Privacy Preserving Collaborative Enforcement of Firewall Policies in Virtual Private Networks

Alex X. Liu, *Member*, *IEEE*, and Fei Chen, *Student Member*, *IEEE*

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 nonoverlapping 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.

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, 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

• *The authors are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824-1266. E-mail: {alexliu, feichen}@cse.msu.edu.*

Manuscript received 10 July 2009; revised 25 Feb. 2010; accepted 13 May 2010; published online 23 Aug. 2010.
Recommended for acceptance by X. Li.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-07-0311.
Digital Object Identifier no. 10.1109/TPDS.2010.155.

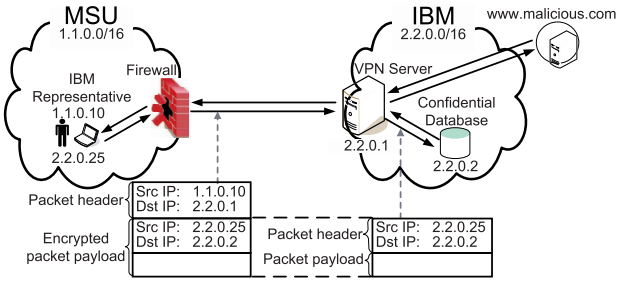

Fig. 1. A typical VPN example.

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 [5], 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 [13] and Secure RPC Authentication (SRA) [16]), 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 VGaurd 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 5,035 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.

1.6 Structure of Supplemental Material

Due to the space limitation, we present four sections: *Background*, *Discussion*, *Related Work*, and *Experimental Results* in the supplemental material (which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.155). In the *Background* section, we formally define the concepts of fields, packets, firewall rules, and firewall policies. In the *Discussion* section, we discuss the eight remaining issues, firewall updates, decision caching, decision obfuscation versus decision encryption, special treatment of IP addresses, *etc*. In the *Related Work* section, we review the previous research of CDCF framework, secure function evaluation, and secure queries. In the *Experimental Results* section, we evaluated the performance of our schemes on both real-life and synthetic firewall policies, and compared our schemes with CDCF side by side.

2 Threat Model

First, we assume that the two parties of policy owner MSU and request owner IBM are semihonest; 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 [7], [10], [14]. 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 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

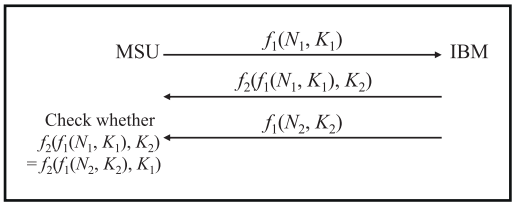

Fig. 2. The oblivious comparison protocol.

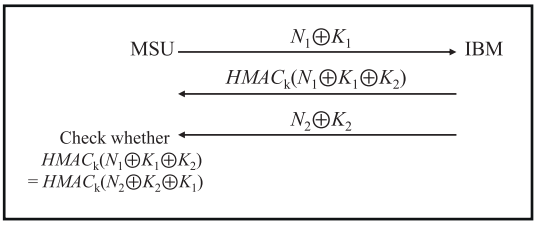

Fig. 3. The Xhash protocol.

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.

3.1 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.

3.2 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.

3.3 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.

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 pseudorandom 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 $x, x_1, x_2, \ldots$ 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.

3.4 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 [13]. A commutative encryption function satisfies the following four properties, where $(x)_K$ denotes the encryption of x using key K:

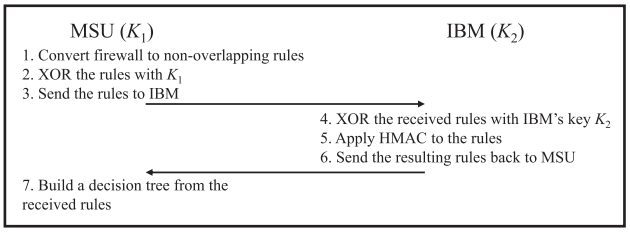

Fig. 4. The bootstrapping protocol.

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 5.

4 Bootstrapping Protocol

In the bootstrapping protocol, MSU first converts its firewall policy to a set of nonoverlapping 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.

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

4.1 FDD Construction

In this step, MSU converts its firewall policy to an equivalent *Firewall Decision Diagram (FDD)* [8]. An 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 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 $F(v) \in \{F_1, \ldots, F_d\}$. 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*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$.

