S-Blocks: Lightweight and Trusted Virtual Security Function with SGX

Juan Wang, Shirong Hao, Hongxin Hu, Bo Zhao, Hongda Li, Wenhui Zhang, Jun Xu, Peng Liu and Jing Ma

Abstract—Despite the advantages of scalability and flexibility, Security Function Virtualization (SFV) raises concerns about its own security. To enhance the security of SFV, a promising approach is to run critical components of off-the-shelf security software inside Software Guard Extensions (SGX) enclaves. This idea, however, is hardly practical due to the difficulty of detaching components from the monolithic security function and the unacceptable cost of executing them inside enclaves. In this paper, we propose S-Blocks, an architecture to modularize virtual security functions (VSFs) and protect crucial modules with SGX in an efficient manner. S-Blocks decomposes VSFs into trusted and untrusted modules and provides dedicated APIs systematically. Only crucial VSF modules are hardened with enclaves. Furthermore, aiming at addressing state consistency and secure migration issues of security function scaling, we design a fine-grained state synchronization and migration mechanism to ensure loss-free, order-preserving, and state security for VSFs. To demonstrate the effectiveness of our approach, we prototype S-Blocks using Fast-Click on a real Skylake platform and implement three critical types of virtual security functions based on the S-Blocks architecture. Our evaluation results show that S-Blocks only imposes a manageable performance overhead, and low latency and resource consumption when protecting VSFs.

Index Terms—Security Virtual Function, Virtual Security Function, Software Guard Extensions, Intrusion Detection System.

1 Introduction

Security functions are of vital importance to an enterprise network. Traditional security functions are built in hardware boxes. These hardware boxes are protected with isolated and closed hardware devices. They have their own CPU, memory, I/O, and OS. Currently, Security Function Virtualization (SFV) [73] provides a promising way to implement security functions in software, while deploying the security functions on high-volume standard servers and executing them as virtual instances instead of proprietary hardware. SFV can not only reduce both Capital Expenditures (CAPEX) and Operating Expenditures (OPEX), but also speed up software-oriented network innovation so as to bring new security services. Most importantly, it enables network operators and service providers to use virtual instances to easily add or remove security functions, which greatly improves flexibility and scalability. Recently, more and more companies are coming to embrace SFV so as to adapt to increasingly complex network environments and IT virtualization.

Despite many benefits, SFV faces some serious security issues. The first critical threat is that virtual security functions (VSFs) lack strong isolated protection provided by proprietary hardware because VSFs are executed in a shared and open environment with same priviliage of other functions [9], [21]. For example, a virtual Intrusion Detection System (IDS) is usually deployed as a virtual instance on a standard server. It shares CPU, memory, I/O and host OS with other virtual instances, which incurs a larger attack surface. To better secure VSFs, one should provide VSFs with isolation protection similar to dedicated hardware boxes. Most important of all is protection for network states and security policies of VSFs. Network states and security policies are stored in network sensitive data formats (e.g. IP, ports number and communication state, etc.). Leaving these sensitive data in an untrusted environment can cause fatal damage to networks. This threat often occurs in VSFs scaling scenarios. When processing capability of a VSF instance reaches a bottleneck or some traffic needs to be processed separately, programmers and administration operators start new virtual instance and migrate current states to the new instance. In this way, they achieve dynamic scaling. During the scaling, if the new virtual security instance lacks the required detection states and obtains forged states. It may misidentify some attacks as non-attacks.

Existing virtualization techniques enable elastic security [41], [74]. They consider a virtual network function as a monolithic piece of software executing in a virtual machine or container. However, this monolithic design has its own limitations. First, it is difficult to customize a security function if it is provisioned as a monolith. However, security function customization is critical in terms of resource efficiency for advanced network attack defense. Second, the monolithic design of VSFs makes it difficult to create a new VSF agilely. However, our modular design of VSFs decomposes a VSF to several relatively independent elements, which make smaller chained security functions reusable. It is also quick and flexible to create a new VSF based on the existing tiny security functions. Third, pro-

- *Associate Professor Juan Wang and Master Shirong Hao, and Professor Bo Zhao are with Key Laboratory of Aerospace Information Security and Trust Computing, Ministry of Education, school of cyber science and engineering, Wuhan University, Wuhan, China.*
  E-mail: jwang@whu.edu.cn, shirong@whu.edu.cn, zhaobo@whu.edu.cn.
- *Associate Professor Hongxin Hu and PhD Hongda Li are with Clemson University.*
- *Professor Peng Liu and PhD Wenhui Zhang are with Pennsylvania State University.*
- *Assistant Professor Jun Xu is with Stevens Institute of Technology.*
- *Jing Ma is with Science and Technology on Information Assurance Laboratory, Beijing, China.*

Manuscript received January 23, 2019.

tecting monolithic VSFs in run-time with Software Guard Extensions (SGX) has an intractable problem, which could result in a large performance overhead. Taking Snort [65] as an example, we divide Snort code into security-sensitive modules and none security-sensitive modules, and only execute security-sensitive modules in a SGX enclave. The performance overhead of executing security-sensitive Snort modules in the SGX enclave is about 10 times higher than executing them without SGX protection [62]. Such an overhead is mainly introduced by (1) memory encryption and decryption of SGX, and (2) ECall and OCall transition in the enclave. Moreover, we have found that it is impractical to protect security policies and their enforcing procedure in Snort, since the policy processing of Snort almost penetrates all operating procedures and modules of Snort. Hence, if we intend to protect the security policies and network states in Snort, almost all modules of Snort need to be executed in the SGX enclave. This could cause unacceptable performance overhead. Furthermore, previous research studies [29], [34], [35], [45], [60] focus on the protection of general virtual network functions. They couldn't consider the trusted protection of policies and processing states of those policies. For example, in previous work, a virtual machine migration mechanism [13], [43] is adopted for virtual instance scaling. However, such a mechanism can only perform coarse-grained state migration, which may cause the missing detection of some attacks. Besides, those approaches mainly make use of shared and unencrypted buffers, which may leak sensitive network data, for state migration. Therefore, our design goal is to enable a modular architecture for virtual security functions, and protect their code, states and policies using SGX, while achieving lower performance overhead.

To address these issues, we propose S-Blocks, a novel, lightweight and trusted VSF architecture based on SGX. S-Blocks leverages a modular and microservice-oriented architecture to design VSFs. It decomposes modules of VSFs to trusted elements and untrusted elements, and provides proprietary APIs. This makes it easy to build a new security function and put its critical modules and elements into an enclave with low performance overhead. Moreover, aiming at addressing state consistency and secure migration issue of virtual security function scaling, we present a fine-grained state synchronization and migration mechanism to ensure loss-free, order-preserving and state security for VSFs. To demonstrate the effectiveness of our approach, we prototype S-Blocks using Fast-Click on real Skylake platform. We design and implement three types of VSFs, including Distributed Denial of Service (DDoS) detection and defense, firewall and Intrusion Detection System (IDS).

To our knowledge, S-Blocks is the first practical work that achieve chain-able Click-based security functions while protects their code, state, policy with SGX. S-Blocks achieves a reasonable performance overhead. In this work, we make the following contributions.

- We propose S-Blocks, a novel lightweight and trusted VSF architecture based on SGX. It protects code, policies, and sensitive states of VSFs. It provides isolated and trusted box similar to the dedicated hardware. S-Blocks provides the modular design and achieves better trade-off between performance and security.
- We propose a fine-grained state synchronization scheme to address the issue of secure state synchronization for VSFs. It considers a data flow as a processing unit to synchronize different types of flow states. It achieves loss-free and order-preserving at a smaller granularity. Furthermore, we leverage SGX remote attestation mechanism to protect internal sensitive states during the scaling and migration of VSFs. We also design and implement the state synchronization scheme and provide corresponding APIs.
- We evaluate function and performance of S-Blocks using three types of virtual security functions as use cases. We design and implement the DDoS detection and defense function, firewall and IDS. We validate them based on real Sky-lake platform. Our evaluation results show that S-Blocks introduces a manageable performance overhead while providing security functions with trusted protection. S-Blocks is open-source and is available at https://github.com/S-Blocks-impl/S-Blocks.

The rest of this paper is organized as follows. In Section 2, we present background. Section 3 gives an illustration of our threat model and S-Blocks overview. Section 4 presents our detailed design. We describe the implementation of S-Blocks in Section 5. Section 6 discusses the evaluation of S-Blocks . Section 7 introduces the related work. Discussion and future work are presented in Section 8. Section 9 concludes the paper.

2 Background

In this section, we describe background of Security Function Virtualization and Intel SGX.

2.1 SFV

Security Function Virtualization (SFV) [5], [32], [73] is an emerging architecture that migrates security functions from dedicated hardware appliances to software. It makes security functions easily execute in commodity computers or cloud. Traditional network security functions consist of proprietary hardware boxes, usually including Application Specific Integrated Circuits (ASIC) to perform specific security tasks (e.g. IDS). These security devices are often costly and cannot be customized. In addition to this, traditional network security devices can hardly provide scalable defense adapted with attack traffic volume. For example, when DDoS traffic volume grows, a DDoS detection and defense function should also increase its processing power accordingly.

2.2 Intel SGX

Intel's Software Guard Extensions (SGX) [14] technology is a set of Instruction Set Architecture (ISA) extensions for the Trusted Execution Environment (TEE). It is released as part of Skylake processor architecture. It contains two sets of extended instruction sets, SGX1 and SGX2. SGX1 allows applications to instantiate a protected container called enclave.

The enclave is a protected area of the application's address space. Even with malicious privileged software, SGX can guarantee confidentiality and integrity of program code and data in the enclave. SGX1 can prevent any unauthorized programs, even privileged ones, from accessing an enclave that does not belong to them. SGX2 provides greater flexibility for resource management and thread management during enclave operation, such as adding memory after enclave creation, and adding threads. However, SGX2 is not available on off-the-shelf commodity computers.

Enclave Page Cache (EPC) is a trusted memory area, currently restricted to 128M. It is encrypted and protected by a memory encryption engine (MEE). Processor uses Enclave Page Cache Map (EPCM) to track the metadata of EPC. This structure can only be accessed by CPU. The MEE executes memory encryption and decryption while writing and reading the EPC. When the EPC is insufficient, the rarely used EPC pages will be swapped to untrusted DRAM pages outside Processor Reserved Memory (PRM) range by using a secure paging mechanism. It incurs very high performance overhead due to memory encryption and decryption and translation look-aside buffer (TLB) flush during swapping EPC pages. Hence, the code and data put in enclaves should be minimized.

The code executing in an enclave is prohibited by system calls (i.e. ECall and OCall). The fundamental reason is that code in the enclave runs in user mode, and these user mode code should go through ECall to evoke kernel mode functions. OCall is just the opposite, which is used when the code in the enclave needs to call the external untrusted code. The enclave needs to perform security checks during ECall and OCall. This brings large overhead cost. Although SGX has been optimized in this respect, the overhead still cannot be avoided in most cases. Therefore, it requires limiting the number of ECalls and OCalls when separating program and putting the trusted part into an enclave.

