TruSDN: Bootstrapping Trust in Cloud Network Infrastructure

Nicolae Paladi, Christian Gehrmann

SICS Swedish ICT
Stockholm, Sweden
{nicolae, chrisg}@sics.se

Summary. Software-Defined Networking (SDN) is a novel architectural model for cloud network infrastructure, improving resource utilization, scalability and administration. SDN deployments increasingly rely on virtual switches executing on commodity platforms. However, such vulnerable commodity operating systems with large code bases are prime targets for adversaries attacking the infrastructure. We describe and implement TruSDN, a framework for bootstrapping trust in SDN infrastructure using Intel Software Guard Extensions (SGX), allowing to securely deploy SDN components and protect communication between network endpoints. We introduce *ephemeral flow-specific session keys* and a novel defense against *cuckoo attacks* on SGX enclaves. TruSDN is secure under a powerful adversary model, with a minor performance overhead.

Key words: Software Defined Networking, trust, integrity, virtual switches

1 Introduction

Renewed and widespread interest in virtualization – along with proliferation of cloud computing – has spurred a series of innovations, allowing cloud service providers to deliver on-demand compute, storage and network resources for highly dynamic workloads. Consequently, more hardware and virtual components are added to already large networks, complicating network management. To help address this, SDN emerged as a novel network architecture model. Separation of the *data* and *control* planes is its core principle, allowing network operators to implement high-level configuration goals by interacting with a single *network controller*, rather than configuring discrete network components. The controller applies the configuration to the *network edge*, i.e. its global view of the data plane [11]. Data and control plane separation in SDN challenges network infrastructure security best practices evolved in the decades since packet-switched digital network communication gained popularity [16], [22].

In the cloud infrastructure model, SDN allows tenants to configure complex topologies with rich network functionality, managed by a network controller. The availability of a global view of the data plane enables advanced controller capabilities – from pre-calculating optimized traffic routing to managing applications that replace hardware middleboxes. However, these capabilities also make the

controller a valuable attack target: once compromised, it yields the adversary complete control over the network [27]. The global view itself is security sensitive: an adversary capable of impersonating network components may distort a controller's global view and influence network-wide routing policies [13].

Virtual switches are another category of security sensitive components in SDN deployments. They execute on commodity operating systems (OS) and are often assigned the same trust level and privileges as *hardware switches* – specialized network components with compact embedded software [28] or application-specific integrated circuits. Commodity OS are likely to contain security flaws which can be exploited to compromise virtual switches. For example, their configuration can be modified to disobey the protocol, breach network isolation and reroute traffic to a malicious destination or compromise other network edge elements through lateral attacks. Such risks are accentuated by the extensive control a cloud provider has over the infrastructure of its tenants.

Security and isolation of tenant infrastructure can be strengthened by confining select SDN components to trusted execution environments (TEE) and attesting their integrity before provisioning security-sensitive data. TEEs with strong security guarantees can be built using SGX, a set of recently introduced extensions to the x86 instruction set architecture and related hardware [1, 18]. Earlier work used SGX to protect computation in cloud environments, by executing modified OS instances in SGX enclaves [2] or a data processing framework in a set of SGX enclaves [32]. However, while both of the above efforts *highlighted* the need to secure network communication, they did not *address* it.

1.1 Contribution

This paper makes the following contributions:

– We present TruSDN, a framework to bootstrap trust in SDN infrastructure.
– We introduce ephemeral flow session keys for communication protection.
– We propose a defense against cuckoo attacks [23], based on properties of the enhanced privacy ID (EPID) scheme [4] used for remote enclave attestation.
– We describe the implementation and a performance evaluation of TruSDN.

1.2 Organization

We introduce the system model in Section 2, describe the adversary model in Section 3 and the design of TruSDN in Section 4. In Section 5 we provide a security analysis, describe the prototype implementation and performance evaluation in Section 6 and review the related work in Section 7. We discuss future work in Section 8 and conclude in Section 9.

2 System Model

In this section we describe the SDN architectural model and the SDN deployment layers. Furthermore, we describe the use of TEEs based on Intel SGX.

2.1 Software Defined Networking

In this paper we target SDN in infrastructure cloud deployments. The system model follows the architecture presented in [5] and depicted in Figure 1.

The *data plane* includes hardware and software switch implementations. *Software switching* is used in cloud deployments due to its scalability and configuration flexibility. Figure 2 illustrates the software switching approaches for inter-virtual machine (VM) communication. In a typical switch implementation, its kernel-space component is optimized for forwarding performance, lacks decision logic and only forwards packets matching rules in its *forwarding information base* (FIB) [20]. The FIB comprises packet forwarding rules deployed to satisfy network administrator goals. Mismatching rules are discarded or redirected to the *control plane* through the *southbound API*. While the data plane uses complementary functionality of both virtual and physical switches, the role of the latter is often reduced to routing IP-tunneled traffic between hypervisors [25]. We do *not* address control of hardware switches and traffic routing between hosts; we assume that the physical network provides uniform capacity across hosts, based on e.g. equal-cost multi-path routing [14], such that if multiple equal-cost routes to the same destination exist, they can be discovered and used to provide load balancing among redundant paths. Overlay networks – e.g. VLANs or GRE [10] – are used for inter-VM communication. We focus exclusively on software switching and use the term "switch" to denote a virtual, software implementation. We refer to hardware switch implementations as "hardware switches".

In the *control plane*, high-level network operator goals are translated into discrete routing policies based on the *global network view*, i.e. a graph representation of the virtual network topology. The main component of a control plane is the *network controller*, which we define as follows:

Definition 1. *Network Controller (NC) is a logically centralized component that manages network communication in a given deployment by updating the FIB with specific forwarding rules. The NC compiles forwarding rules based on three inputs: the dynamic global network view, the high-level configuration goals of the network operator, and the output of the network management applications.*

The NC is typically implemented as part of a logically centralized *network OS*, which builds and maintains the global network view and may include a *net-*
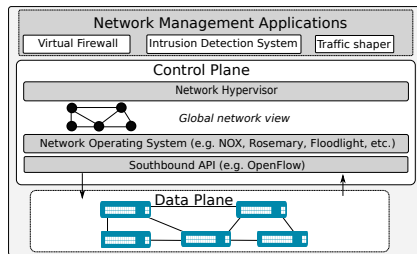

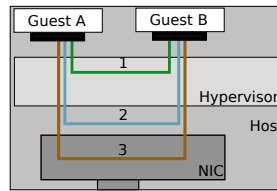**Fig. 1.** The SDN architectural model.

