future internet

MDPI

Article

Design and Implementation of Virtual Security Function Based on Multiple Enclaves

Juan Wang [1,2,*], Yang Yu [1,2], Yi Li [1,2], Chengyang Fan [1,2] and Shirong Hao [1,2]

[1] School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China; 2018202110019@whu.edu.cn (Y.Y.); liyi1023@whu.edu.cn (Y.L.); cyfan@whu.edu.cn (C.F.); shirong@whu.edu.cn (S.H.)

[2] Key Laboratory of Aerospace Information Security and Trusted Computing Ministry of Education, Wuhan 430072, China

* Correspondence: jwang@whu.edu.cn

Abstract: Network function virtualization (NFV) provides flexible and scalable network function for the emerging platform, such as the cloud computing, edge computing, and IoT platforms, while it faces more security challenges, such as tampering with network policies and leaking sensitive processing states, due to running in a shared open environment and lacking the protection of proprietary hardware. Currently, Intel® Software Guard Extensions (SGX) provides a promising way to build a secure and trusted VNF (virtual network function) by isolating VNF or sensitive data into an enclave. However, directly placing multiple VNFs in a single enclave will lose the scalability advantage of NFV. This paper combines SGX and click technology to design the virtual security function architecture based on multiple enclaves. In our design, the sensitive modules of a VNF are put into different enclaves and communicate by local attestation. The system can freely combine these modules according to user requirements, and increase the scalability of the system while protecting its running state security. In addition, we design a new hot-swapping scheme to enable the system to dynamically modify the configuration function at runtime, so that the original VNFs do not need to stop when the function of VNFs is modified. We implement an IDS (intrusion detection system) based on our architecture to verify the feasibility of our system and evaluate its performance. The results show that the overhead introduced by the system architecture is within an acceptable range.

Keywords: NFV; SGX; enclave; hot swapping; click

Citation: Wang, J.; Yu, Y.; Li, Y.; Fan, C.; Hao, S. Design and Implementation of Virtual Security Function Based on Multiple Enclaves. *Future Internet* **2021**, *13*, 12. https://doi.org/10.3390/fi13010012

Received: 1 December 2020
Accepted: 2 January 2021
Published: 6 January 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In traditional solutions, network functions (NF) are usually deployed in proprietary hardware devices. They are increasingly unable to meet the flexible and scalable requirements for the emerging platform, such as cloud computing, edge computing, 5G [1], and Internet of Things. Network function virtualization (NFV) [2] as a new type of network technology can meet these challenges. NFV refers to the use of standard IT virtualization technology to implement network functions in software and deploy on general purpose servers, switches, and storage. Security function virtualization (SFV) has also been proposed to provide agile security functions. Although the simplicity and low cost of NFV accelerate the innovation and market speed of network products, VNFs (virtual network functions), especially virtual security functions, face more security challenges [3,4] than traditional network functions because VNFs typically operate in a shared, open environment, away from the protection of proprietary hardware. Insiders have malicious access to sensitive information such as virtual network function (VNF) codes and states. Moreover, there is an isolation problem in the NFV architecture itself. Attackers may steal and damage the data of other virtual machines through VM escape. Recently, Software Guard Extensions (SGX) [5] proposed by Intel has provided a promising way to solve the security issues of VNF [6,7]. SGX only trusts the CPU. Even the operating system, driver, BIOS, or virtual

machine monitor (VMM) cannot access the data and code in the enclave. Therefore, the VNF can be put into the enclave to ensure its runtime security.

However, previous works [6,8,9] place multiple VNFs in a single SGX enclave which loses the scalability advantage of NFV. First, users cannot scale in/out a single network function because multiple VNFs run in an enclave as a whole. Then, users cannot dynamically modify network functions because the network functions protected by SGX need to be recompiled and redeployed once they are modified. Finally, it makes it difficult to combine new security services by using the existing modules due to the monolithic design with a single enclave.

To address these issues, we propose a multi-enclave-based virtual security function architecture by using the security features of SGX and the modularity of Click [10], which is a new software architecture for building flexible and configurable network middle-boxes. In our architecture, the virtual security functions are decomposed into many small elements and modules and then put into multiple enclaves to provide strong confidentiality and integrity guarantees. The system can freely combine these modules according to user requirements, and increase the scalability of the system while protecting its code and running state security. In addition, we design a new hot-swapping scheme to enable the system to dynamically modify the configuration function at run time, so that the original VNFs do not need to stop when the function of VNFs is modified. We implement an intrusion detection system (IDS) based on our architecture to verify the feasibility of our system and evaluate its performance. The results show that the overhead introduced by the system architecture is within an acceptable range.

To our knowledge, our work firstly designs and implements a multi-enclave-based virtual security function architecture to enable the VNFs to be scaled dynamically with enclave protection. In this work, we make the following contributions.

(1) Propose a multi-enclave-based virtual security function architecture so that the different modules running inside enclaves can be freely combined to new security service according to user requirements.

(2) Design a new hot-swapping scheme to enable users to dynamically modify the configuration of VNFs at runtime without stopping the original VNFs when the function of VNFs is modified.

(3) Implement an IDS based on our architecture and evaluate its performance. The results show that the overhead introduced by the system architecture is within an acceptable range.

The rest of this paper is organized as follows. Section 2 introduces the related work. Section 3 describes SGX technology and Click. Section 4 presents the system architecture and key approaches. Section 5 introduces the implementation of our system. The evaluation of the system is depicted in Section 6. Section 7 concludes this paper.

2. Related Work

Some research efforts have been devoted to building trusted VNFs. Marku et al. [11] provide a summary of current techniques for establishing trusted NFVs, and summarize two approaches to protecting NFVs in terms of cryptographic or trusted hardware-based mechanisms. ESTI [2] proposes to provide trusted protection based on HSM (hardware security module), TPM (trusted platform module) [12], and vTPM (virtual trusted platform module) [13]. NetBricks [14] leverages a safe language (Rust) and LLVM [15] to build a zero copy soft isolation. It provides memory isolation software by using type safe language and achieves high performance by adopting LLVM as an optimization back-end of compilers. NetBricks can also ensure that only a single NF can access a packet, so as to guarantee the packet isolation in common cases. OpenNetVM [16] runs NFs in lightweight Docker containers based on the NetVM architecture. It provides NF isolation through container mechanisms, such as namespace and capability.

In recent years, with the rise of SGX technology, using SGX to protect VNF has also become a hot research direction. SGX can isolate network functions and sensitive data into a secure container called enclave.

Coughlin et al. [8] first proposed a new idea of using SGX to protect a virtual network function. The article proposed to put Click elements into an enclave to provide run-time isolation and compare the performance overhead of using SGX. Wang et al. [17] analyzed the security challenge of using SGX to protect VNF. To improve system performance, a lightweight trusted implementation framework for protecting VNF was proposed based on SGX and Click, and a fine-grained state migration mechanism was presented. In order to verify the feasibility of the framework, the article implements a DDoS detection system based on their architecture. SafeBricks [6] proposed to put all VNFs into an enclave and use the security features of the Rust language for memory isolation of the VNF, thereby avoiding the performance overhead between multiple enclaves. S-NFV [9] uses SGX to design a new architecture to provide integrity protection for NFVs outsourcing. It ensures the stateful security of NFVs and proposes a remote attestation approach to allow remote parties to confirm the security of the VNF. Slick [18] proposed a secure middleware framework for deploying high-performance VNFs on untrusted commercial servers. Slick designs and implements various VNFs based on Click and runs on top of the Scone architecture. Scone [19] is a secure Docker container based on SGX that supports user-level threads and asynchronous system calls to reduce the performance overhead caused by thread synchronization and system calls in enclaves. Slick also optimizes its performance to keep the system's native throughput and latency.

In a word, these solutions combine the security features of NFV technology and SGX in all aspects and provide a solution for protecting VNF. However, the above work puts VNFs into a single enclave that affects the scalability of NFVs because users cannot scale in/out a single VNF. In addition, none of the above work supports the dynamic reconfiguration at run-time when the original VNFs need to be modified.

Compared with the previous work, we propose a virtual security function architecture based on multiple enclaves to ensure the security and scalability of VNFs. In our system, the sensitive modules of VNF can be put in different enclaves and communicate by local attestation so as to improve the scalability of the system while providing run-time security protection for VNFs. In addition, we present a new hot-swapping mechanism to enable the system to be reconfigured at run time without stopping the whole system.