SGX remote attestation is a mechanism by which a third party validates that an application is executing in enclave on the Intel SGX enabled platform. The remote attestation process requires the attestation service provided by the Intel Attestation Server (IAS). IAS is responsible for providing a public critical certificate that verifies the report by the authentication platform. The services provided by using IAS need to be registered with Intel and provide certificates obtained from Intel-approved certificate authorities. For testing purposes, the Independent Software Vendor (ISV) may use a self-signed certificate generated using OpenSSL [52] instead of one signed by an authority. After the registration is passed, the public critical certificate of the Service Provider ID (SPID) and the verification report is obtained.

3 Overview

In this section, threat model and system architecture is briefly explained.

3.1 Threat Model

In this paper, we aim at building an isolated box for virtual security functions and protecting their internal states and policies. In addition, we also focus on the security issues that exist during the dynamic scaling of virtual security functions. Virtual security functions are usually deployed as virtual instances, hence they suffer security threats from virtualization platform. For instance, the vulnerable privileged entities, such as VMM, OS and cloud administrators, often have access to memory and virtual instances so that they may leak sensitive data of virtual security functions. Meanwhile, other vulnerable virtual security instances may also obtain private data due to weak isolation mechanism of cloud platforms.

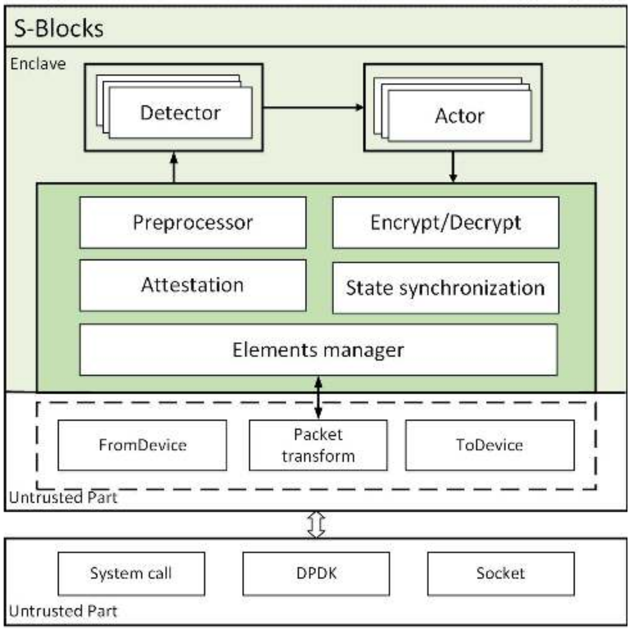


Fig. 1. Overview of S-Blocks.

Therefore, our threat model assumes that only CPU and the code executing in enclave are trusted. Privileged software (i.e. operating systems, hypervisor and BIOS) is untrusted because they may be vulnerable and exploited by attackers. Attackers also can launch physical attacks on memory and I/O devices. Side channel attacks [12], [23], [68], such as time-based side channel and cache-based side channel attacks, on SGX are beyond the scope of this paper. S-Blocks can be compromised if the code in the enclaves contains software vulnerabilities and is subject to controlled side channel attacks [72]. Recently, a number of approaches have been presented to solve and mitigate those attacks [11], [15], [27], [63], [64]. Solutions to preventing SGX side attacks are orthogonal to our contribution.

3.2 System Architecture

Our goal is to design a lightweight and trusted execution environment for virtual security functions based on SGX. Lightweight refers to that the system has a small overhead and minimal modules for a specific security requirement. Our architecture supports extensible and stackable security functions.

Aiming at this goal, we propose S-Blocks, a lightweight and trusted virtual security function architecture. S-Blocks has three advantages. First, the modular architecture of

S-Blocks allows developers to quickly build new virtual security function by using the basic and stackable elements. It is easy to put critical modules and elements into an enclave without decoupling a security function while only imposing a little performance overhead. Second, S-Blocks provides the approach of fine-grained state consistency and secure migration. It can securely process packets in flow context so as to support VSFs working on L2-L6 traffic and enable dynamic and trusted VSFs scaling. Finally, S-Blocks provides trusted protection for the policy of virtual security functions.

Fig. 1 shows the critical components of S-blocks. In the design of S-Blocks, a virtual security function is divided into several critical modules including *Preprocessor*, *Detector*, *Actor*, *Encrypt/Decrypt*, *Attestation*, *State synchronization*, *Elements manager*, and others communication processing modules, such as *Packet transform*, *FromDevice*, *ToDevice*, *DPDK*, *Socket*, etc.. Every module can contain one or more elements. When a virtual security function receives network packets, the *Preprocessor* module performs some basic packets processing operations, such as assembling and so on. Then *Detector* module sends data packets to different action elements in the *Actor* module based on the packets processing results and the security policy rules. The *Encrypt/Decrypt* and *Attestation* module are mainly responsible for encrypt/decrypt packets, seal/unseal policy file and build a secure communication channel. The *State synchronization* module is designed for fine-grained state consistency. In addition, we add the *Elements manager* module in order to connect the related elements and modules.

In order to provide security protection for the critical code, policies and internal states of the virtual security function, we put critical modules into SGX enclaves. Therefore, we carefully divide a virtual security function into two parts: trusted part and untrusted part. The trusted part is run in the SGX enclave and the untrusted part is run outside the enclave. Since, the protected memory size of SGX is restricted to 128M (the available maximum memory is less than 90 MB), the trusted modules or elements put in enclaves should be minimal. In S-Blocks, the trusted part is composed of seven critical modules related with sensitive packets, state and policy processing: *Preprocessor*, *Detector*, *Actor*, *Encrypt/Decrypt*, *Attestation*, *State synchronization*, and *Elements manager*. These modules will be elaborated in the following section.

Considering that the trusted base should be as small as possible, we place security insensitive modules (i.e. Socket, and DPDK) out of the SGX enclave. The untrusted components include the *FromDevice* module, the *ToDevice* module, the *Packet transform* module, etc.. *FromDevice* module reads packets from network device using Intel's DPDK. Each encrypted packet arriving from the network is first copied into the enclave by *Packet transform* module, where its signature is checked and its content is decrypted. It is then processed by middlebox functions, accepted or discarded, and finally encrypted and copied outside the enclave and passed to the network. *Todevice* element sends packets to network device using Intel's DPDK.

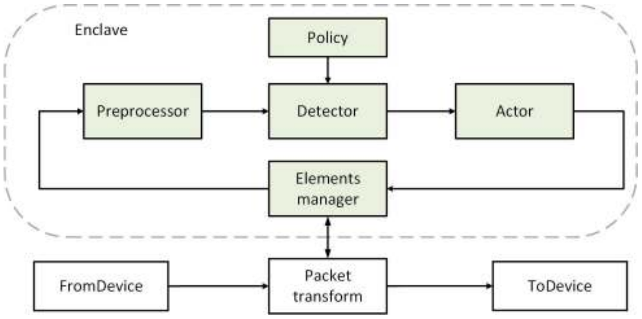


Fig. 2. Workflow of S-Blocks.

4 Design

S-Blocks presents an isolated and trusted box for VSFs. The critical elements of virtual security functions are put into the SGX enclave, which is an isolated environment like the propriety hardware device. Furthermore, we propose a state consistency scheme and secure state migration approach to achieve trusted state protection during multiple virtual security instance scaling. Last but not least, we present our policy protection approach. The details about S-blocks design are described as follows.

4.1 Isolated and Trusted Box

Aiming to provide an isolated box similar to a hardware device for a virtual security function using SGX, we put the critical modules that are related to sensitive data into enclaves. In S-Blocks, we design and put the code of *Preprocessor* module, *Detector* module, *Actor* module, *Encrypt/Decrypt* module, *Attestation* module, *State synchronization* module and *Elements manager* module into enclaves.

There are two solutions to achieve code protection of virtual security functions. The first one is that we put each module in a separate enclave. The system performs local authentication between enclaves and then enclaves call each other to enable packet processing between different modules. Although this scheme can achieve better isolation and flexibility, it incurs huge performance overhead due to authentication and packet transmission between enclaves. Even when only the header of packets are checked when packets are passed between elementsthe performance of S-Blocks overhead is non-trivial. The second solution is to put the important and sensitive modules in an enclave. This solution reduces the overhead when packets are received by and sent from enclaves. Considering performance, we follow the second option. The critical modules and workflow of S-Blocks are shown in Fig. 2.

Trusted components: Trusted code in the SGX enclave performs core functionality on sensitive data, policies, and states. As a result, the trusted code contains several components, which implements the following functions:

(1) *Preprocessor* module: The *Preprocessor* module contains many different elements that perform various packet processing operations depending on its functionality. For example, one Preprocessor elements can check the length of the packet or classifier the packets by contents.

(2) *Detector* module: This module receives the output of the Preprocessor module. It is responsible for forwarding packets according to the security policies and the labels

of the packets. For example, firewall function uses firewall rule-set, a set of security policy, to filter network traffic.

(3) *Actor* module: The *Action* module contains a number of elements that conduct different defense actions on packets, such as discarding, rate limiting, quarantining, alerting, logging, outputting and so on.

(4) *Encrypt/Decrypt* module: This module mainly provides secure cryptography operations for S-Blocks. The critical functions of this module include keys generation, keys storage, encrypt/decrypt operations and seal/unseal operations.

(5) *Attestation* module: This module is responsible for building secure communication between virtual instances or user and virtual functions based on the remote attestation function provided by SGX.

(6) *State synchronization* module: This module provides a fine-grained state consistency and migration, which will be described in detail in Section 4.2.

(7) *Elements manager* module: *Elements manager* module is responsible for initializing and managing the elements in the aforementioned modules. In addition, it passes packets to other modules in the enclave. When initialized, the Elements manager module converts the network function configuration issued by the SFV controller into a directed graph and corresponding policies. The all the other modules are connected according to the directed graph.

After initialization, the *Elements manager* module starts to manage the other modules. The *Elements manager* has three major tasks. First, it receives packets from *Packet transform* module and sends packets to the *Packet processor* module. After finishing processing of packets in enclave, *Elements manager* module sends the results to the *Packet transform* module. Second, it collects the states of each module in the enclave when it needs to copy an instance or copy a packet processing state. Third, it restores the state of each module and communicates with the *State synchronization* module and the *Attestation* module when a new VSF instance is launched.

For S-Blocks, we define generic APIs to assist developers developing security functions with enclaves. Developers provide a configuration file written in Click configuration language, which can define which modules should be put in the enclave and the connections between different modules, for a new security function. Then, the *int CreateSF (file *)* API can create the security function according to the configuration file and returns the id of this security function. Meanwhile, the code of the related elements in the configuration file is put into the enclave. Finally, this enclave code is recompiled to generate a new element. The new element implements the original functionality of the security function. When a security function needs to be updated, the *int UpdateSF (file *, int SFid)* API can be used to update the security function according to a new configuration file, which contains the new modules that should be run in enclave. In addition, *int DestorySF (int SFid)* API can be used to directly destroy a security function based on its ID. If S-Blocks defines the system call in common use with the Ocall function previously, we can avoid hand-tuning to ensure all modules indeed fit in an enclave to some extend. However, if the module contains the system call, which was not defined previously, we need to hand tune the code to ensure all modules could run in an enclave.

