Hindawi
Security and Communication Networks
Volume 2018, Article ID 8392080, 20 pages
https://doi.org/10.1155/2018/8392080

Research Article

A Compatible OpenFlow Platform for Enabling Security Enhancement in SDN

Haosu Cheng,[1,2] Jianwei Liu,[1,2] Jian Mao,[1,2] Mengmeng Wang,[1] Jie Chen,[1] and Jingdong Bian[1,2]

[1]*School of Cyber Science and Technology, Beihang University, Beijing, China*
[2]*School of Electronic and Information Engineering, Beihang University, Beijing, China*

Correspondence should be addressed to Jian Mao; maojian@buaa.edu.cn

Received 7 September 2018; Accepted 31 October 2018; Published 15 November 2018

Guest Editor: Chunqiang Hu

Software-defined networking (SDN) is a representative next generation network architecture, which allows network administrators to programmatically initialize, control, change, and manage network behavior dynamically via open interfaces. SDN is widely adopted in systems like 5G mobile networks and cyber-physical systems (CPS). However, SDN brings new security problems, e.g., controller hijacking, black-hole, and unauthorized data modification. Traditional firewall or IDS based solutions cannot fix these challenges. It is also undesirable to develop security mechanisms in such an ad hoc manner, which may cause security conflict during the deployment procedure. In this paper, we propose OSCO (Open Security-enhanced Compatible OpenFlow) platform, a unified, lightweight platform to enhance the security property and facilitate the security configuration and evaluation. The proposed platform supports highly configurable cryptographic algorithm modules, security protocols, flexible hardware extensions, and virtualized SDN networks. We prototyped our platform based on the Raspberry Pi Single Board Computer (SBC) hardware and presented a case study for switch port security enhancement. We systematically evaluated critical security modules, which include 4 hash functions, 8 stream/block ciphers, 4 public-key cryptosystems, and key exchange protocols. The experiment results show that our platform performs those security modules and SDN network functions with relatively low computational (extra 2.5% system overhead when performing AES-256 and SHA-256 functions) and networking performance overheads (73.7 Mb/s TCP and 81.2Mb/s UDP transmission speeds in 100Mb/s network settings).

1. Introduction

Recent years, cyber-physical systems are widely adopted in areas such as personal healthcare, emergency response, traffic flow control, and electric power management. CPS uses emerging technologies such as 5G and Edge computing as its building components. As one of the core technologies in 5G network [1] and Edge computing [2], software-defined-networking (SDN) is applied as a basic networking infrastructure in the cyber-physical system (CPS), which is a collection of interconnected computing devices interacting with the physical world with its users. Being a representative next generation networking technique, SDN decouples the data plane from the control plane and supports programmatically initialization and dynamic network behavior management. Its good performance in flexibility and scalability brings advances in CPS deployment and configuration.

Figure 1 shows the classic SDN architecture which includes three virtual layers, *control plane*, *data plane*, and *application plane* [3, 4]. The application plane invokes software-based logic in the control plane via REST-APIs (Representational State Transfer Application Programming Interface), which is also called *Northbound Interface*. The deployed logic decisions are executed by the data plane through *Southbound Interfaces*, e.g., OpenFlow, a widely adopted implementation architecture of SDN. The network control traffic is transferred from the infrastructure (data plane) to the controller (control plane). With the help of SDN apps, network operators achieve distinguished properties of network control, automation, and resource optimization.

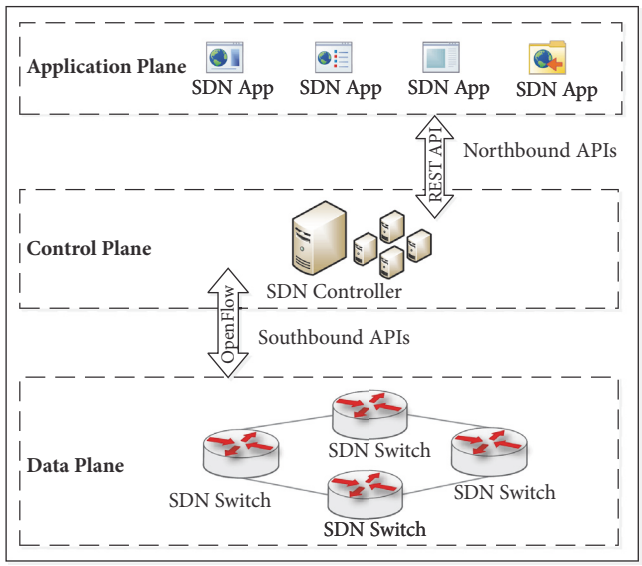

Figure 1: The three planes in SDN architecture.

Table 1: Comparison of SDN attack types.

Attack Type	Targeted SDN Layer
DDOS	Control, Data
DOS	Control, Data
Hijacked Controller	Control, Data, App
Malicious Applications	App
Man-in-the-middle	Control, Data, Control-Data link
Black-hole	Control, Data, Control-Data link
Eavesdropping	Control, Data, App

Although SDN provides a new solution for the network architecture, it exposes many security problems [5]. Since a centralized controller is responsible for managing the entire network, any security compromises of the controller may cause the whole network to crash. Furthermore, security lapses in controller to data plane communication may lead to illegitimate access to network resources. SDN applications enable the users to access network resources, deploy new rules, and manipulate the behavior of the network through the interaction with the control plane. Meanwhile, securing the network from malicious applications or abnormal behavior of applications is a serious security challenge in SDN [6].

Table 1 shows different attacks and the affected layers in the SDN architecture. The security vulnerabilities in SDNs are mainly distributed in application plane, control plane, and data plane. For instance, communication channels between isolated planes might be attacked by other compromised planes. Because of the visible nature of the control plane, it is more attractive to attacker and vulnerable to DoS and DDoS attacks [7]. The lack of authentication and authorization mechanisms is another security limitation in the application plane. The data plane is vulnerable to fraudulent flow rules, flow rule flooding, controller hijacking, and man-in-the-middle attacks.

Many security solutions, targeted at the exposed security vulnerabilities of SDN, are proposed. FRESCO [8] is proposed to enable development of OpenFlow security applications. The control plane security solutions consist of defending malicious applications, optimizing load balancing policies, DoS/DDoS attack mitigation, and reliable controller placement. In order to provide an ideally secure SDN control layer, Security-Enhanced (SE) Floodlight [9] adds a secure programmable northbound API to operate as an intercessor between applications and the data plane. The fine-grained security enforcement mechanisms such as authentication and authorization are used for applications that can change the flow rules. FortNOX [10] is a platform that enables the NOX controller to check flow rule contradictions real-time and authorize applications before they can change the flow rules.

However, rather than focusing on specific types of attacks or planes, it is desirable to develop and deploy security countermeasures in a more general manner to guarantee data and network resource security. Besides, how to systematically test and verify the deployed security mechanisms is another challenge. The current hardware deployment or software simulation are mostly focused on network function implementation and verification. None of them target the security perspective.

In addition, SDN switch hardware is too expensive to deploy massively and it is also difficult to modify the embedded protocols. Furthermore, commercial SDN switches cannot support customized protocols and interfaces. The software simulation (e.g., Mininet [11], NS3 [12]) is able to support large size SDN network topology but it lacks various hardware interfaces to enable hardware-based security schemes, e.g., TPM, special encryption chip, and random number generator.

To address the above problems, a lightweight security development platform—that supports security countermeasure implementation, deployment, and testing in a uniform, extensible, and economical mode—is desired. In this paper, we propose OSCO (Open Security-enhanced Compatible OpenFlow) platform, a uniform testing platform based on Raspberry Pi Single Board Computer (SBC) hardware and SDN network architecture, which supports highly configurable cryptographic algorithm modules, security protocols, flexible hardware extensions, and virtualized SDN networks. (An earlier version of this manuscript was presented in International Conference on Wireless Algorithms, Systems, and Applications (WASA), 2018 [13].) We systematically evaluate critical security modules, which include hash functions, stream/block ciphers, public-key cryptosystems, and key exchange protocols. Furthermore, we verify the scalability of the platform and the corresponding system overhead.

Contributions. In summary, we make the following contributions in this paper:

(i) We proposed an extensible security-oriented SDN network experiment platform, OSCO, allowing various security design schemes to experiment in SDN environment. Our scheme provides an open framework with flexible hardware interfaces and extensible

software modules that fully support standard OpenFlow protocol and its security enhancement solutions.

(ii) We implemented the OSCO platform in a Raspberry Pi hardware, which is acting as the OpenFlow switch. It supports physical SDN network deployment and simulation. In addition, we present a case study for switch port security enhancement to illustrate the extensibility and usability of OSCO platform functions with relatively low computational (extra 2.5% system overhead when performing AES-256 and SHA-256 functions) and networking performance overheads (73.7 Mb/s TCP and 81.2Mb/s UDP transmission speeds in 100Mb/s network settings).

Paper Organization. The rest of this paper is organized as follows: Section 2 presents the overall architecture and critical components of our platform with implementation details; Section 3 describes the case study for port security enhancement; Section 4 evaluates performance of some critical security modules; Section 5 discusses related work; and Section 6 concludes the paper.

2. Overall Architecture of OSCO Platform

In this section, we present the overall architecture of OSCO platform and describe its core functions with auxiliary components in detail.

2.1. Security Threats in SDN. As we discussed previously, the security threats of SDN exist in the application layer, control layer, and data layer, respectively, as well as the control-data-link and application-control-link interfaces. The application plane enables SDN applications to manipulate the behavior of SDN devices through the SDN controller. The variety of SDN applications allows the SDN network to be managed in the flexible and diverse ways via the centralized SDN controller. Because most of the network functions can be implemented as SDN applications, malicious applications can spread devastation in the whole network as well, specifically, attacks on authentication, access control mechanisms [6, 14–16], etc.

The control plane holds the abstract view of the entire network and provides the information of network resources to SDN applications. The control plane is implemented as a logically centralized module which separates the SDN application and data plane. Concrete implementation of control plane is the Network Operation System (NOS), which collects network information via APIs to observe and control network. NOS provides a uniform and centralized programmatic interface to the entire network. For instance, the controller is responsible for flow formation, management, and distribution in data-path elements via OpenFlow protocol between control and data plane. As the control plane is a centralized control entity, it is an attractive target to attackers. The main security challenges and threats existing in the control plane are unauthorized application access, e.g., DoS/DDoS attacks.

