SafeBricks: Shielding Network Functions in the Cloud

Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy, *UC Berkeley*

https://www.usenix.org/conference/nsdi18/presentation/poddar

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-939133-01-4

Open access to the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

SafeBricks: Shielding Network Functions in the Cloud

Rishabh Poddar
UC Berkeley

Chang Lan
UC Berkeley

Raluca Ada Popa
UC Berkeley

Sylvia Ratnasamy
UC Berkeley

Abstract

With the advent of network function virtualization (NFV), outsourcing network processing to the cloud is growing in popularity amongst enterprises and organizations. Such outsourcing, however, poses a threat to the security of the client's traffic because the cloud is notoriously susceptible to attacks.

We present SafeBricks, a system that *shields* generic network functions (NFs) from an untrusted cloud. SafeBricks ensures that only encrypted traffic is exposed to the cloud provider, and preserves the integrity of both traffic and the NFs. At the same time, it enables clients to reduce their trust in NF implementations by enforcing least privilege across NFs deployed in a chain. SafeBricks does not require changes to TLS, and safeguards the interests of NF vendors as well by shielding NF code and rulesets from both clients and the cloud. To achieve its aims, SafeBricks leverages a combination of hardware enclaves and language-based enforcement. SafeBricks is practical, and its overheads range between ∼0–15% across applications.

1 Introduction

Modern networks consist of a wide range of appliances that implement advanced network functions beyond merely forwarding packets, such as scanning for security issues (*e.g.*, firewalls, IDSes) or improving performance (*e.g.*, WAN optimizers, web caches). Traditionally, these network functions (or NFs) have been deployed as dedicated hardware devices. In recent years, however, both industry and academia have proposed the replacement of the devices with software implementations running in virtual machines [55, 62], a model called Network Function Virtualization (NFV). Inevitably, the advent of NFV has spurred the growth of a new industry wherein third-parties offer traffic processing capabilities as a cloud service to customers [4, 51, 62, 79]. Such a service model enables enterprises to outsource NFs from their networks entirely to the third-party service, bringing the benefits of cloud computing and reducing costs.

However, outsourcing NFs to the cloud poses new challenges to enterprise networks—security.

Need to protect traffic from the cloud. By allowing the cloud provider to process enterprise traffic, enterprises end up granting to the cloud the ability to see their sensitive traffic and tamper with NF processing. While the cloud itself might be a benign entity, it is vulnerable to hackers [56], subpoenas [24, 45, 75], and insider attacks [10, 54, 77]. This is doubly worrisome because not only does network traffic contain sensitive information, but some NFs are also designed to protect enterprises against intrusions which an attacker could try to disrupt.

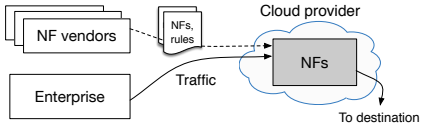

Figure 1: Model for outsourced NFs.

Need to protect traffic from NF. What complicates matters further is that often, an enterprise must also trust another party with its traffic: NF vendors. This is the case when enterprises procure proprietary NF implementations and rulesets from NF vendors [8, 22, 51] instead of using their own, as shown in Figure 1. While such NFs typically need access only to specific portions of the traffic (*e.g.*, IP firewalls only need read access to packet headers), the enterprise by default entrusts the NFs with *both read/write access* over *entire* packets.

Need to protect NF source code. This model threatens the security of the NF vendors as well, who have a business interest in maintaining the privacy of their code and rulesets (often baked into the source code) from both the cloud and the enterprise. NFs have traditionally been shipped as hardware devices, so being shipped as software now exposes them further to untrusted platforms (*e.g.*, it is possible to reverse binaries).

The question is: how can we design an NF processing framework that meets all these security goals?

There has been little prior work in this space, consisting of mostly two approaches. Cryptographic approaches such as BlindBox [63] and Embark [38] are significantly limited in functionality, supporting only simple functions such as $=$ and $>$. They are unable to support more sophisticated operations such as regular expressions (needed in common NFs such as intrusion detection systems) or process custom NF code. Least-privilege approaches such as mcTLS [48] aim to give each NF access to only part of the packet and are designed for *hardware middleboxes*; however, when used in the cloud setting, they provide weak guarantees because the cloud receives the *union* of the permissions of all middleboxes, which often, is everything. Neither of these approaches protects

the NF source code, and both require significant changes to TLS, which is an impediment to adoption.

We present SafeBricks, a system for outsourcing NFs that provides protection with respect to the three security needs above. SafeBricks addresses the discussed limitations of prior work by supporting *generic* NF functionality with significantly stronger security guarantees, without requiring changes to TLS. It builds upon NetBricks [53], a framework for building and executing arbitrary NFs that uses a safe language and runtime, Rust.

To overcome the limited functionality of cryptographic approaches, SafeBricks *shields* [9] traffic processing from the cloud by executing the NFs within *hardware enclaves* (*e.g.*, Intel SGX [43]). This approach promises that neither an administrator with root privileges nor a compromised operating system can observe enclave-protected data in unencrypted form, or tamper with the enclave's execution. Enclaves have already been used to shield *general-purpose* computation from the cloud provider [3, 9, 30, 59]. Applying them to network processing is a natural next step, as recent proposals have pointed out (see §11).

While this idea is simple, designing a system that provides protection with respect to the three security goals above, and simultaneously maintains good performance, is far more challenging.

First, general-purpose approaches result in a large trusted computing base (TCB) inside the enclaves (up to millions of LoC), any vulnerability in which can result in information leakage. In SafeBricks, we investigate *how to partition* the code stack of NF applications (from packet capture to processing) and choose a boundary that reduces the code within the trusted domain without compromising security.

Second, partitioning an application is likely to result in *transitions* between enclave and non-enclave code. These transitions are expensive, introducing a high runtime overhead due to the cost of saving/restoring the state of the secure environment. Consequently, there is a tension between TCB size and the overall performance of the application: the lesser code the enclave contains, the more transitions it is likely to make to non-enclave code. In SafeBricks, we address these challenges simultaneously by developing an architecture that leverages shared memory and splits computation across enclave and non-enclave threads (while verifying the work of the non-enclave threads) without performing transitions.

Third, NFV deployments typically comprise multiple NFs running in a chain, isolated via VMs or containers for safety. In our setting, the straightforward way of achieving this isolation would be to deploy each NF in a separate enclave. However, as we discuss in §6, such an architecture can result in a system that is ∼2–16× slower than the baseline. Instead, SafeBricks supports chains of NFs within the *same* enclave. To isolate them, SafeBricks leverages the semantics of the Rust language.

Nevertheless, this strategy introduces a new difficulty: all NFs must be assembled using a trusted compiler. Though the client enterprise is the natural site for building the NFs safely, doing so would leak the source code of the NFs to the client, which is undesirable for the NF vendors. To address this challenge, SafeBricks runs in an enclave at the cloud a *meta*-functionality: a compiler that creates an encrypted binary, and a loader that runs this binary in a separate enclave. Using the remote attestation feature of hardware enclaves, both the NF vendors and the client can verify that the agreed-upon compiler and loader are running in an enclave, before the vendors share the NF code and the client shares data and traffic.

Finally, none of the above satisfies our requirement for enforcing least privilege across NFs: each NF still has access to entire packets. SafeBricks enforces least privilege by (i) exposing an API to the client for specifying the privileges of each NF, and (ii) ensuring that the SafeBricks framework *mediates* all NF accesses to packets, both reads and writes. To enforce the latter, SafeBricks leverages the safety guarantees of Rust.

We evaluate SafeBricks across four different NF applications using both synthetic and real traffic. Our evaluation shows that the performance impact of SafeBricks is reasonable, ranging between ∼0–15% across NFs.

2 Model and Threat Model

As shown in Figure 1, there are four types of parties in our setting: (1) a *cloud provider* that hosts the outsourced NFs; (2) a *client enterprise* outsourcing its traffic processing to the cloud; (3) *two endpoints* that communicate over the network, at least one of which is within the enterprise; and (4) *NF vendors* that supply the code and rulesets for network functions.

The client enterprise contains a gateway (as shown in Figure 2) which is trusted. The endpoints are trusted only with their communication.

The core of SafeBricks's design builds on the abstract notion of a hardware enclave. Our implementation uses Intel SGX [43], a popular hardware enclave, but few design decisions are tailored to SGX. We provide some relevant background on hardware enclaves, and then define the threat models for the cloud and the NF vendors.

2.1 Hardware enclaves

Hardware enclaves aim to provide an isolated execution environment that preserves the confidentiality and integrity of code and data within the enclave. An important feature of hardware enclaves is remote attestation.