3. Background

3.1. SGX

SGX (Intel® Software Guard Extensions, [5,20]) is a new CPU-based trusted execution environment technology with the goal of implementing advanced protection for confidentiality and integrity. The overall architecture of SGX is shown in Figure 1. It allocates the hardware-protected memory for an application code and data in which an isolated enclave runs [21]. The data in the enclave memory space can only be accessed by code that is also located in the enclave memory space. The privileged software such as the virtual machine monitor (VMM), BIOS, and even the operating system outside the enclave cannot directly access sensitive data and code inside the enclave. SGX has certain hardware limitations on the size of the protected memory, usually up to 128MB (the available memory for user data only is ≈93 M) [22]. Therefore, the number of enclaves running in protected memory is also limited. The use case shows that there are typically 5–20 enclaves that reside in memory, at the same time depending on the memory usage of each enclave. Enclave can be called with special instructions.

The SGX architecture includes 17 new instructions, new processor architecture, and new execution mode. SGX loads the enclave into the protected memory, accesses resources through page table mapping, and executes the enclave application. The system software still maintains control of the enclave's access to resources. The entire application can be packaged into an enclave, or it can be decomposed into multiple small

components, and only the critical security components are placed in the enclave. Due to the limitation of SGX memory size, the latter is our commonly used enclave application development method.

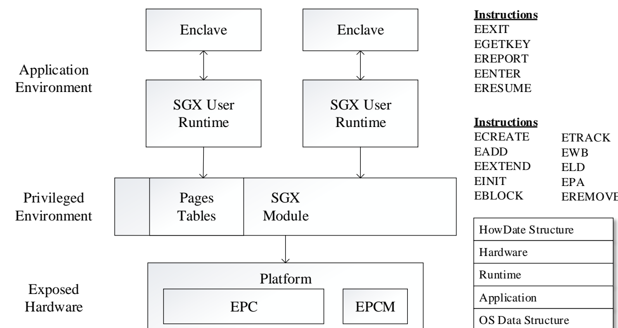

Figure 1. Architecture description of Software Guard Extensions (SGX).

EPC. In order to achieve memory protection, SGX requires new hardware and structure. The enclave pages and SGX structures are stored in a protected memory called the enclave page cache (EPC). EPC is protected by both hardware and software and is divided into many EPC pages, each of which is a 4 KB block. Inside the EPC, there are many different enclaves running when an enclave accesses memory in the EPC; whether to allow access is determined by the CPU processor. The access control information for each page in the EPC is maintained in a hardware structure called the enclave page cache map (EPCM) in the processor.

Enclave created. The enclave binary is loaded into the EPC and an identity for the enclave binary is established in the creation process. The creation of the enclave includes the following steps: initialize the enclave control structure, allocate EPC pages and load the contents of the enclave into the pages, measure enclave content, and eventually establish the enclave identity.

Enclave exit and entry. The most important thing to maintain the confidentiality and integrity of the enclave is to control the data into and out of the enclave. The entry process needs to clear all caches transfers associated with the enclave address area. All memory accesses in enclaves need to be properly checked. The entry process must confirm that the processor is inside the enclave, and the processor should transfer control and enable the enclave execution mode. Before exiting the security zone, all cache translations in the protected area must be cleared, so that other software cannot use the cache translation to access the protected memory area of the enclave.

Interrupts, faults, or anomalies may occur when operating in enclave mode. The processor can pass the specific fault handling program through the system software traditionally. The fault handler saves the register states, and once the event is processed, the system software will restore the register state and return control to the point of interruption. SGX places the system software within the enclave's trust boundary to allow the system software to read and modify the enclave's registration status. Therefore, a new routine is introduced in SGX to protect the integrity and confidentiality of the enclave.

Intel SGX provides special instructions to enter and exit the enclave: EENTER and EEXIT. When an enclave exits, the processor will call a special internal routine called asynchronous exit (AEX), which retains the enclave's register states, clears the register, and sets the address of the error instruction to the value defined by EENTER. The ERESUME instruction is used to restore the state and allow the enclave to continue execution.

SGX attestation. SGX offers two attestation mechanisms-local and remote. Local attestation is used between two enclaves running on the same platform. This paper uses local attestation to ensure that multiple enclaves run on the same platform and transmit information between multiple enclaves through a secure channel established by local attestation.

3.2. Click

Click [10] is a famous modular routing software architecture proposed by MIT university to build flexible and configurable routers. It provides a modular router, allowing us to compose different elements for easily building different VNFs. Click consists of fine-grained packet processing modules called elements, each of which implements simple packet processing functions, such as packet classification, queuing, scheduling, and interface with network devices. Based on the modular and scalable architecture, users can also write new elements or combine existing elements according to their own needs. Click is extended as a Linux module on general purpose PC hardware. On the 700 MHz Pentium III, Click can achieve a maximum lossless forwarding rate of 333,000 64-byte packets per second. This is about four times the standard Linux router on the same hardware. It proves that Click's modularity and flexibility are compatible with good performance.

The Click architecture is inspired by router properties in several ways. First, the packet switching along the connection can be initiated by the source (push processing) or the destination (pull processing). This allows for a clear simulation of the packet flow patterns of most routers, while pull processing can write a combinable packet scheduler. Second, the flow-based router context mechanism allows elements to automatically locate other elements it depends on. These features make individual elements more powerful and configurations easier to write.

The element is the basic unit of packet processing in Click. Each Click element is often only responsible for a single routing function, and each element has multiple entries and exits. Multiple elements can be connected to each other's gateways to form a complex network function using Click's own configuration language. These connections that implement complex network functions are typically stored in a configuration file. Click can complete the initialization and operation of the network function by parsing the configuration file.

The most important attributes of the element are as follows.

Element class. Each Click element belongs to an element class. It specifies the configuration of the element, initialization, number of exits and entries, and processing of the packet. In addition to the underlying element class, Click also has a number of other components built into it for developers to implement a user-defined element after implementing these basic virtual functions.

Port. Elements can have any number of input and output ports. Each connection goes from the output port on one element to the input port on another element. Different ports can have different semantics. For example, by default, the second output port of each element is used to send the wrong packet.

Configuration string. Each element of Click needs to be configured successfully before it can be properly initialized. The configuration string usually includes all the parameters required for element initialization and the private state of each element. Elements can be initialized by reading this information from the configuration file.

The new element is created in Click by defining a new class, and the input ports, output ports, and configuration strings are modified according to the function. Figure 2 shows how we diagram properties for a single element, EleClass(3). The triangular port is input port and rectangular ports are output ports. EleClass receives data from the input port and sends the processed data to two output ports. The '3' in the brackets represents the configuration string of EleClass, which represents the total number of input and output ports (3).

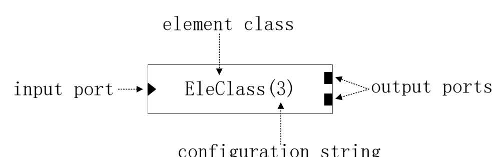

Figure 2. A sample element in Click.

The Click configuration is written in a simple Click language that consists of two important structures: a declaration to create elements and a connection. Among them, the connection statement defines the flow of packets between elements. The Click language contains an abstraction mechanism called composite elements that allows users to define their own element classes. A composite element is a configuration fragment of a router that behaves like an element class. Composite elements can have any number of push/pull characteristics. At initialization, each use of the composite element is compiled into a corresponding collection of simple elements.

Click provides a number of basic packet processing elements, such as checking packet length, packet classification, packet fragmentation, and reassembly, receiving packets from the network card, constructing packets and controlling the number and rate of packet transmissions, dropping packets, etc. In recent years, more and more researchers have used Click's modular architecture to build many useful middleware platforms [18,23–26].

