic firewall policies, VGuard is 252 times faster than CDCF on the MSU side and 5529 times faster than CDCF on the IBM side; VGuard+ is 248 times faster than CDCF on MSU side and 5513 times faster than CDCF on IBM side.

11 CONCLUDING REMARKS

In this paper, we propose VGuard, a privacy preserving framework for collaborative enforcement of firewall policies. In terms of security, compared with the state-of-the-art CDCF scheme, VGuard is more secure because

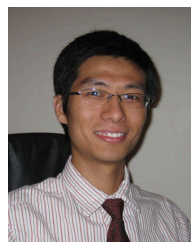
of two major reasons. First, VGuard converts a firewall policy of an ordered list of overlapping rules to an equivalent non-ordered set of non-overlapping rules, which enables rule shuffling and consequently MSU cannot identify which original rule matches the given packet. Second, VGuard obfuscates rule decisions, which prevents MSU from knowing the decision for the given packet. In terms of efficiency, compared with the state-of-the-art CDCF scheme, VGuard is hundreds of times faster than CDCF in processing packets because of two reasons. First, VGuard uses a new oblivious comparison scheme proposed in this paper, which is three orders of magnitude faster than the commutative encryption scheme used in CDCF. Second, VGuard uses firewall decision diagrams for processing packets, which is much faster than the linear search used in CDCF. We want to emphasize that the VGuard framework can be applied to other types of security policies as well. It is also worth noting that the Xhash scheme can be used for other applications that require oblivious comparison.

REFERENCES

- [1] Cisco ios ips deployment guide, www.cisco.com.
- [2] Tippingpoint x505, www.tippingpoint.com/products_ips.html.
- [3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proc. ACM Int. Conf. on Management of Data (SIGMOD)*, pages 563–574, 2004.
- [4] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333C340, June 1975.
- [5] Fabrice Boudot. Efficient proofs that a committed number lies in an interval. In *Proc. Advances in Cryptology (EUROCRYPT)*, volume 1807 of *Lecture Notes in Computer Science*, May 2000.
- [6] Yeim-Kuan Chang. Fast binary and multiway prefix searches for packet forwarding. *Computer Networks*, 51(3):588–605, 2007.
- [7] David Chaum, Claude Crepeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proc. ACM Symposium on Theory of Computing*, pages 11–19, 1988.
- [8] Fei Chen and Alex X. Liu. SafeQ: Secure and efficient query processing in sensor networks. In *Proc. IEEE Int. Conf. on Computer Communications (INFOCOM)*, 2010.
- [9] Jerry Cheng, Hao Yang, Starsky H.Y. Wong, and Songwu Lu. Design and implementation of cross-domain cooperative firewall. In *Proc. IEEE Int. Conf. on Network Protocols (ICNP)*, 2007.
- [10] Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proc. Colloquium, on Automata, Languages and Programming*, pages 118–132, 1979.
- [11] Ronald Cramer, Matthew K. Franklin, Berry Schoenmarks, and Moti Yung. Multi-authority secret-ballot elections with linear work. In *Proc. Advances in Cryptology (EUROCRYPT)*, volume 1070 of *Lecture Notes in computer Science*, 1996.
- [12] D. Eastlake and P. Jones. Us secure hash algorithm 1 (SHA1). *RFC 3174*, 2001.
- [13] Li Fan, Pei Cao, Jussara Almeida, and Andrei Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proc. ACM SIGCOMM*, September 1998.
- [14] Philippe Golle, Jessica Staddon, and Brent Waters. Secure conjunctive keyword search over encrypted data. In *Proc. Int. Conf. on Applied Cryptography and Network Security (ACNS)*, pages 31–45, 2004.
- [15] Mohamed G. Gouda and Alex X. Liu. Structured firewall design. *Computer Networks Journal (Elsevier)*, 51(4):1106–1120, 2007.
- [16] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.
- [17] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. ACM Int. Conf. on Management of Data (SIGMOD)*, pages 216–227, 2002.
- [18] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *Proc. Int. Conf. on Very Large Data (VLDB)*, pages 720–731, 2004.
- [19] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication. *RFC 2104*, 1997.
- [20] Jiangtao Li and Ninghui Li. Oacerts: Oblivious attribute certificates. In *Proc. Conf. on Applied Cryptography and Network Security (ACNS)*, pages 301–317, June 2005.
- [21] Alex X. Liu and Mohamed G. Gouda. Diverse firewall design. In *Proc. Int. Conf. on Dependable Systems and Networks (DSN)*, pages 595–604, June 2004.
- [22] Wenbo Mao. Guaranteed correct sharing of integer factorization with off-line shareholders. In *Proc. Public Key Cryptography (PKC)*, volume 1431 of *Lecture Notes in Computer Science*, February 1998.
- [23] Silvio Micali, Oded Goldreich and Avi Wigderson. How to play any mental game. In *Proc. ACM Conf. on Theory of computing*, 1987.
- [24] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.
- [25] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over $gf(p)$ and its cryptographic significance. *IEEE Transactions Information and System Security*, IT-24:106–110, 1978.
- [26] R. Rivest. The md5 message-digest algorithm. *RFC 1321*, 1992.
- [27] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. Systems Administration Conference (LISA)*, *USENIX Association*, pages 229–238, 1999.
- [28] David K. Hess David R. Safford and Douglas Lee Schales. Secure RPC authentication (SRA) for TELNET and FTP. Technical report, 1993.
- [29] Bo Sheng and Qun Li. Verifiable privacy-preserving range query in two-tiered sensor networks. In *Proc. IEEE Int. Conf. on Computer Communications (INFOCOM)*, pages 46–50, 2008.
- [30] Jing Shi, Rui Zhang, and Yanchao Zhang. Secure range queries in tiered sensor networks. In *Proc. IEEE Int. Conf. on Computer Communications (INFOCOM)*, 2009.
- [31] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proc. IEEE Symposium on Security and Privacy*, 2000.
- [32] Andrew C. Yao. Protocols for secure computations. In *Proc. IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 160–164, 1982.
- [33] Andrew C. Yao. How to generate and exchange secrets. In *Proc. IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 162–167, 1986.
- [34] Rui Zhang, Jing Shi, and Yanchao Zhang. Secure multidimensional range queries in sensor networks. In *Proc. ACM Int. Symposium on Mobile Ad Hoc Networking and Computing*, 2009.



Alex X. Liu received his Ph.D. 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 received the IEEE & IFIP William C. Carter Award in 2004 and the National Science Foundation CAREER Award in 2009. His research interests focus on networking, security, and dependable systems.



Fei Chen received the BS degree in Automation from Tsinghua University in 2005 and the MS degree in Automation from Tsinghua University in 2007. He is currently a PhD student in the Department of Computer Science and Engineering at Michigan State University. His research interests include on networking, algorithms, and security.