In order to facilitate developers to better handle policy file and data packets, we design four APIs for the encryption and decryption module. When the policy file needs to be stored on the hard disk, the *void Seal (file *)* function will encrypt the policy file and write it to the disk. The *void Seal (file *)* function seals the policy file to the current enclave using the current version of the enclave measurement(i.e. MRENCLAVE). Only an enclave with the same MRENCLAVE measurement will be able to unseal the data that was sealed in this manner, which prevents attackers from illegally unsealing policy file.

When S-Blocks starts up, the *void Unseal (file *)* function reads the file from the disk, then decrypts the policy file. In addition, the *void EncryptPacket (packet *, int EncryptLength)* and *void DecryptPacket (packet *, int DecryptLength)* functions are used to process encrypted network traffic. Since packets may be processed by multiple network functions, it is not necessary to seal packets. Therefore we only provide the encrypt/decrypt APIs for secure processing of packets. The functional description of the above APIs is shown in Table 1.

S-Blocks workflow: When the system starts, a directed graph is generated by parsing the configuration file. Then *FromDevice* element reads encrypted packets from network device using Intel's DPDK and sends them to the Packet transform module. The ECalls of *Packet transform* module indirectly bootstrap S-Blocks enclave and copy the traffic into the enclave, where the signature of packets is checked and the content of packets is decrypted. Traffic is then processed by detector and actor modules. After finishing the processing, *Elements manager* module sends the results to the Packet transform module. Finally, the *Packet transform* element sends packets to *ToDevice* element. *ToDevice* element sends packets to network device using Intel's DPDK.

Although we have presented a design and recommended that some trusted modules should be protected by the enclave in S-Blocks, developers can determine which modules should run in the enclave according to their security requirements, and define them by a configuration file. Then, those modules can be loaded and run in the enclave by calling the API *int CreateSF (file *)* shown in Table 1. In addition, S-Blocks decomposes VSFs into several separately deployable and smaller elements. Different elements can be reused to reduce code redundancy, which can further reduce the size of enclave usage. When enclaves are larger than the total memory available to the Enclave Page Cache (EPC), EPC paging [76] can evict the rarely used memory pages to DRAM pages outside the PRM (Processor Reserved Memory) range with the encrypted mode. Since such a process may introduce some overhead, developers should carefully consider the minimum modules that need to be protected in enclaves.

4.2 Trusted State

When processing capability of a VSF instance reaches bottlenecks or some data flows need to be processed separately, we need to create a new VSF instance and migrate the state of the original VSF instance to the new instance. In this way,

TABLE 1
APIs for Security Function Development

API	Functionality
int CreateSF (file *)	Create a security function based on a configure file, return the id of security function.
int UpdateSF (file *, int SFid)	Update the configure file of a security function according to the SFid.
int DestroySF (int SFid)	Destroy a security function according the security function id.
void EncryptPacket (packet *, int EncryptLength)	Encrypt packets.
void DecryptPacket (packet *, int DecryptLength)	Decrypt packets.
void Seal (file *)	Seal a policy file.
void Unseal (file *)	Unseal a policy file.

the dynamic scaling can be achieved. SFV has the ability to instantiate multiple instances executing on different virtual machines and dynamically scale by destroying or creating instances. Each instance takes a part of traffic and maintains its own detection states. If the traffic is delivered to an instance that lacks the required detection states, the VSFs may miss some attacks. For example, a scanning detector usually maintains a counter to count how many flows are generated by each host. If a flow is delivered to an instance that does not maintain its counter, this flow may be overlooked.

When an security function is scaled out, the SFV controller not only needs to reroute traffic to the new instance, but also needs to transmit the state information needed by the new VSFs. Currently, state sharing approach has been proposed that enables state sharing among instances by maintaining global detection states in the shared data storage, such as RAMCloud [53], FaRM [19], and Algologic [46]. This approach does not need to migrate state among virtual instances. However, this approach needs additional tools to extract the states of VSFs and introduces a significant performance overhead [42]. In StatelessNF [36], it is shown that the remote-only state share approach can lead to a 2-3x times of degradation in throughput and a 100-fold increase in packet latency. StatelessNF uses distributed shared object (DSO) to access states of an instance. However, it needs to obtain the states of the instance by Remote Procedure Call (RPC). This also introduces high cost when many states need to be shared. Hence, S-Blocks leverages state migration [13], [43] approach to share states between instances.

In many scenarios, virtual machine migration schemes are adopted. However, such schemes only perform coarse-grained state migration. State consistency issue is not crucial to these schemes. Another thread of works, such as Split/Merge [56] and OpenNF [26], aims to implement state consistency during instance migration. Those schemes stop sending traffic to the instance and cache them in an NFV controller when instances are cloned. Until all states are successfully migrated to new instances, the cached traffic will then be sent to the new instances. These methods introduce high latency and memory usage to to NFV controller.

In S-Blocks, we propose a fine-grained state consistency approach, which ensures loss-free and order-preserving during virtual security functions scaling. Loss-free means that all packets should be processed without any packet loss. Order-preserving indicates that packets should be processed according to the original order when they are forwarded to new VSF instances. Satisfying these two critical points can achieve strong consistency of state. Our method executes a classifier and caches the packets in the migrated instance. The classifier is used to fulfill fine-grained flow differentiation. In addition, we propose a states serialization algorithm to copy group states and flow states of elements. Furthermore, we present the loss-free and order-preserving mechanism to avoid missing migrated states and guarantee correctness of migrated states. Moreover, we design and implement secure state migration based on SGX attestation.

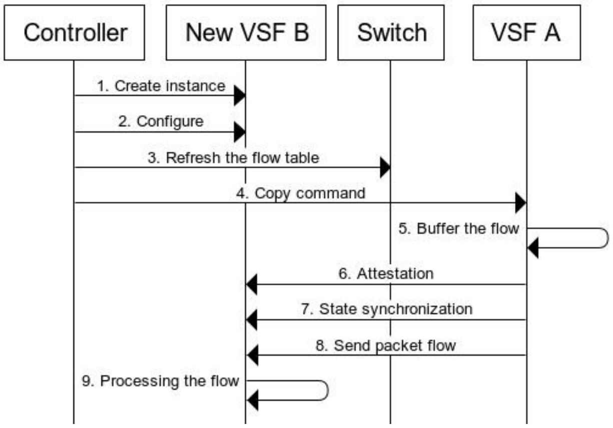


Fig. 3. Timing Diagram of State Synchronization.

As is shown in Fig.3, an SFV controller is used to manage, schedule the VSFs, such as VSFs creation and migration, and reroute flows between VSFs. The VSFs are connected with SDN switches. When a new instance B needs to be created, the state of the existing instance A will be migrated to the new instance B. The SFV controller copies the configuration of the instance A to the instance B. According to the configuration, the instance B completes its launching. Then the instance B waits to synchronize with the instance A, caching the packets forwarded by the switch. Then the controller updates the flow tables, and the switch forwards part of the data flows to the instance B. The controller sends the command of copying instance to the instance A. The command contains the characteristics of the traffic, and the information of instance B. Then characteristics of the traffic will be sent to instance B. Instance A runs a classifier and caches the packets that should have been sent to the instance B. The data packets are actually sent to the instance A, because the flow table has not taken effect due to the time delay. Then, the instance A and the instance

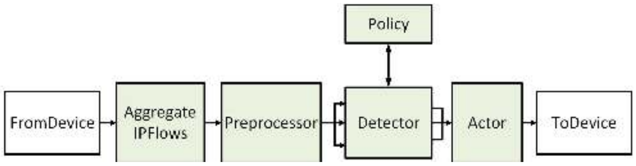


Fig. 4. Packets Processing with Flow Classifier.

B perform remote authentication and exchange keys. The instance A sends the encrypted internal states of the traffic, which needs to be synchronized to the instance B. After the instance B has synchronized all the states, the instance A sends buffered packets to the instance B. The instance B firstly processes the packets from the instance A, and then processes the packets from switch. In the end, the new instance B can process the migrated data flows.

Flow classifier: To distinguish flows from all traffic of a security function instance, we design the <SrcIP, SrcPort, DstIP, DstPort, Protocol> five-tuple and the *AggregateIpFlows* element depicted in Fig. 4. The *AggregateIpFlows* element calculates the flowIDs based on the five-tuple for the distinguished flows in the order of those flows being received. We also design two hash tables for the AggregateIpFlows element to store TCP and UDP flows. The critical of the hash table is the pair of IP Addresses including SrcIP and DstIP, and the value of the hash table is the flows corresponding to the host IP pair. We use FlowInfo, which includes the FlowID, Port, and a pointer to the next record, to describe the flows. After data flow differentiation is finished, data flow is processed by *Preprocessor* module, *Detector* module, and *Actor* module. Each element can recognize different data flow and realize fine-grain state processing. Below are the three categories of states which an element needs to handle.

- ElementStates: states related to elements themselves but not related to traffic, such as the states corresponding elements configuration.
- GroupStates: a set of states corresponding to multiple flows. For example, when packets that belong to multiple flows are processed, GroupStates will be updated or read.
- FlowStates: the private states that only belong to each flow. Only when packets belonging to this flow are processed, will FlowStates be updated or read.

In order to support copying only partial flow states, we design an element-states serialization interface that can serialize the states of an element itself. This interface will be called by the *Elements manager* module when the state is migrated or synchronized. At the same time, we design deserialization interface for state recovery, which will restore the states of each element.

The pseudo-code of element states serialization algorithm is shown in Algorithm 1. The state serialization algorithm has three input parameters. The first parameter represents whether to serialize all states. The second parameter identifies a set of flowID that needs to be serialized. The last parameter shows the state structure after serialization. The state serialization algorithm returns whether the serialization is successful. In the scenario of fine-grained state serialization, an element firstly looks for groupID set corresponding to the flowID set (line 7-8). Secondly, the element serializes group states corresponding to the groupID set and the flow states corresponding to the flowID set (line 10-11). Finally, the elements serialize the element states and add meta information, such as the size of states, the hash value of states, etc. (line 15-16).