4. System Design

4.1. System Structure

Our goal is to design and implement a multi-enclave-based virtual security function architecture based on SGX and Click technology. Click itself can build complex network functions through many small elements with packet processing capabilities, and Click supports developers to design new elements according to their own requirements. In order to ensure the security of VNF, we use SGX technology to protect the VNF in the enclave. However, if we completely adopt the Click architecture, and put each element in a separate enclave, there will be too many enclaves, which has large EPC memory and communication costs, although it can retain its modular advantages. According to our testing, packet transmission across enclaves can result in significant performance overhead. When the number of enclaves is too large, the frequency of packets entering and leaving the enclave is too large, which causes a lot of overhead. Therefore, this paper proposes to split the overall VNF into multiple small modules according to the packet type and put them into a single enclave to protect them. Each module is responsible for one feature of packet processing, and these modules can be freely combined based on user requirements. At this point, the frequency of data packets entering and leaving the enclave is much smaller than that of the small element combination, which not only ensures the runtime security of the VNF but also retains the modular advantage of Click to some extent.

Figure 3 shows the overall architecture of a virtual security function based on multiple enclaves. The architecture consists of a whole VNF composed of multiple enclaves, local attestation module, manager module, Log module, system call lib, socket, DPDK, and NICs.

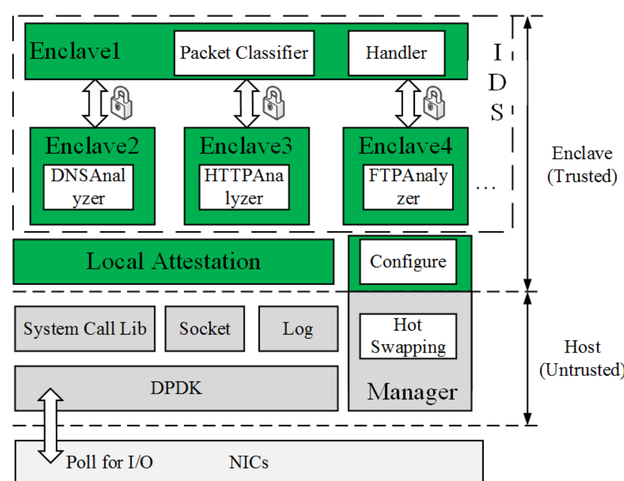

Figure 3. System Architecture.

Due to the limitations of SGX technology [22], it is impractical to protect all the data in SGX and we need to classify the data. The architecture is mainly divided into trusted parts, untrusted parts, and NICs. Among them, the conversion between the trusted part and the untrusted part needs to perform data interaction through the encapsulated ecall and ocall of SGX. Ecall and ocall are key factors in the performance of SGX. When the number of enclaves is too many, the frequent entry and exit of enclaves will cause a lot of extra overhead. Therefore, we need to control the number of enclaves to ensure high performance. Ecall is the function call performed by the untrusted part to enter an enclave, passing some arguments and/or expecting a return value. In addition, ocall is the function call performed by the enclave to call back to the untrusted part, passing some arguments and/or expecting a return value.

The manager is a key part of the system and responsible for the secure boot of the system, mainly including the command line parameters parsing, the enclave configuration and initialization, the overall operation of the control system, and a management of the enclave life cycle. The Manager has both trusted and untrusted parts, including the Configure file stored in the trusted part and the hot swapping in the untrusted part to support hot swapping. The NICs represent network interface controllers, which are responsible for receiving data frames sent by other devices on the network and reassembling the frames into packets.

4.1.1. Trusted Part

The trusted part is mainly composed of an overall VNF module, configure and local attestation. The configuration file stores the configuration information of the virtual network function. The Manager can read the configuration file through ecall and parse the file according to the file. The local attestation module is mainly responsible for the authentication between multiple enclaves to ensure that it runs on the same platform, which is also responsible for the transmission of information between multiple enclaves. This module also provides a cryptographic function library to encrypt and decrypt sensitive packets.

In addition, an overall VNF consists of multiple modules. In our architecture, the sensitive modules of a VNF should run in the different enclaves. This architecture has three advantages. Firstly, it can keep sensitive modules secure and prevent sensitive data leakage. Secondly, it can increase the scalability of the system. Thirdly, since our architecture decomposes an overall VNF 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 provided by SGX can evict the rarely used memory pages to DRAM pages outside the PRM (processor reserved memory) range with encrypted mode. As shown in Figure 3, the sensitive modules of IDS, such as DNSAnalyzer, HTTPAnalyzer, FTPAnalyzer, and packet classifier, are running in different enclaves. These enclaves are freely combined through the Click configuration language and stored in the configure module. The enclave also includes a Handler module. The Handler includes a read Handler and a write Handler. The read Handler can be used to read the information in the enclave. The write Handler can be used to dynamically modify the configuration parameters inside the enclave.

4.1.2. Untrusted Part

The untrusted part includes system call library, socket module, DPDK, and log module. Among them, log module is responsible for recording the log information during the processing of the data packet. The DPDK is primarily responsible for reading the encrypted packets from the NICs queue and put into untrusted memory by polling. The system call library includes multiple ecalls and ocalls functions, which are mainly responsible for the conversion between the trusted part and the untrusted part of the system. The socket module is primarily responsible for communication.

4.2. Secure Boot

The Manager is the core of the system and is responsible for the secure startup of the system, including configuration file parsing, enclave creation and destruction, and dynamic configuration modification. The Manager is mainly composed of Manager_Main (untrusted) and Manager_Enclave (trusted). The Manager_Main is responsible for the static initialization of Click and the parsing of command line parameters and the configuration file. The Manager_Enclave is used to protect the lexer (lexer is used to parse Click configuration files) creation, routing generation, elements connection, etc. In order to ensure the security of the configuration file, we use the SGX seal and unseal functions to seal it. Hence, Manager_Main will ecall into Manager_Enclave to perform configuration file parsing, lexer, element and port connections.

Figure 4 is a flow chart of the secure boot. When the system starts executing, the Manager_Main first initializes the Click static route, which is mainly responsible for the initialization of parameter types and routing parameters, and the initialization of the global Handler. Then the Manager will read the command line parameters and the configuration file encrypted by SGX seal to see if other functions are needed for this operation, such as whether hot swapping is supported. Next, the Manager will enter Manager_Enclave via an ecall. Manager_Enclave first unseals the configuration file, then reads the configuration and parses the configuration file. Manager_Enclave creates a lexer and converts the configuration file to a String type and passes it as a parameter to the lexer. The lexer creates a route, then extends the route based on the passed String parameters, sorts and adds elements and port connections. At this point, Manager_Enclave is executed internally, and the system will return to Manager_Main to continue execution. Next, the system traverses the entire route according to the order and configures each element. The configuration of each element creates an associated enclave, and multiple enclaves need to be locally authenticated to ensure that they each run on the same platform. This includes local attestation between enclaves and local attestation between them and Manager_Enclave. After the local attestation is successful, the Manager_Enclave will communicate with the enclave running the functions in turn, and perform the configuration and initialization of the functions in each enclave in this order. In addition, the functions running inside each enclave are composed of multiple elements of Click. Therefore, when the function in the enclave is initialized, the enclave internally generates routes, adds elements and port connections, and executes the elements in turn.

4.3. Hot Swapping

Click provides support for hot swapping [10]. Hot swapping means that when the system processes packets according to its configuration, the user can dynamically modify the configuration (such as adding new elements, etc.) without stopping the system. If the new configuration is resolved correctly, the new configuration will automatically take over the states of the old configuration and continue processing the packet, for example, any queued packets will be moved to the new configuration. In this way, Click can implement the function of dynamically modifying the configuration without terminating the operation.

Hot swapping can modify (add or delete) elements in the configuration file by creating new routes, hence you need to establish communication through the "ControlSocket element (an element opens control sockets for other programs)" and pass the modified configuration information for Click resolution. When we use SGX to guarantee the runtime security of the elements, it will inevitably cause losing the dynamic configuration modification function of Click. In this paper, we propose a new hot swapping mechanism based on SGX and Click technology to keep the system to dynamically configure without termination.