Remote attestation. This procedure allows a remote client system to cryptographically verify that specific software has been securely loaded into an enclave, us-

ing CPU-based attestation [2]. When a client requests remote attestation, the enclave generates a report signed by the processor that contains a hash measurement of the enclave. As part of the attestation, the enclave can also bootstrap a secure channel with the client by generating a public key and returning it with the signed report.

Intel SGX. Intel Software Guard Extensions [43] is a set of ISA extensions that enables the creation of hardware enclaves. Software running outside the enclave, including privileged software such as the kernel or hypervisor, cannot access or tamper with enclave memory.

2.2 Threat model for the cloud and enclaves

Our threat model for the cloud provider is similar to prior works [3,9,59] that build on hardware enclaves. Enclaves strive to provide an abstract security guarantee so that systems like SafeBricks can build on them in a black-box manner; however, current implementations do not yet fully achieve this guarantee as we discuss below.

Abstract enclave assumption. The attacker cannot observe any information about the protected code and data in the enclave, and the remote attestation procedure establishes a secure connection between the correct parties and loads the desired code into the enclave.

Attacker capabilities. Except the out-of-scope attacks described below, we consider an attacker that can compromise the software stack of the cloud provider outside the enclave, which includes privileged software such as the hypervisor and kernel. In particular, whenever the enclave exits or invokes code outside the enclave, the attacker can instead run arbitrary code and/or respond with arbitrary data to the enclave. For example, the OS can mount an Iago attack [15] and respond incorrectly to system calls. Note that this threat model implies that the attacker can observe communication between hardware enclaves as well as communication on the network.

Out-of-scope attacks. In short, all attacks that violate the abstract enclave assumption above are out of scope for SafeBricks. For example, we consider as out of scope all hardware and side-channel attacks, as well as assume that the enclave manufacturer (*e.g.*, Intel) is trusted. Intel SGX's current implementation does not fully achieve the enclave assumption above because it suffers from side-channel attacks, including those based on access pattern leakage amongst others [12, 14, 17, 26, 28, 39, 47, 60, 73, 74]. While these are important issues with SGX, we treat them as out of scope for SafeBricks because solutions to these are orthogonal and complementary to our contribution here. Recently, a number of solutions have been proposed for solving or mitigating these attacks [16, 18, 27, 65, 66].

2.3 Threat model for network functions

Each NF is trusted only with the permissions given to it by the enterprise for specific packet fields. That is, if the enterprise gives a NAT read/write permissions for the IP header, the NF is trusted to not leak the header to unauthorized entities and to modify it correctly. At the same time, if the NAT attempts to access the packet payload, then SafeBricks must prevent it from doing so.

3 SafeBricks: End-to-end Architecture

APLOMB [62] discusses in detail the architecture for outsourcing NF processing to the cloud by redirecting client traffic, as well as the merits of this architecture. Here, we focus on how SafeBricks enhances this architecture with protection against cloud attackers and TLS compatibility, while maintaining performance.

SafeBricks supports three typical architectures considered in the cloud outsourcing model [38, 62], as shown in Figure 2. These architectures have different merits or constraints, and are useful for different cases.

Let S be the source endpoint, G the client gateway, CP the cloud provider running NFs using SafeBricks (SB), and D the destination endpoint. Let G_1 be the gateway near the source, and G_2 be the gateway near the destination. Note that in the Direct architecture, an enclave in the cloud plays the role of G_2, and in the Bounce architecture, a single gateway plays both G_1 and G_2. CP runs hardware enclaves; code and data are decrypted inside enclaves, but remain encrypted outside. D could either be an external site or an endpoint in another enterprise.

1. **Bounce:** In the bounce architecture, SB tunnels processed traffic to G over the secure channel. G then forwards the processed traffic to the destination. The response from D is similarly redirected by G to SB before forwarding it to S. The bounce setup is the simplest in that it does not place any added burden on SB or D from a functionality and security perspective. However, it inflates the latency between S and D as a result of bouncing the processed traffic to G.
2. **Direct:** The direct architecture alleviates the latency added by the bounce setup. SB directly forwards the enterprise traffic to D after processing it without bouncing it off the gateway. However, this setup comes at the cost of security: since there is no secure channel between SB and D over which traffic can be tunneled, SB must necessarily send the processed packets to D in the clear, revealing the headers to CP. If S and D use TLS, CP will not see the payload.
3. **Enterprise-to-enterprise:** If S and D belong to the same enterprise or to enterprises that trust each other, it is possible to have the combined benefits of the bounce and direct architecture. SB tunnels the processed traffic to G_2, so CP does not see any headers at any time. At the same time, this approach does not suffer from the bounce setup's latency.

Though not the focus of this work, it is worth mentioning that SafeBricks can also be used in a local cloud

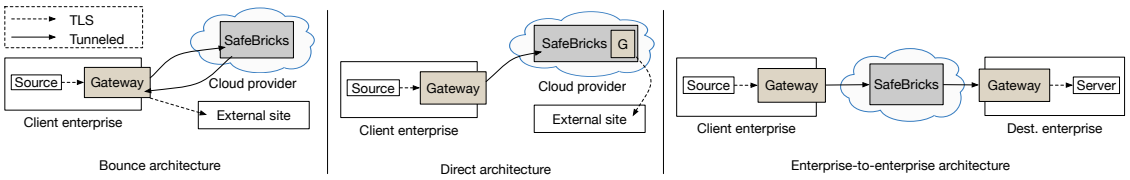

Figure 2: End-to-end system architecture

deployment in which the NFs run within the client enterprise. This benefits the client by providing SafeBricks's isolation and least privilege for NFs, as well as protection against administrators of the local cloud (although the gateway administrators need to remain trusted).

3.1 Overview of the communication protocol

Our protocol for handling connections is the same for all architectures, as we now explain in terms of G_1 and G_2.

System bootstrap. The client enterprise first sets up and verifies the enclaves in the cloud as explained in §7. As part of this process, the gateways are able to set up a set of IPSec tunnels with the cloud in a secure way (such as installing certificates to avoid the risk of a man-in-the-middle attack). To load-balance flows at the cloud server via receive-side scaling (RSS), the number of IPSec tunnels depends on the number of ports at the server.

As with all such interception systems, the source endpoints need to be configured to allow interception. The most common approach is to use an *interception proxy* [34], in which the sources' browsers accept certificates from the proxy which can now terminate the TLS connection. Another approach is to install a browser plugin at the client endpoints, which sends the TLS session keys to the gateway [29] over a secure channel. SafeBricks supports both these approaches.

Upon a new TLS connection from a source. G_1 terminates the connection as described above and informs G_2, which starts the TLS connection to the destination.

Packet processing. G_1 intercepts the TLS traffic from S, decrypts it, and tunnels it over an IPSec connection. Packets from the same flow are sent on the same IPSec connection. Note that as part of this process, the entire packet (including the header) is encrypted and encapsulated in a new header. We use AES in GCM mode as the IPSec encryption algorithm, which includes packet authentication. *SB* receives the packets, decrypts, and processes them. It then tunnels the packets over IPSec to G_2. G_2 terminates the IPSec tunnel and forwards the traffic over TLS to the destination server.

4 Background

Before delving into the design of SafeBricks, we provide a brief overview of NetBricks and some additional details on Intel SGX relevant to our system.

4.1 Intel SGX

Illegal enclave instructions. SGX does not allow instructions within an enclave that result in a change of privilege levels (*e.g.*, system calls) or cause a `VMEXIT`. Applications that need to perform such instructions must exit the enclave and transfer control to host software.

Memory architecture. Enclave pages reside in a protected memory region called the enclave page cache (EPC), whose size is limited to ∼94MB in current hardware. EPC pages are decrypted when loaded into cache lines, and integrity-protected when swapped to DRAM.

4.2 NetBricks

The NetBricks framework [53] enables the development of arbitrary NFs by exposing a small set of customizable programming abstractions (or operators) to developers. In this respect, NetBricks is similar to Click [37], which also enables developers to write NFs by composing various packet processing elements. However, we choose to build our system atop NetBricks instead of Click for the following reasons:

- Unlike Click, the behavior of NetBricks' operators can be heavily customized via user-defined functions (UDFs). This allows us to protect a small number of operators within the enclave (with NetBricks), which are then composed into NFs, as opposed to routinely adding new Click modules.
- More importantly, NetBricks builds upon a safe language and runtime, Rust, to provide isolation between NFs chained together in the same process. In §6, we describe how SafeBricks extends these guarantees to provide least privilege across NFs inexpensively.
- NetBricks' zero-copy semantics also improve performance substantially [53].

We now briefly describe some features of NetBricks relevant to the design of our system.