Fig. 2. Inter-VM communication paths: (1) virtual switch; (2) host-local, e.g. native bridging; (3) virtual queues in the NIC.

work hypervisor, to multiplex network resources among distinct virtual network deployments.

Southbound API is a set of vendor-agnostic instructions for communication between data and control planes. it is often limited to flow-based traffic control of the data plane, with management done through a configuration database [25].

Network operators use *network management applications* (NMAs), e.g. firewalls, traffic shapers, etc., to configure the network using high-level commands.

2.2 Deployment layers

We next describe the deployment layers of an SDN infrastructure (Figure 3).

The *hardware layer* includes infrastructure for data transfer, processing and storage and is comprised of network hardware (including hardware switches and communication channels), hardware server platforms and data storage.

The *infrastructure layer* includes software components for virtualization and resource provisioning to *tenants*. For network resources, this layer includes the network hypervisor, which creates *network slices* by multiplexing physical network infrastructure between tenants. Infrastructure providers expose a *slice* (i.e. a quota) of network resources to infrastructure users, referred to as tenants.

The *service layer* includes components controlled by tenants. Network components operated by tenants are grouped into *network domains*, comprising the virtual network resources and topologies that logically belong to the same organizational unit and network slice, and perform related tasks or provide a common service. The *network hypervisor* ensures that a tenant's control plane can only control switches in its own slice. Within their slice, tenants have exhaustive creation, destruction and configuration privileges over components, such as instances of switches, the NC, NMAs and network domains. We define three *logical* communication segments (Figure 4): between the network controller and switches (α segments); among the switches on each host (β segments); between host-local switches and network endpoints (γ segments).

The *user layer* includes endpoint consumers of network services, e.g. VM guests, containers and applications in a network domain.
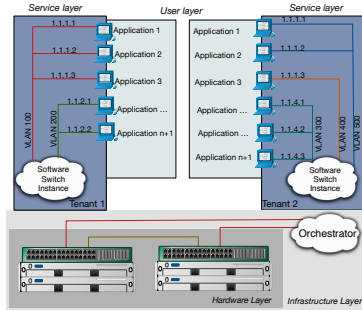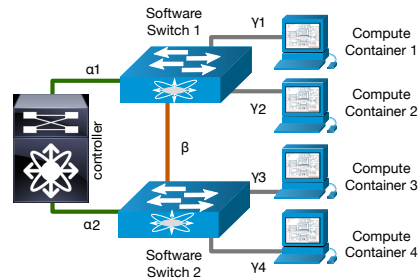

Fig. 3. Deployment Layers.

Fig. 4. Logical communication segments.

2.3 Trusted Execution Environments

Our system model relies on the presence of TEEs. A TEE can be created using Intel SGX enclaves during OS runtime and relies for its security on a trusted computing base (TCB) of code and data loaded at build time, processor firmware and processor hardware. At build time, the CPU measures the loaded code, data and memory page layout. At initialization time, the CPU produces a final measurement, after which the enclave becomes immutable and cannot be externally modified. The CPU maintains the measurement throughout the enclave's lifetime to later assert the integrity of the enclave contents. Processor firmware is the root of trust (ROT) of an enclave. It prevents access to the memory segment of enclave by either the platform OS, other enclaves, or other external agents. Enclaves operate in a separate memory region inaccessible to non-enclave processes, called the enclave page cache (EPC). Multiple mutually distrusting enclaves can operate on the platform. The processor enforces separation of memory access among enclaves based on the layout in the *EPC map*. Program execution within an enclave is transparent to both the underlying OS and other enclaves.

Remote attestation allows an enclave to provide integrity guarantees of its contents [1]. For this, the platform produces an attestation assertion with information about the identity of the enclave and details of non-measureable state (e.g. the mode of the software environment, associated data, and a cryptographic binding to the platform TCB making the assertion). For *intra-platform attestation* (i.e. between enclaves on the same platform), the reporting enclave (*reporter*) invokes the `EREPORT` instruction to create a `REPORT` structure with the assertion and calculate a message authentication code (MAC), using a *report key*, known only to the target enclave (*target*) and the CPU. The structure contains a `user data` field, where the reporter can store a hash of the auxiliary data provided. The target recomputes the MAC with its report key to verify the authenticity of the structure, and compares the hash in the `user data` with the hash of the auxiliary data, to verify its integrity. Enclaves then use the auxiliary data to establish a secure communication channel. For *inter-platform attestation* the remote verifier first sends a challenge to the enclave platform, where the challenge is complemented with the indentity of a *quoting enclave* (QE) and forwarded to the reporter, which appends the challenge response to the `REPORT` and attests itself to the QE. The QE verifies the structure, signs it with a platform-specific key using the *enhanced privacy ID group signature scheme* (EPID) [4] and returns it to the verifier, to check the authenticity of the signature and the report itself [1].

3 Adversary Model

We now describe the adopted adversary model, as well as the core security assumptions on which we base our design. The adversary model we adopt can be described by the capabilities of the adversary at the *network* and *platform* levels respectively (overview in Table 1).

3.1 Network infrastructure

For SDN infrastructure, we adopt the adversary model introduced in [7] and extended with SDN-specific attack vectors in [22]. We assume a powerful adversary ($\mathcal{A}dv$), which controls the cloud deployment network infrastructure; it can intercept, record, forge, drop and replay any message on the network, and is only limited by the constraints of the employed cryptographic methods. Particularly, the $\mathcal{A}dv$ may use forged messages with unmatched packets to induce the NC to update the FIB. Furthermore the $\mathcal{A}dv$ may create own instances of switches and launch Sybil attacks [8] and launch other types of topology poisoning attacks [13] to distort the global network view. Finally, $\mathcal{A}dv$ can store arbitrary quantities of intercepted communication and attempt its decryption with encryption keys intercepted or leaked at a later point. It can analyze the traffic patterns in the network through passive probing and may disrupt or degrade network connectivity to achieve its goals. We explicitly exclude Denial-of-Service attacks on SDN infrastructure.

3.2 Platform

For platform security, we consider a powerful adversary, similar to [2, 32], that may control the entire software stack in the cloud provider's infrastructure.