Algorithm 1 States Serialization

```
Input:
 1: all: whether serializing all states;
 2: flowIDs: unique ID for each flow that needs to be
    transformed;
 3: state: serialized state structure;
Output: whether the serialization operation is successful
 4: if state == null then return false
 5: end if
 6: if all == false & flowIDs!=null then
 7:     for flowID in flowIDs do
 8:         groupIDs.add(find(flowID))
 9:     end for
10:     serialiseGroupStates(groupIDs,state)
11:     serialiseFlowStates(flowIDs,state)
12: else
13:     serialiseAllFlowStates(state)
14: end if
15: serialiseElementStates(state)
16: fillMetaInfo(state)
```

Then we design four APIs, which are shown in Table 2, for the state processing. The *void getflowIDinfo(vector<int> flowIDs)* API is responsible for identifying and providing all flow states that match the flowIDs, such as the number of packets, characteristics of packets from a single endpoint, etc. The *void delflowIDinfo(vector<int> flowIDs)* API is used to delete the obsolete flows, which do not appear for a long time, corresponding to the flowIDs. The function of *void chflowIDinfo(vector<int> flowIDs)* is to change the flow state that matches the flowIDs. In the end, *void migrate(srcInst, dstInst, state and flowIDs)* API is designed to migrate the states of the source instance to the destination instance.

Our approach also guarantees loss-free and order-preserving when achieving fine-grained state consistency.

Loss-free: In order to achieve loss-free, it's necessary to get fine-grained states based on traffic. In S-Blocks, we distinguish traffic with the <SrcIP, SrcPort, DstIP, DstPort, protocol> five-tuple, and attach a flowID to each packet to realize the data flow classification. Then, we implement state storage for each element at data flow granularity and provide the migration module with a serialization and deserialization interface. However, we notice that not all elements need to distinguish the fine-grained internal state. For example, *CheckLength* element processes each packet independently, thus does not need to determine the granularity of the traffic. Since complex elements based on entropy detection need different data streams to calculate entropy values, we cannot implement a uniform serialization method. Therefore, the process of the state serialization and deserialization can only be defined within the element.

In addition, the new instance needs to process the new traffic belonging to the new instance that was not processed

TABLE 2
APIs for Trusted State Management

API	Functionality
void getflowIDinfo(vector<int> flowIDs)	get the information according to flowIDs
void delflowIDinfo(vector<int> flowIDs)	make the information of flowIDs invalid
void chflowIDinfo(vector<int> flowIDs)	change the information according to flowIDs
void migrate(srcInst, dstInst, state, flowIDs)	migrate states (i.e. srcInst, dstInst, state and flowIDs) between two instances

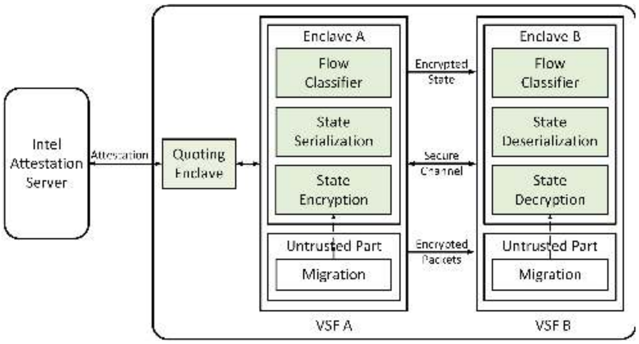


Fig. 5. State Migration.

during the state migration. Our migration scheme uses the traffic classifier to classify and cache the data packets belonging to the new instance, and the cached packets will be handled by new instances when the state synchronization and remote authentication has been finished. The traffic classifier can distinguish the data packets that should be sent to the new instance by determining the characteristic information such as the IP address of the data packet. The distinguished packets are then sent to the new instance, which will process the packets after the state is synchronized.

Order-preserving: The new virtual security instance implements order-processing by first processing the cached data packets from the original instance and then processing the data packets of the new instance itself. The new instance will also run the traffic classifier to distinguish the packet from the switch and the original instance. The original instance sends the data packets to the new instance through socket communication, hence they can be distinguished by the classifier of the new instance. After differentiation, the packets forwarded by the switch are buffered. Then the packets sent from the original instance are processed. Until the packets sent from the original instance have been processed, the packets forwarded by the switch will be read from the buffer and processed.

Secure state migration: During virtual security functions scaling, the states of virtual security function should be securely migrated from original instance to destination instance. Previous work migrates the state mainly based on the shared buffer and plain channel that may lead the states to be tampered with and leaked by malicious programs. Therefore, we present a secure state migration mechanism using SGX attestation in S-Blocks.

Our secure state migration mechanism is depicted in Fig. 5. Before the migration, we should find flows states to be migrated according to Flow Classifier. The states are then serialized for implementing the state synchronization with order-preserving. Then the migration module in the untrusted part performs the migration operation. It communicates with the state encryption module in the enclave, which is responsible for obtaining states of the virtual function. The state protected by the enclave will be encrypted as a disk file outside of the enclave and sent to the destination VSF through a secure channel.

The remote attestation function [33] provided by SGX is used in our architecture to build a secure channel. Both source VSF and destination VSF leverage Diffie-Hellman key exchange protocol to negotiate a shared symmetric key. During the negotiation process, they use the Quote enclave to get the singed measurement value, which contains enclave information and platform information. They communicate with IAS to verify the quote and obtains authentication results. When the validation is passed, both instances obtain two elliptic curve public keys of ga and gb. The symmetric critical can be calculated through the Elliptic-Curve Diffie-Hellman (ECDH) protocol.

After the secure channel is created, the original instance can synchronize the states to the new instance. The SGX decryption library is used to decrypt the states, after which the states will be recovered. Through the secure state migration, the states are securely protected during state migration. We also transmit the migrated packets by the cipher-text through the secure channel and design a migration API for transferring both state and traffic for a set of flowIDs from one instance to another instance (as shown in Table 2).

4.3 Trusted Policy

The trust of policy is crucial to virtual security functions because the rules of policy include lots of sensitive information about a network. Once those sensitive rules have been tampered with or leaked by adversaries, the network security is endangered. However, it is still a challenge to ensure the trust of virtual security function policy while using SGX. Take Snort as an example, its policy processing is almost related to all processing procedures and involves a large number of modules and plugins. If we want to provide complete protection for the policy processing, we need to put almost all modules of Snort into enclaves. That is impractical because it will bring huge expenses since some modules and plugins cannot be put into enclaves due to involving a lot of system calls.

S-Blocks architecture is different from Snort. The architecture makes the complete protection of policy possible. S-Blocks abstracts each function into a separate component (i.e. element), which carries out a single and non-conflicting traffic processing. Therefore, it's easy to extract the modules that are independent with policy processing and only put those modules in enclaves for better protection.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2985045, IEEE Transactions on Cloud Computing

In S-Blocks, the rules of a policy are implemented by assembling multiple package processing elements. For example, a rule is defined as:

```
  drop tcp any any → any 3306 (msg: "rule
detection"; falg:S; content: "|03|"; offset:
4; dsize: <300;)
```

This rule indicates that the system discards TCP syn packets coming from any port and destination port 3306, and the packets payload includes specific content ”|03|” after the first 4 bytes of the payload while the packet payload size is less than 300. In addition, the system prints the message ”rule detection” when detecting packets according to the rule. In order to implement this rule in S-Blocks, we use *Classifier* element to classify IP packets with special content. Then *IPClassifier* element classifies syn packets with destination port 3306 and *CheckLength* Element checks the length of packets. After packets conforming to this rule are distinguished, *Print* element will print the message and *Discard* element will discard the packets. Meanwhile, elements relating to the policy such as *Classifier*, *IPClassifier*, *Discard*, and *CheckLength* are put into the enclave so as to provide protection for the policy processing.

In S-Blocks, we use a configuration file to describe how the elements are configured and connected in detail. An element implements a simple packet processing function which depends on the element configuration. For example, *Classifier* element classifies different types of packets according to the element configuration. Users can connect various elements to achieve various and complex rule processing. If users want to support new types of rules, they can create new elements to support complex rules. Based on the rules, the elements are connected and executed to realize packets detection. In order to realize complete protection for the policy processing, we also design some specific modules or elements which are as shown in Fig. 6.

① Seal/Unseal policy: The policy file stored in plain in disks needs to be protected carefully. We design two APIs, *void Seal(file *)* (i.e. PolicyFile) and *void Unseal(file *)* (i.e. PolicyFile) based on SGX for policy file protection of VSFs. The keys used in two APIs are generated based on platform information so that keys cannot be retrieved and policy file can only be decrypted on the same platform. When the policy file needs to be stored on the hard disk, the Seal function will encrypt the policy file and write it to the disk. When S-Blocks starts up, the Unseal function reads the file from the disk, then decrypts the policy file.

② Policy processor: Policy Processor module parses the policy and generates the elements which are responsible for processing policy. The lexical analyzer in *PolicyParse* element converts the decrypted policy file into four array structures: (1) an array of elements, (2) an array of element configuration, (3) an array of reading handlers and (4) an array of element connections. These elements are connected to each other according to the array of element connections. The relation of element connection defines the process of rules. The elements will be configured according to the array of elements and the array of element configuration, and then generate a router. After that, elements query flow-based router context and place themselves in the task queue. *PolicyChange* element is responsible for changing the policy. Then users can dynamically reconfigure some elements with handlers according to the new policy as the router is executing and the *PolicyParse* will regenerate the new elements which support the new policy.

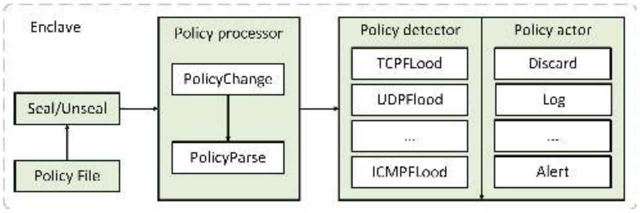


Fig. 6. Trusted Policy.

③ Policy detector and Policy actor: Policy detector module implements various elements to detect the traffic based on the policy rules, such as *UDPFlood* element, *TCPFlood* element. *UDPFlood* element and *TCPFlood* element filter malicious TCP/UDP traffic based on policy rules. Policy actor module performs corresponding actions on data packets based on the rules of security policy, such as *Discard* element, *Log* element, *Alert* element.

In S-Blocks, the policy file and the elements related to policy processing are put into the enclave of SGX. Hence, it can provide security protection in whole life cycle of policy. Thus it prevents the policy from being tampered with and leaked by malicious cloud administrator, tenants or programs.

If a security function is moved from an old Intel SGX system to a new Intel SGX system (i.e. this happens during platform upgrade) or from one processor to another (i.e. this happens in CPU replacement in a system or load balancing in a cloud environment). The enclave will not be able to unseal the policy in the new platform. There are two ways to migrate the policy file. One is based on SGX migration [28] where the original instance and the new instance establish a secure channel based on SGX remote attestation. Then the old instance unseals the policy file, encrypts the policy file with a negotiated session key and sends the policy to a new instance. After receiving the policy file, the new instance decrypts policy file using the session key, then the new instance seals the policy in order to implement the security policy protection of VSFs. The other approach is to use an SFV controller to deliver the policy file of the old instance to the new instance. The policy file is transferred to the new instance over the TLS channel from the SFV controller and then the policy file will be sealed and loaded into the enclave of the new VSF for protection.