Programming abstractions. To construct an NF, the developer specifies a directed graph consisting of NetBricks' operators as nodes. For example, the `parse` operator casts packet buffers into protocol structures; `transform` modifies packet buffers; and `filter` drops packets based on a UDF. All nodes in the NF graph process packets in batches.

Execution environment. The NetBricks scheduler implements policies to decide the order in which different nodes process their packets. Chains of NFs are run in a single process by composing their directed graphs to-

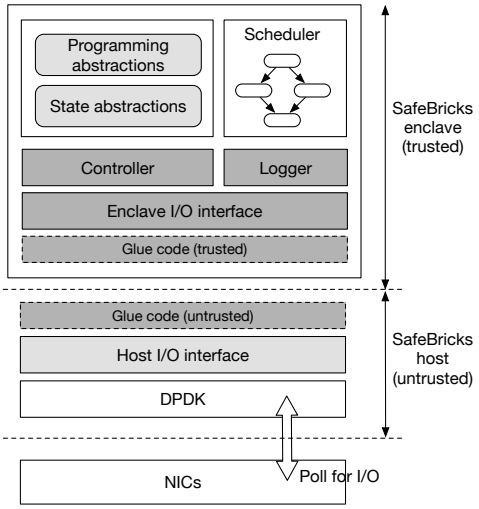

Figure 3: SafeBricks framework: White boxes denote existing NetBricks components, light grey boxes denote modified components, and dark grey boxes denote new components.

gether as function calls, instead of running each NF separately in a container or VM. For isolation between NFs, NetBricks relies on a safe language and runtime, Rust.

Packet I/O. NetBricks builds on top of DPDK [32], a fast packet I/O library. DPDK polls packets from the network devices, buffers them in pools of memory, and maintains a queue of pointers to the packet buffers. NF instances query DPDK via an I/O interface to retrieve pointers to the next batch of packet buffers, and process them in-place without performing any copies.

5 SafeBricks: Framework Design

We now describe how we build our system on top of NetBricks (while redesigning some parts of it). Figure 3 shows the overall design of the framework, highlighting the components modified or introduced by SafeBricks. Our goal in this section is to reduce the size of the TCB while minimizing the overhead of transitions between the enclave and the host. However, these two goals are often at odds with each other—the lesser code the enclave contains, the more transitions it makes to outside code. We now describe how our design balances both these aims.

5.1 Partitioning NetBricks

We carefully split NetBricks into two components—enclave code and host code.

SafeBricks enclave. At a bare minimum, the enclave should include the programming and state abstractions of NetBricks. However, during execution, the NetBricks scheduler takes decisions regarding which node to process next in the directed graph representing the NF (as described in §4.2). These decisions are frequent—every time a node is done processing a batch of packets, it surrenders control to the scheduler. As a result, excluding the scheduler from the TCB would result in a large number of enclave transitions per packet batch. Hence, we include the scheduler in our TCB as well.

SafeBricks host. The remaining components of NetBricks (mostly pertaining to packet I/O) together form the SafeBricks host. As described in §4.2, NFs in NetBricks directly access the packet buffers allocated by the packet capture library (DPDK) without copying them. Simply excluding DPDK from the enclave without other modifications is not a viable option because it would gain access to the packets once they are decrypted. On the other hand, including DPDK within the enclave would drastically inflate the size of the TCB by ∼516K LoC.

We circumvent this issue by introducing two new operators in NetBricks: `toEnclave` and `toHost`. The `toEnclave` operator polls the I/O interface for pointers to packet buffers, reads the encrypted buffers from DPDK-allocated memory and decrypts them inside the enclave. Once the processing is complete, the `toHost` operator re-encrypts the packet buffers and returns them outside the enclave into DPDK's memory pool.

More concretely, `toEnclave` and `toHost` implement endpoints of the IPSec tunnel. As a result, even if the host attacker attempts Iago attacks [15] such as modifying packet buffers or queues outside the enclave, these will be detected by the authenticity provided by IPSec.

Excluding DPDK from the TCB enables us to remove NetBricks' I/O module from the TCB as well. The module interfaces with the packet capture library and is used by the NFs to poll DPDK for packets (Figure 3).

5.2 Packet I/O avoiding enclave transitions

Every receive or send operation for a batch of packets results in an invocation of the I/O interface. Since we exclude the packet capture library from the TCB, every such invocation necessarily results in an enclave transition. Batch processing of packets alleviates the overhead of these transitions to some extent, but as we show in §9.2.1, it is far from being a perfect solution.

Prior works [3, 50] have also explored the reduction of enclave transitions, albeit in a different context—they allow enclave threads to delegate system calls to the host with the help of shared queues. In a similar spirit, we propose an alternative design point that allows enclave code to receive and send packet batches from the host via shared memory, without the need for enclave transitions. To do so, we (i) introduce an additional trusted I/O module within the enclave (called `EnclaveIO`) that exposes the I/O APIs transparently to the rest of enclave code, and (ii) modify the NetBricks I/O interface outside the enclave (`HostIO`) to appropriately interface with the `EnclaveIO` module.

SafeBricks allocates two lockless circular queues (`recvq` and `sendq`) on heap memory outside the enclave during the application's initialization, one for re-

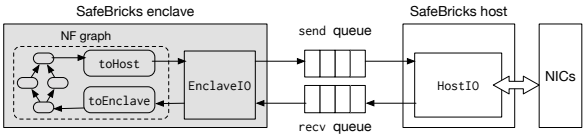

Figure 4: Packet I/O via shared memory.

ceiving pointers to packet buffers and the other for sending. `HostIO` busy polls DPDK for incoming packets and populates `recvq` with the buffer addresses. Enclave code queries the `EnclaveIO` module which in turn reads the packet buffer addresses directly from `recvq` without having to exit the enclave. To send packets, `EnclaveIO` pushes the packet buffer addresses into `sendq`. `HostIO` consumes the buffers asynchronously from this queue, and finally invokes the I/O interface to emit the packets to the network. Figure 4 illustrates the approach.

This mechanism doesn't result in any enclave transitions because (i) enclave code can readily access memory outside the enclave, and (ii) the queue management is asynchronous—the `HostIO` module and the SafeBricks enclave (containing `EnclaveIO`) run in separate threads.

5.3 System calls and other illegal instructions

As described in §4.1, SGX allows neither system calls within enclaves nor instructions that could lead to a `VMEXIT` (such as `rdtsc`, used for reading the timestamp counter). There exist a set of general-purpose systems [3, 9, 30, 50, 67, 72] that add support for such system calls to enclave applications, at the expense of added complexity and/or a significant increase in TCB.

We note that many NFs simply do not make system calls or execute instructions that require VM exits, and those made are typically only of a few types: such as I/O for maintaining logs, or timestamp measurements using `rdtsc`. For example, no application in the NetBricks or Bess source trees [11, 49] implements system calls. This is due to the high-performance goal of NFs, aiming to run exclusively in user-space [32, 33, 52, 57, 70]. The same extends to user-space implementations of networking stacks as well, which are gaining in popularity [20, 35, 40, 41, 46]. Therefore, instead of exposing an exhaustive API within the enclave for these instructions, SafeBricks focuses only on the operations essential for NFs and executes them without the need for enclave transitions. SafeBricks does not expose any other system calls or illegal instructions that would require enclave exits to NFs within the enclave.

Logging. Instead of allowing NFs to write to files, we expose a new state abstraction in SafeBricks that enables them to directly push logs to queues allocated in heap memory outside the enclave (similar to how we perform packet I/O). During system initialization, the Logger module allocates a queue in non-enclave heap per NF that logs information. NFs can push log entries to the respective queue by invoking the Logger module. Host code asynchronously reads the logs off these queues and writes them to files.

However, since this heap memory is untrusted and visible outside the enclave, we need to take additional steps to ensure the security of the logs (as they contain sensitive packet information). We encrypt and chain together log entries via authentication tags, a fairly standard technique. The Logger module encrypts each log item L_i as $C_i = \mathsf{Enc}(\texttt{id}||L_i)$, where `id` identifies the NF. It then computes the authentication tag $T_i = \mathsf{Auth}(C_i||T_{i-1})$, and pushes (C_i, T_i) to the log queue. Including the previous tag in the computation ensures that host code cannot arbitrarily drop or reorder log items. The Logger module maintains the root authentication tag within the enclave. Verifiers can later validate the log by obtaining the latest tag from the enclave over a secure channel and replaying the log. We note that by itself, the approach doesn't prevent rollback attacks on the logs; however, techniques for avoiding such attacks exist and can be deployed in a complementary fashion [69].