On the hardware level, we assume the processor is correctly implemented and remains uncompromised; furthermore, we assume a reliable and secure source of random numbers (which can be provided by the CPU). $\mathcal{A}dv$ has full control over the remaining hardware, including memory, I/O devices, periferials, etc. Similarly, $\mathcal{A}dv$ fully controls the software stack, including the platform OS and the hypervisor. This implies that $\mathcal{A}dv$ may pause indefinitely the execution of the code in the TEE and return arbitrary values in response to OS system calls. We exclude side-channel attacks. While some side-channel attacks – e.g. timing, cache-collision, controlled channel attacks – can be mitigated through software modification [35], preventing other side-channel attacks – such as power analysis – requires hardware modifications. An $\mathcal{A}dv$ with advanced capabilities

Table 1. Summary of the $\mathcal{A}dv$ capabilities in relation to the adversary model.

Type	Network	Platform
Included	Intercept, record, forge, drop, replay messages Analyze the traffic patterns; Disrupt or degrade network connectivity Launch topology poisoning attacks	Control non-processor hardware Control software stack OS, hypervisor Pause execution; Deploy arbitrary software components "Cuckoo attack": Forward function calls to compromised SGX enclaves
Not included, mitigations known		Side-channels: cache-collision, controlled channel Attacks on shielded execution Return arbitrary values to system calls
Excplicitly excluded	Denial-of-Service (DoS) attacks	Side-channels: power analysis; DoS attacks

may leverage its full control over the OS to utilize the class of known attacks on shielded execution; while we do not address such attacks, they have known countermeasures [6, 2].

SGX, similar to other trusted computing solutions, is vulnerable to *cuckoo attacks* [23]. In one attack scenario, malware on the target platform forwards the messages intended for the *local* SGX enclave (SGX_L^E) to a remote enclave under Adv's physical control (*malicious* enclave, SGX_M^E). Having physical access to SGX_M^E, Adv can apply hardware attacks to violate its security guarantees. As a result, Adv controls all communication between the verifier and SGX_L^E, with access to an oracle that provides all of the answers a benign SGX^E would, but without its expected security properties.

Briefly, the adversary model for platform security largely matches the remote administrator capabilities of an infrastructure cloud provider.

4 Solution Description

In this section we present TruSDN, a framework for bootstrapping trust in SDN deployments. Its goal is to allow tenants to securely deploy computing tasks and create virtualized network infrastructure deployments, given the adversary model and sample attacks defined in Section 3. To satisfy this goal, the framework must satisfy the following set of requirements:

– *Authentication:* communication in the domain must the authenticated, and a secure enrollment mechanism for data plane components must be in place.
– *Topology integrity:* the NC must be protected from network components that attempt to distort the global network view.
– *Component integrity:* integrity of switches must be verified prior to enrollment and the cryptographic material required for their network access must be protected with a hardware ROT.
– *Confidentiality protection of domain secrets:* network domain secrets – such as VPN session keys – should not be revealed in plaintext to the Adv.
– *Protected network communication:* network communication in the tenant domain must be confidentiality and integrity protected.

4.1 TruSDN overview

We begin by introducing the building blocks of TruSDN (Figure 5).

Trusted Execution Environments: TruSDN uses TEEs that guarantee secure execution in the given adversary model, assuming the CPU and executed code are correctly implemented.

Protected Compute Tasks: Security sensitive compute tasks (CT) are deployed in TEEs. Such tasks include all operations that tenants aim to protect from the Adv. However, CTs rely on the untrusted OS for I/O and support functionality.

Protected Data Plane: Switches, the main components of the tenant network data plane, are deployed in TEEs – they route traffic between CTs according to forwarding rules communicated through secure channels and maintained in the FIB. Switches, their FIB, and the key material necessary to establish the secure channels are stored in TEEs.

Attested code in TEEs: An orchestrator under tenant control attests the TEEs during network infrastructure deployment, to ensure integrity of the deployed code and data before keys or key material are provisioned to the respective TEE.

In a typical deployment scenario, the tenant invokes an orchestrator to deploy a switch *bootstrap application* on the hosts in the tenant's domain. The bootstrap application invokes a host-local SGX driver to build an SGX enclave containing a switch, and attests it (as described in Section 2.3) prior to enrolling the switch with the NC. The orchestrator uses the enclave's public key from the attestation quote to securely transfer the enclave-specific integrity and confidentiality protection session keys used to establish a protected communication channel between the NC and the TEE. Finally, the NC communicates any remaining security-sensitive payload to the created TEE, e.g. the initial FIB. Next, CTs are deployed in TEEs on the host and the switch forwards packets between the CTs, matching them against the rules in the FIB. Mismatching rules are forwarded to the NC, which may update the FIB with new rules. For clarity, we assume the orchestrator and NC are collocated on a platform under tenant control and view both as a single component, further referred to as "NC".

Secure Communication: TruSDN protects communication between CTs, as well as between switches and the NC, in the above adversary model. Communication security is ensured using confidentiality and integrity protection keys provisioned to authenticated network components and endpoints executing in TEEs. Furthermore, TruSDN leverages SDN principles to introduce a novel mechanism – flow-specific communication protection using ephemeral pre-shared keys.

4.2 Cryptographic Primitives

We now define the cryptographic primitives and notations used in the remainder of this paper. We denote by $\{0,1\}^n$ the set of all binary strings of length n, and by $\{0,1\}^*$ the set of all finite binary strings. In a set U, we refer to the ith element as u_i, and following notation for cryptographic operations:
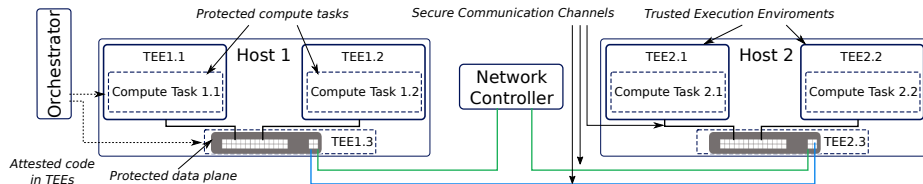
Fig. 5. Illustration of core building blocks of TruSDN.

Table 2. Summary of keys used in the TruSDN framework.

Key	Created by	Access	Usage
K_i^α	NC	NC, switch	Enclave-specific session, segment α
K_j^β	NC	NC , switch	Domain-specific session, segment β
K'	NC	NC, switch	Ephemeral session key
K''	NC	NC, switch	Ephemeral MAC key
EK_i^{pk}	switch	public	Public key of the switch enclave
EK_i^{sk}	switch	switch	Private key of the switch enclave
CK_i^{pk}	CT	public	Public key of the compute task
CK_i^{sk}	CT	CT	Private key of the compute task
QE^{pk}	vendor	public	Public key of the quoting enclave
QE^{sk}	vendor	vendor, QE	Private key of the quoting enclave
SK_{ij}^γ	NC	NC, CT_i, CT_j	Ephemeral flow-specific session key

– Given an arbitrary message $m \in \{0,1\}^*$, we denote by $c = \mathsf{Enc}\,(K,m)$ a symmetric encryption of m using the secret key $K \in \{0,1\}^*$. The corresponding symmetric decryption operation is $m = \mathsf{Dec}(K,c) = \mathsf{Dec}(K, \mathsf{Enc}(K,m))$.

