Trust Anchors in Software Defined Networks

Nicolae Paladi[1](✉), Linus Karlsson[2], and Khalid Elbashir[3]

[1] RISE SICS, Kista, Sweden
nicolae.paladi@ri.se

[2] Lund University, Lund, Sweden
linus.karlsson@eit.lth.se

[3] KTH - Royal Institute of Technology, Stockholm, Sweden
elbashir@kth.se

Abstract. Advances in software virtualization and network processing lead to increasing network softwarization. Software network elements running on commodity platforms replace or complement hardware components in cloud and mobile network infrastructure. However, such commodity platforms have a large attack surface and often lack granular control and tight integration of the underlying hardware and software stack. Often, software network elements are either themselves vulnerable to software attacks or can be compromised through the bloated trusted computing base. To address this, we protect the core security assets of network elements - authentication credentials and cryptographic context - by provisioning them to and maintaining them exclusively in isolated execution environments. We complement this with a secure and scalable mechanism to enroll network elements into software defined networks. Our evaluation results show a negligible impact on run-time performance and only a moderate performance impact at the deployment stage.

1 Introduction

Software Defined Networking (SDN) is a widely used approach to operate network infrastructure in virtualized environments. Separation of forwarding and control logic, a core idea of this model, is often realized by software network elements in a virtualized network infrastructure deployed on commodity hardware. However, by departing from hardware network elements with tightly couped software and hardware often provided by the same vendor [20], SDN broke previous assumptions, outdated best-practices and introduced new vulnerabilities [41,42]. Scott-Hayward et al. outlined a series of attack vectors that can lead to unauthorized access, data leakage or modification, malicious applications on the network, configuration issues, and a wider collection of system-level security vulnerabilities [49]. This concern applies to both the data plane and the application plane in SDN deployments. On the data plane, related literature describes both potential attacks on SDN in case of a virtual switch compromise [2], partly demonstrated

J. Lopez et al. (Eds.): ESORICS 2018, LNCS 11099, pp. 485–504, 2018.
https://doi.org/10.1007/978-3-319-98989-1_24

in [55]. Malicious applications deployed on the SDN infrastructure are a particular concern in virtualized environments. They affect network security both directly (by intercepting or modifying traffic), or indirectly through horizontal attacks aimed to leak authentication credentials and encryption keys [54].

Earlier research addressed SDN security through additional services [21,48,53], formal verification [6] and isolated execution using Intel Software Guard Extensions (SGX) [28,43,44,52], and most popular network element implementation support communication over transport layer security (TLS) [15]. Despite these efforts, the confidentiality and integrity of authentication credentials of network elements in SDN remain unaddressed. In particular, the existing approaches to provision authentication credentials to network elements in SDN are either plain insecure or both insecure and unscalable, requiring manual steps[1] [38]. Moreover, credentials provisioned to network elements in virtualized environments are often stored in plaintext on the file system. Adversaries exploiting vulnerabilities in process and virtualization isolation can access authentication credentials to perform network attacks or impersonate network elements. In this paper, we address two complementary questions: (1) *How can authentication credentials be securely provisioned to software network elements in SDN deployments?* and (2) *How can the TLS context of virtual switches be protected on compromised hosts?*

1.1 Contributions

In this work, we present the following contributions:

- A secure, practical, and scalable mechanism to provision authentication credentials and bootstrap communication between software network elements.
- TLSonSGX[2], a library allowing to maintain authentication credentials and the TLS context exclusively in isolated execution environments.
- A novel approach to restricting the availability of authentication credentials for SDN components to hosts with an attested trusted computing base.
- A first thorough analysis of the performance trade-offs of deploying components of network elements in SGX enclaves.

1.2 Structure

The remainder of this paper is structured as follows. We present the system model and threat model in Sect. 2. Next, we describe the proposed solution in Sect. 3 and its implementation in Sect. 4. We evaluate the approach in Sect. 5, discuss the related work in Sect. 6, outline limitations and future work in Sect. 7 and conclude in Sect. 8.

[1] Indeed, the Open vSwitch manual contains phrases as "Write the fingerprint down on a slip of paper and copy sc-req.pem to the machine that contains the PKI structure".

[2] Source code available: https://github.com/TLSonSGX/TLSonSGX.

2 System and Threat Model

We consider an SDN infrastructure deployed on commodity platforms in a distributed system, such as in a cloud platform or a mobile communications network. The infrastructure is managed by the *administrators* of a *network operator*. Physical access to the platforms is restricted and auditable.

System Model. Administrators use *orchestrators* to manage network infrastructure, software components and network services [20]. They deploy network elements on the *data plane*, *control plane* and *application plane*. The *data plane* consists of hardware or software switches (e.g. Open vSwitch [47]) and communication links between them. The *control plane* consists of a logically centralized *network controller* (e.g. ONOS [7], Floodlight [25]). The network controller manages software switches through protocols such as OpenFlow [34] (to add or remove flows) or OVSDB [46] (to create ports and tunnels); it manages hardware switches through OpenFlow (if supported) or other interfaces, such as NETCONF [17]. The *application plane* comprises *network functions* that implement services such as traffic engineering, monitoring, or caching. A *Virtual Network Function* (VNF) is a virtualisation of a network function [20]. Orchestrators deploy VNFs upon request from the network controller or a tenant. The network controller configures flows and steers traffic to the network functions.

Network elements on the data-, control-, and application planes communicate over two application programming interfaces (APIs). The controller communicates with data plane elements over the *southbound* API, commonly Openflow [8,34,51] and with application plane elements over the *northbound* API.

At deployment time, the orchestrator provisions TLS certificates to network elements during the *enrollment* process. Furthermore, to protect the data within the SDN deployment, the network controller enforces communication over TLS with mutual authentication on both southbound and northbound APIs.

Threat Model. Similar to earlier work on SDN security threats [30,41], we assume physical security of the platforms underlying the SDN infrastructure and correct implementation of cryptographic algorithms and communication security protocols, such as TLS [15]. The adversary has the capabilities of a system administrator with remote access to commodity platforms in the SDN infrastructure. The adversary can intercept, drop and modify packets on the southbound and northbound interfaces. Furthermore, the adversary can run arbitrary network elements in the SDN deployment and elsewhere [20]. The adversary can read the memory of the commodity platforms, exploit vulnerabilities in network elements on the data- and application planes, and circumvent virtualization isolation [2].