The data plane holds the SDN forwarding devices such as routers, switches, virtual switches, and access points and manages configurable device entity using OpenFlow protocol to control plane. The SDN device can be (re)configured for different purposes, including traffic isolation, data-path modification, and device virtualization via a remote procedure call from the SDN controller using an optional secure communication channel in data-control link. The flow rules are installed in the OpenFlow switch's flow tables according to the controller's decisions. The security challenges of data plane are compromising SDN controller and recognizing genuine flow rules and distinguishing them from bogus rules. And it is also vulnerable to saturation attacks, man-in-the-middle attack, and TCP-Level attacks.

2.2. Objectives. The goal of our proposed OSCO platform is to provide a uniform, open, extensible, and economical environment for security countermeasure implementation, deployment, and test in different SDN planes. In the application plane, OSCO is able to support deployment of SDN applications and its corresponding security countermeasures for authentication, authorization and access control, and accountability security risks. In the control plane, OSCO is capable of supporting various security-enhanced controllers via a standard interface to prevent unauthorized application access, DoS/DDoS attacks, etc. In the data plane, it functions as an OpenFlow compatible switch that forwards packages between switches and hosts, and plays an important role in the security countermeasure for saturation attacks, man-in-the-middle attacks, and TCP-Level attacks. OSCO is a testing environment for security schemes in all the three SDN planes.

2.3. Architecture of OSCO Platform. Figure 2 shows the overall architecture of the OSCO platform, which consists of central part enclosed by the dashed rectangle and peripheral system. The central part consists of OSCO core function modules. The peripheral system consists of the REST-API formatted SDN security applications, diversified peripheral security hardware, SDN controllers, and the security-supported network simulator. The OSCO central part functions as an OpenFlow switch by using embedded OpenFlow protocol under the SDN architecture. The SDN controller communicates with applications and OSCO core. The OSCO is able to extend the security function through the peripheral security hardware. The SDN network simulator is available for OSCO platform to simulate a large scale complex SDN network topology with an internal or external SDN controller.

(a) OSCO Core Function Modules: The central part of the OSCO platform includes core function modules, such as the *hardware interface*, the *Linux kernel*, the *OpenFlow module*, the *cipher algorithm library*, and *protocol stack*. The core function modules act as a multifunction OpenFlow switch. The security hardware is supported by the hardware (HW) interfaces, such as the secure chip, random number generators, biological information acquisition, and recognition devices. The security hardware provides better encryption or decryption performance than software implementation. The OpenFlow module is implemented above the Linux kernel, which is one of the most important protocols in an SDN

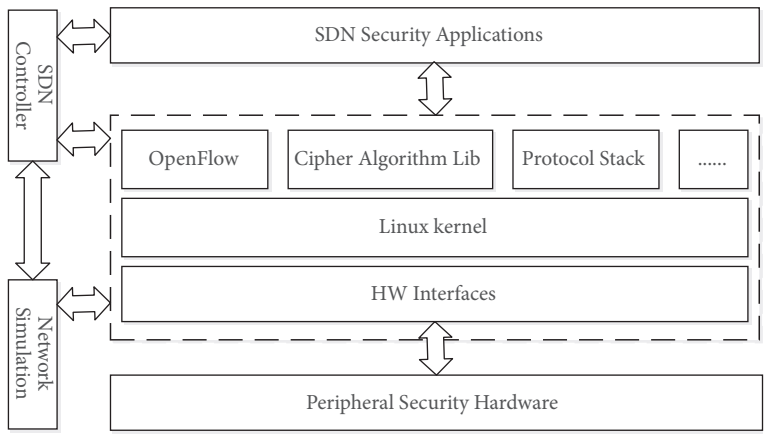

Figure 2: Overall architecture of OSCO platform.

architecture. OpenFlow is used for the interaction between a data plane constituted network switch and a control plane constituted controller. The cipher algorithm library contains the mainstream cryptographic algorithms, e.g., hash function, stream ciphers, and public-key cryptosystems. It provides rich algorithms and the latest implementation from different libraries. The key management, key distribution, and authentication protocols are loaded in the protocol stack module. In addition, other extensible modules such as package capture and package analysis modules can be employed in OSCO platform.

(b) Peripheral System: The peripheral system in OSCO platform supports the core function modules to fulfill their designed functionalities. The peripheral systems include four parts: (1) the SDN security application module, which implements and invokes the security solutions; (2) the peripheral security hardware module which provides hardware-based security solutions; (3) the network simulation module, which enables the simulation of the complex network topology and SDN network functions; (4) the SDN controller module, which makes packages forwarding decisions, maintains the SDN network topology, coordinates network resources, etc.

2.4. Implementation. (1) System Hardware: We select the Single Board Computer (SBC) hardware Raspberry Pi3 model B [17] as computation and HW-interface platform used in core modules implementation. The Raspberry Pi is an open source hardware with a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor and 1 GB of RAM. It supports 10/100 Mbit/s Ethernet, 802.11n wireless, Bluetooth 4.1 On-board network, 4 USB2.0 ports, and 17 general purpose input-output (GPIO) connectors including I2C, SPI, UART, PCM, and PWM interfaces. We set up three external USB Ethernet adapters (10/100Mb/s) as OSCO physical ports. The Ubuntu MATE [18], which is a Linux-based implementation, is chosen as platform operation system (OS). The rest of the core functions modules are deployed above the Linux level. The Open Virtual Switch (OVS) [19] is installed as the implementation of OpenFlow protocol. OpenSSL [20], Crypto++ [21], and PBC [22] libraries are integrated as the cryptology algorithm

Figure 3: The hardware environment of OSCO platform.

module in the platform. Additionally, we use the Wireshark [23] as OpenFlow packet inspector and monitor. Figure 3 presents the hardware environment of our OSCO platform.

(2) Network Configuration: We implemented the POX/NOX [24] and Floodlight [9] SDN controllers in our platform, and a Floodlight controller based topology is shown in Figure 4. We chose Mininet [11] to create/simulate the SDN virtual network infrastructure. We discussed the implementation details of security enhancement based on OSCO platform by using a case study in Section 3.

3. Case Study: Port Security Enhancement on OSCO

3.1. System Configuration. As shown in Figure 4, in our case study, the experiment system consists of two parts, a virtual network and a physical network. We use Mininet to simulate a virtual SDN network with two switches and two hosts on the workstation (Intel i7-4770 CPU, 8G RAM, Ubuntu 16.04 64-bit OS). The host1 connects to the switch1, the host2 connects to the switch2, and the switch1 connects to the switch2. The Mininet configuration allows switch2 to connect to the controller (installing on the workstation with Intel i5-4570 CPU, 4G RAM, Ubuntu 16.04 64-bit OS) via a physical OSCO

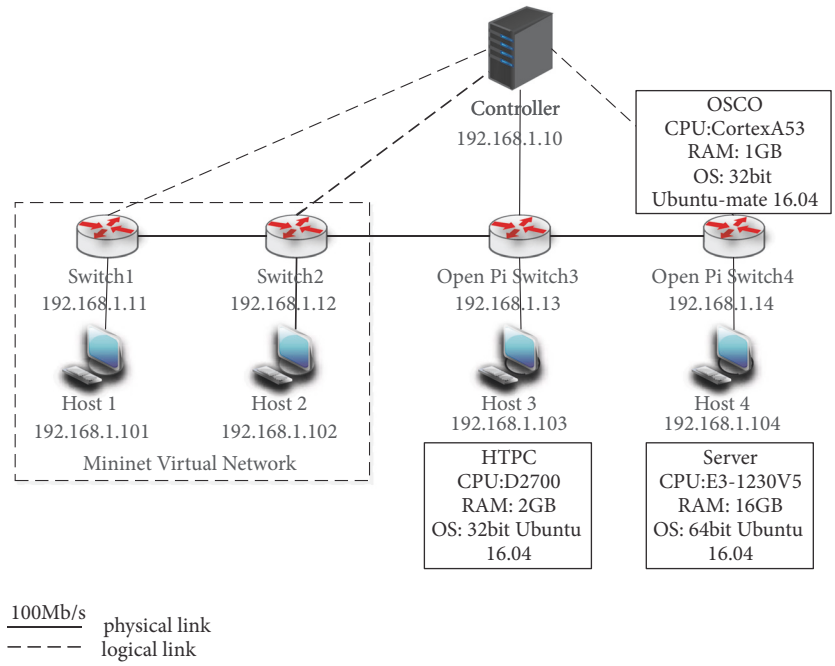

Figure 4: The system configuration.

switch (Cortex-A53 CPU, 1G RAM, Ubuntu MATE 16.04 32-bit OS).

The OSCO switch3 physically connects to OSCO switch4, the virtual switch2, and a controller. The host3 connects to the OSCO switch3, and the host4 connects to the OSCO switch4. The host3 is a HTPC with Intel Atom D2700 CPU, 2G RAM, Ubuntu 16.04 32-bit OS and the host4 is a server with Intel E3-1230v5 CPU, 16G RAM, Ubuntu 16.04 64-bit OS. The OSCO switch includes one original Ethernet port from Raspberry Pi and three USB Ethernet ports. The unified physical Ethernet interfaces are ensured by the core function modules of the platform.

OSCO switch runs OVS to support OpenFlow protocol and logical and physical ports mapping for OpenFlow connection. All switches connect (physically or logically) the controller via the OSCO switch3 and switches work in OpenFlow-only mode with OpenFlow 1.3 protocol.

3.2. Port Security Enhancement. The OVS software module is concrete standard implementation of OpenFlow protocol. It maps the physical port to the OpenFlow logical forwarding port and supports Spanning Tree Protocol to prevent broadcast storm in the loop network. For instance, in the implementation topology shown in Figure 4, the OVS port os1-eth1 in OSCO switch3 is used to connect os2-eth0 port in OSCO switch4 by connecting two physical ports with setting OVS bridge name and port mapping at both sides. The openness of the OVS makes it easier to extend the OpenFlow protocol. The OpenSSL library is used to add OSCO switch port encryption/decryption function by modifying OVS software module.