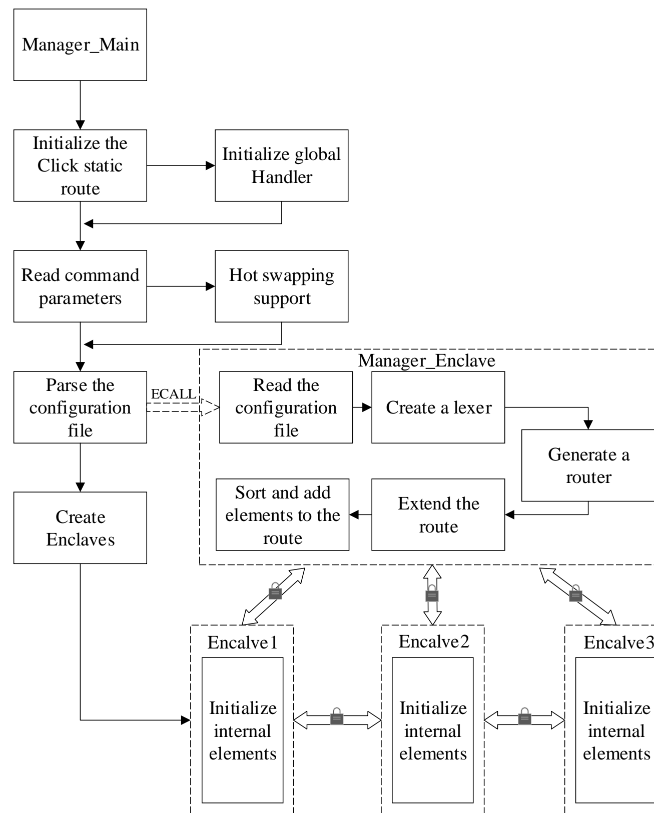

Figure 4. Secure Boot.

Figure 5 is a flow chart of the system hot swapping. When a user needs to use the hot swap function to dynamically modify the configuration, they first add a parameter, such as -r, and add the ControlSocket element to the configuration file (declare the connection, type and port number). When the system starts running, it first reads the command line parameters. According to whether the command line has the -r parameter, it is determined whether the configuration needs to support the hot swap function. If there is a -r parameter, the system will add a global write Handler in the Manger main (global write Handler including element name, Handler name, user data, etc.). After the Handler is added, the Handler is stored as a global Handler to dynamically modify the configuration file through the Handler. The system then begins routing initialization. After the route initialization is complete, the old configuration begins to run normally. At this point, on the one hand, the system routes the packet through the old configuration. On the other hand, the ControlSocket in the old configuration is always listening for the connection. When the user needs to use the hot swap function to modify the running old configuration file, a new terminal is connected to the port of the old configuration that is running through the ControlSocket, and the new configuration is sent to the old system through the ControlSocket connection. When the old system listens to the new connection, it first parses the new command, matches the parsed Handler with the stored Handler according to the Handler name, and if the matching is successful, it creates a new lexer and route, parses the new configuration, and connect the elements and ports. In addition, the enclave as a Click element is created, configured, and initialized in order based on the contents of the new configuration file. After all the enclaves in the new configuration are initialized, the system will destroy the last route used for hot-plugging and copy the newly created route to the hot-plug route. If the new configuration does not require enclave creation, it just performs this step directly. Next, the system will set the running state of the newly created route for subsequent scheduling by the state information. Finally, the system driver starts the execution of the schedule. It first suspends the old configuration that

is running, and then compare the enclaves in the old and new configuration one by one. If there is the same enclave, the state information of the enclave in the old configuration is synchronized by the state synchronization function through a secure channel established by local attestation. After the state synchronization is completed, the enclave in the old configuration can be destroyed, and the new configuration can start running and perform new routing processing on the data packets.

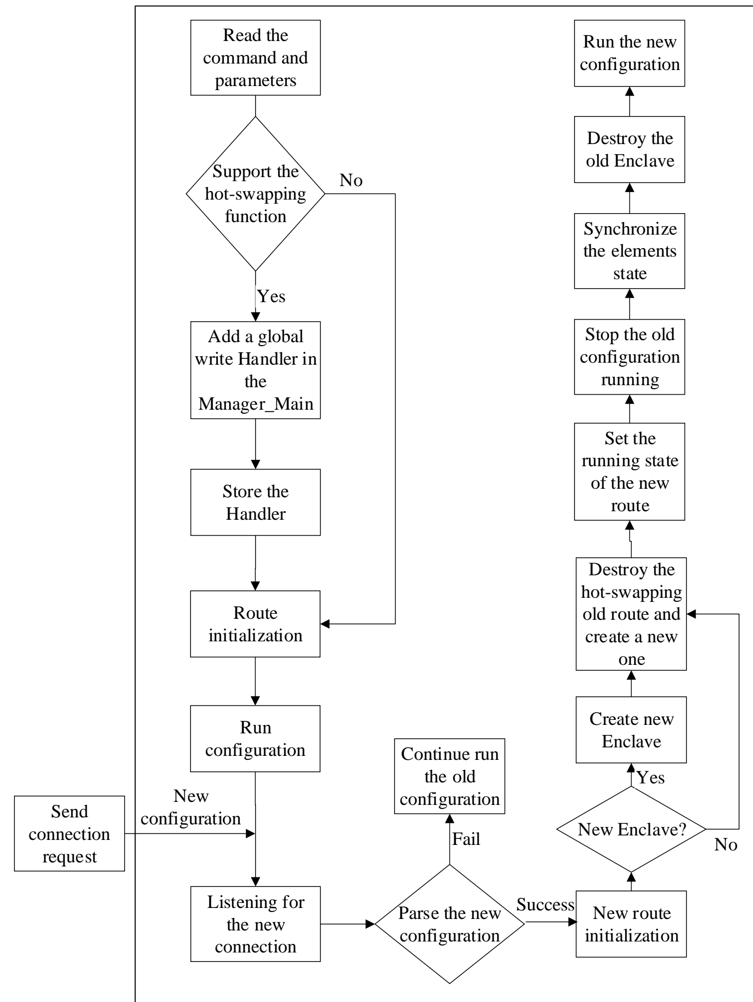

Figure 5. Hot Swapping.

4.4. Local Handler

Unlike global handlers that support hot swapping, a local handler is used to modify the parameters in the same element. However, the local handler of Click cannot support SGX. Therefore, we modify the Handler function of Click to reimplement a Handler class for SGX enclaves. Each enclave can pre-set the parameters that it may need to modify, and set the ecall_write_handler call for the enclave. The modified ecall includes the Handler name, the read and write permissions of the Handler, and the parameters to be modified. When users modify the configuration using a local Handler, the enclave name and Handler name to be modified can be sent to the new running system. The system will also read the passed parameters and configuration information through the ControlSocket connection, then parse it, and use the resolved enclave name and handler name for ecall. The enclave name specifies the eid of the ecall's enclave. When it reaches an enclave, it is matched by the handler name, and the new parameter value is assigned to the matched parameter. In addition to writing Handlers, the read Handlers can also be encapsulated in the form of an ocall, which is responsible for outputting information to the outside.

5. Implementation

This section first introduces the initialization of the VNF in the enclave and then describes how to decompose the overall VNF into multiple small packet processing modules, puts them into different enclaves, and optimize their performance to reduce the performance overhead of multi-enclave architecture. The main codes of the manager and each analyzer in IDS are attached in Appendices A–D.

5.1. VNF Initialization

Each enclave consists of a virtual function responsible for packet processing and a local handler for modifying internal parameters and external output. Therefore, when the virtual function performs initialization, multiple elements of the virtual function running inside the enclave also need to be initialized. The steps include creating routes, elements and port connections, and the initialization of each element. (Internal initialization does not need the lexer to analyze the configuration process because the connections inside each VNF are fixed). In addition, the packets in the enclave also need to be processed by ecalls.

5.2. The Protection of IDS

We implement multiple enclave protected IDS solutions based on vNIDS [27]. vNIDS is an innovative NIDS architecture to address the challenges of efficient intrusion detection and the monolithic NIDS configuration in virtualized NIDS. vNIDS has three microservices: head-based detection, protocol-based parsing, and payload-based detection. The three microservices of vNIDS are implemented based on the Click modular router software. Click provides rich networking processing elements, which can be leveraged to construct a wide range of network functions. In order to achieve better detection results, vNIDS is implemented based on ClickOS [26] and ported the event engine of Zeek [28] (previously named Bro). Zeek is a passive open source network traffic analyzer which supports a variety of traffic analysis tasks, even outside the security domain. Zeek's scripting language facilitates a broader approach to discovering malicious activities, including semantic misuse detection, anomaly detection, and behavior analysis.