Timestamps. SafeBricks relies on the `HostIO` module to capture the timestamp per incoming packet batch and write it to a slot in the packet buffer reserved for external metadata. NFs that need timestamps for their functionality simply read it off the packets. This approach also reduces latency when chains of NFs are deployed together, as the cost of measuring the timestamp is borne only once. Though it is possible to ensure the monotonicity of timestamps, SafeBricks does not guarantee that the timestamps are correct—this is unavoidable in the current SGX implementation as the reporting module is not trusted hardware.

5.4 Execution model

SafeBricks runs the NFs in a multi-threaded enclave, each enclave thread affinitized to a core. We note that our shared memory mechanism for packet I/O adds extra burden on system resources compared to vanilla NetBricks, as it requires an extra thread for running the `HostIO` module. This cost, however, gets amortized by mapping a single `HostIO` instance to multiple enclave threads.

6 SafeBricks: NF Isolation, Least Privilege

SafeBricks gives enterprises the flexibility to source NFs from different vendors and deploy them together on the same platform, while isolating them from each other and controlling which parts of a packet each NF is able to read or write. For example, consider a chained NF configuration wherein traffic is first passed through a firewall, then a DPI, and finally a NAT. The firewall application only needs read access to packet headers; the DPI needs read access to headers and payload; while the NAT needs read and write access to packet headers.

SafeBricks ensures that each NF is given only the minimum level of access to each packet as required for their functions, *e.g.*, the firewall is unable to write to packet headers, or read/write to the payload. In other words, SafeBricks isolates NFs from one another while enforcing the principle of *least privilege* amongst them.

6.1 Strawman scheme

The importance of least privilege access to traffic has been recognized before in mcTLS [48], which relies on physical isolation of NFs and enforces least privilege by encrypting and authenticating each field of the packet separately using different keys. Each NF is given the keys only for fields that it needs access to. To allow read access, the NF is given the encryption keys; for writes, the NF is given the authentication keys as well. Packets are re-encrypted before being transferred from one NF to the other. In the mcTLS model, NFs are isolated by virtue of being deployed on separate systems (hardware or VMs). Correspondingly in our setting, it suffices to run each NF concurrently in a separate enclave isolating their address spaces, as shown on the left of Figure 5.

Such an approach, however, eliminates much of the performance benefits of the underlying NetBricks framework. In addition to adding significant overheads due to repeated re-encryption of packets, it requires packets to cross core boundaries between NF enclaves (for enclaves affinitized to separate cores). Together, this can result in a system that is up to ∼2–16× slower (as we show in §9.2.3). Instead, it would be ideal to keep all NFs in the same enclave and isolate them efficiently within.

6.2 NF isolation in NetBricks

Before describing how SafeBricks enforces least privilege across NFs, we revisit crucial properties of the Rust language that form the basis of our design.

The NetBricks framework provides isolation between NFs running in the same address space by building on a safe language, Rust [7, 53]. Rust's type system and runtime provide four properties crucial for memory isolation: (i) they check bounds on array accesses, (ii) prohibit pointer arithmetic, (iii) prohibit accesses to null objects, and (iv) disallow unsafe type casts.

In addition to memory isolation, NFs also require packet isolation; *i.e.*, NFs should not be able to access packets once they've been forwarded. NetBricks relies on Rust's unique types [7,25] to isolate packets. Rust enforces an *ownership model* in which only a unique reference exists for each object in memory. Variables acquire sole ownership of the objects they are bound to. When an object is transferred to a new variable, the original binding is destroyed. Rust also allows variables to temporarily *borrow* objects without destroying the original binding. By harnessing Rust's ownership model, NetBricks ensures that once an NF is done processing a packet, its ownership is transferred to the next NF and the previous NF can no longer access the packet.

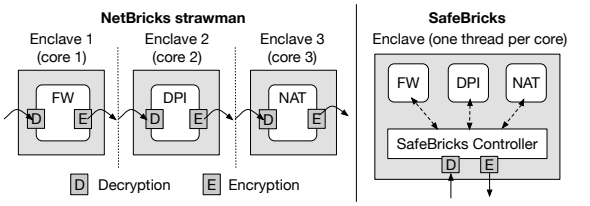

Figure 5: Strawman approach for enforcing least privilege versus SafeBricks. Solid arrows indicates packet transfers. Dotted arrows indicate interaction between NFs and the Controller.

```
pub fn chain<T:'static + Node>(input:T, pmap:HashMap) -> Node {
   let input = input.toEnclave()
      .wList(pmap.get('firewall'));
   let mut chain = firewall(input)
      .wList(pmap.get('dpi'));
   chain = dpi(chain)
      .wList(pmap.get('nat'));
   return nat(chain)
      .toHost();                                              }
```

Figure 6: Code for chaining NFs together (firewall, DPI, and NAT), generated automatically by SafeBricks from a configuration file. Lines in magenta represent code added by SafeBricks over and above NetBricks to enforce least privilege across NFs.

Taken together, the properties of NetBricks suffice for the purpose of running NFs safely within the same address space. However, they do not provide the desired security, as we explain next.

6.3 Isolating NFs within the same enclave

The properties of NetBricks do not satisfy the requirements of our threat model for the following reasons:

- The isolation guarantees only hold if NFs are built using a compiler that *enforces* the safety properties above. In our model, however, enterprises may source NFs from various vendors that compiled them in their own way and lack incentive to enforce these properties.
- Each NF still receives ownership of *entire packets*, instead of limited read / write access to specific fields.

We now describe how SafeBricks addresses both issues.

6.3.1 Ensuring memory safety

SafeBricks needs to ensure that NFs are built using a compiler that prohibits unsafe operations inside NFs. Instead of trusting NF providers, SafeBricks ensures that a trusted compiler gets access to the raw source code of all the NFs which it can then build in a trusted environment.

This strategy is seemingly in conflict with the confidentiality of NF rules. In §7 we show how SafeBricks performs this compilation such that neither the enterprise nor the cloud learns the source code of the NFs.

6.3.2 Enforcing least privilege

SafeBricks extends NetBricks' memory safety guarantees by interposing on its packet ownership model. Instead of transferring packets across NFs, SafeBricks in-

troduces a Controller module that mediates NF access to packets as depicted in Figure 5 (right).

Controlling access to packets. The Controller holds ownership of packet buffers, and NFs can only *borrow* packet fields (or different fragments of the data buffers) by submitting requests to the Controller. To provide least privilege, each packet in SafeBricks encapsulates a bit vector of *permissions*. Each function in the packet API exposed by the Controller is associated with a bit in the permissions vector. Before lending the NF a reference to the requested field, the Controller checks the corresponding bit in the vector and answers the request only if the bit is set. Otherwise, the call returns an error. Furthermore, by controlling whether an API call returns a *mutable* or an *immutable* reference, the framework also disambiguates read access from writes. Rust's type system ensures that once the NF processing completes, the binding between the reference and the field is destroyed, and any later attempt by the NF to access the field will result in a compilation error.

Setting packet permissions. SafeBricks updates the permissions vector in packets with the help of a new packet processing operator: `wList` (whitelist). Chained NFs are interleaved with invocations of the `wList` operator that applies a given vector of permissions to each packet batch before it's processed by the next NF. Figure 6 illustrates the code for chaining NFs together while enforcing least privilege. In §7, we describe how SafeBricks generates this code automatically using a configuration file supplied by the client enterprise.

We need to fulfill two more requirements for the guarantees to hold: (i) NFs should not be able to alter the permissions vector during execution, and (ii) NFs should not be able to parse packet buffers arbitrarily—for example, an NF that has permissions only for IP headers should not be able to incorrectly parse TCP headers as IP, thereby circumventing the policy. SafeBricks therefore does not expose these operations to NFs. NFs in NetBricks invoke the `parse` operator to cast packet buffers into protocol structures before processing them. In contrast, SafeBricks mandates that packets be parsed as required before being processed by NFs (not shown in Figure 6 for simplicity). In §7, we describe how the SafeBricks loader interleaves NFs with `parse` nodes and stitches them together into a directed graph based on enterprise-supplied configuration data.

Runtime overhead. The permissions vector leverages portions of the packet buffers reserved for metadata, and hence does not lead to any memory allocation overhead. Setting and verifying permissions, however, lead to a small overhead at runtime: setting the permissions vector before each NF via the `wList` operator increases the depth of the NF graph, and verifying the permission adds an extra check as all requests are mediated by the Controller. As we see in §9.2.3, the impact on performance is small for real applications.

7 SafeBricks: System Bootstrap Protocol

We now describe the protocol for bootstrapping the overall system. Instead of compiled binaries, SafeBricks needs access to the raw source code of the NFs from the providers so it can pass them through a trusted compiler, which ensures that NFs do not perform unsafe operations and are confined to least privilege access. The natural strategy is to have the client enterprise compile these binaries and upload them to the cloud, as in prior enclave-based systems such as Haven [9]. However, this approach is problematic in our case because NF code is proprietary and the client enterprise may not see it.