OpenFlow protocol processes packets through flow tables and actions in the OpenFlow pipeline. OpenFlow port receives and sends data to controller or switches. The flow tables hold the package forwarding rules and the actions apply the rules that include flow modification, deletion, addition, and forwarding. The process of the OpenFlow packet receiving and forwarding is conducted in the OpenFlow pipeline.

In this case study, one of the requirements is to enable OSCO switch to execute encryption/decryption operations in a specific port. To achieve this, the source code of OpenFlow pipeline part needs to be modified to enable os1-eth1 and os2-eth0 ports to perform packages encryption/decryption actions between them. The datapath.c file located in the OVS project data-path directory holds the implementation of OpenFlow pipeline. It contains the most important function calls and processing logic for the package forwarding. Pseudocode 1 presents the OVS original code and the key part of modified code for the implementation of port encryption.

Pseudocode 2 list is developed for package encryption, which is put in line 27 before the dp_xmit_skb function in original code.

The dp_output_put function processes the flow packages and sends them to the corresponding ports, which include the flood port, table port, controller port, and physical port. Only the os1-eth1 and os2-eth0 port encrypt or decrypt the packages. The check_port function checks the output port before a series of encryption efforts. The package needs to attach its hash value for the later integrity verification. The do_hash function calculates the package hash value by using SHA-256 implementation in OpenSSL library. The add_md_to_tail function adds the calculated hash value to the end of the package. To facilitate the implementation and evaluation, the encryption key and the initialization value are hard code inside the do_encrypt_data function.

```
1 int dp_output_port
2   (struct datapath *dp, struct sk_buff *skb, int out_port, int ignore_no_fwd)
3 {
4       BUG_ON(!skb);
5       switch (out_port){
6       ...
7       ...
8       case 0 ... DP_MAX_PORTS - 1: {
9          struct net_bridge_port *p = dp->ports[out_port];
10          if (p == NULL)
11             goto bad_port;
12          if (p->dev == skb->dev) {
13               /* To send to the input port, must use OFPP_IN_PORT */
14             kfree_skb(skb);
15             if (net_ratelimit())
16             printk(KERN_NOTICE "%s: can't directly "
17                 "forward to input port\n",
18                 dp->netdev->name);
19           return -EINVAL;
20           }
21          if (p->config & OFPPC_NO_FWD && !ignore_no_fwd) {
22            kfree_skb(skb);
23            return 0;
24           }
25           skb->dev = p->dev;
26
27           return dp_xmit_skb(skb);
28                    }
29
30        default:
31           goto bad_port;
32       }
33       ...
34       ...
35 }
```

Pseudocode 1:

```
1 if (chech_port (out_port,"os1-eth1","os2-eth0")    {
2             int32_t p_size = get_ksb_package_size (skb);
3             unsigned char package [p_size] = get_package_data (skb);
4
5             unsigned char md[33] = {0};
6             do_hash (package, p_size, md); // do SHA256
7             add_md_to_tail (md, skb); // to add hash value
8
9             int32_t size = get_data_size (skb);
10            unsigned char data [size] = get_data (skb); // get data from skb
11
12            int32_t padding_size =0;
13            encrypt_buff = (unsigned char *)malloc (padding_size);
14            do_encrypt_data (data, &encrypt_buff, padding_size);
15            put_encypted_data (encrypt_buff, skb);
16            free (encrypt_buff);
17 }
```

Pseudocode 2:

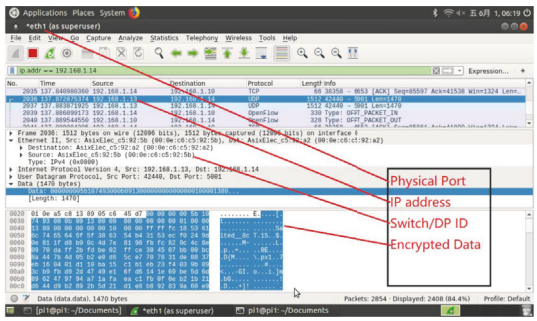

Figure 5: Package capturing in port os1-eth1.

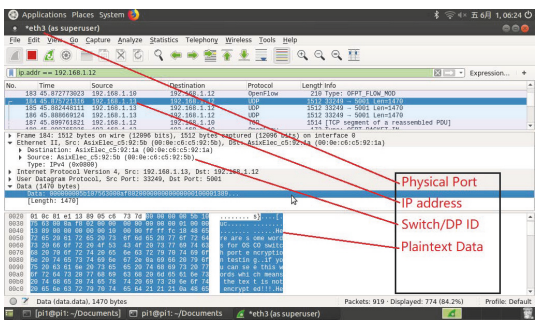

Figure 6: Package capturing in port os1-eth3.

The do_encrypt_data function encrypts data via invoking OpenSSL library to implement AES-CBC algorithm with 256-bits key size. Finally, the new package with encrypted data is sent to output port buffer via the dp_xmit_skb function.

Pseudocode 3 indicates the do_port_input function that handles the package receiving process in the OVS original code. The decryption action should be executed at line 10 before fwd_port_input function. The decryption action code is shown in Pseudocode 4.

The do_port_input function is responsible for retrieving the packages from input port in OpenFlow pipeline. The encrypted packages need to be decrypted before any further processing. It needs to check the input port before performing decryption action, because it only encrypts package that goes through os1-eth1 or os2-eth0 port. The get_data function gets a package from a network device buffer, and the do_decrypt_data function decrypts package with hard coded encryption/decryption key and initialized value. Next, it needs to calculate a hash value of decrypted package via calling do_hash function. The check_not_the_same_md function verifies the integrity of decrypted package by comparing two hash values. If two hash values are equal, it will call fwd_port_input function for the further package processing; otherwise it drops this package. The port encryption/decryption function is only available between the connection of OSCO switch3 and switch4 in our case. By doing this, the different block/stream encryption module can be easily tested and verified in OSCO platform.

Figures 5 and 6 represent the packages capturing on os1-eth1 and os1-eth3 ports in OSCO switch3. They show packages that contain some text data going through the physical port os1-eth1 and os1-eth3 in OSCO switch3, respectively.

Figure 5 indicates that the encrypted data is sent from OSCO switch3 to OSCO switch4 via port os1-eth1. 192.168.1.13 and 00:0e:c6:c5:92:5b are the IP address and the switch logical MAC address (it is also the ID of OpenFlow Data-Path) for OSCO switch3. The IP (192.168.1.14) and MAC (00:0e:c6:c5:92:a2) address are used by OSCO switch4.

As shown in Figure 6, the nonencrypted data (plaintext) are sent out from OSCO switch3 via port os1-eth3. The package will not be encrypted when using other port rather than the port os1-eth1.

All ports of OSCO switch3 are listed by SDN controller in Figure 7. It shows OSCO switch3's MAC address, IP address, OpenFlow version, and five registered ports in SDN controller. There are one OpenFlow logical port (local:os1) and four physical ports, but only one port (os1-eth1) allows encryption function. In addition, OpenFlow local port contains all physical ports, and its MAC address is the switch/OpenFlow Data-Path ID.

In this section, we present a case study for OSCO switch port security enhancement and illustrate the details to implement the port encryption/decryption function in OSCO switch. We evaluate the system and network performances in Section 4.

4. Performance Evaluation

In this section, we evaluate the OSCO platform in two aspects, computational overhead caused by the cryptographic security modules and its network performance. Table 2 indicates the hardware and software configuration of cryptographic algorithms performance testing.

We adopted the benchmark in OpenSSL project for the cryptographic algorithms performance testing, which is the most widely used in cryptographic evaluation. Besides Raspberry Pi-based OSCO platform, we also conduct the performance evaluation on two other hardware platforms deployed in our experiment (as shown in Figure 4). The Home Theatre PC (HTPC) has a duel core Intel Atom D2700, 2GB of RAM and running 32bit Ubuntu 16.04 OS. The Server platform has a quad-core Intel Xeon E3-1230 V5 processor with 16GB RAM and 64bit Ubuntu 16.04 OS installed. The experiment results disclose the hardware preference for different crypto settings. It could be a reference for SDN hardware selection according to the network and system performance requirements.

We mainly conducted the performance evaluation on four kinds of cryptographic operations, hash functions, symmetric encryption, public-key encryption, and digital signature, which are the basic primes of most SDN security solutions.

4.1. Hash Algorithm Performance. Hash function is used to ensure the integrity of transmitted data. We test performances of five widely used hash functions on three different hardware platforms. Table 3 shows the hash function parameters such

```
1 static void do_port_input (struct net_bridge_port *p, struct sk_buff *skb)
2 {
3         skb = skb_share_check (skb, GFP_ATOMIC);
4         if (!skb)
5             return;
6         /* Push the Ethernet header back on. */
7         skb_push (skb, ETH_HLEN);
8         skb_reset_mac_header (skb);
9
10        fwd_port_input (p->dp->chain, skb, p);
11 }
```

Pseudocode 3:

```
1 if (chech_bridge_ports (p,"os1-eth1","os2-eth0")    {
2
3             int32_t size = get_data_size (skb);
4             unsigned char data [size] = get_data (skb);
5             // get data from skb
6             int32_t padding_size =0;
7
8             decrypt_buff = (unsigned char *)malloc (padding_size);
9             do_decrypt_data (data, &decrypt_buff, padding_size);
10            // do AES CBC 256 decryption
11            put_decypted_data (decrypt_buff, skb);
12            free (decrypt_buff);
13
14            int32_t p_size = get_ksb_package_size (skb);
15            unsigned char package [p_size] = get_package_data (skb);
16
17            unsigned char md[33] = {0};
18            do_hash (package, p_size, md); // do SHA256
19            unsigned char oldmd = get_hash_value (skb, 256);
20
21            if (check_not_the_same_md (md, oldmd)) {
22                kfree_skb (skb);
23                return;
24            }
25 }
```

Pseudocode 4:

as output message size, maximum input message size, input processing block size, basic word size, and the number of processing steps.