3 Solution Space

We next present the approach for provisioning and protecting authentication credentials on the data and application planes of SDN deployments. We first introduce three building blocks to create trust anchors in SDN deployments: Software Guard Extensions (SGX), Trusted Platform Module (TPM) and Integrity Measurement Architecture (IMA).

3.1 Trust Anchors

We use SGX enclaves [1,32,33,61] to create trusted execution environments (TEEs) during operating system execution. We use the TEEs to store authentication credentials and execute cryptographic operations for network elements. SGX enclaves rely on a trusted computing base (TCB) of code and data loaded at enclave creation time, processor firmware and processor hardware. Program execution within an enclave is transparent to the underlying operating system and other mutually distrusting enclaves on the platform. Enclaves operate in a dedicated memory area called the Enclave Page Cache, a range of DRAM that cannot be accessed by system software or peripherals [23,33]. The CPU firmware and hardware are the root of trust of an enclave; it prevents access to the enclave's memory by the operating system and other enclaves. Remote attestation [12] allows an enclave to provide integrity guarantees of its contents [1].

We use TPMs to store platform integrity measurements collected during boot, and attest the integrity of platforms hosting the SDN infrastructure. A TPM is a discrete component on the platform motherboard and its state is distinct from the state of the platform. TPMs provide secure non-volatile storage, cryptographic key generation and use, sealed storage and support (remote) attestation [56]. TPMs assume platform integrity by identifying and reporting the platform state that comprises the hardware and software components [36]. In this context, *trust* is based on the conjecture that a certain behaviour can be expected based on the reported platform state [42]. TPMs can prove the association between a cryptographically verifiable identity and the host platform [56,57].

We use Linux IMA to measure the integrity of the TCB. Linux IMA measures a predefined set of files on the system by hashing their contents and storing the values in a measurement list; it can be configured to detect modifications of files at runtime. To guarantee the integrity of the measurement list, its trust can be rooted in the TPM. The system's trustworthiness can be assessed by a remote appraiser by comparing the measurement list to an expected configuration [12]. We utilize IMA to collect measurements of the network elements on the platform. During the remote attestation of the platform, we use the measurement list to verify the integrity - and implicitly the trustworthiness - of network elements.

3.2 Data Plane

At cloud platform deployment time, an orchestrator deploys and runs virtual switches on the underlying compute resources. To enable network connectivity, the orchestrator instructs virtual switches to add (or delete) ports whenever virtualized execution environments are instantiated or torn down.

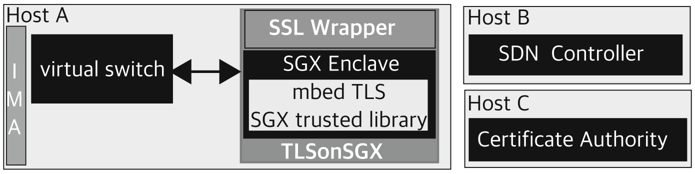

Fig. 1. TLSonSGX system design

For a secure deployment, the administrator must ensure both a secure installation of hardware and software, as well as provision the correct initial configuration of the virtual switch instances in the cloud infrastructure. In turn, secure generation of keys and provisioning of certificates is a precondition to ensuring security of the initial deployment configuration. Furthermore, ensuring the integrity of virtual switch binaries and configurations is a precondition for ensuring the run-time security of the deployed instances.

We address this with a new library, **TLSonSGX**, that enables virtual switches to use a cryptographic library running in a TEE (see Fig. 1). TLSonSGX provides an abstraction layer and a wrapper around the cryptographic library deployed in a TEE, allowing to easily substitute the implementation depending on performance, functionality and licensing aspects. Following this approach, TLS sessions originate and terminate within the TEE and the generated keys and certificates are confined to the TEE, ensuring the confidentiality and integrity of core assets, such as generated keys, certificates and TLS context, even in the event of a host compromise. This, in combination with an infrastructure monitoring system and a file integrity subsystem (such as Linux IMA), prevents the adversary from impersonating data plane network elements [55] and from enrolling additional network elements into the infrastructure.

Secure provisioning of authentication certificates is challenging, especially at scale, and depends on the capability to establish a secure communication channel between the certificate authority (CA) and the target component. Several vendor-specific solutions exist [27,33]. To support the deployment, we introduced a CA with extended functionality to sign certificates for the virtual switches and the SDN controller. CA certificates are provisioned to the virtual switches and the SDN controller in the deployment and are subsequently used for mutual authentication. Beyond secure certificate provisioning, the extended CA verifies the integrity of the virtual switches before signing their certificates. We leverage the remote attestation capability provided by the TPM to verify the TCB integrity on the host platform. The TPM is in this protocol the root of trust that stores and provides a signed quote of the integrity measurements of the virtual switch binary and ancillary libraries, collected by IMA.

3.3 Application Plane

Network elements on the application layer, such as VNFs, must be authenticated and integrity verified prior to enrollment into the SDN infrastructure. As the

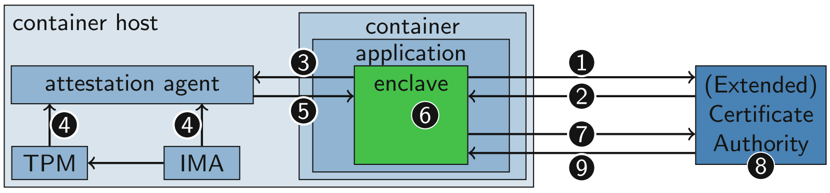

Fig. 2. Enrollment steps in the application layer.

controller requires mutual authentication with all its clients, this ensures that only trustworthy VNFs can communicate with the controller. Similar to the approach above, the TPM is used as a root of trust.

We use SGX enclaves to ensure integrity and confidentiality of the authentication credentials for enrolled VNFs. Storing the credentials in SGX enclaves reduces the attack surface to the enclave TCB and offers an additional layer of protection even in the case of a breach of the platform TCB. We discuss the limitations of this approach in Sect. 6.3.

We next provide an overview of the proposed solution (see Fig. 2). The extended certificate authority (CA) introduced above determines whether or not a VNF configuration is valid, by matching against a list of known good configurations. If a configuration is valid, the CA can also sign certificates. This component can be collocated with the network elements in the deployment, or be deployed and operated by a third party. We assume that the CA root certificate is provided to the SDN controller during initial setup.