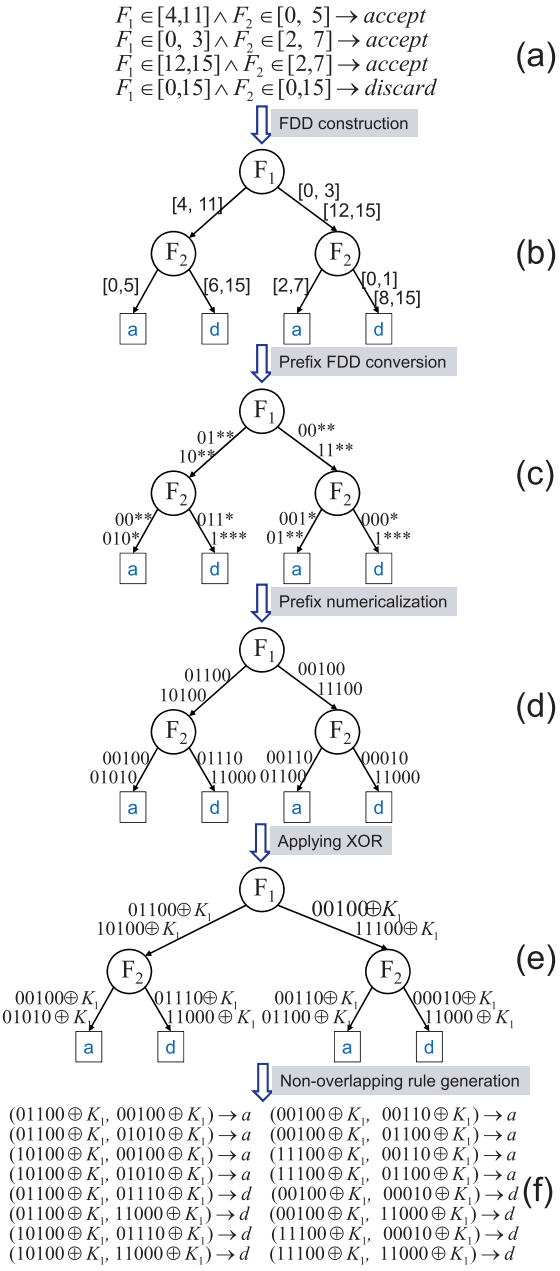

Fig. 5. Example of bootstrapping at MSU.

Fig. 5a 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. 5b. 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 [11].

4.2 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 the 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 $2w - 2$ [9]. We call such FDDs, where each edge is labeled by a set of prefixes, *prefix FDDs*. Fig. 5c shows the prefix FDD converted from the FDD in Fig. 5b.

4.3 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 [4]. Given a prefix $b_1 b_2 \cdots b_k * \cdots *$ of w bits, we first insert 1 after b_k. The bit 1 represents a separator between $b_1 b_2 \cdots b_k$ and $* \cdots *$. 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. 5c becomes the one in Fig. 5d.

4.4 Applying XOR by MSU

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

$$F_1 \in 01100 \oplus K_1 \wedge F_2 \in 00100 \oplus K_1 \rightarrow a,$$
$$F_1 \in 01100 \oplus K_1 \wedge F_2 \in 01010 \oplus K_1 \rightarrow a,$$
$$F_1 \in 10100 \oplus K_1 \wedge F_2 \in 00100 \oplus K_1 \rightarrow a,$$
$$F_1 \in 10100 \oplus K_1 \wedge F_2 \in 01010 \oplus K_1 \rightarrow a.$$

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

After nonoverlapping rules are generated, MSU sends the resulting policy to IBM. If MSU needs to prevent IBM from knowing the number of nonoverlapping 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 nonoverlapping 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 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 nonoverlapping prefix rules that a firewall is converted to far exceeds the number of original rules.

4.5 Applying XOR and HMAC by IBM

Upon receiving a sequence of nonoverlapping numerical rules from MSU, IBM further applies XOR and HMAC to every number in the received policy using its secret key K_2.

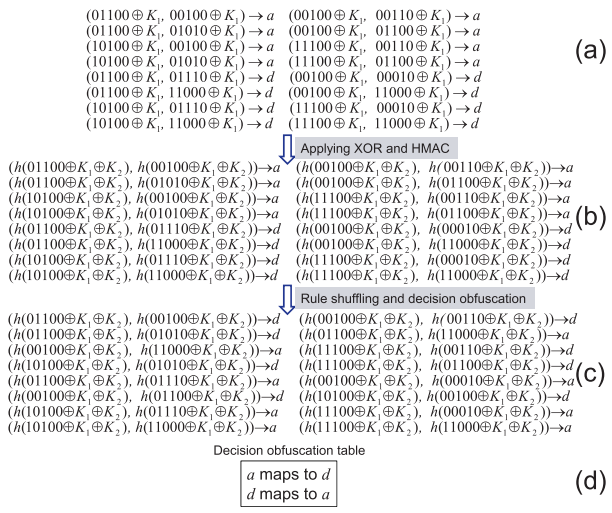

Fig. 6. Example of bootstrapping at IBM.