Figure 8 shows the testing results of Message-Digest Algorithm (MD5) [25], Keyed-Hashing for Message Authentication (HMAC) [26], Secure Hash Algorithm (SHA) [27], and SHA-256 and SHA512 hash functions, which are executed by single thread with 16-byte input size in three platforms. The x-axis indicates the amount of processed data per second. The y-axis labels the five hash functions. The performance metric is defined as the parameter p_1, $p_1 = d_k/t$, where d_k denotes the size of data being processed with input buffer size k and t denotes the processing time.

For instance, we run MD5 hash routine in a loop for 3 seconds with a 16-byte input in OSCO platform. It performs over 1.7 million iterations after 3 seconds, that is, about 26.7 million bytes processed. Because of the use of Xeon E3 processor, the server has the best performance in three platforms; especially SHA1 has the best score in all five hash functions. The rest of the results also indicate that SHA1 is designed to perform better than MD5 when handling small input data size (16 bytes). The Server platform is the only one which runs 64-bit implementation of OpenSSL. Because of less number of steps, SHA-256 is the fastest hash function in both HTPC and OSCO platforms when handling 16-byte input data in 32-bit OS. HMAC generates a message authentication code by performing an embedded hash function. Its performance is close to embedded hash function (MD5 in our case). Figures 9, 10, and 11 show the hash function performances with different input data sizes

Table 2: Testing environment of cryptographic performance.

Platform	OSCO	Server	HTPC
CPU	Quad-core ARM Cortex-A53	Quad-core Intel Xeon E3-1230 V5	Quel core Intel Atom D2700
RAM	1GB	16GB	2GB
OS	32Bit Ubuntu-mate 16.04	64Bit Ubuntu 16.06	32Bit Ubuntu 16.04
Benchmark	OpenSSL 1.0.2g	OpenSSL 1.0.2g	OpenSSL 1.0.2g

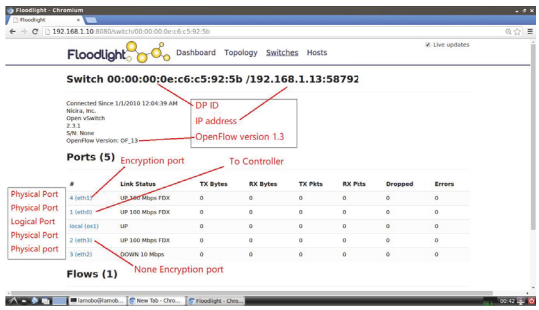

Figure 7: OSCO switch3 ports in SDN controller.

Table 3: Hash functions parameters.

Algorithm	MD5	SHA1	SHA256	SHA512
Message digest size (bits)	128	160	256	512
Message size (bits)	unlimited	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$
Block size (bits)	512	512	512	1024
Word size (bits)	32	32	32	64
Number of steps	80	80	64	80

on the three platforms, respectively. The x-axis indicates the different size of input data. The y-axis shows the amount of processed data per second (same as Figures 10, 11, 13, 15, and 14). According to Figure 9, on the OSCO platform, the performance is influenced by the input size, that is, the bigger the input size, the better the performance.

Furthermore, MD5 is somewhat less CPU-intensive than SHA1 in 32-bit OS [28]. The result shows MD5 getting better performance with input data larger than 64 Bytes (512 Bits) in Figures 9 and 10, because the additional block size padding operations reduce when input data size is larger than block size (it means larger than 512 bits). The similar result (large input size with better results) is presented in Figure 11, which is acquired from the Server platform. The 64-bit SHA-512 implementation is the only function among those four which benefits from a Streaming SIMD Extensions 2 (SSE2, Intel processor supplementary instruction sets) implementation [28]. Thus, it explains SHA512 performing better than SHA-256 in Server platform.

4.2. Stream/Block Ciphers Performance. Stream/block ciphers operate as important elementary components in many cryptographic protocols and are widely used to implement encryption of bulk data, for instance, RC4 [29], Blowfish [30], International Data Encryption Algorithm (IDEA) [31], Data Encryption Standard (DES) [32], and Advanced Encryption Standard (AES) [33] stream/block ciphers algorithms.

In this section, we evaluated the implementation performances of Blowfish, IDEA, DES, Triple DES (3DES), and AES in the Cipher Block Chaining (CBC) mode. Table 4 shows the block cipher parameters such as block size, key size, and round of iterations.

Figure 12 indicates that the RC4 is the fastest algorithm of all five algorithms in three platforms with single thread and 16-byte input size. RC4 is stream cipher; it simply uses bitwise exclusive-OR (XOR) operation between pseudorandom key-stream and plaintext stream for encryption. The decryption requires the use of the same key-stream and XOR operation. The RC4 only needs the hardware to perform XOR operation and to generate a pseudorandom key-stream. The simplest operations and less system hardware overhead guarantee the highest execution efficiency. The x-axis indicates the amount of processed data per second. The y-axis shows the type of algorithms.

DES is the deprecated data encryption standard. Its key size is 56 bits long, which is too short for proper security. 3DES is a security-enhanced version of DES. The 3DES module uses Encryption-Decryption-Encryption mode implementation, which applies two keys (k_1, k_2, 56-bits for each key) for each data block. 3DES provides a relatively simple method of increasing the key size of DES to protect against brute-force attack, but it increases the system overhead and reduces the speed of encryption and decryption. 3DES is the slowest algorithm of all cryptographic modules on all three platforms.

The DES has been superseded by the AES. AES supports 128, 192, or 256-bit key size with 128-bit block size. Recently, new processor hardware has been optimized for AES and SHA algorithms. For example, Intel's AES-NI and ARM's ARMv8-A instruction sets can provide faster hardware acceleration when using AES encryption. Intel XEON E3-1230 V5 and ARM Cortex-A53 processors support AES-NI and ARMv8-A instruction set, respectively. This explains why AES has better performance than DES in both server and OSCO platforms as shown in Figures 13 and 14.

Blowfish and IDEA are two commonly used symmetric-key block cipher algorithms with 64-bit data block size and 128-bit key size in our implementation. Blowfish has a variable key length from 32 bits up to 448 bits and employs 16-round interactions with key-dependent S-boxes. Blowfish provides a good encryption rate in software implementation on three platforms. IDEA consists of a series of 8 identical transformations and an output transformation, and the processes for encryption and decryption are similar. Blowfish

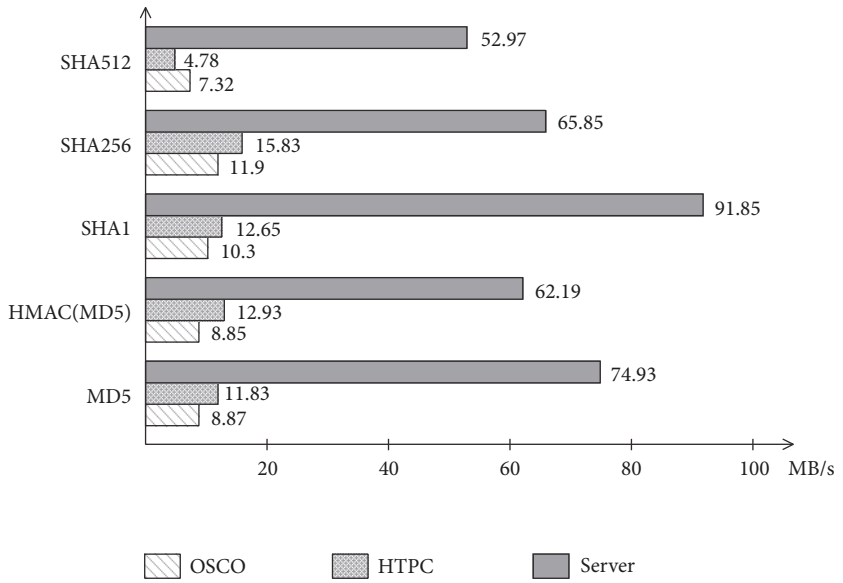

Figure 8: Hash functions performance in the three platforms.

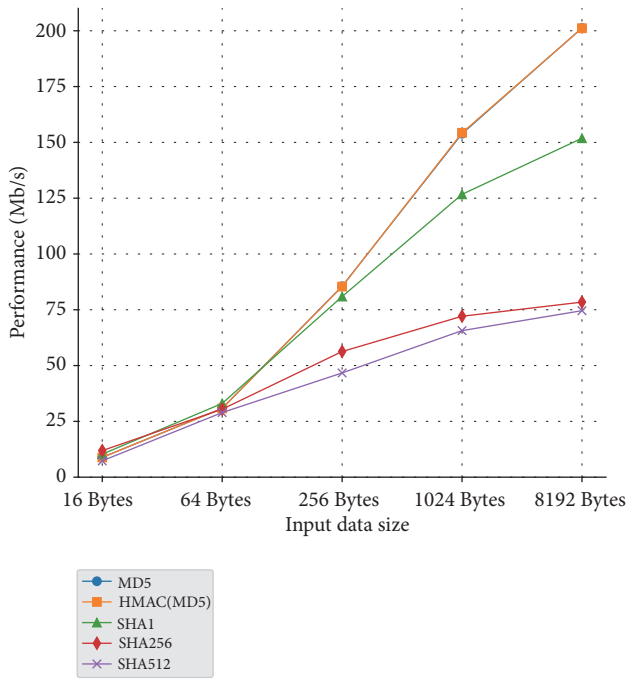

Figure 9: Hash functions performance in OSCO platform.

Table 4: Block cipher parameters.