– We denote by pk/sk a public/private key pair for a public key encryption scheme. We denote by $c = \mathsf{Enc}_{\mathsf{pk}}\,(m)$ the encryption of message m with the public key pk, and the decryption by $m = \mathsf{Dec}_{\mathsf{sk}}(c) = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(m))$.

– We denote a digital signature over a message m by $\sigma = \mathsf{Sign}_{\mathsf{sk}}(m)$ and the corresponding verification of a digital signature by $\nu = \mathsf{Verify}_{\mathsf{pk}}(m,\sigma)$, where $\nu = 1$ if the signature is valid and $\nu = 0$ otherwise.

– We denote a Message Authentication Code (MAC) using a secret key K over a message m by $\mu = \mathsf{MAC}(K,m)$.

We next describe key sharing and communication protection mechanisms on the identified logical segments. Table 2 summarizes the keys used by TruSDN.

4.3 SDN Trust Bootstrapping and Secure Communication

The first step in deploying a TruSDN infrastructure is to launch a set of trusted switches for connectivity and topology building. The NC requests creation of switch enclaves to deploy switches in TEEs on hosts in its domain. Switches are deployed based on parameters provided by the NC in plaintext (application code and configuration). Next, the NC *attests* the integrity of switch enclaves and only *enrolls* the successfully attested ones (Figure 6). A TEE E_i is attested following the protocol introduced in [1]. With TruSDN however, the reporter generates an enclave-specific public-private keypair and submit its public key EK_i^{pk} along with the attestation data; a hash of the public key is stored in the `user data` field. The switch enclave is only enrolled to the global network view if its reported state matches the one expected by NC.

Having attested enclave E_i, NC communicates an Enrollment message with the following key material for communication protection: certificate with a signature of EK_i^{pk} ($\mathsf{Cert}_{EK_i^{pk}}$); certificate with a signature of NC^{pk} ($\mathsf{Cert}_{NC^{pk}}$); root certificate of the deployment certificate authority (Cert_{CA}); enclave-specific pre-shared key K_i^α and domain-specific pre-shared key K_j^β, encrypted with an
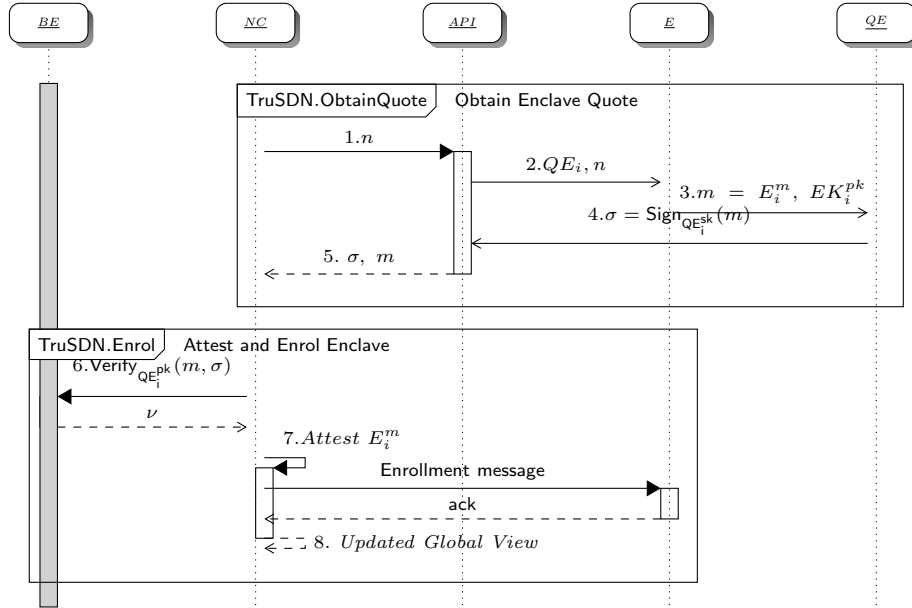
Fig. 6. TruSDN enclave attestation and enrollment;

ephemeral key K_i'. Switches within a domain use K_j^β to protect communication on β segments. The NC appends a MAC of the message calculated using an ephemeral key K_i'' and encrypts the keys K_i', K_i'' with EK_i^{pk} (Table 3).

Once switches are deployed and enrolled, tenants may configure the network topology using the NC to update the switch FIBs. Communication on α segments – e.g. FIB updates or unmatched packets forwarded to the NC – is protected using the session key K_i^α (e.g. using TLS [9]), which never leaves the TEE.

Similarly, a secure channel is established among the switches within the same domain, using the pre-shared key K_j^β, to protect communication between switches on different hosts (e.g. TEEs 1.2 and 2.3 in Figure 5). K_j^β never leaves the TEEs, has a limited validity time and is periodically redeployed by the NC. On β segments, traffic may traverse multiple hardware switches, forwarded to the host over tunnels deployed on top of a standard routing protocol (e.g. [14]).

Next, the tenant may deploy CTs in TEEs and attest their integrity using the very same scheme and principles as for the switch deployment described above. The CTs and the network controller use the Enrollment message to establish a secure communication channel (e.g. TLS).

Table 3. Enrollment message sent by the NC upon switch enrollment.

$\mu = \mathsf{MAC}(K_i'', m)$				
$\mathsf{Cert}_{EK_i^{pk}}$	$\mathsf{Cert}_{NC^{pk}}$	Cert_{CA}	$\mathsf{Enc}(K_i', (K_i^\alpha, K_j^\beta))$	$\mathsf{Enc}(EK_i^{pk}, (K_i', K_i''))$

Once the NC has deployed and attested the TEEs with switches and CTs, intra-host communication (i.e. between two CT enclaves on the same host) is straightforward (Figure 7). When a packet m sent from C1 (e.g. a TLS `ClientHello` message) reaches the local host *switch A*, it attempts to match m against a FIB entry. If no suitable flow rule f is present, the switch forwards $\mathsf{Enc}(K_A^\alpha, m)$ to NC, which processes the packet, generates and deploys on the CTs C1, C2 an ephemeral, flow session key SK_{12}^γ and finally updates the switch FIB with f, after which steps 2 and 3 are ignored. Once the FIB is updated, the switch forwards m to C2, which continues the message exchange and uses SK_{12}^γ to protect the communication with C1, using e.g. PSK TLS [9].