Based on our architecture, we use SGX and Click to implement four different protocol analyzer elements (HTTPAnalyzer, SSHAnalyzer, FTPAnalyzer, and DNSAnalyzer) to detect whether it is subject to session hijacking, DNS tunneling attacks, and Trojan horses. When a packet is sent to the IDS, it is first classified based on the header information. Packets belonging to different application protocols are sent to different protocol analyzers for further analysis. The protocol analyzer then generates an event based on the analysis results. The IDS then passes these events to the event engine. The event engine sends the packet to the behavior element, such as discard, delay, etc. or sends it to the next virtual function for further processing.

5.3. Performance Optimization

5.3.1. DPDK

In order to achieve high throughput and low latency, this paper uses a high-performance packet I/O library-DPDK [29], which is a data plane development tool kit.

In the traditional Click architecture, the FromDevice element, which acts as a sniffer function, sends packets captured from the NIC to the kernel thread in a hardware interrupt and then processes the packet in the protocol stack. Compared with the traditional mode, DPDK has the following advantages: (1) DPDK abandons the traditional interrupt mode and reads the data packet from the network card by polling, avoiding the interrupt overhead. (2) DPDK allocates large memory pages to replace normal memory and reduce cache-miss. (3) Using user space I/O technology (UIO) to intercept interrupts, thus bypassing the subsequent processing of the kernel protocol stack, so as to greatly improve network performance.

The use of the DPDK causes the system to no longer use the Click buffer, and it can no longer use Click's packet object, instead of using the DPDK packet processing mechanism. Placing the DPDK in the enclave increases the size of the TCB (approximately 516K) [6], so for reasons of TCB size and performance, we place the DPDK in an untrusted area outside the enclave.

5.3.2. Reduced System Calls

First of all, to reduce the number of ecalls and ocalls, multiple elements are put into different enclaves according to the function rather than in one single enclave. In this way, the conversion frequency of the enclave can be reduced to some extent.

In addition, the VNF often needs to determine whether it has been attacked based on the rate of the packet. At this point, we need to get the time through the system call, but if you put these VNFs into the enclave, getting the trusted time becomes a big problem. On the one hand, if we frequently use the ocall call to get the time, even if we ignore the performance overhead, its security cannot be guaranteed. Because the ocall call gives control to the untrusted part, there is no guarantee that the time acquired by the ocall is trustworthy, which loses the meaning of determining whether the system is attacked by the network by calculating the rate. On the other hand, the SGX platform service enclave (PSE) currently provides trusted time services in seconds only for enclaves [30]. However, it is obvious that this granularity is far from meeting the current network needs. There are many alternatives for trusted time, the most common being the time service of the system through shared memory. In addition, ShieldBox [31] proposes a time source from hardware, such as a PTP clock on the NIC. This method can achieve a higher time resolution than the former. However, they all get time from untrusted sources and are therefore vulnerable to malicious tampering. Zhang et al. [32] proposed to obtain time from a remote trusted source, but due to the use of a large number of web interfaces, the resolution is still very low (100 milliseconds). Therefore, providing trusted and high-accuracy time for SGX applications remains an open question [33].

We use the shared timestamp scheme to provide an untrustworthy timestamp for VNFs based on the above considerations. When a packet arrives at the FromDevice element, it uses shared memory to get the timestamp and then write it to the annotation section of the packet. When the VNF needs to use the timestamp, it can read from the packet annotation section. In addition, the ocall is also required to maintain log I/O. In order to improve system performance, we also optimize some code to reduce unimportant log operations.

5.3.3. Batch

We introduce the batch processing function to improve the performance of the system. The BatchElement class is extended based on the Element class of Click. Similar to the Element class, BatchElement also extends the push method of the packet, called push_batch, while the received parameter of this method is no longer Packet * instead of PacketBatch *. The pull method also has a corresponding pull_batch method that supports batch processing. We add batch processing functions to the packet processing elements, such as packet reception and discarding. In this system, the default value for batch settings is 128. The system will modify the number of packets per batch based on the packet sending rate.

5.3.4. Shared Memory

In Click architecture, the transmission of data packets in elements is based on the mechanism of copy-on-write. When a packet needs to be copied, only the header of the data packet needs to be copied, because most of the processing of the data packet is only necessary to read the header of the packet instead of the data of the packet. This mechanism greatly improves the efficiency of Click's execution.

Based on the above observation, we design our shared memory mechanism for data packet transmission. In our approach, the DPDK caches packets from network devices in a memory pool via polling and maintains a pointer queue to the packet buffer. Enclave does not need to transfer a large number of copied packets through the secure channel established by local attestation, instead of passing directly the address of the packet. The enclave can directly query the DPDK and retrieve the pointer to the next batch of packet buffers, then directly process the packets without any copy. This solution greatly reduces the frequent conversion of data packets between multiple enclaves and the frequent copying of data packets, so as to improve system performance.

6. Result

6.1. Experimental Set-Up

We evaluate our system on the local server with SGX and Click installed. The server running system version is Ubuntu 18.04.1 LTS, and the CPU is Xeon E3-1280 v6 quad-core 3.9 GHZ.

For testing the system function of detecting packets of different attack types, we use Scapy as the main tool for traffic generation. Scapy [34] is a powerful Python-based interactive packet operator and library. It can forge and decode packets of a large number of protocols, send them over the wire, capture packets, use pcap files to store or read packets, match requests, and replies. Scapy can handle most network testing tasks, such as scanning, trace routing, probing, unit testing, attack, or network discovery with a fast sending rate. Scapy is installed on the same server as our system and sends the generated network packets to the system. Ten threads of Scapy run at the same time to send packets of a certain length, and our system is overloaded according to our experiment. Scapy generates TCP and UDP traffic and sends it to our system, some of which are malicious network traffic generated by flightsim [35]. We record the data of the system running for 30 min and repeat the test 10 times for each package length, taking the average data of all tests as the final result.

6.2. Performance of System

We evaluate the performance of our system from various aspects including system throughput, packet processing time, and CPU occupancy. We consider several cases of our architecture:

- Baseline represents the unprotected Click-based IDS;
- S-VNFs represents our system with the protection of multiple enclaves.
- S-VNFs (w/opt.) represents an optimized system using DPDK, batch processing, etc.

6.2.1. Throughput

Throughput is the most important performance index in this system. We test the system throughput in three modes by controlling the byte size of the packet (from 64 bytes to 1024 bytes). The result is shown in Figure 6. As the byte size of the packet increases, the throughput increases in the three modes. When the byte size reaches 1024 bytes, the baseline, S-VNFs, and VNFs (w/opt.) throughputs reach 1550 Mb/s, 1060 Mb/s, and 1250 Mb/s respectively. Compared to baseline, S-VNFs decreased about 31.3% more in terms of throughput. The throughput of optimized S-VNFs (w/opt.) is about 19.7% lower than the baseline. The results prove that the use of DPDK, batch processing, and shared memory can improve the throughput of the system to a certain extent. In addition, the throughput reduction caused by the multi-enclave-based system architecture proposed in this paper is within a certain acceptable range. The reason for this decline is mainly due to the fact that the SGX consumes more CPU and memory, as well as the transmission of packet addresses between multiple enclaves.

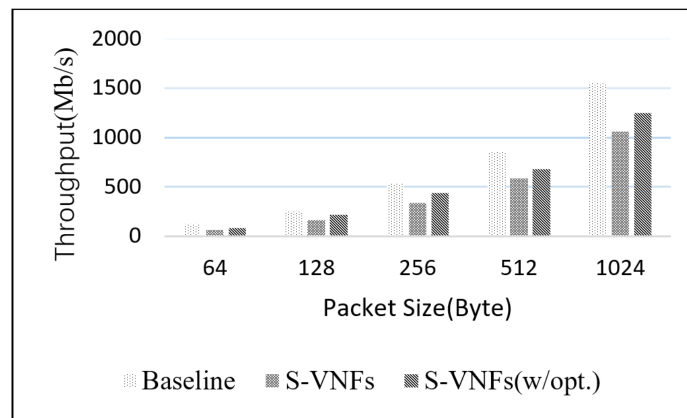

Figure 6. System throughput at different packet sizes.