To address this problem, the idea in SafeBricks is to run inside the enclave a *meta*-functionality: the enclave assembles the NFs and *compiles them* using a trusted compiler, and only then starts running the resulting code. The key to why this works is that *both* the client enterprise and the NF vendors can invoke the remote attestation procedure to check that the enclave is running an agreed upon SafeBricks loader and compiler (both being public code). In this way, (i) each NF vendor can ensure that the enclave does not run some bad code that exfiltrates the source code to an attacker, and (ii) the client enterprise makes sure the NF vendor cannot change what processing happens in the enclave. The bootstrap process consists of two phases, assembly and deployment.

7.1 Phase 1: NF assembly

For assembly, SafeBricks uses a special enclave provisioned with two trusted modules—a loader and a compiler—that combine the NFs into a single binary.

Loader. The loader exposes a simple API that allows the client enterprise to specify (i) *encrypted* NF source codes, (ii) optionally, unencrypted NF source codes that might be interspersed with the proprietary encrypted NFs, (iii) a configuration file outlining the placement of each NF in the directed graph (when chained together), and (iv) a whitelist of permissions per NF indicating the fields each NF is allowed to access.

For the first two, the loader exposes the following API to the client: `load(name, code, is_encrypted)`. For the third, the client specifies the NF graph as a set of edges: ($\texttt{name}_i \rightarrow \texttt{name}_j$). For the fourth, the client supplies a configuration file with a list of items of type: (`name, op, proto:field`) where `op` $\in$ `[read, write]` and `proto:field` indicates a field within a protocol that access is given to. For example, for a firewall, one such entry is `(firewall, read, IP:src)`, in addition to entries for other fields of the IP header.

The loader decrypts the NFs and stitches them together based on the specified graph, before invoking the com-

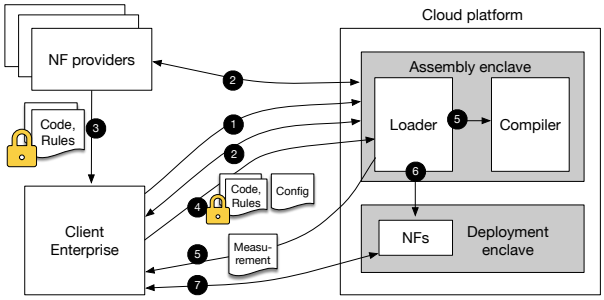

Figure 7: SafeBricks's NF assembly and deployment phases during bootstrap. Locks indicate that the data is encrypted.

piler. (In §7.2, we discuss how the enclave obtains the keys to decrypt this code.) In doing so, it adds the following additional nodes to the composite graph: (i) a `toEnclave` node at the root of the graph, (ii) a `toHost` node at the end of the graph, and (iii) `parse` nodes followed by a `wList` node before each NF. The loader infers the arguments to the `parse` and `wList` nodes automatically from the configuration file. Thus, `parse` is run by the trusted SafeBricks framework and not by an NF or the client enterprise, ensuring that the packets are not parsed in an unintended way.

Compiler. The compiler is a standard Rust compiler that implements a lint prohibiting unsafe code inside the enclave, as discussed in §6.3. Since launching the compiled binary requires OS support, the binary must be placed into main memory where the OS can access it post compilation. However, giving the OS access to the binary unencrypted would violate NF confidentiality.

In order to maintain the privacy of NF code while still allowing its execution by the OS, we take inspiration from VC3 [59]. Similarly to VC3, our compiler links the compiled NF code to a small amount of public code $\texttt{NF}_{\texttt{load}}$, and then encrypts the NF code because it will be placed in main memory for the OS to load in the deployment enclave. We refer to the encrypted code as $\texttt{NF}_{\texttt{priv}}$. Post compilation, $\texttt{NF}_{\texttt{load}} + \texttt{NF}_{\texttt{priv}}$ are loaded and run in a separate deployment enclave by the OS. $\texttt{NF}_{\texttt{load}}$ will be responsible for decrypting and interfacing with $\texttt{NF}_{\texttt{priv}}$ within the deployment enclave once it's initialized.

The loader and compiler are generic modules independent of the NFs. Hence, the NF providers need to audit them only once, across all customer deployments.

Assembly protocol. Figure 7 illustrates the assembly and deployment protocol. ❶ The cloud provisions an enclave with the SafeBricks loader and compiler modules. ❷ Next, the client as well as the NF providers verify that the loader and compiler have been securely provisioned into the enclave using the remote attestation feature of SGX, as described in §2.1. During the attestation, the enclave also returns a securely generated public key to each NF provider. ❸ Each provider then encrypts the NF source code and rulesets with the received public key and submits it to the client enterprise. ❹ The enterprise loads the encrypted source codes and rulesets along with configuration files into the enclave via APIs exposed by the loader module. ❺ The loader decrypts the source codes, stitches them together, and builds and encrypts the assembled code using the compiler, producing $\texttt{NF}_{\texttt{load}} + \texttt{NF}_{\texttt{priv}}$. It then returns to the client a hash measurement of the compiled code so that the client can later verify it once it's deployed in a separate enclave.

7.2 Phase 2: NF deployment

❻ The loader finally requests the OS to deploy $\texttt{NF}_{\texttt{load}} + \texttt{NF}_{\texttt{priv}}$ in a separate enclave on the cloud platform. It attests the deployed enclave, establishes a secure channel with $\texttt{NF}_{\texttt{load}}$, and transfers to it the decryption key for $\texttt{NF}_{\texttt{priv}}$. $\texttt{NF}_{\texttt{load}}$ decrypts the private code and starts execution. Note that since the assembly enclave attests the deployment enclave and the NF vendors attested the assembly enclave, the NF vendors are assured that the deployment enclave will not send the decrypted binary anywhere but merely run it. ❼ The client then attests the deployed enclave using the measurement it received at the end of the assembly phase, after which it establishes a secure channel of communication with the enclave.

8 Security Guarantees

We describe SafeBricks's guarantees assuming the threat model in §2, including the enclave assumption.

SafeBricks's main benefit to confidentiality is that it exposes only encrypted traffic to a cloud attacker, so the attacker does not see the contents of the packets and is limited to observing only packet sizes, timing, and NF access patterns to packets and data. SafeBricks protects in this manner the packet payload and, except in the direct architecture, the header as well.

As with any system with complex processing, encryption does not mean perfect confidentiality because of the existence of side-channels. In §2.2 we mentioned some categories of side-channels that SafeBricks, and SGX in general, does not protect against. In addition, there are a few other SafeBricks-specific side channels. First, an attacker in SafeBricks knows which (encrypted) packets belong to which flow because each flow is affinitized to an IPSec tunnel for scalability. If this issue is of concern, it can be fixed by using a single tunnel for all flows at the expense of performance. Second, an attacker can measure the time taken by NFs to process a batch of packets. This could leak information in some cases, *e.g.*, whether an expensive regular expression was triggered or not. This is a classical problem, already investigated by prior work [6, 13, 78] with common solutions involving padding, *i.e.*, bounding the running time of NFs by executing dummy cycles. Third, an attacker can learn

the action taken by an NF, *e.g.*, whether a connection was dropped simply by noticing that fewer packets were sent out. Like many other side-channels, this leakage can also be removed via padding—for example, the gateways could continue sending dummy traffic.

SafeBricks also protects the integrity of the traffic and of the NF processing. A cloud attacker cannot drop, insert, or modify packets, nor can it tamper with NF execution. Integrity of the NFs is guaranteed by SGX, while the integrity of the traffic is guaranteed by the IPSec tunnels between the enclave and the client.

Via the isolation and least privilege design, SafeBricks further ensures that each NF is confined to accessing only parts of the packet the enterprise desires. For the NF vendors, SafeBricks guarantees that NF source codes are hidden from all untrusted parties, including the client enterprise, a cloud attacker or other NF vendors.

8.1 Comparison to prior approaches

Prior approaches leak significantly more information about the traffic to the cloud provider than SafeBricks.

Cryptographic approaches. BlindBox [63] and Embark [38] encrypt the traffic in a special way that allows the cloud to match encrypted tokens against the traffic and detect if a match occurs. In these schemes, the cloud learns the offset at which any string from any rule in an NF occurs in the packet, regardless of whether or not the rule as a whole matched (rules often contain several such strings). If the rule is known (as in public rulesets), the attacker learns the exact string at that offset in the packet. Even if the rule string is not known, the attacker learns its frequency, which could lead to decryption via frequency analysis. Assuming an enclave employing side-channel protections as in §2.2, SafeBricks does not reveal this information. The attacker does not know which rule or part of a rule triggered on a packet. Moreover, BlindBox and Embark do not protect against *active attackers* who modify the traffic flow and, for example, drop packets.