At the start of the enrollment protocol, the orchestrator launches an execution environment (such as a bare-metal host, virtual machine or container) with TPM and IMA support. Together, these two mechanisms record both the software and hardware configuration in a measurement log, including the TCB of the VNFs. The measurement log is anchored in the TPM located of the host, allowing the use of the TPM's remote attestation functionality. Note that both a native and a virtualized TPM can be used in this case.

Similar to [62] we use an *attestation agent* running on the container host. This agent proxies the communication between the container and the TPM and IMA. We propose a solution where the attestation agent is only accessible from the container running on the same host. This prevents direct communication between the attestation agent and the CA. To prevent cuckoo attacks [45], the communication passes through the container application and the enclave and ensures that the enclave is running on the same host.

The enrollment phase consists of the following steps (see Fig. 2): Upon initialization of the container and application, the latter requests a nonce from the CA ❶, ❷. Next, the application requests from the attestation agent a quote for the given nonce, together with the IMA measurement list ❸. The agent communicates with the TPM and the IMA to retrieve the data ❹, and returns the data to the application ❺. The enclave generates a new private key and a certificate signing request (CSR) and stores it in the SGX enclave ❻. The application sends the quote, measurement list, and the CSR to the CA ❼, that verifies the message ❽. As the measurement list covers both the host system and the container TCB, the integrity of the host and target containers can be validated. If the measurement values match known good configurations, the CA signs the CSR and returns the signed certificate to the enclave ❾. At this point, the VNF can establish a secure TLS connection with the SDN controller. The proposed solution ensures that only trustworthy VNFs receive valid certificates and can be enrolled in the SDN infrastructure.

4 Implementation

To facilitate adoption and obtain reproducible results, we implemented the proposed solution using common open-source libraries and execution isolation features available on commodity platforms. We used Open vSwitch (OvS), a popular software switch implementation and the Ryu and Floodlight SDN controllers, mainly due to their popularity and simple configuration. In the remainder of this section, we first describe the implementation of TLSonSGX on the data plane. Next, we describe the security mechanisms deployed on the application plane.

4.1 TLSonSGX

The SGX programming model requires that applications deployed in SGX enclaves have an external component that can be called by other processes running on the operating system, and that in turn maps such calls to software in the enclave. This external component is not part of the enclave and its integrity cannot be attested using the SGX integrity attestation mechanisms, thus is considered *untrusted*; in contrast, the code running in the enclave is considered *trusted* once its integrity has been attested. Following the SGX programming model, the untrusted code portion of the TLSonSGX library is a wrapper that maps OpenSSL external methods (used by Open vSwitch) internally into enclave calls (ECALLs). The trusted portion of the code, contained within the SGX enclave, implements the ECALLs by utilizing the SGX trusted TLS library. Support for TLS libraries in SGX varies and evolves continuously; we have chosen the mbed TLS [31] library considering its sufficient support for SGX enclaves.

Considering that authentication keys and certificates are confined to the enclave, we modified OvS to use only a limited set of OpenSSL external methods that we subsequently implemented in TLSonSGX. The OpenSSL library implements three data structures: `SSL_METHOD`, `SSL_CTX`, and `SSL`.

These data structures all contain crucial information for TLS connection security, therefore we create and confine them within the enclave. The objects are passed by the OvS instance via an unmodified API using the external methods we implemented. They are created, confined, and handled inside the enclave during the operation of the virtual switch, and hence discarded and not passed to ECALLs. There is no one-to-one mapping in mbed TLS for these three structures, hence we redefine these structures using mbed TLS primitives (specifically the `mbedtls_ssl_config` and `mbedtls_ssl_context` data structures).

The code in `stream-ssl.c` implements the interface between OvS and the OpenSSL library. We extended the OvS configuration script and `stream-ssl.c` with a new compilation flag, `SGX`. If the `SGX` flag is set at compilation time, `stream-ssl.c` will use the TLSonSGX static library instead of the OpenSSL library. Moreover, the sections of `stream-ssl.c` that load keys and certificates from the file system become redundant and are omitted.

4.2 Application Plane

On the application plane, the solution consists of three major components: the network application, the attestation agent, and the certificate authority.

The attestation agent is a service running on the container host, setup to listen to connections from containers running on the same host, as those are the only containers able to request a quote from this host. The attestation agent can return both a copy of the measurement list, and a quote from the TPM. The quotes are made over the appropriate PCR registers to capture the current configuration, together with a nonce to prevent replay attacks. Interfacing with the TPM is implemented using the TrouSerS TSS library [22] on Linux. Using an attestation agent reduced the code base of the containers, since they do not have to interface directly with a TPM or Linux IMA.

Next, the CA fulfills two goals. First, it validates the integrity of the components by validating the quote, and compares the configuration and measurement list to known good values. Second, if the two values match, the CA signs the applications CSR. We implemented this using the OpenSSL C library to create the signature with a pre-configured root certificate. This root certificate is distributed to the SDN controller, allowing it to validate the certificate chain.

The final component is the container application. Using mbed TLS [31], we implemented an application that supports the attestation sequence described earlier, and communicates with both the attestation agent and the CA. Once the attestation sequence is finished, the application can connect to an SDN controller using the credentials generated and confined within the enclave.

5 Evaluation

5.1 Testbed

We evaluated the solution on the testbed described below (see Fig. 3).

Hardware. The host platform is a Lenovo Thinkpad T460s with a dual-core Intel® Core™ i7-6600U CPU clocked at 2.60 GHz with SGX support. VM_1 was created with 1 virtual CPU, and VM_2 with 2 virtual CPUs; both VMs had 4 GB RAM, 30 GB of storage, and used virtio as vNIC. We used Ubuntu 16.04.1 (with OvS and SGX drivers and SDKs) on both the host and VMs. To enable the use of SGX within the VM environment, we created VM_2 using patched versions of QEMU and KVM provided by the SGX project[3] and Intel SGX SDK, v1.8.

[3] SGX Virtualization, 01.org/intel-software-guard-extensions/sgx-virtualization.