6.2.2. Packet Processing Time

Secondly, we use Scapy to construct a variety of packet formats, including TCP, UDP, HTTP, etc. During the testing, the maximum number of packets sent from Scapy per batch in batch mode was 32. In three modes, we send packets of different byte sizes to calculate the average running time of the packet. The result is shown in Figure 7. In the three modes, the byte size of the packet has little effect on the average running time. In baseline mode, the average running time of the packet is approximately 3.88 µs. However, the average run time of S-VNFs is approximately 5.57 µs, and its run time is increased by 43%. The average run time of optimized S-VNFs (w/opt.) is approximately 4.63 µs, which is a 16% improvement over pre-optimization, due to the use of shared memory and batch processing which reduce enclave conversion. Compared with baseline, the optimized runtime increased by approximately 19%.

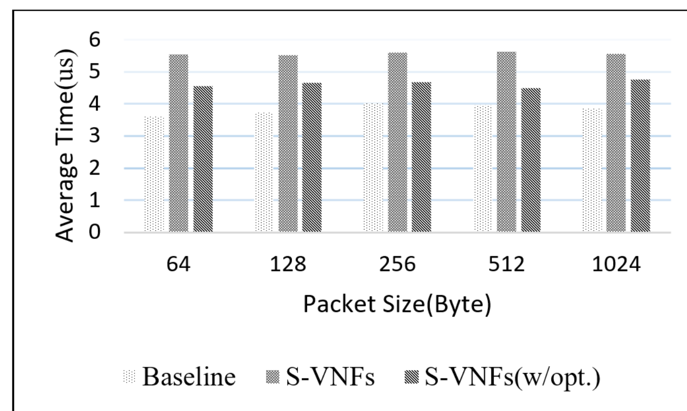

Figure 7. Average running time of the system at different packet sizes.

6.2.3. CPU Average Usage

Finally, we generate packets of different byte sizes at the highest rate and compare the average CPU usage from 64 bytes to 1024 bytes in baseline and S-VNFs (w/opt.) mode. The result is shown in Figure 8. Under different packet byte sizes, when the same mode is running, the CPU usage is not much different. In the baseline mode, the average CPU usage is 10.5%. In S-VNFs (w/opt.) mode, the average CPU usage of the system is 13.7%. In all, the average CPU usage increased by 30%, mainly due to the use of SGX technology.

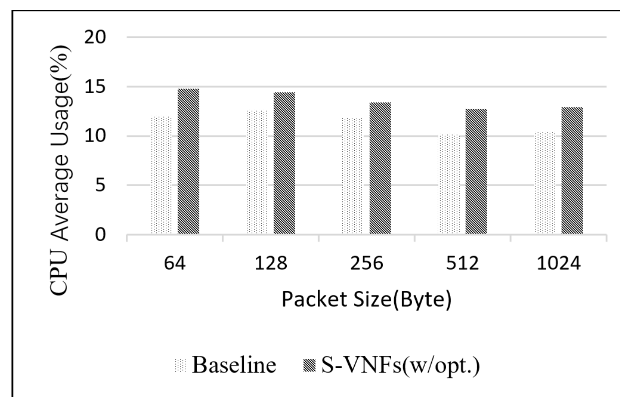

Figure 8. CPU usage of the system at different packet sizes.

6.2.4. Result Analysis

In conclusion, the performance of our system is affected by several factors, and the additional overhead is caused by SGX protection. Firstly, most of the overhead is caused by ecall and ocall instruction, which is the most influential factor in the performance of SGX, according to our test. When the number of enclaves is too large, the performance of the whole system is seriously affected because of entering and exiting enclaves. Secondly, the security check will also affect the performance of the system. It is necessary to use remote attestation and local attestation to ensure that the data will not be tampered with or stolen. As a result, a series of encryption and decryption operations are used and bring some extra overhead. Besides, the parameter transmission process also affects system performance. In our multi-enclave-based design, the result of packet classification needs to be passed to other enclaves. If the entire packet is transferred between multiple enclaves, the overhead is too large. Therefore, we have used shared memory to optimize packet transmission, and a packet pointer is designed to transfer the packet. Besides, there are three other performance optimizations for improving the performance of the entire system. By using the high-performance I/O library DPDK, the performance of I/O operations is improved. Moreover, by reducing the number of system calls, the use of ecall and ocall instruction can be reduced. Through batch processing, multiple small data packets can be processed uniformly, which increases the length of data packets processed at one time and improves efficiency.

6.3. Security Analysis

Putting VNFs into a single enclave that affects the scalability of NFVs, because users cannot scale in/out a single VNF. In our design, the sensitive modules of a VNF are put into different enclaves as elements of Click. The system can freely combine these security modules according to user requirements, and increase the scalability of the system while protecting its running state security.

The security of our system is protected by Intel SGX technology. The code, sensitive data, and system configuration of elements in VNF cannot be stolen or modified. In our experiments, various IDS security rules are put into enclaves for protection. Network packets are encrypted during the communication between multiple enclaves and are decrypted only inside enclaves. Therefore, the attackers can only capture the encrypted traffic, but cannot obtain the content of the packets, and cannot obtain and modify the IDS rules and detection results. The system is divided into two parts: the trusted part and the untrusted part. The trusted part stores IDS-related sensitive data, states, and policies in isolated and secure EPC memory. The code and data in the trusted part cannot be accessed by the operating system, drivers, BIOS, or VMM. The security of the system depends on the SGX hardware. Meanwhile, the untrusted part provides the necessary system call interfaces for the trusted part and the transmission of the packets between them.

In addition, the transmission of information between multiple enclaves is a very important security issue. In this architecture, multiple enclaves must first pass SGX's local

attestation to ensure that they each run on a trusted platform and establish a secure channel. Thereafter, multiple enclaves must pass this channel for information transmission, ensuring the security of information transmission.

7. Conclusions

In this paper, we propose a virtual security function architecture based on multiple enclaves to ensure the security and scalability of VNFs. In our system, the modules of VNF can be put into different enclaves and the communication can be done through local attestation to improve the scalability of the system while providing runtime security protection for VNFs. Furthermore, a new hot swapping mechanism is also presented to enable the system to be reconfigured at runtime without stopping the whole system. In order to reduce the overhead caused by SGX protection, we use DPDK to speed up data packet processing and present the methods of reduced system calls, batch and shared memory to improve system performance. Finally, an IDS based on our architecture is implemented and evaluated. The results show that the performance overhead is within an acceptable range while improving the scalability of VNFs protected by SGX.

To our knowledge, our work is the first attempt to put the elements of virtual network functions in different enclaves, to protect the security of key modules while supporting the scalability and composability of virtual network functions. In future work, we will design more virtual security functions based on our architecture and evaluate their performance. Moreover, we will optimize the method of hot-swapping and improve its security.

Author Contributions: Conceptualization, J.W.; methodology, J.W. and Y.Y.; software, Y.L. and C.F.; validation, Y.Y.; formal analysis, Y.Y.; investigation, Y.L.; resources, S.H.; data curation, Y.Y. and C.F.; writing—original draft preparation, Y.Y. and Y.L.; writing—review and editing, J.W., Y.Y., S.H.; visualization, Y.L.; supervision, J.W.; project administration, J.W.; funding acquisition, J.W. All authors have read and agreed to the published version of the manuscript.