Communication between CTs C1 and C3 deployed on distinct hosts is similar, with the only notable difference that the NC updates the FIB of the local switches on both hosts where C1, C3 are deployed.

In the above scenarios TruSDN leverages two aspects of the SDN model – **(1)** the deployment has a central authority (the NC) and **(2)** the first packet of a flow is forwarded to the central authority – to deliver on demand ephemeral PSKs to communication endpoints. This allows to relax the need for high-quality entropy being available to CTs (a known issue in virtualized environments [29]). Furthermore, this approach ensures communication security without compromising packet visibility – having control over the keys used to protect communication between the CTs allows the NC to maintain fine-grained control over the traffic.

4.4 Preventing Cuckoo Attacks

To prevent cuckoo attacks [23], we propose a solution that leverages cryptographic properties of the EPID group signature scheme used by the QE [4] and the *SIGn and Message Authentication* (SIGMA) protocol [34]. The EPID scheme supports two signature modes: *fully anonymous mode* – the verifier cannot associate a given signature with a particular member of the group; *pseudo-anonymous mode* – the verifier can determine whether it has verified the platform previously. The unlinkability property distinguished in the two modes depends on the chosen *base*. A signature includes a pseudonym B^f, where B is the base chosen for
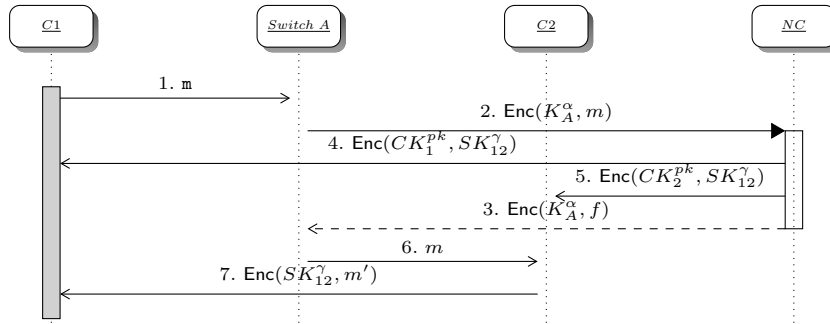

Fig. 7. Intra-host communication with TruSDN.

a signature and revealed during the signature; f is unique per member and private. For a *random base*, the pseudonym is R^f, where R is random – in this case the signatures are unlinkable. For a *name base*, the pseudonym is N^f, where N is the name of the verifier – in this case the signatures remain unlinkable for *different* verifiers, while signatures with a common N can be linked. For privacy reasons, the current implementation of the EPID scheme only accepts *name base* pseudonyms from verifiers authorized by the EPID authority [30]. Authorization is done by issuing to the qualified verifier an X.509 certificate – e.g. a leaf certificate, or an intermediate certification authority (CA) certificate – signed by the EPID authority, which in this context acts as root CA.

We use the following algorithm to prevent cuckoo attacks. At deployment time, the cloud provider requests the EPID authority to issue an intermediate CA verifier certificate, for the set of platforms that are part of the cloud provider's data center and creates the authorized verifier V_P. Next, V_P follows the SIGMA protocol to attest its platforms and publishes a list of resulting platform EPID signatures and the name base chosen for the signature, B_P^N. To guard against cuckoo attacks, tenants first request V_P to sign an X.509 certificate, to themselves become *authorized verifiers*. Next, tenants choose the same pseudonym base B_P^N (and a private f), follow the SIGMA protocol, and verify that the resulting signature *can be linked* to a signature in the published list. The cloud provider has multiple tools to protect platform privacy and prevent untrusted tenants from fingerprinting the platform infrastructure, e.g. limiting the validity of issued certificates, changing the name base and re-creating platform signatures, etc.

5 Security Analysis

In this section we analyze the security properties of the proposed framework in the adversary model described in Section 3.

On the network level, many of the $\mathcal{A}dv$ capabilities are thwarted by first authenticating the switches deployed on the data plane, as well as the network edge (i.e. the compute tasks that generate or receive the network traffic), in combination with confidentiality and integrity protection of the traffic on the three identified segments. Authenticating the network components prevents topology poisoning attacks (a countermeasure mentioned in [13]), while confidentiality and integrity protection of all of the network traffic in the deployment prevents the $\mathcal{A}dv$ from either learning the contents of the exchanged packets or successfully forging packets. The $\mathcal{A}dv$ may in this case still intercept and record messages. However, collecting encrypted traffic does not yield the $\mathcal{A}dv$ any more information about the contents of the exchanged packets. Similarly, the $\mathcal{A}dv$ does not gain an advantage by simply dropping or replaying messages, since these actions would at most simply reduce the channel capacity (as would the ability of the $\mathcal{A}dv$ to disrupt network connectivity). Finally, the proposed framework does not prevent the $\mathcal{A}dv$ analyzing the traffic patterns and does not prevent it from fingerprinting the components of the deployment, making it vulnerable rule scanning and denial of service attacks. While the goals of TruSDN did not

include this, such traffic analysis could be prevented using anti-fingerprinting techniques, as proposed in [3].

On the platform level, the security of the proposed framework relies to a large extent on the security properties of Intel SGX enclaves. This allows to protect the execution of switches and network edge components deployed in TEEs from the capabilities of an *Adv* controlling non-processor hardware, the software stack of the OS and the hypervisor. Similarly, pausing execution of switches executing in TEEs, while possible, would have no further effect than degrading network connectivity, already discussed above. While the *Adv* may attempt to deploy own arbitrary components on the data plane or the network edge in order to launch Sybill attacks, the integrity of such components would not be successfully attested, unless they are identical to legitimate components, which are assumed to be executing correctly – rendering Sybill behavior impossible. The *Adv* is prevented from launching cuckoo attacks by enabling tenants to verify the platforms, as described in Section 4.4. As presented in Table 1, several relevant classes of attacks are not addressed by TruSDN, but have known mitigations, namely cache-collision, controlled channel and attacks on shielded execution (addressed in [35, 32]. The capability of the *Adv* to return arbitrary values to system calls, while not addressed in this work, can be mitigated by a validation component as described in [2].

6 Implementation and Evaluation

We now describe the implementation and evaluation of TruSDN.