We remark, however, that these prior approaches rely on cryptography alone, and not on trusted hardware as SafeBricks, which makes it much more challenging for them to achieve the properties SafeBricks achieves.

mcTLS [48] aims to provide least privilege in a setting where each NF is a separate hardware middlebox and belongs to a *different trust perimeter*. Running mcTLS in the cloud in software, however, removes essentially all its security guarantees: the cloud receives the *union* of the permissions of all NFs, which often, is everything.

9 Evaluation

We now measure the impact of SafeBricks on NF performance versus an insecure baseline. We also measure the reduction in TCB size as a result of our design. We do not discuss the performance of SafeBricks's gateway as the protocols it implements are well understood.

9.1 Setup

We evaluate the performance of SafeBricks using SGX hardware on a single-socket server provisioned with an Intel Xeon E3-1280 v5 CPU with 4 cores running at 3.7GHz. We disable hyperthreading for our experiments. The server has 64GB of memory, and runs Ubuntu 14.04.1 LTS with Linux kernel version 4.4. The hardware supports the SGX v1 instruction set which does not allow dynamic page allocation. Further, the total enclave page cache memory (EPC) available to all enclaves is limited to ∼94MB. For test traffic, we use another server that runs a DPDK-based traffic generator and is directly connected to the SGX machine via Intel XL710 40Gb NICs. The SGX machine acts as the cloud, and the traffic generator is both source and sink for the client traffic.

9.2 Performance

We evaluate the performance of SafeBricks using (i) synthetic traces of different packet sizes, from 64B to 1KB, and (ii) the ICTF 2010 trace [31], captured during a wide-area security competition and commonly used in academic research. We report throughput in millions of packets per second (Mpps). In all experiments, we exchange traffic between the traffic generator and the SGX machine over an encrypted tunnel (per §3). As a result, the size of each packet exchanged between the enterprise and the cloud increases by a fixed amount, equal to the headers added by the IPSec protocol.

We compare SafeBricks against an insecure baseline comprising vanilla NetBricks augmented with support for the encrypted tunnel. The baseline represents a setup in which traffic is sent to the cloud over an encrypted channel (hence safe from network attackers), but lacks the protection of SafeBricks at the cloud. Finally, we report the median of 10 iterations for each experiment.

9.2.1 Framework overheads

We first measure the overhead introduced by SafeBricks as a result of redesigning the core NetBricks framework. To illustrate the benefits of our architecture, we also compare the overheads of the strawman approach that performs packet I/O via enclave transitions (per §5).

The net overhead of both approaches varies with the complexity of NFs and the latency the NF introduces as a result of packet processing. In this experiment, we use CPU cycles as a proxy for NF complexity, and evaluate a simple NF that first modifies each batch of packets by interchanging the source and destination IP addresses, and then loops for a given number of cycles. We use packet batches of size 32 for both NetBricks and SafeBricks.

Figure 8 (left) presents the results with varying packet sizes when the NF is deployed on a single core, and Figure 8 (right) shows the performance for 64B packets when the deployment is scaled to two cores. In the worst

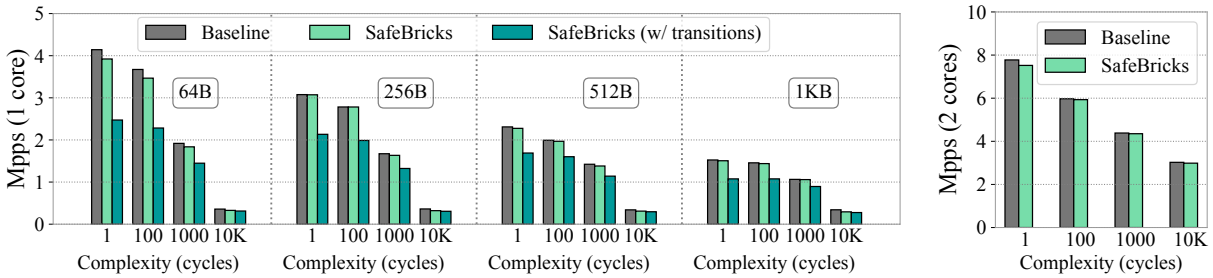

Figure 8: (**Left**) SafeBricks framework performance on 1 core compared to the baseline across different packet sizes, and with increasing NF complexity (*i.e.*, processing time in CPU cycles). (**Right**) Performance with 64B packets and NFs on 2 cores.

case with 64B packets and a delay of 1 cycle, the overhead introduced by SafeBricks is $< 5\%$. As the processing time begins to dominate (with increasing NF complexity), the overhead of SafeBricks becomes negligible.

The results also confirm that the design of SafeBricks outperforms the strawman approach, the overhead of which is ∼40% in the worst case. It's worth noting, however, that the relative overhead of the strawman approach decreases with larger packet sizes, as the rate of I/O falls.

9.2.2 Impact on real NFs

Unlike the simple NF in the previous experiment, real NFs have varying state requirements. Since the sizes of both the processor caches and enclave memory are limited, the overheads of SafeBricks are also governed by the memory access patterns of the NFs in addition to their complexity. In particular, L3 cache misses are more expensive for enclave applications because cache lines need to be encrypted/decrypted before being evicted/loaded. In this experiment, we characterize the effect of state on the performance of SafeBricks by evaluating the following sample applications:

- **Firewall**: We use a stateful firewall application that linearly scans a list of access control rules and drops connections if it finds a match. We evaluate it using a ruleset we obtained from our department (643 rules).
- **DPI**: We use a simple deep packet inspection (DPI) application that implements the Aho-Corasick pattern matching algorithm [1] on incoming packets, similar to the core signature matching component of the Snort IDS [58]. We evaluate the DPI using patterns extracted from the Snort Community ruleset [68].
- **NAT**: Our implementation is based on MazuNAT [42].
- **Load balancer**: We use a partial implementation of Google's Maglev [20], that spreads traffic between backends using a consistent hashing lookup table.

Figure 9 shows the normalized overhead of SafeBricks on application performance across different packet sizes with synthetic traffic. Figure 12 summarizes the worst-case results corresponding to 64B packets. Figure 12 also presents the performance results with the ICTF trace. Across applications, the overhead ranges between an acceptable ∼0–15% for both synthetic and real traffic, and is a result of page faults triggered by L3 cache misses.

Impact of larger memory footprint. In the previous experiment, the working sets of the applications exceeded the L3 cache but remained less than the size of the EPC (∼94MB). However, accessing memory beyond the EPC is doubly expensive because evicted EPC pages need to be encrypted and integrity-protected. We now assess the impact of a large memory footprint using the DPI application. The application builds a finite state machine over all the patterns in the ruleset, and as such has a significantly larger memory footprint than other NFs.

Figure 10 shows the results of our experiment using an increasing number of rules from the Emerging Threats ruleset [21] and the ICTF trace. At 18K rules, the working set of the DPI breached the ∼94MB EPC boundary causing its performance to sharply deteriorate thereafter.

This experiment indicates the limits of SafeBricks with regard to the nature of applications it can efficiently support. However, we note that the ∼94MB limit is only an artifact of existing hardware and isn't fundamental to SGX enclaves. The next generation of SGX machines is likely to support larger EPC sizes.

9.2.3 Cost of NF isolation

We now evaluate the overhead as a result of our mechanisms for enforcing least privilege. Given a chain of NFs, SafeBricks increases the overall depth of the NF graph by one node per NF (§6.3). In this experiment, we first measure this extra cost as a function of the length of the NF chain. We then compare our approach against an mcTLS-like strawman that relies on encryption for selectively exposing packet fields to NFs (§6.1).

Effect of chain length. For this part of the experiment, we use a simple NF that decrements the time-to-live (TTL) field in the IP header of each packet, composed together into chains of varying length. Before executing subsequent NFs in the chain, SafeBricks whitelists access to the TTL field in the permissions vector per packet.

Figure 11 compares the performance of SafeBricks with and without least privilege. Since the NF is stateless, in the absence of isolation SafeBricks does not introduce any discernible overhead against the baseline. With least privilege enforcement, the latency added

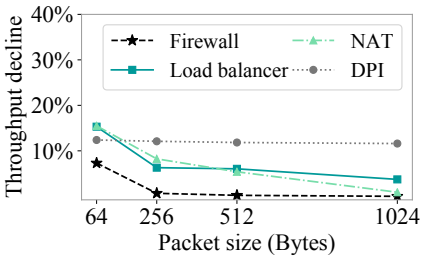

Figure 9: Normalized overhead (Mpps) across NFs for different packet sizes.