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. 6b shows the rules after IBM applies XOR and HMAC, and Fig. 6c shows the rules after IBM shuffles rules and obfuscates decisions. The obfuscation mapping function is shown in Fig. 6d. Note that in these figures, h denotes the HMAC function. Finally, IBM sends the resulting rules to MSU.

5 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, and 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.

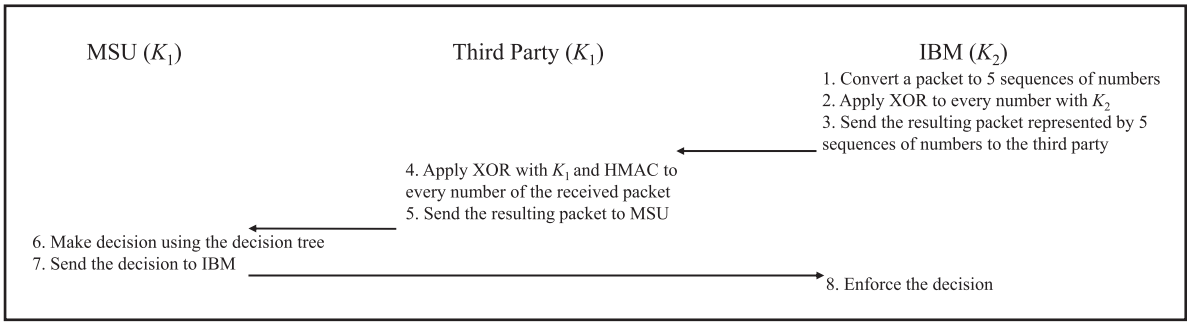

Fig. 7. The filtering protocol.

5.1 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 the IBM representative's MSU IP address. Otherwise, it is likely that the 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.

5.2 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 $b_1b_2\cdots b_w$ of w bits, the prefix family of this number is the set of $w+1$ prefixes $\{b_1b_2\cdots b_w, b_1b_2\cdots b_{w-1}*, \cdots, b_1*\cdots*, **\ldots*\}$, where the ith prefix is $b_1b_2\cdots b_{w-i+1}*\cdots*$. We use $PF(x)$ to represent the prefix family of x. For example, $PF(0101) = \{0101, 010*, 01**, 0***, ****\}$. 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 $\mathcal{P} \in PF(x)$.

5.3 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. 8a, the prefix family of each field is shown in Fig. 8b. The result of prefix numericalization is shown in Fig. 8c. The final two sequences of numbers are shown in Fig. 8d.

5.4 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 (i.e., Fig. 8d) 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. 5d). Let $\langle r_1, \ldots, r_n\rangle$ denote these rules, where each rule $r_j (1 \le j \le n)$ is in the form $(m_1^j, \ldots, m_d^j) \rightarrow \langle dec^j\rangle$. In the filtering protocol, when MSU finds that a packet $(p_1, p_2, \ldots, p_d)$ matches a rule, according to the property of prefix membership verification, for each $1 \le i \le d$, there must be a number n_i in $PF(p_i)$ that is equal to one number in the set $\{m_i^1, \ldots, m_i^n\}$. Third, for each $1 \le i \le d$, MSU XORs $n_i \oplus K_2$ received from IBM with every number in the set $\{m_i^1, \ldots, m_i^n\}$. Because one of the numbers in $\{m_i^1, \ldots, m_i^n\}$ is equal to n_i, the resulting set, denoted S_i, must contains K_2. For example, if $m_i^1 = n_i$, then $m_i^1 \oplus n_i \oplus K_2 = K_2$. Thus, for each packet P, we can

