ShieldBox: Secure Middleboxes using Shielded Execution

Bohdan Trach[†], Alfred Krohmer[†], Franz Gregor[†], Sergei Arnautov[†],
Pramod Bhatotia[‡], Christof Fetzer[†]
[†]Technische Universität Dresden [‡]University of Edinburgh

Abstract

Middleboxes that process confidential data cannot be securely deployed in untrusted cloud environments. To securely outsource middleboxes to the cloud, state-of-the-art systems advocate network processing over the encrypted traffic. Unfortunately, these systems support only restrictive functionalities, and incur prohibitively high overheads.

This motivated the design of ShieldBox—a secure middlebox framework for deploying high-performance network functions (NFs) over untrusted commodity servers. ShieldBox securely processes encrypted traffic inside a secure container by leveraging shielded execution. More specifically, ShieldBox builds on hardware-assisted memory protection based on Intel SGX to provide strong confidentiality and integrity guarantees. For middlebox developers, ShieldBox exposes a generic interface based on Click to design and implement a wide-range of NFs using its out-of-the-box elements and C++ extensions. For network operators, ShieldBox provides configuration and attestation service for seamless and verifiable deployment of middleboxes. We have implemented ShieldBox supporting important end-to-end features required for secure network processing, and performance optimizations. Our extensive evaluation shows that ShieldBox achieves a near-native throughput and latency to securely process confidential data at line rate.

CCS Concepts

• **Networks → Middle boxes / network appliances**;

ACM Reference Format:
Bohdan Trach[†], Alfred Krohmer[†], Franz Gregor[†], Sergei Arnautov[†], Pramod Bhatotia[‡], Christof Fetzer[†]. 2018. ShieldBox: Secure Middleboxes using Shielded Execution. In *SOSR '18: SOSR '18: Symposium on SDN Research, March 28–29, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3185467.3185469

SOSR '18, March 28–29, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5664-0/18/03…$15.00
https://doi.org/10.1145/3185467.3185469

1 Introduction

Modern enterprises ubiquitously deploy network appliances or "middleboxes" to manage the networking infrastructure. These middleboxes are extensively used to maintain a wide range of workflows for improving the efficiency (e.g., WAN optimizers), performance (e.g., caching, proxies), reliability (e.g., load balancers, monitoring), and security (e.g., firewalls, IDS). Due to their widespread usage, they incur significant deployment, maintenance, and management costs [50].

To overcome these limitations, many enterprises are contemplating outsourcing middleboxes to the cloud [38, 50]. Cloud computing offers the economies of scale for computational resources with the ease of management, elasticity, and fault tolerance. Realizing the vision of middleboxes as a service in the cloud is strengthened by the advancements in network function virtualization (NFV) [33]. NFV offers a flexible and modular architecture that can be easily deployed on commodity hardware. Thus, NFV is a perfect candidate to reap the outsourcing benefits of the cloud infrastructure.

However, middleboxes that process confidential data cannot be securely deployed in the untrusted cloud environments. In cloud environment, an accidental or, in some cases, intentional action from a cloud administrator could compromise the confidentiality and integrity of execution. These threats of potential violations to the integrity and confidentiality of customer data are often cited as a key barrier to the adoption of cloud services [43]. Furthermore, cloud providers are increasingly offering edge computing resources in collaboration with third-party ISPs and CDN operators to meet stringent low-latency performance requirements (SLAs) of modern online applications [17]. Since the underlying infrastructure is operated by multiple third-party providers, such a hybrid cloud-edge computing infrastructure further exacerbate secure deployment of middleboxes.

To securely outsource middleboxes in the cloud, state-of-the-art systems advocate network processing over encrypted traffic [29, 51]. However, these systems support only restrictive type of functionalities, and incur prohibitively high performance overheads since they require complex computations over encrypted network traffic.

These limitations motivated our work—we strive to answer the following question: *How to securely outsource middleboxes on the untrusted third-party platform without sacrificing performance while supporting a wide range of enterprise NFs?*

To answer this question, we present ShieldBox—a secure middlebox framework for deploying high-performance network functions (NFs) on untrusted commodity servers. The architecture of ShieldBox is based on four design principles: (1) *Security* — we aim to provide strong confidentiality and integrity guarantees against a powerful adversary, (2) *Performance* — we strive to achieve near-native throughput and latency, (3) *Generality* — we aim to support a wide range of network functions (same as plain-text processing) with the ease of programmability, and, (4) *Transparency* — we aim to provide a transparent, portable, and verifiable environment for deploying middleboxes, without major changes to the systems source code and deployment procedure.

To achieve these design goals, ShieldBox leverages hardware-assisted secure enclaves based on Intel SGX [15] for providing strong security properties. In particular, ShieldBox builds on Scone [42]—a shielded execution framework to securely process network packets on commodity untrusted infrastructure. However, the architectural limitations of Intel SGX present a significant challenge for middleboxes requiring high-performance network I/O. To achieve high performance despite the inherent limitations of the SGX architecture, we have designed a high-performance I/O library for shielded execution using Intel DPDK [2] to efficiently process packets in the userspace secure enclave memory.

For the developers, ShieldBox provides a flexible and modular framework to build a rich set of NFs by adapting the Click [26] architecture. In this way, ShieldBox supports a wide range of NFs with the ease of programmability using Click's out-of-the-box elements and C++ extensions. Finally, ShieldBox builds on the Docker container technology with a remote attestation and configuration service, which provides network operators a portable and cryptographically verifiable deployment mechanism.