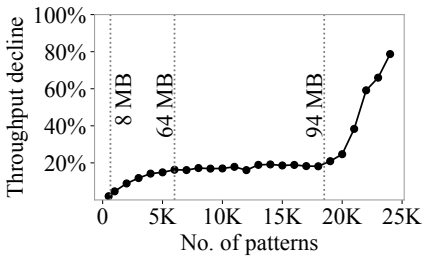

Figure 10: DPI performance (Mpps) using the ICTF trace, with increasing no. of rules.

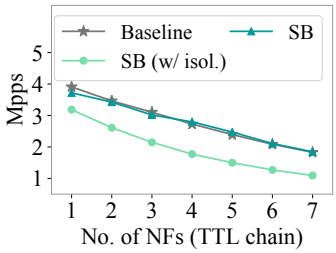

Figure 11: Cost of least privilege with increasing no. of NFs.

	Synthetic (64B packets)		ICTF trace	
NF	**Baseline**	**SB**	**Baseline**	**SB**
Firewall	3.86	3.58	1.96	1.93
DPI	1.10	0.96	0.29	0.25
NAT	3.80	3.21	1.97	1.80
Maglev	3.59	3.04	1.92	1.73

Figure 12: Performance of sample NFs (Mpps)

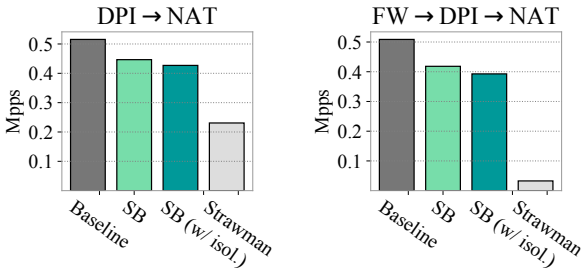

Figure 13: Cost of least privilege across NF chains (2 cores)

by the additional nodes increases as the length of the chain increases. Consequently, the overhead climbs from ∼14–40% as the chain grows to a size of seven NFs. We note that these numbers represent an upper bound on the overhead of SafeBricks. As we show in the next part of this experiment, the percentage overhead is much smaller for real, more complex NFs.

Comparison with encryption-based strawman. We now measure the performance of SafeBricks using a chain of real NFs each of which accesses different parts of packets—a firewall (given read permissions on packet headers), a DPI (with read permissions on both headers and payload), and a NAT (with read and write permissions on packet headers). The NF implementations are identical to the ones described in §9.2.2.

To quantify the benefit of our approach for enforcing least privilege, we also compare SafeBricks to an mcTLS-like strawman in which each NF in the chain is run in a separate enclave (as described in §6.1). In all setups (including the baseline), we allocate two cores for running the NFs, and reserve one core for I/O.

Figure 13 shows the results with two different chains: (i) a DPI followed by a NAT, and (ii) a firewall chained to a DPI and then a NAT. In the former scenario, SafeBricks results in an overhead of 15% in the absence of least privilege enforcement. With least privilege, the throughput declines by a further 3%, confirming that the cost of enforcing least privilege across real NFs is minimal. In contrast, an mcTLS-like approach (with each NF running in a separate enclave, affinitized to distinct cores) results in a sharper decline of 2.2× the performance being bottlenecked at the DPI along with the added encryption and copying of packets as they move across NFs in different enclaves. In the latter scenario with three NFs in a chain, the performance of the strawman approach falls further, by 16×. In this scenario, however, the NFs (and hence enclaves) outnumbered the available cores in our setup, leading to resource contention.

9.3 Comparison with BlindBox and Embark

Both SafeBricks and Embark tunnel packets to a third-party service in the cloud. For the ICTF trace, IPSec tunneling inflates the bandwidth by 16% due to both encryption and encapsulation. Embark introduces a further 20-byte overhead per IPv4 packet because it converts them to IPv6, resulting in a net overhead of 21%. BlindBox, in contrast, does not pay the cost of tunneling as it is targeted at in-network DPI applications. However, the BlindBox encryption protocol (also used by Embark for DPI processing) inflates bandwidth consumption by up to 5× in the worst case, unlike SafeBricks which only uses standard encryption schemes.

As regards throughput, both Embark and BlindBox are competitive with unencrypted baseline NFs and incur negligible overhead, whereas SafeBricks impacts performance by ∼0–15% across NFs due to its use of SGX enclaves (§9.2.2). At the same time, both BlindBox and Embark impact performance at the client considerably—with BlindBox, client endpoints need to implement its special encryption protocols over and above TLS and take 30× longer to encrypt a packet; Embark centralizes this overhead at the enterprise's gateway instead. Clients do not need to pay these costs with SafeBricks.

9.4 TCB size

SafeBricks involves the use of two types of enclaves: one for assembling the NFs during system bootstrap (per §7), and another for deploying the NFs. The assembly enclave primarily contains the Rust compiler, which is nec-

essarily part of the TCB of applications with or without SafeBricks. The deployment enclave, on the other hand, represents the TCB which we aim to reduce in redesigning the NetBricks framework.

To evaluate the reduction in TCB, we thus compare the size of the deployment enclave components in SafeBricks with that of NetBricks. The size of the enclave binary in SafeBricks is ∼1MB. In comparison, the aggregate size of NetBricks components is 21.3MB, representing a TCB reduction of over 20×. The reduction can largely be attributed to the exclusion of DPDK from the TCB as a result of partitioning NetBricks, which itself comprises ∼516K LoC. Furthermore, by designing for our specific use case, we avoid including a library OS within our trust perimeter, the size of which can be as large as 209MB (as in Haven [9]).

10 Limitations and Future Work

SafeBricks inherits three primary limitations owing to its use of Intel SGX.

First, enclave memory is limited to ∼94MB in existing hardware, making SafeBricks impractical for applications with larger working sets. Exploring alternate architectures that combine cryptographic approaches and SGX, thereby reducing the memory burden on the enclaves, is an interesting open problem in this context.

Second, SafeBricks is unsuitable for NFs relying on operations that are illegal within SGX enclaves, such as system calls and timestamps. Though SafeBricks supports timestamps, it can only ensure their monotonicity and not correctness.

Third, SGX enclaves, and consequently SafeBricks, are vulnerable to side-channel attacks (per §2.1). Though a number of potential solutions have been proposed in recent work [16, 18, 27, 65, 66], their impact on application performance is often non-trivial. Investigating the viability of these proposals in the NFV context, or developing targeted solutions for NFs is potential future work.

11 Related Work

We divide related work largely into two categories: (i) cryptographic approaches for securing NFs, and (ii) proposals based on trusted hardware. We do not discuss the mcTLS protocol [48] further as we have already compared SafeBricks with mcTLS in §6 and §8.

Cryptographic approaches. Recent systems propose the use of cryptographic schemes that enable NFs to operate directly over encrypted traffic [5, 38, 44, 63, 76]. When compared to SafeBricks, these approaches have the advantage that they do not rely on trusted hardware. However, this comes with two significant limitations. (1) Their functionality is severely constrained, as discussed in §1, and hence are not applicable to a wide range of NFs. To provide full functionality with cryptography, one needs schemes such as fully-homomorphic encryption [23], which is orders of magnitude too slow. (2) Regarding security, we explained in §8 how these systems leak more information to the cloud than SafeBricks.

Trusted hardware proposals for legacy applications. Other work has shown how to use hardware enclaves to run applications in the cloud without having to trust the cloud provider [3, 9, 30, 50, 67, 72]. The mandate of these systems is to support arbitrary, legacy applications instead of optimizing for any in particular. As a result, some of these systems inflate the size of the TCB by introducing a library OS within the enclave (to support illegal enclave instructions), or impact performance because of enclave transitions.

Trusted hardware proposals for network applications. Recent work has proposed the use of hardware enclaves for securing network applications. Kim *et al.* use SGX to enhance the security of Tor [61], and also identify NFs as a potential use case [36]. Other proposals develop prototypes for specific functions: Coughlin *et al.* [19] present a proof-of-concept Click element for pattern matching within enclaves; and Shih *et al.* [64] propose SGX for isolating the state of NFs, applying it to a subset of the Snort IDS. In contrast, SafeBricks is a *general*-purpose framework that additionally enforces least privilege across NFs. At the same time, SafeBricks balances the interests of NF vendors by maintaining the confidentiality of NF code and rulesets.

Concurrent to our work, SGX-Box [29] and ShieldBox [71] also propose frameworks for executing NFs within enclaves. SGX-Box [29] does not explicitly handle NF isolation or chaining; ShieldBox integrates SGX with Click and isolates each NF in a separate enclave. In such cases, ShieldBox reports a throughput decline of up to 3×. SafeBricks, in contrast, avoids this overhead by isolating NFs within the same enclave with the help of language-based enforcement. However, unlike SafeBricks, ShieldBox also supports NFs with system calls by leveraging the Scone framework [3]. Both SGX-Box and ShieldBox also allow NFs to access entire packets, while SafeBricks enforces least privilege.