5 Implementation

We prototype S-Blocks leveraging Fast-Click [3], [24] based on SGX SDK v2.13, and instantiate it for three uses cases. FastClick is an extension of the Click modular router [37], which is a most commonly used software architecture for building modular and extensible network function. FastClick features an improved Netmap support and a new DPDK [18] support and provides fast user-space packet I/O and easy configuration via automatic handling of multi-threading and multiple hardware queues. However, FastClick currently does not support any security functions.

5.1 Case Studies

We design and implement three types of virtual security functions on S-Blocks architecture, including virtual DDoS detection and defense, firewall and IDS. We implement several new elements for virtual security functions. We use Element manager to provide management functions such as policy analysis, configuration, initialization and operational state management. We also modify some basic FastClick elements such as *Discard*, *ToDevice*, *Packet*, *Element* and so on. At last, we put the critical elements related to security functions to the enclave.

- **DDoS Detection and Defense:** As is shown in Fig. 7, we implement *UDP Flood*, *SYN Flood*, *ICMP Flood*, *Packet Switch* elements as DDoS detection and defense module. The *UDP Flood*, *SYN Flood*, *ICMP Flood* elements receive a packet and output a tagged packet to *Packet Switch* element. The *Packet Switch* element sends the packet to different action elements such as *ToDevice*, *Discard*, *Delay* according to the tag and rule tables.
- **Firewall:** We develop a stateful firewall that consists of the *rule table*, *state table*, *match engine*, and *manager* modules. Each module is realized via one or multiple elements. The *rule table* is the basic storage of firewall rules, and the *state table* maintains corresponding states of each firewall rule. When the packet comes in it is handled by the *match engine*, which first matches the packet against *state table*. If the packet is not matched with any states, the *match engine* matches the packet against the *rule table* to determine the actions for this packet. The *manager* handles the control commands from the user, such as inserting, removing, and updating the firewall rules.
- **IDS:** We implement four differnt protocol analyzer elements: (1)*httpanalyzer*, (2)*sshanalyzer*, (3)*ftpanalyzer* and (4)*dnsanalyzer*) and six different detection logic programs (*DNS Tunneling Detector*, *Cookie Hijacking Detector*, *Trojan Detector*, *Scanner Detector*, *Flow Monitor*, and *HTTP Monitor*) for the IDS. When a packet is sent to the IDS, the packet is first classified based on its header information. Packets belonging to different application protocols are sent to different protocol analyzers for analysis. The protocol analyzers then generate events according to the analysis results. Those events are passed to various detection logic programs according to the configuration of the IDS. The detection logic programs are responsible for detecting any threats and reporting the threats. By connecting the IDS to the action elements, such as *Discard*, *Delay*, and *ToDevice* elements, IDS can block or redirect the traffic.

In *State synchronization* module, we implement the interface of states serialization and deserialization to support copying the partial state. When the state is migrated or synchronized, the state information is serialized. After remote authentication between the two VSFs, the state is transmitted through socket communication. At the same time, we achieve the recovery state through the deserialization interface which will restore the state of each element. On the other hand, we offer four state-related APIs to get, delete or change the states information according to flowIDs and migrate states between two instances.

5.2 Performance Optimization

S-Blocks builds on hardware-assisted memory protection based on Intel SGX to provide strong confidentiality and integrity guarantees. However, the architecture of SGX suffers from two major limitations which incur performance overhead in VSFs.

- For strong security, SGX allows neither system calls within enclaves nor instructions that could lead to a VMEXIT. Enclave transitions are expensive, introducing a high runtime overhead due to the cost of saving/restoring the state of the secure environment. Each enclave transition imposes a cost of 8,400 CPU cycles [2].
- The Enclave Page Cache (EPC) is restricted to 128MB. To overcome this limitation, SGX supports a secure paging mechanism to an unprotected memory region. However, the paging mechanism incurs some overheads depending on the memory access pattern [1].

In order to achieve high performance despite the inherent limitations of the SGX architecture, we have implemented the following performance optimization in S-Blocks:

Reduced system call. As one of the goals of the S-Blocks is high performance, we minimize the number of system calls. We find that many VSFs simply do not make system calls or execute instructions that require VMEXIT. In S-Blocks , the sum of ECalls and Ocalls is about ten. However, we need system call to get time when we need to judge where S-Blocks suffers a DDoS attack according to the rate of receiving packets. In the previous work, researchers have thus sought a few alternatives for in-enclave timing. The most common one uses OS time service with system call by using enclave exit or shared memory. However, they all emphasize that it is still an open problem to provide trusted and high-resolution for SGX applications. For example, SafeBricks [55] relies on the host I/O to get timestamps but it does not guarantee the timestamp is correct. Taking overhead into consideration, S-Blocks leverages the *FromDevice* element to capture the timestamp from the per incoming packet batch. The following elements can directly obtain the timestamp of the packet. In addition, I/O for maintaining logs requires Ocall operations which cause system overhead. Therefore, we have optimized the code to reduce some unimportant logs information for better performance.

Batch operation. Every packet receiving or sending operation will result in an invocation of the I/O interface and an enclave transition, hence we exclude the packet capture library from the TCB. Using batch processing can reduce the number of enclave transition and improve system performance. In practical work, we add batch operation to the security function to improve performance of S-Blocks.

DPDK. DPDK allows achieving line speed throughput in software-based network function. The design of putting DPDK code and state inside the enclave would bring high performance and inflate the size of the TCB. Therefore, we put DPDK out of the enclave.

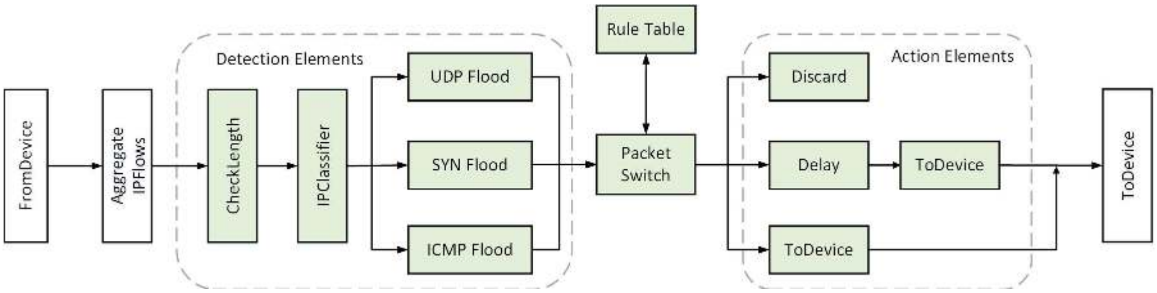


Fig. 7. A Virtual Security Function Case based on S-Blocks: DDoS Detection and Defense.

6 Evaluation

S-Blocks uses SGX and Fast-Click to achieve trusted protection for VSFs. However, SGX will bring additional overhead that affects system throughput, traffic processing time, and hardware resources. In this section, we will perform a detailed performance analysis of S-Blocks and present its security analysis.

6.1 Experimental Set-up

We evaluate the performance of S-Blocks using the real Skylake platform with an Intel E3-1280 v6 CPU with 4 cores executing at 3.9GHz. The server has 16GB of memory and runs Ubuntu 18.04.1 LTS with Linux kernel version 4.15. Besides, we use Mininet [48] and Floodlight [22] as network simulation and controller to build SFV experimental environment. Mininet is a network simulation system. It runs a series of hosts, switches, and links on a single Linux kernel. It uses lightweight virtualization technology to build a virtual network. Floodlight is based on Java and it can modify the behavior of network device with a well-defined instruction sets.

We evaluated S-Blocks using three types of security functions for the case-studies: (1) DDoS detection and defense; (2) firewall; and (3) IDS. For the performance measurements, we consider several cases of S-Blocks :

- Baseline: the system executing VSF based on Fast-Click without SGX protection.
- S-Blocks: the system executing VSF based on Fast-Click with SGX protection.

In addition, we also make a performance comparison between three types of security functions based on S-Blocks to complex, full-featured applications such as Dshield, IPtables, Snort.

6.2 Performance of System

For monitoring the systems elasticity, we deployed a client generating traffic in varied sizes and rates to S-Blocks. Then the traffic was passed to and processed by the virtual DDoS detection and defense, Firewall, and IDS. In the DDoS scenario, we use Scapy [58] and TFN [66] to generate normal traffics and attack traffics. Attack traffic contains UDP flood traffic, SYN flood traffic, ICMP flood traffic and so on. TFN is a set of computer programs to conduct various DDoS attacks such as ICMP flood, SYN flood, UDP flood, and Smurf attack. It can be used to control any number of remote machines to generate random anonymous denial of service attacks and remote access. In Firewall and IDS scenario, we use Scapy [58] to generate traffic in various sizes and rates to the system. Scapy is a powerful interactive packet handler based on python. It can forge or decode mostly protocols packets, send them online, capture them, match requests and replies, etc..

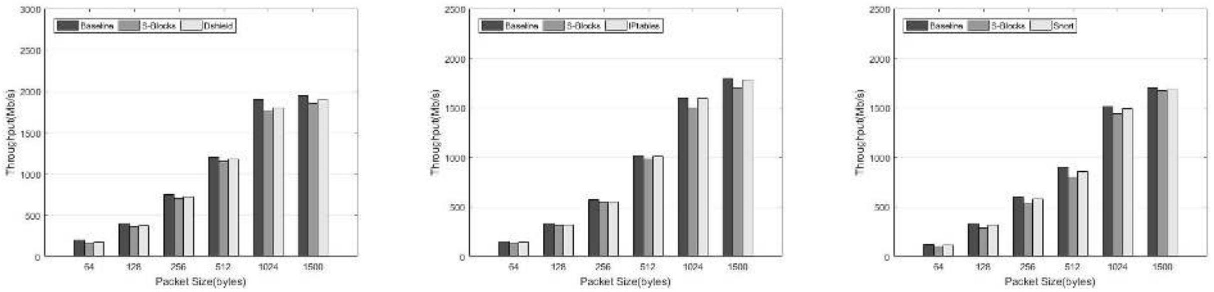
6.2.1 Throughput

Throughput is the most important performance criteria for evaluating our approach. We evaluate the throughput of our approach through traces of different packet sizes, from 64 Bytes to 1500 Bytes, in two different modes: a Baseline mode and a S-Blocks mode, where the Baseline mode means we run the virtual functions developed in Click without SGX protection. We then compare the throughput of those two different modes for three different virtual security functions, including a DDoS detection and defense function, a firewall function, and an IDS function. We also compare those two modes with three state-of-the-art open-source security applications. Fig. 8 (a), Fig. 8 (b) and Fig. 8 (c) show our throughput comparison results.