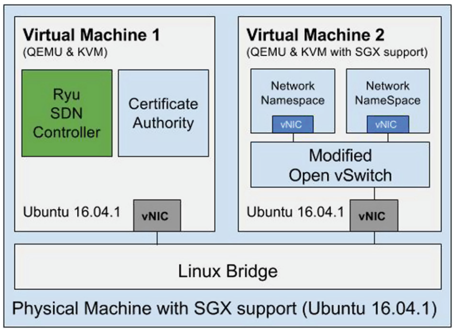

Fig. 3. Testbed architecture

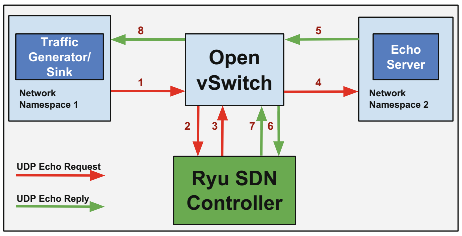

Fig. 4. UDP packet path

We enabled hyper-threading on the host platform, yielding 4 logical CPUs. We pinned VM_1 to CPU 2 and VM_2 to CPUs 1 and 3 (same core). In VM_2, we pinned the virtual switch to CPU 1 and the traffic generator/sink and echo server to CPU 2, in order to reduce inter-core communication overhead [50]. However, due to the limited number of cores on the host (2 cores) we were unable to implement strict CPU isolation by dedicating entire cores. In Sect. 5.3 we discuss the potential implications of this.

Software. We used OvS release 2.6.0[4]. In VM_2, we deployed OvS binaries compiled and linked with our TLSonSGX (as explained in Sect. 3.2). We created two network namespaces, each with a port connected to the OvS instance.

The CA uses OpenSSL 1.1.0d for TLS communication with OvS and to sign the OvS and the SDN controller certificates. We used OpenSSL, rather than TLSonSGX for the CA implementation for two reasons: (1) the CA implementation is trusted according to the threat model; and (2) to ensure interoperability between TLSonSGX (on the client side) and OpenSSL (on the server side).

We chose the Ryu SDN open-source controller as it supports TLS communication with OpenFlow switches[5]. It is written in Python and is widely used in research [3] and in commercial products[6].

5.2 Evaluation Targets

SDN Controller Program. In the SDN model, the virtual switch forwards the first packet in a new flow to the SDN controller. The controller replies with a flow table update, the action to be executed by the switch to handle the packet, and the packet itself. The virtual switch handles subsequent packets in the flow according to the newly installed rule in the flow table.

[4] Commit `4b27db644a8c8e8d2640f2913cbdfa7e4b78e788`.
[5] See Ryu 4.9 Documentation, https://ryu.readthedocs.io/en/latest/tls.html.
[6] See SmartSDN Controller, https://osrg.github.io/ryu-book/.

To exercise the communication between the SDN controller and the virtual switch and to capture latency measurements, we designed the SDN controller as a learning L2 switch, with a MAC address to port number mapping table. To collect measurements of the controller-induced latency, the SDN controller sends no flow updates to the virtual switch (otherwise we would get one measurement per new destination). As a result, the virtual switch sends all the packets in the flow to the SDN controller and the controller returns the packets to the virtual switch along with the action to send the packet through the corresponding port.

Performance Measurements. We are primarily interested in the latency and the time required to generate key pairs and to obtain a signed certificate from the CA. When it comes to latency, the choice of traffic generators was limited to those that can provide latency measurements. Moreover, such measurements require that clocks of both traffic source and sink are synchronized (or co-exist in the same host). Having investigated several traffic generators (qperf[7], pktgen [37], moongen [16], and Click [35]), we chose Click due to its flexibility and versatility.

We implemented a traffic generator and sink using the Click Modular Router. This allows us to measure round trip latency for UDP packets of varying sizes, at a rate of 500 Packets Per Second (pps) using the Click element `StoreUDPTimeSeqRecord`. Increasing the rate beyond that results in much higher latency variance (see Sect. 5.3).

We deployed the traffic generator and sink in network namespace *(i)* and a UDP echo server in network namespace *(ii)*. The echo server echoes the received UDP packet back to the traffic generator and sink. The two network namespaces communicate through Open vSwitch, as illustrated in Fig. 4. To benchmark the performance, we replicated the measurements in a clone of VM_2, using a vanilla QEMU and KVM, with a default Open vSwitch implementation that uses OpenSSL.

5.3 TLSonSGX Performance Evaluation

Keys and Certificate Generation Time. This measurement concerns the time from `SSL_library_init` invocation in the Open vSwitch until the key pairs and signed certificate are loaded to the enclave's memory. See measurement results in Table 1. There is no corresponding measurement in a vanilla Open vSwitch, since keys and certificates are handled manually [38]. However, as this operation is only executed once when `ovs-vswitchd` starts, the measurements show that there is little *de facto* overhead introduced by the implementation.

Packet Round Trip Latency. In this section we discuss and analyze the packet round trip latency. The measurements do not include the key generation time; likewise, the time to establish a TLS session is not included, as it must already be established before packets can flow. The TLS session remains active unless one of the two ends (Open vSwitch or SDN controller) terminates the session.

[7] See `qperf` man page.

Table 1. Keys and certificate generation time. 1000 measurements.

Mean	0.344 s
Variance	0.0488
1st Quartile	0.186 s
Median	0.276 s
3rd Quartile	0.434 s

Table 2. Packet rate vs. average CPU utilization.

Packet rate	OpenSSL	TLSonSGX
500 pps	25%	61%
1000 pps	40%	78%
2000 pps	49%	96%

Packet Size. The IP packet size received by the Open vSwitch from the traffic generator is bounded by the Maximum Transmission Unit (MTU) of the network namespace port connected to the Open vSwitch (1500 bytes in our tests). Open vSwitch encapsulates the received packet in an OpenFlow `Packet In` message, adding an 18 bytes header [13], that is in return encapsulated in a TLS record sent from the Open vSwitch to the SDN controller. If the packet sent by the traffic generator is larger than the MTU, then it is fragmented and Open vSwitch handles it as two separate `Packet In` messages to the SDN controller.

The TLS record adds a 5-byte header. Depending on the cipher suite negotiated between the server and the client, a padding field (up to 15 bytes) is added, and the TLS record is appended with a Message Authentication Code (MAC) computed over the data. In the handshake messages exchanged between Open vSwitch and the SDN controller in our tests, the negotiated cipher suite was `ECDHE-RSA-AES256-SHA`, which provides perfect forward secrecy through the use of an Elliptic Curve Diffie-Hellman key exchange [9], while the bulk encryption use 256-bit AES in CBC-mode with SHA-1 for MAC [11].