Algorithm	Blowfish	IDEA	DES	3DES	AES-128	AES-192	AES-256
Block size (bits)	64	64	64	64	128	128	128
Key size (bits)	128	128	56	112	128	192	256
Rounds	16	8	16	48	10	12	14

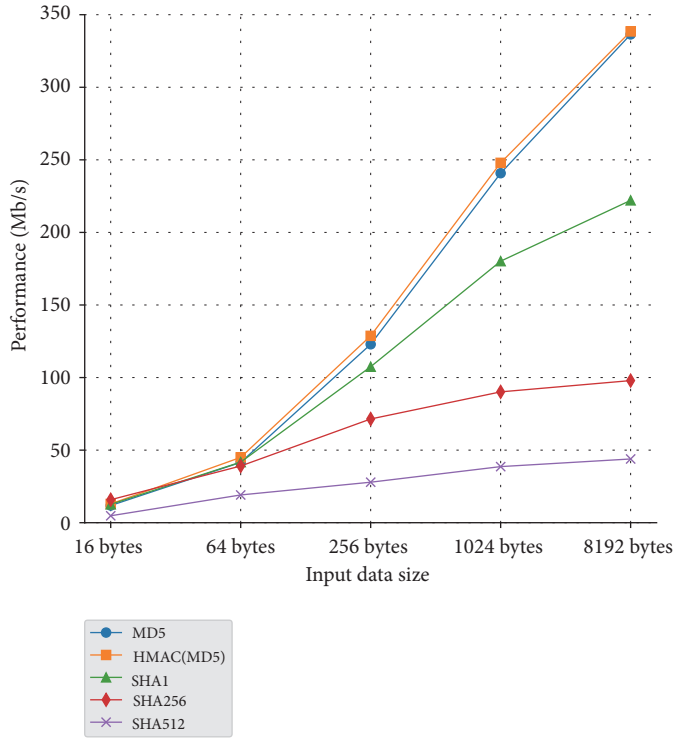

Figure 10: Hash functions performance in HTPC platform.

Table 5: Key parameters used in the experiment.

Algorithm	Modules size or size of n in bits
RSA	2048, 4096
DSA	1024, 2048
ECDSA	192, 224, 256, 384, 521
ECDH	192, 224, 256, 384, 521

and IDEA both show much better performance than DES in OSCO and Server platform.

HTPC platform has an old Atom D2700 processor which cannot support any new security instruction set such as AES-IN or Intel SHA extensions. This is the reason that AES is slower than DES in HTPC platform, as shown in Figure 15.

The performance of IDEA is very similar to AES-128 on HTPC platform. Blowfish also shows a good performance (the second best performance) on HTPC platform. In particular, longer key size leads to lower performance in AES. The performance of block cipher algorithm (Blowfish, IDEA, DES, 3DES, and AES) is not significantly affected by the size of input data on the three platforms.

4.3. Public-Key Cryptosystems Performance. The public-key cryptosystems are widely used for key distribution, confidentiality, and authentication. Table 5 shows the key parameters used by the algorithm in the experiment.

RSA (Rivest-Shamir-Adleman) [55] is one of the first practical public-key cryptosystems. In the RSA scheme, the parameter n is the block size and also participates in generating the public key and the private key. A typical size for n is 1024 bits. In our experiment, the length of the parameter n is set as 2048 bits and 4096 bits.

As a commonly used digital signature algorithm, DSA scheme [56] provides similar performance in signing and verifying operation as shown in Figures 16 and 17. The signing and verifying operation performance metrics are defined as $p_2 = O_s/t$ and $p_3 = O_v/t$, where O_s and O_v denote the number of executed signing operations and verification operations, respectively, and t denotes the time cost of the operation.

Figure 16 shows that ECDSA-192 [57] has the best signing result and RSA-4096 has the worst signing operation in all platforms. The RSA-2048 has the best verification performance and ECDSA-521 shows the worst performance in verification in Figure 17. The Server platform is the fastest one running both signing and verifying operations. The OSCO platform is better than HTPC platform in both operations. The x-axis indicates the amount of accomplished operations per second. The y-axis represents the type of algorithms with different key size (same as Figures 16, 17, and 18).

In our platform, we did not deploy the classic textbook RSA scheme due to its insecure features. Instead, our implementation supports the probabilistic encryption by using the RSA-PKCS1-PADDING padding mode to satisfy the desired security requirements. The RSA-PKCS1-PADDING padding mode adds random nonzero data to each data block to ensure that the result differs each time. The similar technique is used by other algorithms such as DSA and ECDSA.

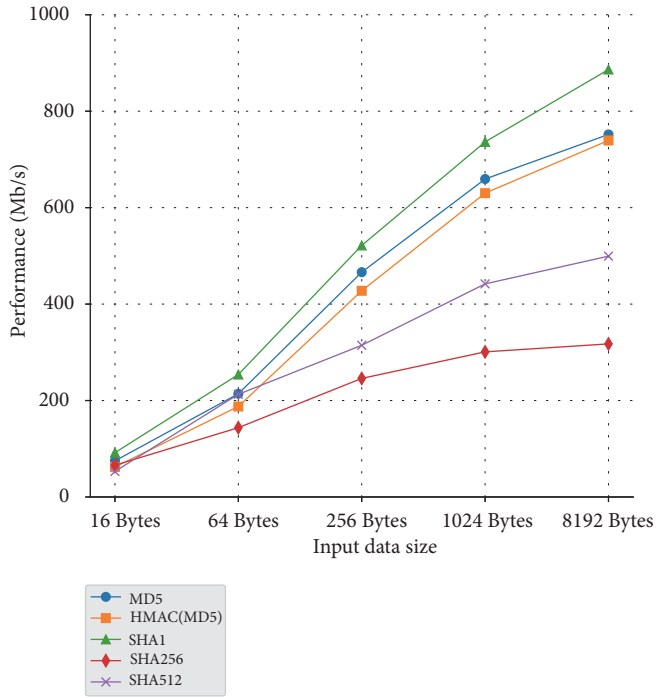

Figure 11: Hash functions performance in Server platform.

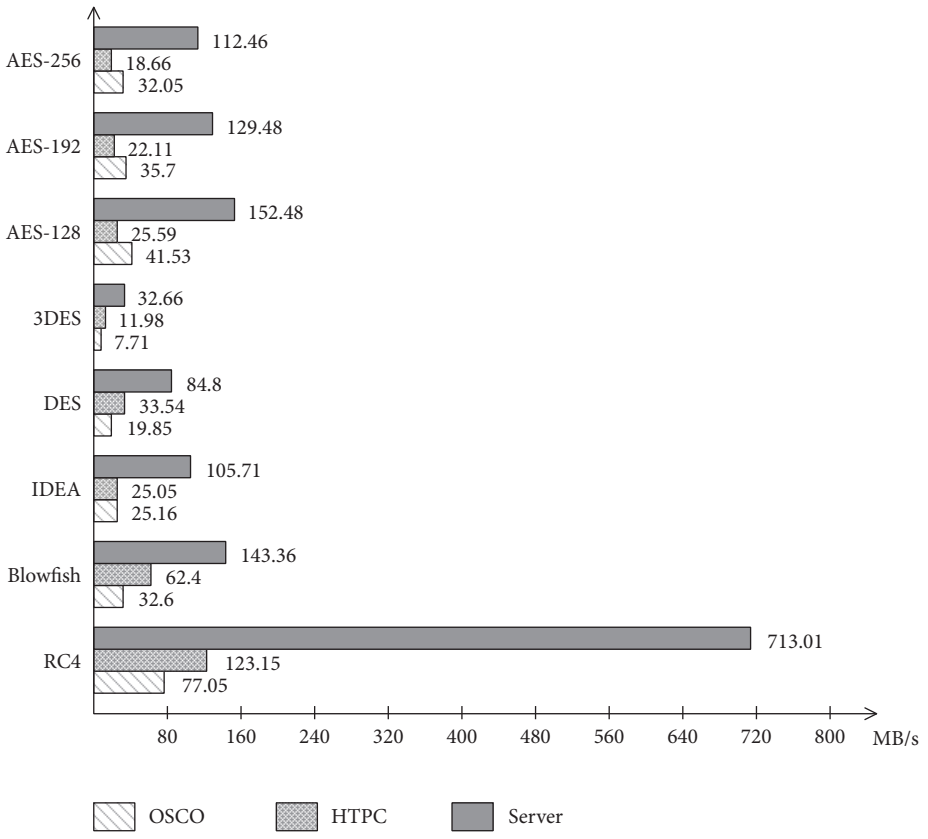

Figure 12: Stream/block ciphers performance in the three platforms.

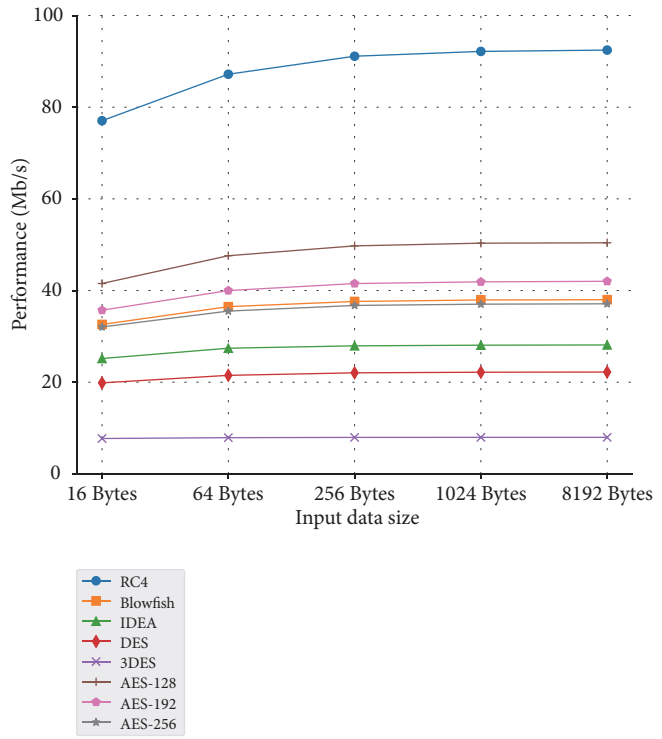

Figure 13: Stream/block ciphers performance in OSCO platform.