6.1 TruSDN Implementation

The TruSDN prototype deployments follows the design presented in Section 4 and is illustrated in Figure 8. *Host 1* and *Host 2* are VM instances of Ubuntu OS 15.04. In each VM, we deployed Linux Containers[1], based on Ubuntu OS 15.04. Containers create an environment with own process and network space, implemented using *namespaces*, with a distinct user ID, network stack, mount points, file systems, processes, inter-process communication, and hostname. We chose containers to facilitate prototype implementation, using their lightweight process isolation. Containers are part of the untrusted OS and this implementation choice is orthogonal to the security of TruSDN. Compute tasks are deployed in TEEs created using SGX enclaves (Figure 8): *enclaves* E1, E2, E4, E5 are placed respectively within containers C1, C2, C3, C4. The switches are deployed in TEEs created using SGX enclaves (enclaves E3, E6 in Figure 8).

Considering that platforms with hardware and software support for SGX were not publicly available at the time of writing, we used *OpenSGX* [15] to emulate the TEEs. It is a software SGX emulator and a platform for SGX development, implemented using binary translation of QEMU and emulating Intel

---

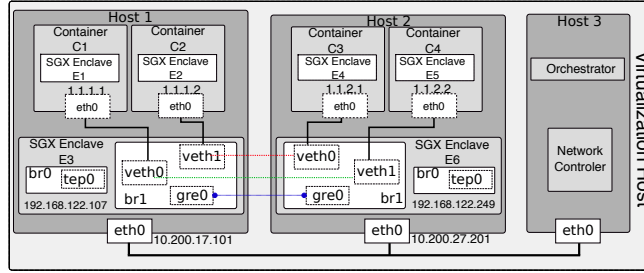[1] Linux Containers Project Website: `https://linuxcontainers.org/`

Fig. 8. Prototype deployment of TruSDN

SGX hardware components at instruction level. It includes emulated hardware and OS components, enclave program loader, OpenSGX user library, debugging and performance monitoring support. The emulator allows to implement, debug, and evaluate SGX applications, but does not support binary compatibility with Intel SGX. Furthermore, OpenSGX does not implement all instructions, e.g. debugging instructions. While OpenSGX does not provide security guarantees, it allows us to obtain performance estimates for the proposed approach. We used $mbedTLS^2$ v1.3.11 (distributed with the emulator) for attestation of the SGX enclaves. We used OpenSSL v1.0.2d (distributed with the emulator) to set up protected communication channels between the CT enclaves and the local switches, and among switches within the same domain.

An SDN network controller is deployed in a third VM (*Host 3*). We used the Ryu^3 SDN framework, due to its flexibility and versatile APIs.

6.2 TruSDN Evaluation

We now analyze the performance impact, present evaluation results and discuss aspects that cannot be measured with the current prototype.

Sources of Performance Impact TruSDN introduces several potential sources of performance impact (Table 4). We distinguish between *transient* performance overhead, which occurs occasionally (e.g. TLS key negotiation) and *continuous* performance overhead, present throughout the deployment operation. We *do not* consider the one-time cost of infrastructure deployment, e.g. provisioning the software, attesting TEEs and enrolling the components.

Measured Performance Impact To evaluate the performance impact, we measured the footprint of establishing TLS sessions on α and γ segments. We used *iperf*, *openssl s_time* and an own Ryu application (Table 5).

---

[2] mbed TLS project website `https://tls.mbed.org/`

[3] Ryu SDN framework: `https://osrg.github.io/ryu/`

Table 4. Sources and types of performance overhead in TruSDN

Source	Type	Clarification
TLS negotiation all segments	transient	Negotiate session keys for TLS
PSK distribution	transient	Distribute PSK for γ segments
TLS protection all segments	continuous	Overhead induced by TLS
Compute task execution in TEEs	continuous	Overhead induced by TEE
Switch execution in TEEs	continuous	Overhead induced by TEE

TLS overhead on the α segment: We measured the round-trip latency of packets sent in plaintext and with TLS, over 1000 tests, each request sending messages of 100 bytes with the 80 bit OpenFlow header. Furthermore, we measured the data transfer rates for plaintext and TLS communication. Use of TLS increased total transfer time by 14.2% and reduced the transfer rate by 15.98%.

Delay on γ segment As mentioned above, the first packet of the flow is intercepted by the switch and forwarded to the NC in a `packet_in` message [24]. At this point the NC processes the flow and installs a flow rule on the switch. TruSDN *extends* this procedure by generating and distributing to the communicating CTs a pre-shared key (PSK), to be used for communication protection. Since this must be done prior to both forwarding the message to the destination CT and installing the flow rule, generating and distributing the PSK would normally delay the installation of the flow rule and increase the latency of the *first* packet (all subsequent packets are forwarded according to the flow rule). To measure the introduced delay, we have sequentially established 1000 TLS sessions between compute tasks C1 and C2 (according to Figure 8). After each TLS session, we flushed the installed flow rules (with `ovs-ofctl del-flows br0`), which resulted in a `packet_in` message upon each new session. The latency of the first packet is shown in Figure 6.2, and compared against the latency of a first packet without the TruSDN extension.

Table 5. Summary of performance evaluation of TruSDN

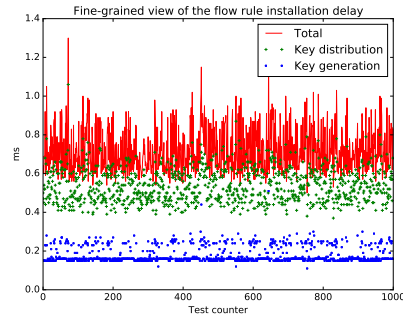| Data	Minimum	Maximum	Mean	Median	Stddev
Total transfer time, ms	0.4	1.1	0.66	0.7	0.07
Total transfer time w. TruSDN, ms	0.5	7.1	0.8	0.8	0.22
TruSDN overhead, **total transfer time**			**21.2%**	**14.2%**	
Transfer rate, bytes per second	1225	2095	1595	1583	98.07
Transfer rate w. TruSDN, bytes per second	919	1589	1338	1330	64.86
TruSDN overhead, **transfer rate**			**16.11%**	**15.98%**	
First packet latency γ	1.53	6.50	3.48	3.38	0.42
First packet latency γ w. TruSDN	3.35	10.7	5.37	5.14	0.93
TruSDN overhead, **first packet latency**			**54.31%**	**52.07%**	
TLS handshake, ms	36.53	77.72	67.97	67.48	7.42
TLS handshake w. TruSDN, ms	52.35	76.44	67.15	66.53	3.93
TruSDN overhead, **TLS handshake**			**-2.21%**	**-2.41%**	
Key generation NC, ms	0.11	0.51	0.178	0.16	0.04
Key distribution γ, ms	0.37	1.06	0.54	0.53	0.08
Key total γ, ms	0.50	1.30	0.71	0.7	0.11