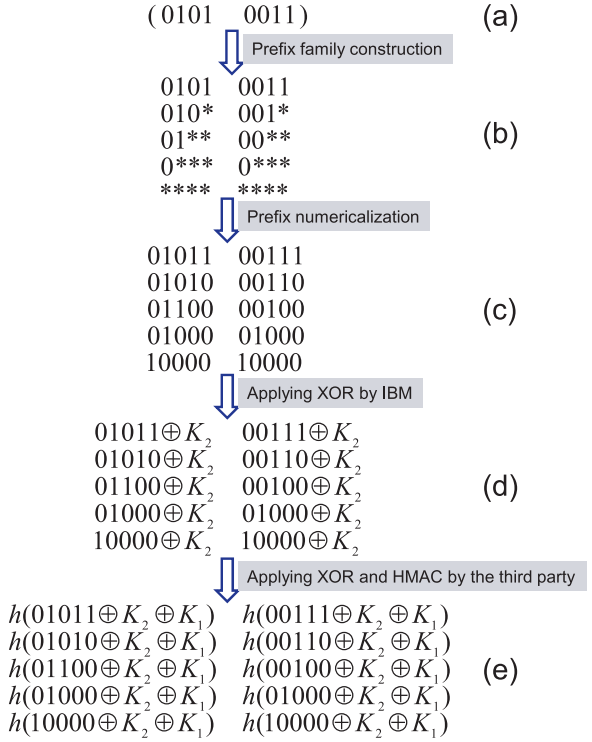

Fig. 8. Example of packet preprocessing.

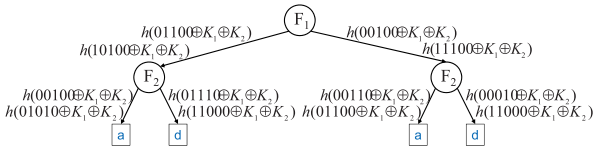

Fig. 9. Decision tree constructed from Fig. 6c.

compute a set $S(P) = S_1 \cap \cdots \cap S_d$, which contains K_2. When MSU receives a large set of packets $P_1, \ldots, P_g$, the set $S(P_1) \cap \cdots \cap S(P_g)$ may only contain 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.

5.5 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 $(p_1, \ldots, p_d)$ matches a rule $(m_1, \ldots, m_d) \rightarrow \langle obfuscated\ decision \rangle$ if and only if the condition $m_1 \in PF(p_1) \wedge \cdots \wedge m_d \in PF(p_d)$ holds. Therefore, MSU can use the 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 nonoverlapping, there exists one and only one rule that the packet matches. For example, given the resulting firewall rules in Fig. 6c and the preprocessed packet in Fig. 8e, 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 nonoverlapping 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. 6c. 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 $PF(p_i)$.

6 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 [15], Bro [12], 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 $a_1 a_2 \cdots a_n$ and a packet payload $s_1 s_2 \cdots s_m$ 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 $s_1 s_2 \cdots s_m$ contains the substring $s_{k+1} s_{k+2} \cdots s_{k+n}$ that is the same as the string $a_1 a_2 \cdots a_n$. If so, the packet payload $s_1 s_2 \cdots s_m$ matches the string $a_1 a_2 \cdots a_n$.

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.

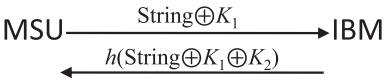

Fig. 10. Bootstrapping for deep packet inspection.

6.1 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 ith $(1 \leq i \leq n)$ 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 $O(\sum_{i=1}^{n} c_i)$.

Considering three strings "eb, ebf, ecg" in Fig. 11a, Fig. 11b shows the anonymized string s after MSU applies XOR to these strings. Fig. 11c shows the resulting strings after MSU adds the random string $r_1 r_2$, where r_1 and r_2 denote two random characters. Fig. 11d shows the resulting strings after IBM adds the dummy string $d_1 d_2$, where d_1 and d_2 denote two random characters. Fig. 11e shows the strings after IBM applies XOR and HMAC, and Fig. 11f 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.

6.2 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 [3] and Commentz-Walter algorithm [6]. MSU can use these algorithms to search the