It is clear that increasing packet sizes reduces the throughput of all three virtual security functions in both Baseline mode and S-Blocks mode. The throughput in the Baseline mode for three virtual security functions is a little bit lower than the S-Blocks mode, since in the S-Blocks mode, running virtual functions in SGX enclaves increases memory and CPU consumption. We additionally compare the throughout overhead of our approach with three state-of-the-art security applications. Fig. 8 (a) shows a throughout comparison between our DDoS detection and defense function implementation based on S-Blocks and an open-source DDoS detection tool, Dshield. The result shows that the throughput of Dshield is 1.03x than our solution. Fig. 8 (b) shows a throughout comparison of our firewall implementation and IPtables. The result shows that the throughput of IPtables is 1.04x than our solution. Fig. 8 (c) shows a throughout comparison considering IDS. The result depicts that the throughput of Snort is 1.05x than our IDS implementation. In summary, the evaluation results show that S-Blocks only impacts overall throughput of security functions slightly due to SGX protection.

6.2.2 Latency

We also measure the latency of average packet processing considering two different modes and three different types of security functions by using Scapy to send both UDP and

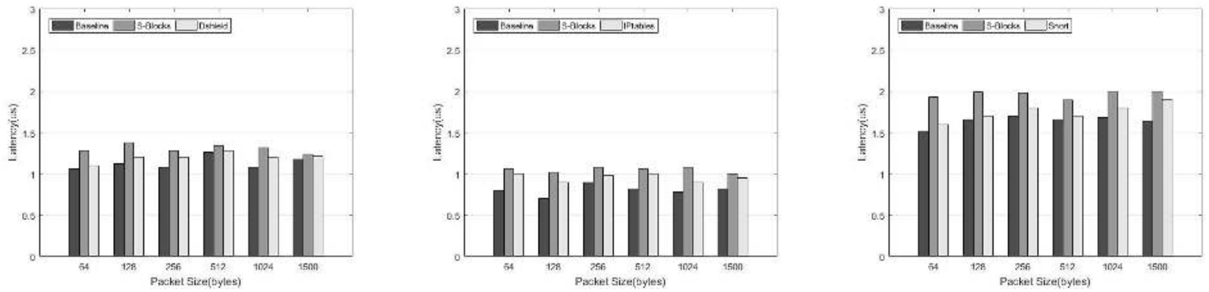


(a) The Throughput of DDoS Detection and Defense

(b) The Throughput of Firewall

(c) The Throughput of IDS

Fig. 8. The Throughput Measurement of Virtual Security Functions.



(a) The Latency of DDoS Detection and Defense

(b) The Latency of Firewall

(c) The Latency of IDS

Fig. 9. The Latency Measurement of Virtual Security Functions.

TCP traffic with different packet sizes in our experiments. Our evaluation results are summarized in Fig. 9.

Fig. 9 (a) presents the latency measurements for the DDoS detection and defense function with various packet sizes. In the Baseline mode, the average processing time is about 1.125us. In the S-Blocks mode, the average processing time is about 1.4us. On the whole, the DDoS detection and defense system executing on top of SGX in the S-Blocks mode gives 24% additional executing time delay compared to the Baseline mode. Fig. 9 (b) and Fig. 9 (c) show the latency measurements for the firewall and the IDS, respectively, with various packets sizes.

In the Baseline mode, the average processing time of the firewall is about 0.80us, and the average processing time of the IDS is about 1.55us. In the S-Blocks mode, the average processing time of the firewall is about 1.05us, which gives 31% additional executing time delay compared to the Baseline mode. The average processing time of the IDS is about 1.92us, which gives 23.8% additional executing time delay compared to the Baseline mode. we also show the latency comparison between our solutions and the state-of-the-art applications with various packets sizes, respectively. The average processing time of the Dshield is about 1.2us, which is 86% of S-Blocks, the average processing time of IPtables is 0.955us, which is 91% of S-Blocks, and the average processing time of Snort is 1.75us, which is 91% of S-Blocks.

6.2.3 State Synchronization

We then evaluate the overhead of state synchronization in S-Blocks. The measurement results are presented in Fig. 10.

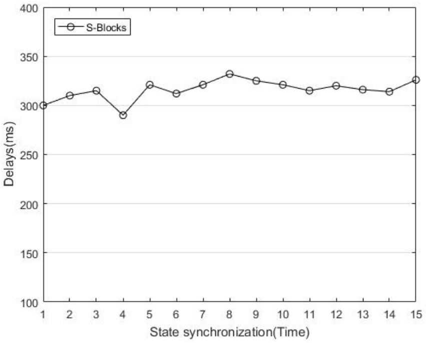


Fig. 10. State Synchronization Latency.

The average time of state synchronization in 15-time tests is 326ms. This time is primarily dictated by the time required for the S-Blocks to serialize and deserialize states. The results show that these data consumption is within the acceptable range of the system and do not result in significant overhead in the system.

6.2.4 CPU consumption

Finally, we compare the CPU consumption with respect to two different modes and three state-of-the-art security functions by sending traffic with various packet sizes, which are generated by Scapy in the maximum rate.

As is shown in Fig 11, in the Baseline mode, the average CPU usages of the DDoS detection and defense function, the firewall, and the IDS are about 5.9%, 6.2%, and 7.2%,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2985045, IEEE Transactions on Cloud Computing

respectively. In the SGX mode, the average CPU usages of the DDoS detection and defense function, the firewall, and the IDS are about 6.4%, 7%, and 7.6%, respectively. The results show that the CPU usages in two different modes for three security functions are increased slightly.

In addition, the average CPU usages of Dshield is 6.06% shown in Fig. 11 (a), which is 95% of the CPU usage of the DDoS detection and defense function based on S-Blocks. Fig. 11 (b) presents that the average CPU usages of IPtables is 5.8%, which is 83% of the average CPU usage of firewall based on S-Blocks. Fig. 11 (c) depicts that the average CPU usages of Snort is 7.2%, which is 95% of the average CPU usage of IDS based on S-Blocks.

6.3 Security Analysis

In this section, we conduct security analysis from two aspects: (1) security analysis of VSF and (2) security analysis of state migration.

6.3.1 VSF Security

Our lightweight and trusted VSF architecture, S-Blocks, uses Intel SGX to protect the code, state, and policy of VSFs. The security architecture is mainly that packets out of VSFs are encrypted so that attackers from the cloud can only intercept encrypted traffic. Attackers cannot get the contents of the packets and can only observe the size and time of packets. It also protects the packets and policies of IDSes and data flow status in such a way. The system is divided into trusted parts and untrusted parts. The trusted part stores sensitive data, sensitive states. DDoS attack processing detection includes a data packet preprocessing module, an action processing module, a rule matching module, and other sensitive modules. The operating system, drivers, BIOS, and VMM cannot obtain the code and data in the enclave. Using SGX security mechanisms, S-Blocks protects against security threats, such as data and sensitive rules leakage, code tamper, targeting VSFs in an untrusted cloud environment, to realize the security of VSF internal states and policies. It also provides the trusted part with the interface of the system call and implements the packets transfer between the trusted and untrusted parts.

6.3.2 State Migration Security

In the process of state migration [51], a critical issue is how to establish a secure channel between two VSF instances for secure state migration. In order to solve this problem, S-Blocks uses the SGX remote attestation mechanism [33] to achieve the integrity certification between two instances. In the process of remote attestation, the developer's server and the SGX remote authentication server communicate through a standard TLS network protocol. The SGX remote attestation service provides necessary infrastructure to allow the server to determine whether the client is requesting the service executing in a secure and trusted environment (hardware + software) and establish an encrypted channel. After successful attestation, the server sends the confidential data to the client.

7 Related Work

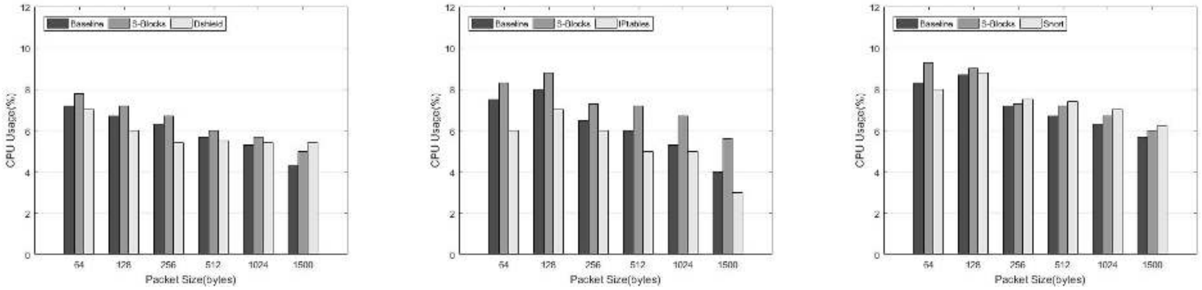
In this section, we review previous works on security hardening of NFV and state migration of OpenNF and VMs.

7.1 Security of NFV

ESTI NFV Security and Trust Guidance [9] propose to provide trusted protection based on Hardware Security Module (HSM), Trusted Platform Module (TPM) [49], and virtual Trusted Platform Module (vTPM) [6]. OpenNetVM [75] runs network functions in Docker containers based on the NetVM architecture. It provides isolation between NFs based on container mechanisms, such as namespace and capability. NetBricks [10] leverages a safe language (Rust) and LLVM [39] to enable zero copy soft isolation. It provides memory isolation by using the type-safe language and achieves high performance by adopting LLVM as an optimization back-end of compilers. However, those approaches cannot provide the virtual network function with strong security level protection since they lack strict memory encryption and isolation mechanisms [31], [47].

Recently, some research efforts have proposed to use Intel SGX to protect virtual network functions. SGX can isolate an application with a hardware sandbox, called enclave, using memory encryption and access control mechanism so that OS, driver, BIOS, or virtual machine monitor (VMM) cannot access the code and data in the enclave. S-NFV [62] attempts to use SGX to protect Snort. However, it ignores the policy protection and state trust, and is implemented with an open-source simulator, OpenSGX [35], instead of a real SGX platform. In addition, the authors simply put the entire Snort into the SGX enclave, which could cause a large performance overhead. LightBox [20] introduces etap, a virtual network device, to access the fully protected network packets so as to secure the traffic based on TSL, which is put in the SGX enclave. However, this work uses monolithic architecture, which has significant limitations to protect the virtualized environments since users can not customize VNFs and reuse existing modules to create new network functions. Trusted Click [16] extends the Click modular router to perform secure packet processing with SGX, but it does not discuss the flow-based state protection because Click lacks the stateful traffic processing functionality. ShieldBox [67] securely processes encrypted traffic by using SCONE [1] and Click. It lacks the necessary state serialization routines. SafeBricks [55] builds upon NetBricks [54] and leverages SGX and Rust language-based enforcement to keep network function outsourcing secure. However, this work only provides the isolation based on processes and does not consider the state protection and migration. SGX-Box [30] proposes a high-level programming language, called SB Lang for handling encrypted traffic in SGX-protected middleboxes. This work provides a few of API based on SB Lang and automatically convert SB Lang code to C/C++ code. AirBox [7] proposes a platform for fast, scalable and secure onloading of edge function with SGX. SPX [8] presents a solution for edge-ready and end-to-end secure protocol extensions, which allows edge functions to operate on encrypted traffic while ensuring that security semantics of secure protocols still hold. Nevertheless, they do not consider state security of middleboxes.