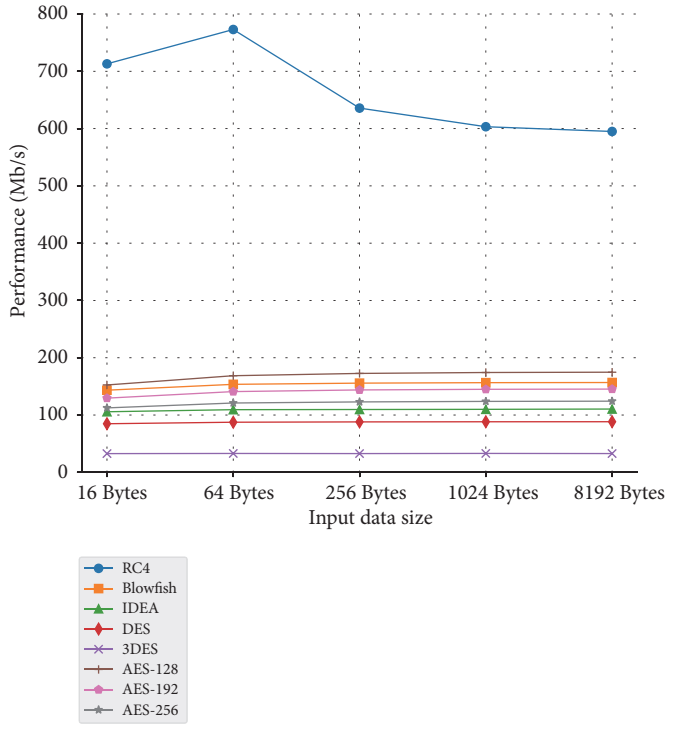

Figure 14: Stream/block ciphers performance in the Server platform.

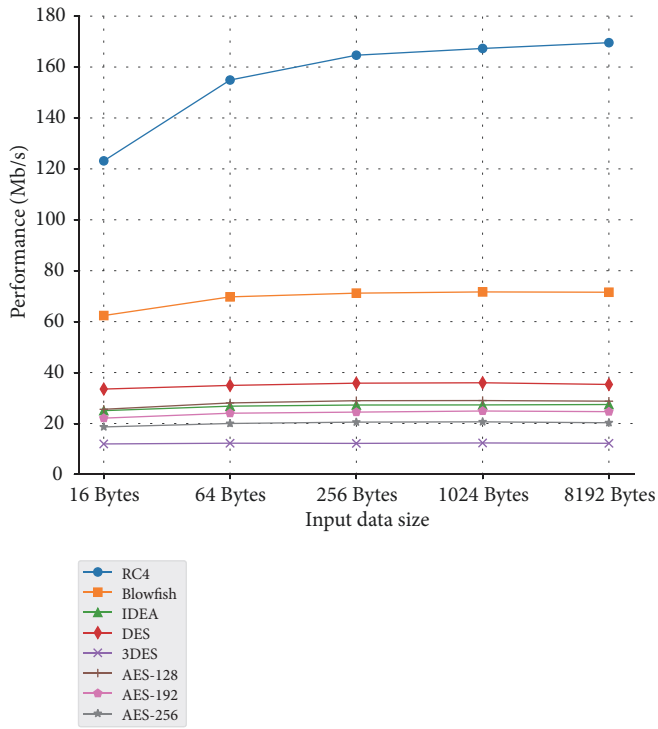

Figure 15: Stream/block ciphers performance in HTPC platform.

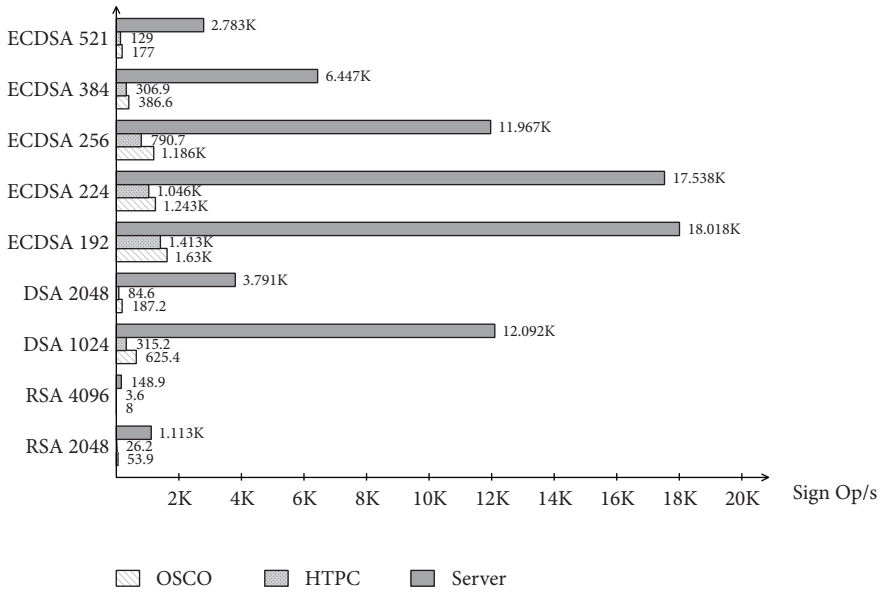

Figure 16: Sign operation performance in the three platforms.

4.4. ECDH Key Exchange Performance. The ECDH (elliptic-curve Diffie-Hellman) [58] is a key agreement protocol that allows two parties to establish a shared secret over an insecure channel. The ECDH key exchange performance is influenced by the size of key as shown in Figure 18. ECDH-192 has the best performance in OSCO platform, while on the Server platform, ECDH-224 is much faster than ECDH-192. ECDH and ECDSA-224/256 implementation are optimized in 64-bit OpenSSL after version 1.0.0h [59]. ECDH-192 and ECDSA-192 only have 32-bit portable implementation. The ECDH operation performance metrics are defined by $p_4 = O_i/t$, where O_i denotes the number of executed ECDH operations with key size i and t denoting the time cost of the operation. The benchmark runs a standard NIST (National

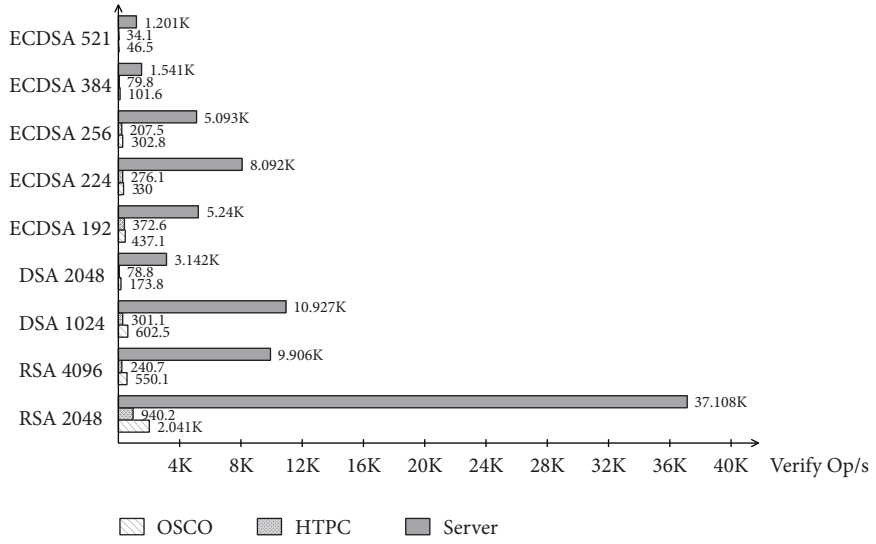

Figure 17: Verify operation performance in three platforms.

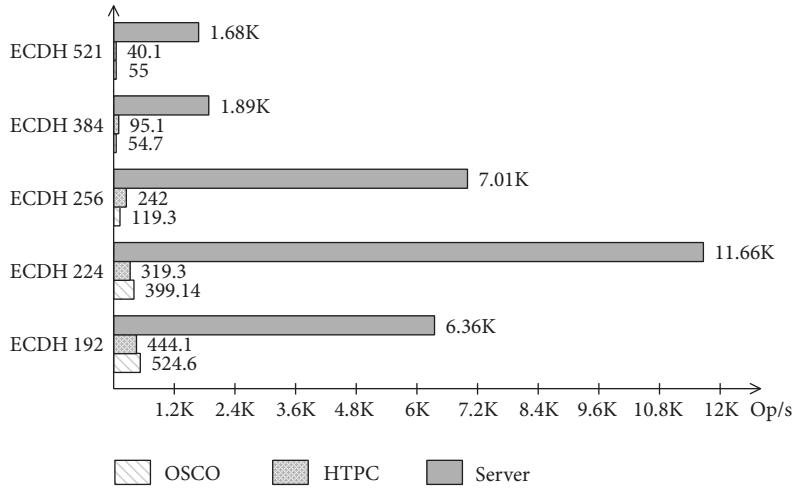

Figure 18: ECDH key exchange performance in the three platforms.

Institute of Standards and Technology) implementation of elliptic-curve with different key lengths, such as P-192, P-224, P-256, P-384, and P-521.

4.5. OSCO Platform Network Performance. The OSCO acts as an OpenFlow switch in the SDN network topology with some enhanced security abilities. Its data forwarding performance, which influences the whole SDN network performance, is one of the key factors besides the computation power of the cryptosystem.

The experiment uses IPerf, a Linux network tool, to test the max bandwidth and minimum response time of two hosts, which use the TCP and UDP network connection, respectively. The UDP connection testing applies -ub 100m as the input parameters and the other part keeps using the default setting; e.g., UDP buffer size is 160Kbyte and port is 5001. The TCP connection uses all default parameters for setting; e.g., TCP windows is 43.8Kbyte, etc. The network performance metrics are defined by $p_5 = d_p/t$, where d_p denotes the number of data being transmitted under protocol p mode and t denotes the transmission time.

Figure 19 shows two different network connections in TCP and UDP mode. The first connection (the virtual host1 to the physical host3) is indicated by lined bar, and the second connection is the link between two physical hosts (host4 to host3), indicated by dotted bar (the network topology is shown in Figure 4). The solid bar indicates the maximum theoretical network connection bandwidth (100Mb/s). Both connections get pretty good results compared with maximum theoretical bandwidth. In addition, the minimum response time of the 1st connection is 7.902ms, and the 2nd connection gets 0.029ms (both use UDP connection).