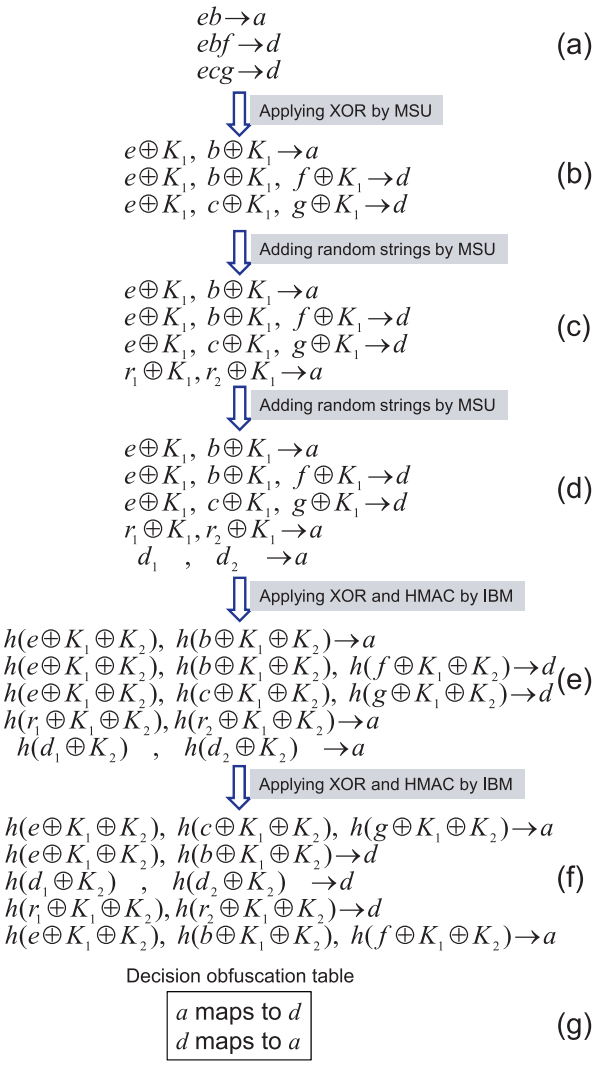

Fig. 11. Example of string processing.

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

7 Summary of 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

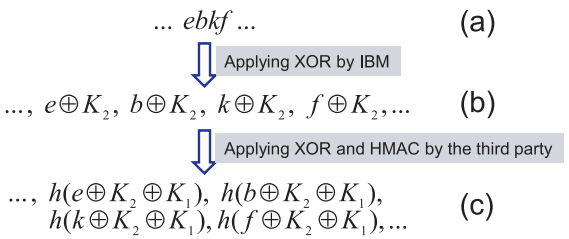

Fig. 12. Example of packet payload processing.

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 5,035 times faster than CDCF on IBM side; VGuard+ is 544 times faster than CDCF on MSU side and 5,021 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 5,529 times faster than CDCF on IBM side; VGuard+ is 248 times faster than CDCF on MSU side and 5,513 times faster than CDCF on IBM side.

8 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 nonoverlapping 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.

Acknowledgments

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

References

[1] "Cisco IOS IPS Deployment Guide," www.cisco.com, 2010.
[2] "TippingPoint X505," www.tippingpoint.comproducts_ips.html, 2009

[3] A.V. Aho and M.J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Comm. ACM,* vol. 18, no. 6, pp. 333-334, June 1975.
[4] Y.-K. Chang, "Fast Binary and Multiway Prefix Searches for Packet Forwarding," *Computer Networks,* vol. 51, no. 3, pp. 588-605, 2007.
[5] J. Cheng, H. Yang, H.Y. Starsky Wong, and S. Lu, "Design and Implementation of Cross-Domain Cooperative Firewall," *Proc. IEEE Int'l Conf. Network Protocols (ICNP),* 2007.
[6] C.-W. Beate, "A String Matching Algorithm Fast on the Average," *Proc. Sixth Colloquium Automata, Languages and Programming,* pp. 118-132, 1979.
[7] D. Eastlake and P. Jones, "US Secure Hash Algorithm 1 (SHA1)," RFC 3174, 2001.
[8] M.G. Gouda and A.X. Liu, "Structured Firewall Design," *Computer Networks J.,* vol. 51, no. 4, pp. 1106-1120, 2007.
[9] P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network,* vol. 15, no. 2, pp. 24-32, Mar./Apr. 2001.
[10] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, 1997.
[11] A.X. Liu and M.G. Gouda, "Diverse Firewall Design," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* pp. 595-604, June 2004.
[12] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks,* vol. 31, nos. 23/24, pp. 2435-2463, 1999.
[13] S.C. Pohlig and M.E. Hellman, "An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance," *IEEE Trans. Information and System Security,* vol. IT-24, no. 1 pp. 106-110, Jan. 1978.
[14] R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, 1992.
[15] M. Roesch, "Snort: Lightweight Intrusion Detection for Networks," *Proc. USENIX Conf. Systems Administration,* pp. 229-238, 1999.
[16] D.K. Hess, D.R. Safford, and D.L. Schales, "Secure RPC Authentication (SRA) for TELNET and FTP," *Proc. Fourth USENIX Security Conf.,* 1993.

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 received the IEEE and IFIP William C. Carter Award in 2004 and a US National Science Foundation (NSF) CAREER Award in 2009. His research interests include networking, security, and dependable systems. He is a member of the IEEE.

Fei Chen received the BS and MS degrees in automation from Tsinghua University, P.R. China, in 2005 and 2007, respectively. He is currently working toward the PhD degree in computer science with a focus on applying algorithmic techniques to networking and security problems. His research interests include networking, algorithms, and security. He is a student member of the IEEE.