We measure the latency for increasing packet sizes ranging from 64 bytes up to 1408 bytes (in increments of 64 bytes), including the Ethernet and IP headers (minus the Cyclic Redundancy Check). The upper limit is set to avoid subsequent fragmentation between the Open vSwitch and the SDN controller.

Packet Rate Selection and CPU Utilization. We excluded outliers with a round trip latency over 2.5 ms from the captured data: 5237 outliers when testing OpenSSL and 11622 outliers when testing TLSonSGX, out of 220000 samples for each implementation. We investigated the CPU utilization to identify the cause of the outliers and the order-of-magnitude difference in the outlier numbers between the two implementations. In both implementations, inside the VM, the first vCPU reaches 100% utilization due to the Click packet generation process pinned to it, even at rates lower than 500 pps (i.e., 50, 100, 200 pps). However, the second vCPU, where `ovs-vswitchd` process is pinned, has a higher average CPU utilization when TLSonSGX is used compared to OpenSSL (see Table 2). Increasing the rate beyond 500 pps leads to increasing the second vCPU's utilization and average latency. Thus, we chose 500 pps as a suitable and optimal maximum rate for further measurements and analysis. Using SGX causes increased CPU utilization due to the overhead of transitioning to and from the memory enclave.

Latency and Packet Size. The packet round trip latency measurements are plotted in a boxplot comparing TLSonSGX with the vanilla Open vSwitch with OpenSSL when forwarding UDP packets of a range of sizes (outliers were excluded, as stated above). Figure 5 shows a plot of latency versus packet size.

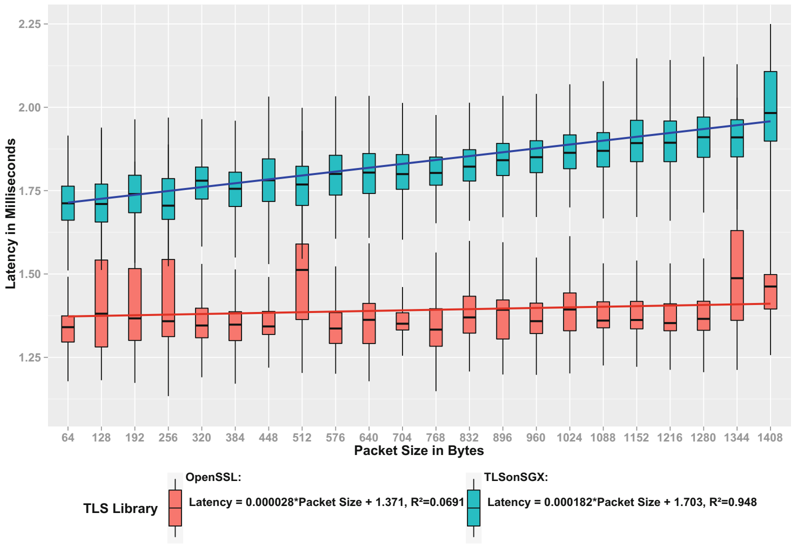

Fig. 5. UDP packet round trip latency vs. packet size

Each box represents the data between first and third quartile, the thick line in the box represents the median. The upper whisker is the minimum value between the data maximum and 3rd Quartile + 1.5 * IQR, where IQR is the interquartile range. The lower whisker is the maximum value between the data minimum and 1st Quartile − 1.5 * IQR [18].

A linear regression analysis of means shows that at zero byte TLSonSGX adds an overhead of 0.33 ms compared to OpenSSL. In implementations the latency increases linearly with packet size; we estimate this increase to 28 ns per byte for OpenSSL, and 182 ns per byte for TLSonSGX. While the linear increase is consistent with our expectations (larger packets require more processing time), the increase per byte is higher in TLSonSGX than in OpenSSL (154 ns per byte). This, and the extra cost of 0.33 ms at zero byte are also expected due to the transition overhead to and from the memory enclave.

Once a packet is received at an Open vSwitch port from the network name space, `ovs-vswitchd` triggers `ecall_ssl_write` to encrypt and send the packet to the SDN controller, while checking the SSL state (`ecall_ssl_get_state`) before and after the write ECALL. Since `ovs-vswitchd` uses non-blocking sockets, `ovs-vswitchd` keeps reading and returning from the socket (`ecall_ssl_read`), while comparing the SSL state before and after the read (`ecall_ssl_get_state`). If a negative value is returned (WANT_READ) from `ecall_ssl_read` then it triggers (`ecall_ssl_get_error`) to retrieve the error code which indicates that the read call must be repeated and accordingly continue the loop. If a positive value is returned, there is a response from the controller. The controller will respond with two packets: (1) the original packet itself; (2) the action needed by the switch to forward the packet to the second network name space. The same flow will run during the return trip from the second network name space to the first one.

Table 3. Analysis of packet latency (all measurements are in milliseconds**)

Size (B)	TLSonSGX	OpenSSL	Diff	`ecall_ssl_`				Total enclave access
				`read`	`write`	`get_state`*	`get_error`*	
64	1.6500	1.2682	0.3817	0.0047	0.0646	0.0047	0.0043	0.2966
128	1.6667	1.2722	0.3944	0.0048	0.0676	0.0047	0.0043	0.3040
256	1.6820	1.2844	0.3976	0.0049	0.0725	0.0047	0.0043	0.3146
512	1.6852	1.2955	0.3897	0.0049	0.0828	0.0047	0.0043	0.3350
1024	1.6963	1.3145	0.3818	0.0049	0.1022	0.0047	0.0043	0.3740

* `ecall_ssl_get_state` and `ecall_ssl_get_error` are independent of packet size.
** Measurements captured in a different iteration than in Fig. 5.

To analyze and break down the time difference between OpenSSL and TLSonSGX, we traced the ECALLs indirectly called by `ovs-vswitchd` during the packet's round trip. We measured the time consumed for each ECALL and repeated the measurement 10000 times per packet size. Table 3 lists the mean values for each of the four different ECALLs. The last column in the table shows the sum of all ECALLs times per packet round trip.

We noticed that the duration of `ecall_ssl_write` is longer (and increases with packet size) than that of other ECALLs. This is because `ecall_ssl_write` is the only ECALL that writes from a buffer with a pointer outside the enclave (unprotected memory) to the enclave memory. All other ECALLs do the opposite. According to the manual, ECALLs that pass an external pointer into the enclave are slow, since a buffer is allocated inside the enclave memory[8]. Before copying the contents of the external buffer into the enclave memory, the content and the size of the buffer referenced by the external pointer are verified for every call to prevent overwriting enclave code or data.