Fig. 9. Coarse-grained view   **Fig. 10.** Fine-grained view

The induced delay is primarily caused by two operations performed by the NC: *generating* a 256-bit PSK and distributing it to the CTs. Figure 6.2 displays a fine-grained picture of the induced delay. Key generation lasted on average 0.178 ms, while key distribution on average 0.54 ms (Table 5). We remind that the test environment is fully virtualized and posit that the measured overhead of key generation can be reduced in a production environment, either by using pre-generated keys or with specialized hardware (e.g. crypto processors). In our tests, the duration of establishing a TLS session with ephemeral flow-specific session keys using the PSK-AES256-CBC-SHA cipher suite was 2.41% *less* compared to the use of e.g. ECDH-RSA-AES128-SHA256. Thus, TruSDN *improves* the performance, by enabling flexible use of pre-shared keys. This reduces the duration of the TLS handshake, by avoiding expensive public key cryptographic operations [17]. Moreover, it allows to reduce the CPU utilization for key derivation for the compute tasks, at the cost of a minimal flow rule installation delay.

Unmeasured Performance Overhead Implementing TEEs with OpenSGX limits the level of detail when it comes to performance evaluation, since: **(a)** the OpenSGX emulator *is not* binary compatible with Intel SGX [15]; **(b)** in its current version[4] and unlike Intel's description of SGX [1], OpenSGX has yet to implement support multithreaded applications[5]. Thus, a fully accurate measurement on TruSDN performance cannot be done until Intel SGX hardware and software is made available. However, our experiments yield a fair picture of the expected performance impact.

7 Related work

In this section we present related work on adversary models for SDN deployments, SDN controller security, as well as principles and software model of SGX.

---

[4] Commit e0713c7 on `https://github.com/sslab-gatech/opensgx`

[5] Issue #34 on `https://github.com/sslab-gatech/opensgx/issues/34`

Adversary models: Kreutz et al. presented a list of attack vectors in SDN [16] (forged traffic flows, vulnerabilities in switches and NCs, lack of trust establishment mechanisms, etc.). However, only part of the described attack vectors are exclusively relevant to SDN networks and no specific solutions are proposed. Work in [22] introduced an adversary model, attack vectors, and security requirements towards multi-tenant SDN infrastructure, highlighting the need to limit the effect of NC vulnerabilities, protect internal SDN communication, verify integrity of SDN components prior to enrollment, and enforce policy and quota isolation. TruSDN addresses several of the attack vectors described in [16, 22].

Secure SDN controllers: The "NOX" network OS [11] presents NMAs with a centralized programming model and a global view, allowing to operate with higher-level abstractions and apply graph processing algorithms to compute paths. It consists of several controller processes operating on the global view; the processes use the global view for network management decisions and update switch FIBs over the OpenFlow API [19]. FortNOX [26] extends NOX with role-based authorization (RBA) and enforcement of security constraints. It translates high-level threats into flow rules to handle suspicious traffic as well as detects rule conflicts, resolves them depending on the authorization of the rule requestor and enforces least privilege authorization. Neither NOX nor FortNOX address malicious network components and Sybill attacks, addressed by TruSDN. "Rosemary" NOS [33] uses NMA sandboxing to improve network resilience, by launching each NMA in a separate process context with access to the required libraries, along with a resource monitor to supervise NMA compliance. However, "Rosemary" does not address data plane security. TruSDN complements it and creates a foundation for trusted deployment of a secure NOS. TopoGuard [13] detects network topology poisoning and mitigates this through port property management, network edge probing and verification of topology updates. TruSDN complements this by verifying the *integrity* of switches prior to enrollment into the topology.

Software Guard Extensions: SGX was introduced in [18] with a description of the software model, extensions to the x86 ISA and hardware modifications for isolated execution; work in [1] described CPU based attestation; finally, several prototypes using SGX to create trustworthy applications are described in [12].

SGX-based solutions in a cloud setting are first described in [2, 32]. "Haven"[2] is a modified version of Windows 8 OS ported to an SGX enclave, evaluated with Apache Web Server and SQL Server using synthetic data sets. It includes a mechanism to protect the enclave from a malicious kernel and a semantically secure data store protecting data and file metadata confidentiality against malicious hosts. TruSDN protects network communication for a similar adversary model. While we deploy compute tasks in SGX enclave-based TEEs, the work in [2] is largely complementary, and similar "Haven"-like OSs could be used.

"VC3" [32] is a Map-Reduce deployment using SGX enclaves. *Map* and *reduce* functions are compiled into private (encrypted) code and public code implementing key exchange and job execution protocols. Code is initialized in enclaves and attested by the users. Public code performs the key exchange, decrypts the pri-

vate code and runs the job execution protocol. To defend against cuckoo attacks, *cloud quoting enclaves* are created on *each* platform in the cloud provider data centers, to "countersign" quotes produced by the QE. The approach is largely complementary to protecting communication between CTs with TruSDN. However, the proposed defense against cuckoo attacks increases the complexity of the attestation protocol and does not prevent $\mathcal{A}dv$ from exploiting a compromised cloud QE outside of the physically secure datacenter perimeter. Instead, the approach described in Section 4.4 leverages the cryptographic properties of EPID scheme, without modifying the attestation protocol.

8 Future Work

Along with security guarantees, the use of Intel SGX imposes certain limitations on TruSDN. Further performance evaluation may be done once software and hardware support for Intel SGX enclaves becomes available; moreover, some security limitations of TruSDN must be mentioned.

Controlled-channel attacks [35] are a novel type of side-channel attacks allowing the OS to extract sensitive information from protected applications. They were successfully applied to "Haven" [2] and TruSDN could also be vulnerable; however, such attacks were explicitly excluded from the adversary model. Known mitigations are: rewriting enclave applications to decouple memory access patterns from sensitive data, prohibiting paging by the underlying OS, or using techniques to obfuscate the memory access patterns [35].

Another limitation stems from the reliance on the platform vendor, which could leak QE^{sk}, to create a "deniable back-door" and allow *person-in-the-middle* attacks on attestation [31]. This challenge remains unaddressed.

We aim to integrate TruSDN with previous work [21], to provide complete cloud infrastructure deployments secure in the given adversarial model.