Figure 20 compares network performances between two OSCO switches when they enable or disable real-time encryptions. Based on the reviewed results of different cryptographic algorithms, AES-256 and SHA-256 algorithms

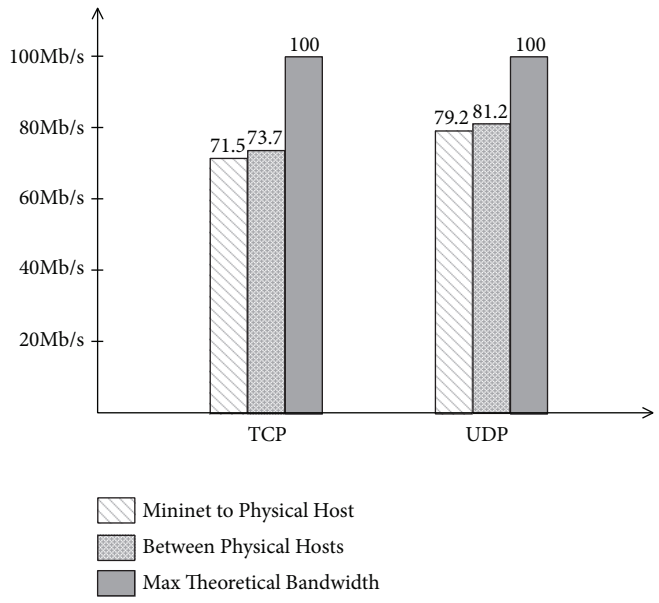

Figure 19: Network performance in OSCO platform.

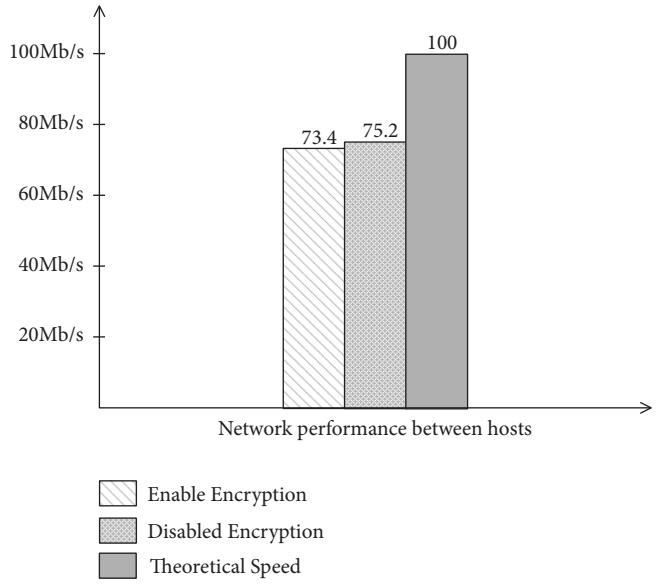

Figure 20: Network performance with security function enabled/disabled in OSCO.

are chosen to build a custom real-time encryption protocol on the OSCO platform. The experiment shows that the encryption overhead is relatively low (around 2.5%) via a UDP connection between two switches.

In this section, we evaluate the performance of most used cryptographic algorithms in different platforms. Moreover, our experiments confirm that OSCO platform is not only capable of carrying out the various cryptographic algorithms but also able to support good network bandwidth and speed requirements. Furthermore, the OSCO platform has the lowest power consumption of 4W (system power consumption) comparing to HTPC and Server platform which approximately consume 15W and 80W only for the processors (Thermal Design Power from Intel), respectively.

5. Related Work

With the rapid development of SDN, more and more studies are carried out to improve SDN network performance and security. The centralized SDN architecture makes network management much more easier and flexible than before, but it introduces new security challenges too. Therefore, many security-oriented solutions are proposed. The proposed solutions focus on the security challenges in the data, controller,

Table 6: SDN security solutions.

Solutions	Type	SDN Plane			Interface	
		Data	Ctrl.	App.	App-Ctrl	Ctrl-Data
PERM-GUARD [15]	Cross-layer protection framework	x	x	x		
VeriFlow [34]	Verify and debug flow rules	x				
CPRecovery [35]	Controller response framework	x	x			
FortNOX [10]	Controller framework	x	x			
FlowChecker [36]	Configuration verification tool	x	x			
FRESCO [8]	Anomaly diction and mitigation framework		x		x	
PermOF [37]	Permission control system for Apps.	x	x			
Flover [38]	Flow policy verification	x	x	x		
OFTesting [39]	OF Apps testing and debugging			x		
SE-Floodlight [40]	Conflict resolution, authorization, audit system	x	x		x	
Mini-Cut placement [41]	Controller reliability, switch-controller connectivity	x	x			x
Monitoring [42]	Data plane connectivity monitoring	x				x
POCO [43]	Controller resilience and failure tolerance		x			x
DCP [44]	Dynamic controller provisioning, scalability, availability		x			x
Resonance [45]	Access control and dynamic policy enforcement	x				
AVANT-GUARD [46]	Controller plane security enhancement framework		x			

Table 7: Platform solutions.

Platform solutions	Software environment	Hardware environment
Mininet [11]	x	
NS3 [12]	x	
OpenNet [47]	x	
OMNET++ [48]	x	
EstiNet [49]	x	
Donatini et al. [50]		x
Ariman et al. [51]		x
OpenSample [52]		x
Mercury [53]		x
XSM [54]		x

and application plane of SDN as well as the interfaces in between. For instance, AVANT-GUARD [46] protects the control plane from a variety of attacks. The authors in [10] propose FortNOX, a mechanism which monitors the flow rules states from application plane to data plane in the SDN network. PERM-GUARD [15] protects all the three SND planes through management of flow rule production permissions. Gao et al. [60] proposed CPSTCS, a cyber-physical systems testbed based on cloud computing and SDN in Industry 4.0. Yoon et al. [61] proposed a fault-tolerant mechanism that responds to controller failures in CPS that have multiple controllers for resilient control of physical objects. Piedrahita et al. [62] proposed a solution that detects and responds automatically to sensor attacks and controller attacks in industrial network based on SDN. Han et al. [63] proposed a dynamic route strategy to decrease the delay and congestion in cyber-physical power system based on SDN. Table 6 shows more SDN security solutions. The software simulation or the physical device deployment are used to test and verify those proposed theories or prototypes.

Generally speaking, there are two ways to verify a newly designed protocol or mechanism in a computer network system. One is software simulation verification, which models a real-system or hypothetical situation on a computer software so that it can be studied to see how the system works. The other way for physical device deployment verification deploys a designed protocol or mechanism to a real hardware environment. Table 7 shows more platform related solutions for the networking system.

Software simulation. As the network environment of SDN is different from the traditional network, the challenge of SDN network simulation has drawn a great attention in recent years. Lantz et al. [11] proposed the Mininet, a lightweight tool simulating the SDN networks by using OS-level virtualization features, including processes and network namespaces to scale hundreds of SDN nodes. Riley et al. [12] introduced NS3, a powerful tool for network simulation. Chan et al. [47] proposed OpenNet, which is an open source simulator for wireless network simulation. OpenNet is a combination of Mininet and NS3 and it supports various SDN controllers by using Mininet and wireless modeling through NS3. OpenNet also supports wireless channel scan mechanism. Wang et al. [49] introduced EstiNet, an SDN network simulator, which enables external SDN controller to control simulated OpenFlow switches.

Physical device deployment. Donatini et al. [50] introduced an approach to develop a SDN network measurement platform by using NetFPGA under the LTE network infrastructure. The SDN implementation part is based on OpenFlow and OpenSketch [64] projects. Ariman et al. [51] implemented a real-time testbed for software-defined wireless networks (SDWN) by using Raspberry Pi as OpenFlow

(OF) switches. The implementation provides a practical development and testing environment for SDWNs. Suh et al. [52] proposed and implemented a sampling-based measurement platform which decreases the latency to gather accurate measurements of network load by using sFlow [65] samples. Hu et al. [66] proposed an SDN based big data platform for social TV analytics. It provides three main components, distributed date crawler, big data processing, and social TV analytics. Skowyra et al. [67] introduced an SDN-Enabled application verification platform which verifies composed applications and SDN system models to assure their safety, security, and performances.

Unlike the related work mentioned above, we do not just focus on simulating some specific SDN network functions. Instead, we are interested in exposing the new security challenges introduced by SDN deployment and verifying corresponding countermeasures on an extensible customized platform.

6. Conclusion

In this paper, to effectively integrate security countermeasures and systematically verify and test developed security mechanisms in the SDN environment, we propose a lightweight security enhancement and security testing platform, OSCO (Open Security-enhanced Compatible OpenFlow) platform. The design of OSCO is based on Raspberry Pi Single Board Computer (SBC) hardware and SDN network architecture, which supports highly configurable cryptographic algorithm modules, security protocols, flexible hardware extensions, and virtualized SDN networks. We prototyped our design and present a case study for port security enhancement to illustrate the details of the OSCO platform implementation and deployment. We evaluate the prototype system, and the experiment results show that our system conducted security functions with relatively low computational and networking performance overheads.

Data Availability

No data were used to support this study.

Disclosure

An earlier version of this manuscript was presented in "International Conference on Wireless Algorithms, Systems, and Applications (WASA) 2018" [13].

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by the National Key R&D Program of China (No. 2017YFB1400700), the National Natural Science Foundation of China (No. 61402029, U17733115), the National Natural Science Foundation of China (No. 61379002, No. 61370190), and Beijing Natural Science Foundation (No. 4162020).

References

[1] HPE Marie Paule-Odini, "Sdn and nfv evolution towards 5g," 2017, http://https://sdn.ieee.org/newsletter/september-2017/sdn-and-nfv-evolution-towards-5g.

[2] R. Wang, J. Yan, D. Wu, H. Wang, and Q. Yang, "Knowledge-Centric Edge Computing Based on Virtualized D2D Communication Systems," *IEEE Communications Magazine*, vol. 56, no. 5, pp. 32–38, 2018.

[3] N. McKeown, T. Anderson, H. Balakrishnan et al., "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[4] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: a comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[5] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in software defined networks: a survey," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2317–2346, 2015.