Recall from the system model (consistent with a typical SDN deployment) that only the first packet in the flow is sent to the SDN controller. As a result, crafting a small enough first packet (64 bytes) allows to optimize the latency and reduce the time to add the flow rule in the Open vSwitch flow table.

[8] Pointer Handling, Intel® Software Guard Extensions SDK, https://software.intel.com/en-us/node/708975.

Table 4. Attestation time in application plane for various stages of the attestation sequence. Stages with execution time <0.010 s removed.

Stage	Mean	Variance	Median
TPM quote	0.332 s	0.000159	0.335 s
Key generation	0.326 s	0.050746	0.266 s
CSR signing	0.011 s	0.000002	0.010 s
Total attestation time	0.686 s	0.050849	0.622 s

5.4 Application Plane Evaluation

In the application plane, we are mostly interested in performance measurement regarding the attestation time. Every time a container is launched, both the container itself and the host it is running on must be attested. In this section, we focus on measuring the attestation time for the proposed application plane design. There are of course other relevant performance aspects, such as time required for the actual TLS connection to the controller, but we refer to previous work for such measurements [19].

The benchmarks were made by repeatedly launching the application which triggers the attestation. We ran 1000 tests, and calculated the mean and median values of the total attestation time (see results in Table 4). As seen from the table, the attestation time is well below one second in the average case. Breaking down the execution time to various stages of the attestation, and presenting those with an execution time of $\geq$0.010 s, we see that the majority of the attestation time is spent in two different stages: (1) waiting for the TPM chip to generate the quote, and (2) generating the private key within the enclave. Stage (1) is implemented in the TPM chip itself, while stage (2) depends on the size and type of key generated. A 2048-bit RSA key was used for the measurements presented above. We also note that our current implementation is not optimized, and it may be possible to reduce the execution time even further.

6 Related Work

6.1 Isolating Network Elements

Protecting the sensitive code and data of network elements is a topic of active on-going research. Jacquin proposed an architecture that used a hardware root of trust to remotely attest the integrity of virtualization hosts in SDN infrastructure [26]. Furthermore, commodity TEEs were used in case studies on securing network applications [29,52], implemented using OpenSGX, an emulator of SGX [27]. TruSDN is a framework for bootstrapping trust in an SDN infrastructure implemented using OpenSGX [43]. It supports secure provisioning of switches in SGX enclaves, a secure communication channel between switches and SDN controller, and secure communication between endpoints in the network using session keys that are generated per flow and used only during the lifetime of the flow. Similarly, *Trusted Click* [14] explores the feasibility of performing network processing in SGX enclaves.

SCONE enables operators to protect confidentiality and integrity of computation in application containers against an adversary with root access to the container host [4]. SCONE achieves this by deploying containers within SGX enclaves and relies on a `libc` library ported to the SGX environment to reduce performance impact of context switches between SGX enclaves and the underlying OS, at the cost of expanding the TCB.

Our solution addresses both confidentiality of long-term credentials and session keys, as well as integrity of the network element platform. In particular, we enable network elements on remotely attested hosts to protect their communication with the network controller using a TLS library and credentials in a local SGX enclave. This allows us to protect core assets with insignificant performance overhead and minimal changes to network element implementations. Porting entire applications into SGX enclaves - as proposed in the related work above - expands the attack surface to both software vulnerabilities and side-channel attacks. We avoid this by only porting to the enclaves a minimal TCB of the network elements. We reduce the TCB by only confining the TLSonSGX library, credentials, and TLS session information to the enclave.

6.2 Enrolling Network Elements

Incomplete or incorrect network views are an attack vector in SDN deployments [40]. The Secure Network Bootstrapping Infrastructure (SNBI) protocol [39] bootstraps secure communication channels of network elements and controllers and provisions the keys required for secure communication. To enable connectivity to the network devices, SNBI assigns unique IPv6 addresses (based on the unique device identifier) or and bootstraps devices with the required keys. However, the SNBI protocol is not resistant against impersonation attacks on network elements and fails to specify a protocol for software network elements with similar security features. We address the shortcomings of SNBI by attesting the integrity of the trusted computing base of the platforms hosting network elements prior to provisioning authentication credentials; the credentials are stored in a secure enclave and as described in Sect. 4.1, never leave the enclave.

6.3 TLS Implementations for SGX

There are several known TLS libraries ported to SGX enclaves. TaLoS [5] terminates TLS communication inside the container enclave by providing a port of LibreSSL library into SGX and thus maintaining OpenSSL API, including APIs to set private keys and certificates from outside the enclave. In this paper, keys and certificates are maintained inside the enclave and no APIs are exposed to manipulate them. Furthermore, TaLoS was not available at the time of writing.

Initially, mbed TLS was the only available port of a TLS library into SGX in Linux [31]. Intel$^{®}$ [24] and wolfSSL [59] provided a port to Linux in May

2017 and June 2017 respectively. However, none of these three provided an unmodified OpenSSL API that is exposed outside the enclave. Thus, none of the TLS libraries for SGX enclaves expose the required functionality. We implemented TLSonSGX to address the lack of usable implementations. TLSonSGX implements a wrapper around mbed TLS Trusted SGX library that exposes the OpenSSL APIs (that are needed for Open vSwitch TLS operations) outside the enclave.

Popular TLS libraries with support for execution in SGX enclaves (OpenSSL, GnuTLS, mbed TLS, WolfSSL, LibreSSL) are vulnerable to Bleichenbacher attacks [10] and a modified version padding oracle attacks [58] on branch level, cache line level and page level [60]. Such attacks can be mitigated by using the Diffie-Hellman (DH) key exchange instead of RSA-based key exchanges and *Authenticated Encryption with Associated Data* (AEAD) mode for encryption [60]. TLSonSGX is compatible with the mitigation suggested in [60] and can be configured to enforce DH key exchanges and AEAD encryption mode.

7 Limitations and Future Work

We implemented a prototype and tested it using one dual-core laptop and used VMs with SGX support to host the virtual switches, the SDN controller, and network namespaces (See in Sect. 5.1). While this sufficient to demonstrate the feasibility of TLSonSGX and compare it to OpenSSL, the platform choice limited possible performance measurements. Dedicated multi-core platforms, or cloud resources, with SGX support could be used to refine the performance measurements.