9 Conclusion

We have presented TruSDN, a framework for bootstrapping trust in SDN infrastructure. It isolates network endpoints and switches in SGX enclaves, remotely attests the integrity of enclaves, and establishes secure communication channels. We leveraged the functioning principles of SDN to introduce *ephemeral flow-specific session keys*: by securely distributing session keys at flow creation time, this reduces overhead of key derivation in compute tasks. Our evaluation shows that using ephemeral flow-specific session keys reduces the total time of establishing protected channels, at the cost of a minor delay in the flow rule installation. Finally, we have presented an improved method to prevent cuckoo attacks, which relies on the properties of the EPID group signature scheme. TruSDN contributes towards creating a complete framework that allows to bootstrap a trusted infrastructure with compute, communication and storage resources that can be practically deployed in public cloud infrastructure.

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. p. 10 (2013)
2. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with Haven. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2014)
3. Bifulco, R., Cui, H., Karame, G.O., Klaedtke, F.: Fingerprinting software-defined networks. In: 2015 IEEE 23rd International Conference on Network Protocols (ICNP). pp. 453–459 (Nov 2015)
4. Brickell, E., Li, J.: Enhanced Privacy ID: A Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities. Dependable and Secure Computing, IEEE Transactions on 9(3), 345–360 (May 2012)
5. Casado, M., Foster, N., Guha, A.: Abstractions for software-defined networks. Communications of the ACM 57(10), 86–95 (2014)
6. Checkoway, S., Shacham, H.: Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. SIGARCH Comput. Archit. News 41(1), 253–264 (Mar 2013), http://doi.acm.org/10.1145/2490301.2451145
7. Dolev, D., Yao, A.C.: On the security of public key protocols. Information Theory, IEEE Transactions on 29(2), 198–208 (1983)
8. Douceur, J.: The Sybil Attack. In: Druschel, P., Kaashoek, F., Rowstron, A. (eds.) Peer-to-Peer Systems, Lecture Notes in Computer Science, vol. 2429, pp. 251–260. Springer Berlin Heidelberg (2002)
9. Eronen, P., Tschofenig, H.: Pre-shared key ciphersuites for transport layer security (TLS). Tech. rep., RFC 4279, December (2005)
10. Farinacci, D., Traina, P., Hanks, S., Li, T.: Generic routing encapsulation (GRE) (1994)
11. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., Shenker, S.: NOX: towards an operating system for networks. ACM SIGCOMM Computer Communication Review 38(3), 105–110 (2008)
12. Hoekstra, M.: Using Innovative Instructions to Create Trustworthy Software Solutions. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. p. 10 (2013)
13. Hong, S., Xu, L., Wang, H., Gu, G.: Poisoning network visibility in software-defined networks: New attacks and countermeasures. NDSS (2015)
14. Hopps, C.: Analysis of an Equal-Cost Multi-Path Algorithm. Tech. rep. (2000)
15. Jain, P., Desai, S., Kim, S., Shih, M.W., Lee, J., Choi, C., Shin, Y., Kim, T., Kang, B.B., Han, D.: OpenSGX: An Open Platform for SGX Research. In: NDSS. San Diego, CA (February 2016)
16. Kreutz, D., Ramos, F., Verissimo, P.: Towards secure and dependable software-defined networks. In: Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. pp. 55–60. ACM (2013)
17. Kuo, F.C., Tschofenig, H., Meyer, F., Fu, X.: Comparison studies between pre-shared and public key exchange mechanisms for transport layer security. In: INFO-COM 2006. 25th IEEE International Conference on Computer Communications. Proceedings. pp. 1–6. IEEE (2006)
18. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative Instructions and Software Model for Isolated Execution. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. pp. 1–1. ACM (2013)

19. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review 38(2), 69–74 (2008)
20. Nadeau, T.D., Gray, K.: SDN: Software Defined Networks. " O'Reilly Media, Inc." (2013)
21. Paladi, N., Gehrmann, C., Michalas, A.: Providing User Security Guarantees in Public Infrastructure Clouds. IEEE Transactions on Cloud Computing PP(99), 1–1 (2016)
22. Paladi, N., Gehrmann, C.: Towards Secure Multi-tenant Virtualized Networks. In: Trustcom/BigDataSE/ISPA, 2015 IEEE. vol. 1, pp. 1180–1185. IEEE (2015)
23. Parno, B.: Bootstrapping trust in a "trusted" platform. In: HotSec (2008)
24. Pfaff, B., Lantz, B., Heller, B.e.a.: Openflow switch specification, version 1.3. 0. Open Networking Foundation (2012)
25. Pfaff, B., Pettit, J., Koponen, T., Jackson, E.J., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al.: The design and implementation of Open vSwitch. In: 12th USENIX Symposium on Networked Systems Design and Implementation (2015)
26. Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., Gu, G.: A security enforcement kernel for OpenFlow networks. In: Proceedings of the first workshop on Hot topics in software defined networks. pp. 121–126. ACM (2012)
27. Porras, P., Cheung, S., Fong, M., Skinner, K., Yegneswaran, V.: Securing the software-defined network control layer. In: Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS), San Diego, California (2015)
28. Qazi, Z.A., Tu, C.C., Chiang, L., Miao, R., Sekar, V., Yu, M.: SIMPLE-fying middlebox policy enforcement using SDN. In: ACM SIGCOMM Computer Communication Review. vol. 43, pp. 27–38. ACM (2013)
29. Ristenpart, T., Yilek, S.: When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In: NDSS (2010)
30. Ruan, X.: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine. Apress, Berkely, CA, USA, 1st edn. (2014)
31. Rutkowska, J.: Thoughts on Intel's upcoming Software Guard Extensions (Part 2) (2013), `http://theinvisiblethings.blogspot.de/2013/09/thoughts-on-intels-upcoming-software.html`, [Online; March 2016]
32. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: VC3: Trustworthy Data Analytics in the Cloud Using SGX. In: Security and Privacy (SP), 2015 IEEE Symposium on. pp. 38–54 (May 2015)
33. Shin, S., Song, Y., Lee, T., Lee, S., Chung, J., Porras, P., Yegneswaran, V., Noh, J., Kang, B.B.: Rosemary: A Robust, Secure, and High-Performance Network Operating System. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 78–89. ACM (2014)
34. Walker, J., Li, J.: Key exchange with anonymous authentication using DAA-SIGMA protocol. In: Trusted Systems, pp. 108–127. Springer (2011)
35. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: Security and Privacy (SP), 2015 IEEE Symposium on. pp. 640–656. IEEE (2015)