Compared with above work, the goal of S-Blocks is to design a new modular security architecture and implement a platform so that users can leverage the platform and its elements to easily create new security functions and enable



(a) The CPU Usage of DDoS Detection and Defense

(b) The CPU Usage of Firewall

(c) The CPU Usage of IDS

Fig. 11. CPU Usage Measurement of Virtual Security Functions.

the SGX protection of those functions. S-Blocks focuses on the modular design and run-time protection of virtual security functions instead of general network functions. We propose the modular architecture of virtual security functions while considering their code, state, policy protection using SGX, and achieving lower performance overhead. We also design several APIs for developers to produce new security functions and put them into enclaves for protection. Furthermore, a fine-grained state synchronization and migration approach is proposed for secure state sharing among virtual instances. Finally, we provide an extensive evaluation based on three typical security functions, including DDoS detection and defense, firewall, and IDS.

In Table 2, we have summarize the state of art of the SGX-based network function protection frameworks and shown the difference between the previous works and S-Blocks. As can be seen from the table, S-Blocks is the first practical work that proposes the design architecture of Click-based security functions while considering their code, state, policy protection using SGX, and achieving low performance overhead.

7.2 State Migration

The state sharing is crucial for virtual instance scaling. Currently, enabling state sharing among instances by maintaining global detection states in the shared data storage, such as RAMCloud [53], FaRM [19], and Algo-logic [46], has been proposed. Such an approach does not need to migrate states among virtual instances. However, this approach needs additional tools to extract the states of VSFs and introduces an additional performance overhead [42]. In StatelessNF [36], its evaluation results show that the remote-only state share approach can lead to a 2-3x degradation in throughput and a 100-fold increase in packet latency. Stateless uses distributed shared object(DSO) to access states of an instance. However, it needs to obtain the states of the instance by RPC, which also causes high cost when many states need to be shared.

VM (Virtual Machine) cloning [4] allows full cloning of NF (Network Function) instances. This will result in additional resource consumption because some unwanted states will also be migrated when cloning NF instances. In addition, this approach cannot move and merge states from multiple NF instances, so it cannot realize fast and elastic scale-in.

Some research work [25], [44], [50], [71] has proposed to mitigate and replicate internal NF states, allowing us to maintain correct NF behaviors during NF scaling. Split/Merge [56] proposes a state consistency approach by suspending current traffic, caching them in a controller, and sending the packets after migration. However, when the current traffic is suspended, some data packets may have arrived at the network. Split/Merge directly discards these data packets, which loses some state information and cause processing packets out of order. Furthermore, caching a large amount of data stream in the controller requires a large cache memory in the controller. OpenNF [26] proposes to cache the data traffic to be synchronized in the controller and then performs state synchronization operation. After the state synchronization is completed, the buffered data traffic is sent to the destination instance for processing. The destination instance processes the cached data packets firstly and then processes the subsequent new data packets to implement loss-free and order-preserving. However, OpenNF has the same problem as Split/Merge: these methods cause high delay and require the controller to have enough memory capacity.

Aiming at this issue, we propose the fine-grained state consistency and secure migration approach. Compared to the previous works, our approach proposes to cache data traffic to be synchronized in an instance rather than a controller in order to reduce the overhead of the controller. Our approach synchronizes different types of flow states through fine-grained flow classification and state serialization. In addition, our approach uses the remote attestation provided by SGX to establish a trusted channel between two instances for secure state migration.

8 Discussion

In this section, we discuss over limitations and future work of S-Blocks.

8.1 Limitations

S-Blocks has some limitations that are mainly caused by using SGX.

First, SGX needs ECall and OCall interfaces between the enclave and the untrusted part incurring expensive overhead. To mitigate such a limitation, at high level, S-Blocks puts all elements together in one enclave instead of multiple enclaves (as described in Section 4) so as to reduce the performance overhead. If developers define the system

TABLE 3
Properties of the Representative Frameworks for SGX-based NFV Security

Frameworks	Modular	Click Support	Isolation	State Protection	Policy Security	DPDK	Overhead	
							Throughput	Latency
SafeBricks [55]	✓	×	✓(Rust+SGX)	N/A	N/A	✓	low	N/A
ShieldBox [67]	✓	✓	✓(SGX)	N/A	N/A	✓	low	high
LightBox [20]	×	×	✓(SGX)	✓	N/A	×	low	N/A
SGX-Box [30]	N/A	×	✓(SGX)	×	×	×	low	N/A
TrustedClick [16]	✓	✓	✓(SGX)	✓	×	×	low	N/A
S-NFV [62]	×	×	✓(SGX)	✓	×	×	high	N/A
S-Blocks	✓	✓	✓(SGX)	✓	✓	✓	low	low

* N/A: the feature is not considered or not explicitly elaborated in a work.

call in common use with the Ocall function previously, we can avoid recompiling to ensure all modules indeed fit in an enclave to some extend. However, if the module contains the system call, which was not defined previously, we need to hand tune the code to ensure all modules could run in an enclave. In addition, a large number of system calls also incur extra performance overhead. However, we found that the process of VSFs almost does not make system calls or execute instructions that lead to VMEXIT except the system calla related to time, such as timestamp measurements using rdtsc. In S-Blocks , we use *FromDevice* element to make timestamp measurements. When the *FromDevice* element gets an incoming packet, it captures the timestamp and writes it to the annotation part of the packet. VSFs that need timestamps for their functionalities just simply read it from the elements, though these timestamps are not guaranteed to be very precise. Intel SGX SDK provides a trusted time, however it is coarse-grained, which can't fulfill the requirement of VFS. SGX needs to make changes to the hardware to support a trusted, efficient, and precise time source for SGX enclaves [61].

Secondly, the available memory of enclaves provided by SGX is limited. Hence, we must control the size of sensitive code and data. In our implementation, we use an enclave to protect the trusted elements. However, when the number and size of the programs are increased, more pages are required to be swapped in and out. In order to ensure security, the system should protect the integrity and confidentiality of the pages, which results in some system overheads. Fortunately, several solutions [76] have been proposed recently to improve the performance of EPC paging to an acceptable range.

Thirdly, provisioning VSF as a monolithic piece of software executing as a whole has significant limitations to protect the virtual environments. Hence, S-Blocks only supports the modular security functions based on Click. Other monolithic security applications such as Snort can not be supported. Modifying a monolithic security application like Snort to adapt to S-Blocks is a daunting and complicated task, which changes the logic of the original application and increases the possibility of errors. In the future, we may try this task, but the cost is very high, probably more than rewriting a modular security function based on S-Blocks.

Finally, S-Blocks does not address the defense mechanism to side-channel attacks on SGX. The side channel attacks to SGX can be used to obtain the enclaves code and data information, such as encryption keys and privacy data. There are various types of side channel attacks, such as timing-based side channel attacks, cache-based side channel attacks [59], TLB-based side channel attacks, Page-table-based side channel attacks, attacks based on the CPU internal structure [40] and mixed side channel attacks [70]. It is possible to defend those side channel attacks by hiding the control flow and data flow. Recently, Recently, a number of solutions have been proposed to solve and mitigate these attacks [11], [15], [27], [63], [64], which are orthogonal to our work.

8.2 Future Work

We have implemented three types of VSFs to verify S-Blocks design and performance. In the future, we plan to develop more security functions. Besides, we plan to increase the flexibility of S-Blocks so that the developers and users can more easily add and put other elements to the SGX platform. Furthermore, we should also consider the architecture with multiple enclaves and dynamic configuration so as to increase the flexibility of S-Blocks.

In addition, we plan to make S-Blocks support hot swapping in order to dynamically modify and delete elements in the future.

Hot swapping is a method that can implement new elements and realize new configurations without stopping the current system. One solution for supporting hot swapping is launching a new enclave and put both new elements and old elements into the enclave. Another solution is to put each element in a separate enclave. However, some issues should be considered for the above solutions. For example, the frequency of enclave transition can introduce additional performance overhead, and how to handle the shared elements.

9 Conclusion

In this paper, we have proposed S-Blocks, a lightweight and trusted VSF architecture based on SGX. Aiming to achieve practical high performance, address the high cost challenges, and employ the SGX to protect the large-scale security applications, S-blocks leverages modular and microservice-oriented architecture to achieve a trade-off between security and performance. Moreover, we have proposed trusted state synchronization and migration mechanisms to provide fine-grained state consistency and ensure loss-free and order-preserving migration for security function scaling. Furthermore, we have presented a trusted policy mechanism in S-Blocks to protect security policies in VSFs. We have implemented a DDoS detection and defense function, a firewall function, and an IDS function based on our architecture as use cases. We have also evaluated S-Blocks and our evaluation results showed that S-Blocks only introduces very low performance overhead when securing VSFs.

10 Acknowledgment

This work is sponsored by the National Natural Science Foundation of China granted No.61872430, 61402342, 61772384 and the National Basic Research Program of China 973 Program granted No.2014CB340601, and Foundation of Science and Technology on Information Assurance Laboratory (No. KJ-17-103).

References

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, volume 16, pages 689–703, 2016.

[2] Pierre-Louis Aublin, Florian Kelbert, Dan OKeeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. Talos: Secure and transparent tls termination inside sgx enclaves. *Imperial College London, Tech. Rep*, 5, 2017.

[3] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pages 5–16. IEEE, 2015.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven H, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *Proc. of SOSP 2003*, 37(5):164–177, 2003.

[5] L. R. Battula. Network security function virtualization(nsfv) towards cloud computing with nfv over openflow infrastructure: Challenges and novel approaches. In *International Conference on Advances in Computing, Communications and Informatics*, pages 1622–1628, 2014.

[6] Stefan Berger, Ramn Cceres, Kenneth A Goldman, Ronald Perez, Reiner Sailer, and Leendert Van Doorn. vtpm: virtualizing the trusted platform module. In *Conference on Usenix Security Symposium*, page 21, 2006.

[7] Ketan Bhardwaj, Ming-Wei Shih, Pragya Agarwal, Ada Gavrilovska, Taesoo Kim, and Karsten Schwan. Fast, scalable and secure onloading of edge functions using airbox. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 14–27. IEEE, 2016.

[8] Ketan Bhardwaj, Ming-Wei Shih, Ada Gavrilovska, Taesoo Kim, and Chengyu Song. Spx: Preserving end-to-end security for edge computing. *arXiv preprint arXiv:1809.09038*, 2018.

[9] B Briscoe et al. Network functions virtualisation (nfv)-nfv security: Problem statement. *White paper, ETSI NFV ISG*, 2014.

[10] Romolo Camplani, Gabriele Viscardi, Manuel Roveri, and Cesare Alippi. Netbrick: A high-performance, low-power hardware platform for wireless and hybrid sensor networks. In *IEEE International Conference on Mobile Ad-hoc & Sensor Systems*, 2012.

[11] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: closing hyper-threading side channels on sgx with contrived data races. In *Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races*, page 0. IEEE, 2018.