[6] J. Spooner and S. Y. Zhu, "A Review of Solutions for SDN-Exclusive Security Issues," *International Journal of Advanced Computer Science and Applications*, 2016.

[7] M. Kuerban, T. Yun, . Y. Qing, and P. David, "Flowsec: Dos attack mitigation strategy on sdn controller," in *Proceedings of the IEEE International Conference on Networking*, pp. 211-212, 2016.

[8] S. Shin, P. Porras, Y. Vinod, M. Fong, G. Gu, and T. Mabry, "Fresco: Modular composable security services for software-defined networks," *Proceedings of Network & Distributed Security Symposium*, 2013.

[9] "Floodlight controller," http://www.projectfloodlight.org/.

[10] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proceedings of the 1st ACM International Workshop on Hot Topics in Software Defined Networks, HotSDN 2012*, pp. 121–126, Association for Computing Machinery, Helsinki, Finland, August 2012.

[11] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets '10)*, 19 pages, ACM, October 2010.

[12] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," *Modeling and Tools for Network Simulation*, pp. 15–34, 2010.

[13] H. Cheng, J. Liu, J. Mao, M. Wang, and J. Chen, "OSCO: An Open Security-Enhanced Compatible OpenFlow Platform," in *Proceedings of the International Conference on Wireless Algorithms, Systems, and Applications (WASA)*, Springer, 2018.

[14] R. Klöti, V. Kotronis, and P. Smith, "OpenFlow: A security analysis," in *Proceedings of the 2013 21st IEEE International Conference on Network Protocols, ICNP 2013*, pp. 1–6, Institute of Electrical and Electronics Engineers, Goettingen, Germany, 2013.

[15] M. Wang, J. Liu, J. Chen, X. Liu, and J. Mao, "Perm-guard: Authenticating the validity of flow rules in software defined networking," in *Proceedings of the IEEE International Conference on Cyber Security and Cloud Computing*, pp. 1–17, 2016 (Chinese).

[16] P. P. Naga, K. Fang Hao, and T. V. Lakshman, *Securing software defined networks via flow deflection*, 2014.

[17] "Raspberry pi hardware specification," 2011, http://www.raspberrypi.org/documentation/hardware/.

[18] "Ubuntu mate," http://ubuntu-mate.org/.

[19] "Open vswitch," http://www.openvswitch.org/.

[20] "Open ssl," http://www.openssl.org/.

[21] "Crypto++ library," http://www.cryptopp.com/.

[22] "Pairing-based cryptography library," http://crypto.stanford.edu/pbc/.

[23] "Wireshark," http://www.woreshark.org/.

[24] "Nox controller," http://www.noxrepo.org/.

[25] R. Rivest, "The MD5 Message-Digest Algorithm," *RFC*, vol. 473, no. 10, pp. 492-492, 1992.

[26] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," RFC2104, 1997.

[27] D. Eastlake 3rd and P. Jones, "Us secure hash algorithm 1 (sha1)," Technical Report, 2001.

[28] T. Pornin, "Is calculating an md5 hash less cpu intensive than sha family functions?" stackoverflow, https://stackoverflow.com/questions/2722943/.

[29] R. Thayer and K. Kaukonen, *A Stream Cipher Encryption Algorithm*, 1999.

[30] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (Blowfish)," in *Fast Software Encryption, Cambridge Security Workshop*, pp. 191–204, 1993.

[31] X. Lai and J. L. Massey, *A Proposal for a New Block Encryption Standard*, Springer Berlin Heidelberg, 1991.

[32] M. E. Smid and D. K. Branstad, "Data encryption standard: past and future," *Proceedings of the IEEE*, vol. 76, no. 5, pp. 550–559, 1988.

[33] J. Daemen and V. Rijmen, *The Design of Rijndael: AES-The Advanced Encryption Standard*, Springer, Berlin, Germany, 2002.

[34] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[35] P. Fonseca, R. Bennesby, E. Mota, and A. Passito, "A replication component for resilient OpenFlow-based networking," in *Proceedings of the 2012 IEEE Network Operations and Management Symposium, NOMS 2012*, pp. 933–939, USA, April 2012.

[36] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration, SafeConfig '10*, pp. 37–44, USA, October 2010.

[37] X. Wen, Y. Chen, C. Hu, C. Shi, and Y. Wang, "Towards a secure controller platform for OpenFlow applications," in *Proceedings of the 2013 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013*, pp. 171-172, China, August 2013.

[38] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in OpenFlow," in *Proceedings of the 2013 IEEE International Conference on Communications, ICC 2013*, pp. 1974–1979, Hungary, June 2013.

[39] M. Canini, D. Kostic, R. Jennifer, and D. Venzano, "Automating the testing of openflow applications," in *Proceedings of the International Workshop on Rigorous Protocol Engineering*, 2011.

[40] "OpenFlow Sec Security Enhanced Floodlight," SRI International, https://www.sdxcentral.com/projects/openflow-sec-security-enhanced-floodlight/.

[41] Y. Zhang, N. Beheshti, and M. Tatipamula, "On resilience of split-architecture networks," in *Proceedings of the Global Telecommunications Conference*, pp. 1–6, 2012.

[42] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Sköldström, "Scalable fault management for OpenFlow," in *Proceedings of the IEEE International Conference on Communications (ICC '12)*, pp. 6606–6610, June 2012.

[43] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia, "Pareto-optimal resilient controller placement in SDN-based core networks," in *Proceedings of the 25th International Teletraffic Congress (ITC '13)*, pp. 1–9, September 2013.

[44] M. F. Bari, A. R. Roy, S. R. Chowdhury et al., "Dynamic controller provisioning in software defined networks," in *Proceedings of the 9th International Conference on Network and Service Management (CNSM '13)*, pp. 18–25, Zürich, Switzerland, October 2013.

[45] A. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control for enterprise networks," in *Proceedings of the 1st Workshop: Research on Enterprise Networking, WREN 2009, Co-located with the 2009 SIGCOMM Conference, SIGCOMM'09*, pp. 11–18, Spain, August 2009.

[46] S. Shin, V. Yegneswaran, and P. Porras, "AVANT-GUARD: scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 413–424, Berlin, Germany, 2013.

[47] M.-C. Chan, C. Chen, J.-X. Huang, T. Kuo, L.-H. Yen, and C.-C. Tseng, "OpenNet: A simulator for software-defined wireless local area network," in *Proceedings of the 2014 IEEE Wireless Communications and Networking Conference, WCNC 2014*, pp. 3332–3336, Turkey, April 2014.

[48] A. Varga, "The omnet++ discrete event simulation system," in *European Simulation Multiconference*, 2001.

[49] S.-Y. Wang, C.-L. Chou, and C.-M. Yang, "EstiNet openflow network simulator and emulator," *IEEE Communications Magazine*, vol. 51, no. 9, pp. 110–117, 2013.

[50] L. Donatini, R. G. Garroppo, S. Giordano et al., "Advances in LTE network monitoring: A step towards an SDN solution," in *Proceedings of the 2014 17th IEEE Mediterranean Electrotechnical Conference, MELECON 2014*, pp. 339–343, Lebanon, April 2014.

[51] M. Ariman, G. Secinti, M. Erel, and B. Canberk, "Software defined wireless network testbed using Raspberry Pi of switches with routing add-on," in *Proceedings of the IEEE Conference on Network Function Virtualization and Software Defined Network, NFV-SDN 2015*, pp. 20-21, USA, 2016.

[52] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "OpenSample: A low-latency, sampling-based measurement platform for commodity SDN," in *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014*, pp. 228–237, July 2014.

[53] K. Lorincz, B.-R. Chen, and G. W. Challen, "Mercury: a wearable sensor network platform for high-fidelity motion analysis," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)*, pp. 183–196, Berkeley, California, USA, November 2009.

[54] P. Dutta, M. Grimmer, A. Arora, and S. Bibykt, "Design of a wireless sensor network platform for detecting rare, random, and ephemeral events," in *Proceedings of the 4th International*

Symposium on Information Processing in Sensor Networks, IPSN 2005, pp. 497–502, USA, April 2005.

[55] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[56] National Institute Of Standards, *Digital signature standard (dss)*, vol. 25, Federal Information Processing Standards Publication 186-2, 2000.

[57] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[58] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology-CRYPTO'85*, H. C. Williams, Ed., vol. 218 of *Lecture Notes in Computer Science*, pp. 417–426, Springer, 1986.

[59] Changelog, "Changes between 1.0.0h and 1.0.1," OpenSSL, https://www.openssl.org/news/changelog.html#x32/.

[60] H. Gao, Y. Peng, K. Jia, Z. Wen, and H. Li, "Cyber-Physical Systems Testbed Based on Cloud Computing and Software Defined Network," in *Proceedings of the International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pp. 337–340, 2016.

[61] S. Yoon, J. Lee, Y. Kim, S. Kim, and H. Lim, "Fast controller switching for fault-tolerant cyber-physical systems on software-defined networks," in *Proceedings of the 22nd IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2017*, pp. 211-212, New Zealand, January 2017.

[62] A. F. M. Piedrahita, V. Gaur, J. Giraldo, A. A. Cardenas, and S. J. Rueda, "Leveraging Software-Defined Networking for Incident Response in Industrial Control Systems," *IEEE Software*, vol. 35, no. 1, pp. 44–50, 2017.

[63] Y. Han, H. E. Yiqian, F. Lou, Y. Wang, and C. Guo, "Analysis and application of sdn based dynamic optimal route strategy for cyber layer in cascading failures of cyber-physical power system," *Power System Technology*, 2018.

[64] "Opensketch project," https://github.com/lavanyaj/opensketch.git.

[65] sflow, http://sflow.org/about/index.php/.

[66] H. Hu, Y. Wen, Y. Gao, T.-S. Chua, and X. Li, "Toward an SDN-enabled big data platform for social TV analytics," *IEEE Network*, vol. 29, no. 5, pp. 43–49, 2015.

[67] R. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury, "A verification platform for SDN-enabled applications," in *Proceedings of the 2nd IEEE International Conference on Cloud Engineering, IC2E 2014*, pp. 337–342, USA, March 2014.