Funding: 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).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Manager Code

```
class Manager : public Element {
public:
Manager() CLICK_COLD;
~Manager() CLICK_COLD;
const char *class_name() const { return "Manager"; }
const char *port_count() const { return PORTS_1_1; }
const char *flags() const { return "A"; }

int configure(Vector<String> &conf, ErrorHandler *errh) CLICK_COLD;

int initialize(ErrorHandler *) CLICK_COLD;
Packet *simple_action(Packet *);
static void* com_module(void*);
void ocall_to_device(void *packet, int packet_length);
static clock_t beginTime;
static clock_t frontTime;

private:
pthread_t com_thread;
```

```
static Communicate* com;
};

Manager::~Manager()
{
cout<<"there are "<<packet_number<<"packets"<<endl;
cout<<"average time in manager:"<<sum_time/packet_number<<endl;
printf("Manager::~Manager()\n");
if(sgx_destroy_enclave(eid) != SGX_SUCCESS)
{
printf("error: destroying enclave!\n");
}
}

int Manager::configure(Vector<String> &conf, ErrorHandler* errh)
{
printf("Manager::configure\n");
struct controller_ip c_ip;
Args(conf, this, errh)
.read("ip", c_ip.ip)
.read("port", c_ip.port)
.complete();

sgx_launch_token_t token = {0};
sgx_status_t ret = SGX_SUCCESS;
int update = 0;
ret = sgx_create_enclave(ENCLAVE_FILE, DEBUG, &token,
&update, &eid, NULL);
if(ret != SGX_SUCCESS)
{
printf("Manager::error: creating enclave!\n");
exit(Exit_FAILURE);
}
printf("Manager::ok: creating enclave!\n");

//create router
int generate_ret = 0;
ret = ecall_generat_router(eid, &generate_ret);
if(ret != SGX_SUCCESS || generate_ret != 1)
{
printf("Manager::error: generat_router!\n");
exit(Exit_FAILURE);
}
printf("Manager::ok: generat_router!\n");

int conf_size = 0;
char* conf_begin = NULL;
printf("Manager::start: ecall_configure!\n");
int configure_ret = 0;
ret = ecall_configure(eid, &configure_ret, conf_begin, conf_size);
if(ret != SGX_SUCCESS || configure_ret != 2)
{
printf("Manager::error: ecall_configure!\n");
exit(Exit_FAILURE);
```

```
}
printf("Manager::ok: ecall_configure!\n");
return 0;
}

int Manager::initialize(ErrorHandler* errh)
{
sgx_status_t ret = SGX_SUCCESS;
int initialize_ret = 0;
printf("Manager::initialize start!\n");
ret = ecall_initialize(eid, &initialize_ret);
if(ret != SGX_SUCCESS || initialize_ret != 3)
{
printf("Manager::error: enclave_initialize!\n");
}
}

Packet* Manager::simple_action(Packet* p)
{
sgx_status_t ret =SGX_SUCCESS;
int simple_ret = 0;
beginTime=clock();
frontTime=beginTime;
packet_number++;

ret = ecall_simple_action(eid, &simple_ret,(void *)const_cast
<unsigned char*>(p->data()),p->length());
if(ret != SGX_SUCCESS || simple_ret != 0)
{
printf("Error: ecall_simple_action!\n");
}
clock_t finish=clock();
sum_time += (finish-beginTime)*1000000/CLOCKS_PER_SEC;
return p;
}

CLICK_ENDDECLS
EXPORT_ELEMENT(Manager)
ELEMENT_REQUIRES(userlevel)
```

Appendix B. DNSAnalyzer Code

```
class DNSAnalyzer: public Analyzer { public:
~DNSAnalyzer();
const char *class_name() const { return "DNSAnalyzer"; }
virtual void push(int port, Packet *p);
};

void DNSAnalyzer::push(int port, Packet* p)
{
my_printf("dnsanalyzer::push\n");
(void)port;
const click_dns* dns = p->dns_header();
```

```
click_dns_info info;
info.dh_ancount = 0;
if(dns_parse_info((const unsigned char*)(dns+1),
p->end_data(),
dns, &info))
{
LOGE("DNS parse failed, packets may invalid!");
output(1).push(p);
}
if(DNS_TYPE_A==info.dns_type && DNS_CLASS_IN==info.dns_class)
{
uint32_t q_len;
if(info.qname) q_len = strlen(info.qname);
else q_len = 0;

event_t * event = alloc_event_data(2,
sizeof(uint32_t), q_len);
event->event_type = DNS_REQUEST;
event->fill_connect(p);
event_t::DataWriter writer = event->get_writer()(info.
    dns_record_ip);
if(info.qname)
writer(q_len, info.qname);

LOG_DEBUG("Save state: dns info %u", info.dns_record_ip);
LOG_DEBUG("Save state: dns qname %s", info.qname);
send_event(event, p->timestamp_anno());

dealloc_event(event);
p->kill();
} else {
output(1).push(p);
}
}

CLICK_ENDDECLS
EXPORT_ELEMENT(DNSAnalyzer)
ELEMENT_REQUIRES(userlevel)
```

Appendix C. FTPAnalyzer Code

```
class FTPAnalyzer: public Analyzer { public:
const char* class_name() const { return "FTPAnalyzer"; }
virtual void push(int, Packet*);
};

void FTPAnalyzer::push(int port, Packet* p)
{
my_printf("ftpanalyzer::push\n");

const char* payload = (const char*)p->transport_header() +
(p->tcp_header()->th_off << 2);
{
```

```
LOG("FTP_DOWNLOAD_ZIP");
event_t *event = alloc_event_data(0);
event->event_type = FTP_DOWNLOAD_ZIP;
event->fill_connect(p);
send_event(event, p->timestamp_anno());
dealloc_event(event);
p->kill();
}
}

CLICK_ENDDECLS
EXPORT_ELEMENT(FTPAnalyzer)
ELEMENT_REQUIRES(userlevel)
```

Appendix D. HTTPAnalyzer Code

```
class HTTPAnalyzer: public Analyzer { public:
const char *class_name() const { return "HTTPAnalyzer"; }
virtual void push(int port, Packet *p);
typedef stlpmtx_std::map<String, String> HttpHeaders;
static inline const unsigned char* _match_string(const unsigned
char* begin,
const unsigned char* limit,
const unsigned char* pattern,
size_t n);
static inline const unsigned char* _http_parse_version
(const unsigned char* data,
const unsigned char* end);
const unsigned char* _http_parse_status_line
(const unsigned char* data,
const unsigned char* end);
const unsigned char* _http_parse_method
(const unsigned char* data,
const unsigned char* end);
const unsigned char* _http_parse_request_line
(const unsigned char* data,
const unsigned char* end);
const unsigned char* _http_parse_header
(const unsigned char* data,
const unsigned char* end,
HttpHeaders *headers);
int http_parse(const unsigned char* ,
const unsigned char*, HttpHeaders*);
};

void HTTPAnalyzer::push(int port, Packet* p)
{
(void)port;
const unsigned char* payload = (const unsigned char*)p->
transport_header() + (p->tcp_header()->th_off << 2);
HttpHeaders headers;

if('M' == *payload && 'Z' == *(payload+1))
```

```
{
LOG_DEBUG("HTTP_RESPONSE_EXE");
event_t * event = alloc_event_data(0);
event->event_type = HTTP_RESPONSE_EXE;
event->fill_connect(p);
send_event(event, p->timestamp_anno());
dealloc_event(event);
p->kill();
}

if(0 == http_parse(payload, p->end_data(), &headers) &&
headers.size() > 0)
{
String cookie = headers.find("Cookie")->second;
String content_type = headers.find("Content-Type")->second;
if (content_type)
{
LOG_DEBUG("Content-Type: %s", content_type.c_str());
if(strncmp(content_type.c_str(), "text/html", 9) == 0)
{
event_t * event = alloc_event_data(0);
event->event_type = HTTP_RESPONSE_HTML;
event->fill_connect(p);
send_event(event, p->timestamp_anno());
dealloc_event(event);
p->kill();
}
else if(strncmp(content_type.c_str(), "application/octet-stream",
24) == 0 || strncmp(content_type.c_str(),
"application/x-msdos-program" , 27) == 0)
{
event_t * event = alloc_event_data(0);
event->event_type = HTTP_RESPONSE_EXE;
event->fill_connect(p);
send_event(event, p->timestamp_anno());
dealloc_event(event);
p->kill();
}
else if(strncmp(content_type.c_str(), "application/zip", 15)== 0)
{
event_t * event = alloc_event_data(0);
event->event_type = HTTP_RESPONSE_ZIP;
event->fill_connect(p);
send_event(event, p->timestamp_anno());
dealloc_event(event);
p->kill();
}
}
if(cookie)
{
String useragent = headers.find("User-Agent")->second;
if(!useragent)
useragent = String(click_random());
event_t * event = alloc_event_data(2, cookie.length(),
```

```
useragent.length());
event->event_type = HTTP_COOKIE_USERAGENT;
event->fill_connect(p);
event->get_writer()(cookie.length(), cookie.c_str())
(useragent.length(), useragent.c_str());
LOG_DEBUG("save cookie: %s", cookie.c_str());
LOG_DEBUG("save useragent: %s", useragent.c_str());
send_event(event, p->timestamp_anno());
dealloc_event(event);
}
}
p->kill();
}

CLICK_ENDDECLS
EXPORT_ELEMENT(HTTPAnalyzer)
ELEMENT_REQUIRES(userlevel)
```