[12] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with dj vu. In *ACM on Asia Conference on Computer and Communications Security*, pages 7–18, 2017.

[13] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Nsdi'05 Proceedings of the Conference on Symposium on Networked Systems Design & Implementation, Boston, U.s.a*, pages 273–286, 2005.

[14] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[15] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[16] Michael Coughlin, Eric Keller, and Eric Wustrow. Trusted click: Overcoming security issues of nfv in the cloud. In *ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 31–36, 2017.

[17] Wanfu Ding, Wen Qi, Jianping Wang, and Biao Chen. Openscaas: an open service chain as a service platform toward the integration of sdn and nfv. *IEEE Network*, 29(3):30–35, 2015.

[18] DPDK. http://dpdk.org/.

[19] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[20] Huayi Duan, Xingliang Yuan, and Cong Wang. Lightbox: Sgx-assisted secure network functions at near-native speed. *arXiv preprint arXiv:1706.06261*, 2017.

[21] Mahdi Daghmehchi Firoozjaei, Jaehoon Jeong, Hoon Ko, and Hyoungshick Kim. Security challenges with network functions virtualization. *Future Generation Computer Systems*, 2016.

[22] Floodlight. http://www.projectfloodlight.org/.

[23] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 357–380. Springer, 2017.

[24] Massimo Gallo and Rafael Laufer. Clicknf: a modular stack for custom network functions. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 745–757, 2018.

[25] Aaron Gember-Jacobson and Aditya Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, pages 43–48, 2015.

[26] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *ACM Conference on SIGCOMM*, pages 163–174, 2014.

[27] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, pages 217–233, 2017.

[28] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. Secure live migration of sgx enclaves on untrusted cloud. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 225–236. IEEE, 2017.

[29] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[30] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 99–105. ACM, 2017.

[31] Hassan Hawilo, Abdallah Shami, Maysam Mirahmadi, and Rasool Asal. Nfv: State of the art, challenges and implementation in next generation mobile networks (vepc). *IEEE Network*, 28(6):18–26, 2014.

[32] Huawei Huang, Song Guo, Jinsong Wu, and Jie Li. Service chaining for hybrid network function. *IEEE Transactions on Cloud Computing*. to be published.

[33] Intel. Intel software guard extensions remote attestation end-to-end example. https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example.

[34] Bernd Jaeger. Security orchestrator: Introducing a security orchestrator in the context of the etsi nfv reference architecture. In *IEEE Trustcom/bigdatase/ispa*, pages 1255–1260, 2015.

[35] Prerit Jain, Soham Jayesh Desai, Ming-Wei Shih, Taesoo Kim, Seong Min Kim, Jae-Hyuk Lee, Changho Choi, Youjung Shin, Brent Byunghoon Kang, and Dongsu Han. Opensgx: An open platform for sgx research. In *NDSS*, 2016.

[36] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 97–112, 2017.

[37] Eddie Kohler. *The click modular router*. 2001.

[38] Eddie Kohler. Click for measurement. *UCLA Computer Science Department, Tech. Rep*, 2006.

[39] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[40] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*, pages 16–18, 2017.

[41] Hongda Li, Juan Deng, Hongxin Hu, Kuang-Ching Wang, Gail-Joon Ahn, Ziming Zhao, and Wonkyu Han. Poster: On the safety and efficiency of virtual firewall elasticity control. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, pages 129–131. ACM, 2017.

[42] Hongda Li, Hongxin Hu, Guofei Gu, Gail-Joon Ahn, and Fuqiang Zhang. vnids: Towards elastic security with safe and efficient virtualization of network intrusion detection systems. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–34. ACM, 2018.

[43] Liu, Haikun, Jin, Hai, Liao, Xiaofei, Xu, and Cheng-Zhong. Performance and energy modeling for live migration of virtual machines. *Cluster Computing*, 16(2):249–264, 2013.

[44] Libin Liu, Hong Xu, Zhixiong Niu, Peng Wang, and Dongsu Han. U-haul: Efficient state migration in nfv. In *ACM Sigops Asia-Pacific Workshop on Systems*, pages 1–8, 2016.

[45] Yeping Liu, Zhigang Guo, Guochu Shou, and Yihong Hu. To achieve a security service chain by integration of nfv and sdn. In *International Conference on Instrumentation & Measurement*, pages 974–977, 2016.

[46] Algo logic systems. http://algo-logic:com/.

[47] Rashid Mijumbi, Joan Serrat, Juan Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys and Tutorials*, 18(1):236–262, 2016.

[48] Mininet. http://mininet.org/.

[49] Thomas Morris. Trusted platform module. 2011.

[50] Leonhard Nobach, Ivica Rimac, Volker Hilt, and David Hausheer. Slim: Enabling efficient, seamless nfv state migration. In *IEEE International Conference on Network Protocols*, pages 1–2, 2016.

[51] Vladimir Andrei Olteanu and Costin Raiciu. Efficiently migrating stateful middleboxes. *ACM SIGCOMM Computer Communication Review*, 42(4):93–94, 2012.

[52] openssl. https://www.openssl.org/.

[53] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[54] Aurojit Panda, Keon Jang, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: taking the v out of nfv. In *Usenix Conference on Operating Systems Design and Implementation*, pages 203–216, 2016.

[55] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18), Renton, WA*, 2018.

[56] Shriram Rajagopalan, Williams Dan, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Usenix Conference on Networked Systems Design and Implementation*, pages 227–240, 2013.

[57] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed data: a new abstraction for building trusted cloud services. In *Usenix Conference on Security Symposium*, pages 10–10, 2012.

[58] Scapy. https://scapy.net/.

[59] Sebastian Schinzel and Sebastian Schinzel. Cache attacks on intel sgx. In *European Workshop on Systems Security*, page 2, 2017.

[60] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *IEEE Symposium on Security and Privacy*, pages 38–54, 2015.

[61] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[62] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-nfv: Securing nfv states by using sgx. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 45–48. ACM, 2016.

[63] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[64] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.

[65] Snort. https://www.snort.org/.

[66] TFN. https://en.wikipedia.org/wiki/Tribe_Flood_Network.

[67] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, page 2. ACM, 2018.

[68] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 991–1008, 2018.

[69] Ricard Vilalta, Arturo Mayoral, Raul Munoz, Ramon Casellas, and Ricardo Martínez. Multitenant transport networks with sdn/nfv. *Journal of Lightwave Technology*, 34(6):1509–1515, 2015.

[70] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434. ACM, 2017.

[71] Yang Wang, Gaogang Xie, Zhenyu Li, Peng He, and Kavé Salamatian. Transparent flow migration for nfv. In *Network Protocols (ICNP), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[72] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy*, pages 640–656, 2015.

[73] Pouya Yasrebi, Sina Monfared, Hadi Bannazadeh, and Alberto Leongarcia. Security function virtualization in software defined infrastructure. *integrated network management*, pages 778–781, 2015.

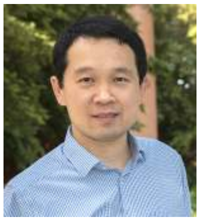
[74] Tianlong Yu, Seyed Kaveh Fayaz, Michael P Collins, Vyas Sekar, and Srinivasan Seshan. Psi: Precise security instrumentation for enterprise networks. In *NDSS*, 2017.

[75] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, KK Ramakrishnan, and Timothy Wood. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*, pages 26–31. ACM, 2016.

[76] Taassori, Meysam and Shafiee, Ali and Balasubramonian, Rajeev. : Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *Proceedings of the ASPLOS, March 2428, 2018, Williamsburg, VA, USA*, pages 665–678. ACM, 2018.

Wang Juan is an Associate Professor at School of Cyber Science and Engineering of Wuhan University. She received her M.E. and Ph.D degrees in computer school from Wuhan University, China in 2004 and 2008. In 2018 and Jan. 2010, she did research as a visiting scholar in Pennsylvania State University and Arizona State University, USA. Her research has been supported by NSF projects. She has authored and co-authored over 40 papers and holds 10 patents in security areas. Her current research interests include cloud security, trust computing, SDN and NFV security. Email: jwang@whu.edu.cn.

Shirong Hao received her bachelors degree in information security from Wuhan University in 2017. And she is currently working towards the masters degree at School of Cyber Science and Engineering of Wuhan University. Her research interests include NFV, system security and IoT security.



Hongxin Hu is an Associate Professor with the Division of Computer Science, School of Computing, Clemson University. He received his Ph.D. degree in computer science from Arizona State University, Tempe, AZ, USA, in 2012. He has published over 100 refereed technical papers, many of which appeared in top conferences and journals. His current research spans security, privacy, networking, and systems. He received the NSF CAREER Award in 2019. He was a recipient of the Best Paper Award from ACM SIGCSE 2018 and ACM CODASPY 2014 and the Best Paper Award Honorable Mention from ACM SACMAT 2016, IEEE ICNP 2015, and ACM SACMAT 2011.

Bo Zhao is a Professor at School of Cyber Science and Engineering of Wuhan University. He is engaged in the research of trusted computing theory and technology. At present, he is the director of China cryptography society, Senior member of CCF (China Computer Society). He presided over two national 863 projects, NSFC key projects of Hubei natural fund, and many cooperation projects of ministries, departments, enterprises and institutions, practiced and applied trusted computing and information system security theory and technology in Cyberspace Security, and achieved certain results. He has published more than 70 papers on SCI, EI and core journals. He has 12 domestic patents and 1 international patent.

Hongda Li is a PhD student in the School of Computing at Clemson University. He received his M.S. in Computer Science from Clemson University and M.E. in Software Engineering from University of Science and Technology of China. His research interest lies in network and system security. Networks and systems keep evolving in the past decade, especially thanks to the advances in softwarization, Internet of Things (IoT), and artificial intelligence (AI).

Wenhui Zhang is a PhD student in Pennsylvania State University. Her research interests include systems security, which includes IoT security, PLC security and network system security (e.g. NFV security) and security and verification for distributed systems. She is serving as committee members and committer roles in several popular NFV related open source projects, including OpenNetVM and Akraino Edge Stack.

Jun Xu is an Assistant Professor in the Computer Science Department at Stevens Institute of Technology. He received his PhD degree from Pennsylvania State University and his bachelor degree from USTC. His research mainly lies in the areas of software security and system security. He is a recipient of ACM CCS Outstanding Paper Award, Penn State Alumni Dissertation Award, RSA Security Scholar Award and USTC Guo-moruo Scholarship.

Peng Liu is a Professor of Information Sciences and Technology, College of Information Sciences and Technology. He is also Director of Center for Cyber-Security, Information Privacy, and Trust at Pennsylvania State University. He has more than 300 papers. His teaching and research interests include system security and survivability, distributed systems, and peer-to-peer systems in the contexts of E-Commerce, digital health care, digital government, command and control, digital infrastructure systems.

Jing Ma was born in April, 1982. She is an engineer of Science and Technology on Information Assurance Laboratory. Her research interests include information security, trusted computing, and computer application.