The current implementation supports only one virtual switch connecting multiple VMs per physical host, as only one SSL context is created and kept inside the enclave. This can be improved by introducing support for multiple switches per host by extending the library to support multiple SSL contexts. TLSonSGX could also be extended to protect the flow table or OVS database content from tampering by storing them in the enclave.

For keys and certificates to survive host reboots, the enclave could deploy sealing mechanisms to seal the enclave, i.e. encrypt it, export it from the enclave, and store it on the local hard disk. We did not prioritize this, as generating new keys and obtaining a new certificate takes approximately 0.3 s (See Sect. 5.3).

8 Conclusion

Protecting network elements on the data and application planes is essential for the security of SDN deployments and the network isolation between tenants. However, both state of art network elements and the underlying platforms are vulnerable to software attacks, potentially exposing authentication credentials stored in plaintext. To address this, we implement the TLSonSGX library that provides a secure and scalable mechanism for network elements to generate keys and obtain signed certificates, while keeping them secure within a memory enclave. TLSonSGX confines all the TLS connections to the SDN controller within the enclave to ensure that keys, certificates, and session data remain inaccessible outside the enclave. We complement TLSonSGX with additional mechanisms to asses the network element trustworthyness and apply the approach on both data- and application planes.

Our evaluation results show that TLSonSGX does not significantly impact the time to generate credentials and only adds an insignificant overhead when processing the first packet in each flow. TLSonSGX reduces the TLS configuration overhead and improves the security of SDN deployments.

Acknowledgements. This research was conducted within the 5G-ENSURE and COLA projects and received funding from the European Union's Horizon 2020 research and innovation programme, under grant agreements No. 671562 and 731574.

References

1. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP 2013, p. 10. ACM, June 2013
2. Antikainen, M., Aura, T., Särelä, M.: Spook in your network: attacking an SDN with a compromised OpenFlow switch. In: Bernsmed, K., Fischer-Hübner, S. (eds.) NordSec 2014. LNCS, vol. 8788, pp. 229–244. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11599-3_14
3. Arbettu, R.K., Khondoker, R., Bayarou, K., Weber, F.: Security analysis of OpenDaylight, ONOS, Rosemary and Ryu SDN controllers. In: 2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks), pp. 37–44, September 2016
4. Arnautov, S., et al.: SCONE: secure Linux containers with Intel SGX. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI 2016, pp. 689–703. USENIX, November 2016
5. Aublin, P.L., et al.: TaLoS: secure and transparent TLS termination inside SGX enclaves. Technical report 2017/5, Imperial College London, March 2017
6. Ball, T., et al.: VeriCon: towards verifying controller programs in software-defined networks. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 282–293. ACM, June 2014
7. Berde, P., et al.: ONOS: towards an open, distributed SDN OS. In: Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking, HotSDN 2014, pp. 1–6. ACM, August 2014
8. Bifulco, R., Boite, J., Bouet, M., Schneider, F.: Improving SDN with InSPired switches. In: Proceedings of the Symposium on SDN Research, SOSR 2016, pp. 1–12. ACM, March 2016
9. Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Moeller, B.: The open vSwitch database management protocol. RFC 4492, IETF, May 2006. http://www.rfc-editor.org/rfc/rfc4492.txt
10. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 1–12. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055716

11. Chown, P.: Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS). RFC 3268, IETF, May 2002. http://www.rfc-editor.org/rfc/rfc3268.txt
12. Coker, G., et al.: Principles of remote attestation. Int. J. Inf. Secur. **10**(2), 63–81 (2011)
13. OpenFlow Switch Consortium: OpenFlow switch specification, v. 1.5.1. Technical report, ONF TS-025, Open Networking Foundation, March 2015
14. 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, SDN-NFVSec 2017, pp. 31–36. ACM, March 2017
15. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, IETF, August 2008. http://www.rfc-editor.org/rfc/rfc3268.txt
16. Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., Carle, G.: MoonGen: a scriptable high-speed packet generator. In: Proceedings of the 2015 Internet Measurement Conference, IMC 2015, pp. 275–287. ACM, New York (2015)
17. Enns, R., Bjorklund, M., Schoenwaelder, J.: Network configuration protocol (NETCONF). RFC 6241, IETF, June 2011. http://www.rfc-editor.org/rfc/rfc6241.txt
18. Frigge, M., Hoaglin, D.C., Iglewicz, B.: Some implementations of the Boxplot. Am. Stat. **43**(1), 50–54 (1989). http://www.jstor.org/stable/2685173
19. Girtler, D., Paladi, N.: Component integrity guarantees in software-defined networking infrastructure. In: Proceedings of the 2017 IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2017, pp. 292–296, November 2017
20. Group Specification: Network Functions Virtualisation (NFV), Architectural Framework, v. 1.1.1. Technical report, GS NFV 002, European Telecommunications Standards Institute, October 2013
21. Hu, H., Han, W., Ahn, G.J., Zhao, Z.: FLOWGUARD: building robust firewalls for software-defined networks. In: Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking, HotSDN 2014, pp. 97–102. ACM, August 2014
22. IBM Corp.: TrouSerS: The open-source TCG Software Stack. http://trousers.sourceforge.net/. Accessed 13 Apr 2018
23. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. Technical report, 325462–063US, Intel Inc., July 2017
24. Intel Corp.: Intel SGX SSL. https://github.com/01org/intel-sgx-ssl. Accessed 20 July 2017
25. Izard, R.: Floodlight REST API. https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Floodlight+REST+API. Accessed 16 Dec 2016
26. Jacquin, L., Shaw, A.L., Dalton, C.: Towards trusted software-defined networks using a hardware-based integrity measurement architecture. In: Proceedings of the 1st IEEE Conference on Network Softwarization, NetSoft 2015, pp. 1–6, April 2015
27. Jain, P., et al.: OpenSGX: an open platform for SGX research. In: Proceedings of the 2016 Network and Distributed System Security Symposium, NDSS 2016. Internet Society, February 2016
28. Kim, S., Han, J., Ha, J., Kim, T., Han, D.: Enhancing security and privacy of Tor's ecosystem by using trusted execution environments. In: 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, pp. 145–161. USENIX (2017)