References

1. Zhang, Q.; Liu, F.; Zeng, C. Adaptive Interference-Aware VNF Placement for Service-Customized 5G Network Slices. In Proceedings of the IEEE INFOCOM 2019-IEEE Conference on Computer Communications, Paris, France, 29 April–2 May 2019; pp. 2449–2457. [CrossRef]
2. Cui, C.; Deng, H.; Telekom, D.; Michel, U.; Damker, H. Network Functions Virtualisation. In Proceedings of the SDN and OpenFlow World Congress, Darmstadt, Germany, 22–24 October 2012.
3. Cotroneo, D.; De Simone, L.; Iannillo, A.K.; Lanzaro, A.; Natella, R.; Fan, J.; Ping, W. Network function virtualization: Challenges and directions for reliability assurance. In Proceedings of the 2014 IEEE International Symposium on Software Reliability Engineering Workshops, Naples, Italy, 3–6 November 2014; pp. 37–42. [CrossRef]
4. Han, B.; Gopalakrishnan, V.; Ji, L.; Lee, S. Network function virtualization: Challenges and opportunities for innovations. *IEEE Commun. Mag.* **2015**, *53*, 90–97. [CrossRef]
5. Wang, J.; Fan, C.; Cheng, Y.; Zhao, B.; Wei, T.; Fei, Y.; Zhang, H.; Ma, J. Analysis and research on SGX technology. *Ruan Jian Xue Bao/J. Softw.* **2018**, *29*, 2778–2798. [CrossRef]
6. Poddar, R.; Lan, C.; Popa, R.A.; Ratnasamy, S. Safebricks: Shielding network functions in the cloud. In Proceedings of the 15th (USENIX) Symposium on Networked Systems Design and Implementation (NSDI' 18), Renton, WA, USA, 9–11 April 2018; pp. 201–216.
7. Wang, Q.; Shou, G.; Liu, Y.; Hu, Y.; Guo, Z.; Chang, W. Implementation of Multipath Network Virtualization With SDN and NFV. *IEEE Access* **2018**, *6*, 32460–32470. [CrossRef]
8. Coughlin, M.; Keller, E.; Wustrow, E. Trusted click: overcoming security issues of NFV in the cloud. In Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, Scottsdale, AZ, USA, 22–24 March 2017; pp. 31–36. [CrossRef]
9. Shih, M.W.; Kumar, M.; Kim, T.; Gavrilovska, A. 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, New Orleans, LA, USA, 11 March 2016; pp. 45–48. [CrossRef]
10. Kohler, E.; Morris, R.; Chen, B.; Jannotti, J.; Kaashoek, M.F. The Click modular router. *ACM Trans. Comput. Syst. (TOCS)* **2000**, *18*, 263–297. [CrossRef]
11. Marku, E.; Biczok, G.; Boyd, C. Securing Outsourced VNFs: Challenges, State of the Art, and Future Directions. *IEEE Commun. Mag.* **2020**, *58*, 72–77. [CrossRef]
12. Morris, T. Trusted platform module. In *Encyclopedia of Cryptography and Security*; Springer: Boston, MA, USA, 2011; pp. 1332–1335. [CrossRef]
13. Perez, R.; Sailer, R.; van Doorn, L. vTPM: virtualizing the trusted platform module. In Proceedings of the 15th Conference on USENIX Security Symposium, Vancouver, BC, Canada, 31 July–4 August 2006; pp. 305–320.
14. Alippi, C.; Camplani, R.; Roveri, M.; Viscardi, G. Netbrick: A high-performance, low-power hardware platform for wireless and hybrid sensor networks. In Proceedings of the 2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012), Las Vegas, NV, USA, 8–11 October 2012; pp. 111–117. [CrossRef]
15. Lattner, C.; Adve, V. 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, IEEE Computer Society, Palo Alto, CA, USA, 21–24 March 2004; pp. 75–86. [CrossRef]

16. Zhang, W.; Liu, G.; Zhang, W.; Shah, N.; Lopreiato, P.; Todeschi, G.; Ramakrishnan, K.; Wood, T. OpenNetVM: A platform for high performance network service chains. In Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization, Florianópolis, Brazil, 26 August 2016; pp. 26–31. [CrossRef]
17. Wang, J.; Hao, S.; Li, Y.; Fan, C.; Wang, J.; Han, L.; Hong, Z.; Hu, H. Challenges Towards Protecting VNF With SGX. In Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, Tempe, AZ, USA, 21 March 2018; pp. 39–42. [CrossRef]
18. Anwer, B.; Benson, T.; Feamster, N.; Levin, D. Programming slick network functions. In Proceedings of the 1st ACM Sigcomm Symposium on Software Defined Networking Research, Santa Clara, CA, USA, 17–18 June 2015; pp. 1–13. [CrossRef]
19. Arnautov, S.; Trach, B.; Gregor, F.; Knauth, T.; Martin, A.; Priebe, C.; Lind, J.; Muthukumaran, D.; O'Keeffe, D.; Stillwell, M.L.; et al. SCONE: Secure Linux Containers with Intel SGX. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; pp. 689–703.
20. CORP, I. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. Available online: https://software.intel.com/sites/default/files/managed/57/0e/ww10-2016-sgx-provisioning-and-attestation-final.pdf (accessed on 6 January 2021).
21. Jain, P.; Desai, S.J.; Shih, M.W.; Kim, T.; Kim, S.M.; Lee, J.H.; Choi, C.; Shin, Y.; Kang, B.B.; Han, D. *OpenSGX: An Open Platform for SGX Research*. Available online: https://cysec.kr/publications/jain-opensgx.pdf (accessed on 6 January 2021). [CrossRef]
22. Weichbrodt, N.; Aublin, P.L.; Kapitza, R. sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In Proceedings of the 19th International Middleware Conference, Rennes, France, 10–14 December 2018; pp. 201–213. [CrossRef]
23. Bremler-Barr, A.; Harchol, Y.; Hay, D. OpenBox: a software-defined framework for developing, deploying, and managing network functions. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianópolis, Brazil, 22–26 August 2016; pp. 511–524. [CrossRef]
24. Kablan, M.; Caldwell, B.; Han, R.; Jamjoom, H.; Keller, E. Stateless network functions. In Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization, London, UK, 21 August 2015; pp. 49–54. [CrossRef]
25. Li, B.; Tan, K.; Luo, L.L.; Peng, Y.; Luo, R.; Xu, N.; Xiong, Y.; Cheng, P.; Chen, E. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianópolis, Brazil, 22–26 August 2016; pp. 1–14. [CrossRef]
26. Martins, J.; Ahmed, M.; Raiciu, C.; Olteanu, V.; Honda, M.; Bifulco, R.; Huici, F. ClickOS and the art of network function virtualization. In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), Seattle, WA, USA, 2–4 April 2014; pp. 459–473.
27. Li, H.; Hu, H.; Gu, G.; Ahn, G.J.; Zhang, F. 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, Toronto, ON, Canada, 15–19 October 2018; pp. 17–34. [CrossRef]
28. Siwek, J. The Zeek Network Security Monitor. Available online: https://www.zeek.org/ (accessed on 6 January 2021).
29. INTEL. Intel Data Plane Development Kit (DPDK). Available online: http://dpdk.org/ (accessed on 6 January 2021).
30. Cen, S.; Zhang, B. Trusted Time and Monotonic Counters with Intel Software Guard Extensions Platform Services. Available online: https://software.intel.com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services.pdf (accessed on 6 January 2021).
31. Trach, B.; Krohmer, A.; Gregor, F.; Arnautov, S.; Bhatotia, P.; Fetzer, C. ShieldBox: Secure middleboxes using shielded execution. In Proceedings of the Symposium on SDN Research, Los Angeles, CA, USA, 28–29 March 2018; pp. 1–14. [CrossRef]
32. Zhang, F.; Cecchetti, E.; Croman, K.; Juels, A.; Shi, E. Town crier: An authenticated data feed for smart contracts. In Proceedings of the 2016 ACM sIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 270–282. [CrossRef]
33. Chen, S.; Zhang, X.; Reiter, M.K.; Zhang, Y. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi, UAE, 2–6 April 2017; pp. 7–18. [CrossRef]
34. Biondi, P.; The Scapy Community. "Scapy". Available online: https://scapy.net/ (accessed on 6 January 2021).
35. Grodzki, T. Network Flight Simulator. Available online: https://github.com/alphasoc/flightsim/ (accessed on 6 January 2021).