Acknowledgments

We thank our shepherd, Emin Gün Sirer, and the anonymous reviewers for their helpful comments. Aurojit Panda helped us with NetBricks; Jethro Beekman helped us with his SGX SDK; Assaf Araki and Intel supplied the SGX cluster; Jon Kuroda helped us manage our testbed. We are also grateful to Chia-che Tsai, Mona Vij, Amin Tootoonchian, Mihai Christodorescu, Rohit Sinha, and Takeshi Mochida for valuable discussions and feedback. This work was supported by the Intel/NSF CPS-Security grants #1505773 and #20153754, the UC Berkeley Center for Long-Term Cybersecurity, and gifts to the RISELab from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

References

[1] AHO, A. V., AND CORASICK, M. J. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* (1975).

[2] ANATI, I., GUERON, S., JOHNSON, S. P., AND SCARLATA, V. R. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[3] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., GOLTZSCHE, D., EYERS, D., KAPITZA, R., PIETZUCH, P., AND FETZER, C. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[4] Aryaka. http://www.aryaka.com/.

[5] ASGHAR, H. J., MELIS, L., SOLDANI, C., DE CRISTOFARO, E., KAAFAR, M. A., AND MATHY, L. SplitBox: Toward Efficient Private Network Function Virtualization. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)* (2016).

[6] ASKAROV, A., ZHANG, D., AND MYERS, A. C. Predictive Black-box Mitigation of Timing Channels. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2010).

[7] BALASUBRAMANIAN, A., BARANOWSKI, M. S., BURTSEV, A., PANDA, A., RAKAMARIĆ, Z., AND RYZHYK, L. System Programming in Rust: Beyond Safety. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)* (2017).

[8] Barracuda Networks. https://www.barracuda.com/.

[9] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[10] BEEBY, D. Rogue tax workers snooped on ex-spouses, family members, 2010. https://goo.gl/WNKoCS.

[11] Berkeley Extensible Software Switch (BESS). http://span.cs.berkeley.edu/bess.html.

[12] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)* (2017).

[13] BRAUN, B. A., JANA, S., AND BONEH, D. Robust and Efficient Elimination of Cache and Timing Side Channels. *arxiv:1506.00189* (2015).

[14] BULCK, J. V., WEICHBRODT, N., KAPITZA, R., PIESSENS, F., AND STRACKX, R. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2017).

[15] CHECKOWAY, S., AND SHACHAM, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[16] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *Proceedings of the ACM Asia Conference on Computer & Communications Security (AsiaCCS)* (2017).

[17] COSTAN, V., AND DEVADAS, S. Intel SGX Explained. *Cryptology ePrint Archive, Report 2016/086* (2016). http://eprint.iacr.org/2016/086.

[18] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2016).

[19] COUGHLIN, M., KELLER, E., AND 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-NFV Security)* (2017).

[20] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[21] Emerging Threats Open Rulesets. https://rules.emergingthreats.net/.

[22] Fortinet. https://www.fortinet.com/.

[23] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)* (2009).

[24] GOOGLE. Transparency Report. https://www.google.com/transparencyreport/userdatarequests/US/.

[25] GORDON, C. S., PARKINSON, M. J., PARSONS, J., BROMFIELD, A., AND DUFFY, J. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (2012).

[26] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache Attacks on Intel SGX. In *Proceedings of the European Workshop on Systems Security (EuroSec)* (2017).

[27] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2017).

[28] HÄHNEL, M., CUI, W., AND PEINADO, M. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2017).

[29] HAN, J., KIM, S., HA, J., AND HAN, D. SGX-Box: Enabling Visibility on Encrypted Traffic Using a Secure Middlebox Module. In *Proceedings of the Asia-Pacific Workshop on Networking (APNet)* (2017).

[30] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[31] ICTF data. https://ictf.cs.ucsb.edu/.

[32] Intel Data Plane Development Kit (DPDK). http://dpdk.org/.

[33] JAMSHED, M. A., MOON, Y., KIM, D., HAN, D., AND PARK, K. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[34] JARMOC, J. SSL/TLS Interception Proxies and Transitive Trust. In *Black Hat* (2012).

[35] JEONG, E., WOOD, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[36] KIM, S., SHIN, Y., HA, J., KIM, T., AND HAN, D. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)* (2015).

[37] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)* (2000).

[38] LAN, C., SHERRY, J., POPA, R. A., RATNASAMY, S., AND LIU, Z. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[39] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2017).

[40] LKL: Linux Kernel Library. https://lkl.github.io.

[41] lwIP: A lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip/.

[42] MazuNAT. https://github.com/kohler/click/blob/master/conf/mazu-nat.click.

[43] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[44] MELIS, L., ASGHAR, H. J., DE CRISTOFARO, E., AND KAAFAR, M. A. Private Processing of Outsourced Network Functions: Feasibility and Constructions. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security)* (2016).

[45] MICROSOFT. Law Enforcement Requests Report. https://www.microsoft.com/en-us/about/corporate-responsibility/lerr.

[46] Mirage TCP/IP stack. https://github.com/mirage/mirage-tcpip.

[47] MOGHIMI, A., IRAZOQUI, G., AND EISENBARTH, T. Cachezoom: How SGX amplifies the power of cache attacks. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)* (2017).

[48] NAYLOR, D., SCHOMP, K., VARVELLO, M., LEONTIADIS, I., BLACKBURN, J., LÓPEZ, D. R., PAPAGIANNAKI, K., RODRIGUEZ RODRIGUEZ, P., AND STEENKISTE, P. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (2015).

[49] NetBricks. http://netbricks.io.

[50] ORENBACH, M., LIFSHITS, P., MINKIN, M., AND SILBERSTEIN, M. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)* (2017).

[51] Palo Alto Networks. https://www.paloaltonetworks.com/.

[52] PALO ALTO NETWORKS. Virtualization Features. https://goo.gl/ezntv6.

[53] PANDA, A., HAN, S., JANG, K., WALLS, M., RATNASAMY, S., AND SHENKER, S. NetBricks: Taking the V out of NFV. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[54] POULSEN, K. Five IRS employees charged with snooping on tax returns, 2008. https://www.wired.com/2008/05/five-irs-employ/.

[55] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. NFV Whitepaper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[56] PRIVACY RIGHTS CLEARINGHOUSE. Chronology of Data Breaches. http://www.privacyrights.org/data-breach.

[57] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2012).

[58] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the Large Installation System Administration Conference (LISA)* (1999).

[59] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)* (2015).

[60] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2017).

[61] SEONGMIN KIM AND JUHYENG HAN AND JAEHYEONG HA AND TAESOO KIM AND DONGSU HAN. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[62] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (2012).

[63] SHERRY, J., LAN, C., POPA, R. A., AND RATNASAMY, S. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (2015).

[64] SHIH, M.-W., KUMAR, M., KIM, T., AND GAVRILOVSKA, A. S-NFV: Securing NFV States by Using SGX. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security)* (2016).

[65] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2017).

[66] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the ACM Asia Conference on Computer & Communications Security (AsiaCCS)* (2016).

[67] SHINDE, S., TIEN, D. L., TOPLE, S., AND SAXENA, P. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2017).

[68] Snort Community Rulesets. https://www.snort.org/downloads.

[69] STRACKX, R., AND PIESSENS, F. Ariadne: A Minimal Approach to State Continuity. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2016).

[70] THE FAST DATA PROJECT. Vector packet processing. https://www.fd.io/technology.

[71] TRACH, B., KROHMER, A., GREGOR, F., ARNAUTOV, S., BHATOTIA, P., AND FETZER, C. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the ACM Symposium on SDN Research (SOSR)* (2018).

[72] TSAI, C.-C., PORTER, D. E., AND VIJ, M. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2017).

[73] WANG, W., CHEN, G., PAN, X., ZHANG, Y., WANG, X., BINDSCHAEDLER, V., TANG, H., AND GUNTER, C. A. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2017).

[74] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)* (2015).

[75] YAHOO! Transparency Report. https://transparency.yahoo.com/.

[76] YUAN, X., WANG, X., LIN, J., AND WANG, C. Privacy-preserving Deep Packet Inspection in Outsourced Middleboxes. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)* (2016).

[77] ZETTER, K. Ex-Googler allegedly spied on user emails, chats, 2010. https://www.wired.com/2010/09/google-spy/.

[78] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Predictive Mitigation of Timing Channels in Interactive Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011).

[79] Zscaler. https://www.zscaler.com/.