29. Kim, S., Shin, Y., Ha, J., Kim, T., Han, D.: A first step towards leveraging commodity trusted execution environments for network applications. In: Proceedings of the 14th ACM Workshop on Hot Topics in Networks, HotNets-XIV, pp. 7:1–7:7. ACM, November 2015
30. Kreutz, D., Ramos, F., Verissimo, P.: Towards secure and dependable software-defined networks. In: Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, pp. 55–60. ACM, August 2013
31. mbedTLS: TLS for SGX: a port of mbedTLS. https://github.com/bl4ck5un/mbedtls-SGX. Accessed 23 Apr 2018
32. McKeen, F., et al.: Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave. In: Proceedings of the 2016 Hardware and Architectural Support for Security and Privacy, HASP 2016, pp. 10:1–10:9. ACM, June 2016
33. McKeen, F., et al.: Innovative instructions and software model for isolated execution. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP 2013, p. 10:1. ACM, June 2013
34. McKeown, N., et al.: OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. **38**, 69–74 (2008)
35. Morris, R., Kohler, E., Jannotti, J., Kaashoek, M.F.: The click modular router. ACM Trans. Comput. Syst. **18**(3), 263–297 (2000)
36. Nyman, T., Ekberg, J.E., Asokan, N.: Citizen electronic identities using TPM 2.0. In: Proceedings of the 4th International Workshop on Trustworthy Embedded Devices, TrustED 2014, pp. 37–48. ACM (2014)
37. Olsson, R.: Pktgen the Linux packet generator. In: Proceedings of the Linux Symposium, Ottawa, Canada, pp. 11–24, May 2005
38. Open vSwitch: Open vSwitch Manual. https://github.com/openvswitch/ovs/blob/master/INSTALL.SSL.rst. Accessed 10 Nov 2017
39. OpenDaylight Community: Secure Network Bootstrapping Infrastructure, October 2017. http://docs.opendaylight.org/en/stable-boron/user-guide/snbi-user-guide.html. Accessed Oct 2017
40. Paladi, N., Gehrmann, C.: Towards secure multi-tenant virtualized networks. In: 2015 IEEE TrustCom/BigDataSE/ISPA, vol. 1, pp. 1180–1185, August 2015
41. Paladi, N.: Towards secure SDN policy management. In: Proceedings of the 8th International Conference on Utility and Cloud Computing, UCC 2015, pp. 607–611, December 2015. https://doi.org/10.1109/UCC.2015.106
42. Paladi, N.: Trust but verify: trust establishment mechanisms in infrastructure clouds. Ph.D. thesis, Department of Electrical Engineering, Lund University, September 2017
43. Paladi, N., Gehrmann, C.: TruSDN: bootstrapping trust in cloud network infrastructure. In: Deng, R., Weng, J., Ren, K., Yegneswaran, V. (eds.) SecureComm 2016. LNICST, vol. 198, pp. 104–124. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59608-2_6
44. Paladi, N., Karlsson, L.: Safeguarding VNF credentials with Intel SGX. In: Proceedings of the SIGCOMM Posters and Demos, SIGCOMM Posters and Demos 2017, pp. 144–146. ACM, August 2017
45. Parno, B.: Bootstrapping trust in a "trusted" platform. In: Proceedings of the 3rd Conference on Hot Topics in Security, HOTSEC 2008, pp. 9:1–9:6. USENIX, July 2008
46. Pfaff, B., Davie, B.: The open vSwitch database management protocol. RFC 7047, IETF, December 2013. http://www.rfc-editor.org/rfc/rfc7047.txt

47. Pfaff, B., et al.: The design and implementation of open vSwitch. In: Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015, pp. 117–130. USENIX, May 2015
48. Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., Gu, G.: A security enforcement kernel for OpenFlow networks. In: Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks, HotSDN 2012, pp. 121–126. ACM, August 2012
49. Scott-Hayward, S., Natarajan, S., Sezer, S.: A survey of security in software defined networks. IEEE Comm. Surv. Tutor. **18**, 623–654 (2015)
50. Sekar, V., Egi, N., Ratnasamy, S., Reiter, M.K., Shi, G.: Design and implementation of a consolidated middlebox architecture. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, p. 24. USENIX Association (2012)
51. Sherwood, R., et al.: Carving research slices out of your production networks with OpenFlow. ACM SIGCOMM Comput. Commun. Rev. **40**, 129–130 (2010)
52. 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, SDN-NFV Security 2016, pp. 45–48. ACM, March 2016
53. Shin, S., Porras, P.A., Yegneswaran, V., Fong, M.W., Gu, G., Tyson, M.: FRESCO: modular composable security services for software-defined networks. In: Proceedings of the 20th Annual Network & Distributed System Security Symposium, NDSS 2013. Internet Society, February 2013
54. Telecommunication Standardization Sector of ITU: Security requirements and reference architecture for software-defined networking. Technical report, X.1038, International Telecommunications Union, October 2016
55. Thimmaraju, K., et al.: The vAMP attack: taking control of cloud systems via the unified packet parser. In: Proceedings of the 2017 on Cloud Computing Security Workshop, CCSW 2017, pp. 11–15. ACM, New York (2017)
56. Trusted Computing Group: TPM Main Specification Level 2 Version 1.2, Revision 116. Parts 1–3. Technical report, 116_01032011, Trusted Computing Group Inc., March 2011
57. Trusted Computing Group: Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.16. Technical report, 120_01102013, Trusted Computing Group Inc., October 2014
58. Vaudenay, S.: Security flaws induced by CBC padding—applications to SSL, IPSEC, WTLS. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 534–545. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46035-7_35
59. WolfSSL: wolfSSL with Intel SGX on Linux. https://www.wolfssl.com/wolfSSL/Blog/Entries/2017/6/14_wolfSSL_with_Intel_SGX_on_Linux.html. Accessed 20 July 2017
60. Xiao, Y., Li, M., Chen, S., Zhang, Y.: Stacco: differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. arXiv preprint arXiv:1707.03473 (2017)
61. Xing, B.C., Shanahan, M., Leslie-Hurd, R.: Intel software guard extensions (Intel SGX) software support for dynamic memory allocation inside an enclave. In: Proceedings of the 2016 Hardware and Architectural Support for Security and Privacy, HASP 2016, pp. 11:1–11:9. ACM, June 2016
62. Zhu, S.Y., Scott-Hayward, S., Jacquin, L., Hill, R.: Guide to Security in SDN and NFV, 1st edn. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-64653-4