Furthermore, we have designed several important end-to-end features required for secure middleboxes:

- New Click elements for secure packet processing.
- Efficient shared memory packet transfer in the multiple SGX enclaves setup for NFVs chaining [23].
- Secure state persistence layer for fault-tolerance and stateful migration of middleboxes [49].
- On-NIC PTP clock as time source for the SGX enclaves.
- Memory safety mechanism to defend against DPDK-specific Iago attacks [14].

We have implemented the aforementioned security features, and also added several SGX-specific performance optimizations to ShieldBox. Lastly, we have evaluated the system using a series of micro-benchmarks, and two case-studies: a multiport IP Router, and IDS. Our evaluation shows that ShieldBox achieves near-native throughput and latency. A detailed version of this paper with additional evaluation results is available as a technical report [54].

2 Shielded Execution

Shielded execution provides strong confidentiality and integrity guarantees for unmodified legacy applications running on untrusted platforms. Our work builds on Scone [42]—a shielded execution framework based on Intel SGX [15].

Intel SGX is a set of ISA extensions for Trusted Execution Environments (TEE) released as part of the Skylake architecture. Intel SGX provides an abstraction of secure *enclave*—a memory region for which the CPU guarantees the confidentiality and integrity of the data and code residing in it. Specifically, the enclave memory is located in Enclave Page Cache (EPC)—a dedicated memory region protected by MEE, an on-chip Memory Encryption Engine. The MEE encrypts and decrypts cache lines with writes and reads in the EPC, respectively.

The architecture of SGX suffers from two major limitations: First, EPC is a limited resource, currently restricted to 128MB (out of which only 94MB is available to all enclaves). To overcome this limitation, SGX supports a secure paging mechanism to an unprotected memory region. However, the paging mechanism incurs very high overheads depending on the memory access pattern (2× to 2000×). Second, the execution of system calls is prohibited inside the enclave. To execute a system call, the executing thread has to exit the enclave. Such enclave transitions are expensive—especially, for middleboxes—because of security checks and TLB flushes.

Scone is a shielded execution framework for unmodified POSIX applications based on Intel SGX [42]. In Scone, legacy applications are statically compiled and linked against a modified standard C library (Scone `libc`). In this model, application's address space is confined to the enclave memory, and interaction with the outside world (or the untrusted memory) is performed only via the system call interface. Scone `libc` executes system calls outside the enclave on behalf of the shielded application. The Scone framework protects the executing application from the outside world, such as untrusted OS, through *shields*. Furthermore, Scone provides a *user-level threading* mechanism inside the enclave combined with the *asynchronous system call* mechanism in which threads outside the enclave asynchronously execute the system calls without forcing the enclave threads to exit the enclave [53]. Lastly, Scone provides a transparent integration to Docker using which users can seamlessly deploy container images, and *remote attestation and configuration system* to securely provision secrets to the application.

3 Overview

Basic design. At a high-level, the core of our system consists of a simple integration of a DPDK-enabled Click [26] that is running inside the SGX enclave using Scone [42]. Figure 1 shows the high-level architecture of ShieldBox.

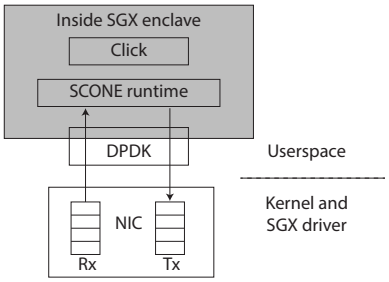

Figure 1: ShieldBox basic design

While designing ShieldBox, we need to take into account the architectural limitations of Intel SGX. As described in §2, an enclave context switch (or exiting the enclave synchronously for issuing system calls) is quite expensive in the SGX architecture. The Scone framework overcomes this limitation using an asynchronous system call mechanism [53]. While the asynchronous `syscall` mechanism is good enough for common Web services like HTTP servers or KV stores, it is not sufficient to sustain the line rate as required by modern middleboxes. Especially, numerous modern middleboxes require a *fast path* bypassing kernel network stack to achieve the line rate [5]. Therefore, we designed a high-performance I/O library for shielded execution based on the userspace DPDK library [2] as a better fit for the SGX enclaves.

Furthermore, we need to ensure that the memory footprint of ShieldBox code and data is minimal, due to several reasons: As described in §2, enclaves that use more than 94MB of physical memory suffer high performance penalties due to EPC paging (2× to 2000×). In fact, to process data packets at line rate, even stricter resource limit must be obeyed—the working set of the application must fit into the L3 cache. Therefore, our design diligently ensures that we incur minimum cache misses, and avoid EPC paging.

Besides performance reasons, minimizing the code size inside the enclave allows reducing the attack surface as it leads to a smaller Trusted Computing Base (TCB). The core of Click is already quite small (6MB for a statically linked binary section that is loaded in the memory). We decrease its size by removing the unnecessary Click elements at the build time. Importantly, we designed ShieldBox with the packet-related DPDK data structures running outside of the enclave. More specifically, the TCB in our case comprises of the following components: the CPU and the microcode that implements the SGX functionality; code and data of Scone's C library as well as its remote attestation mechanism, DPDK (except for the actual packet buffers), and Click. All other components are untrusted.

Threat model. We target a scenario where the middleboxes that process confidential data are deployed in the untrusted cloud environment (or at the edge computing nodes) [50]. In this context [29, 51], attackers might try to learn the contents

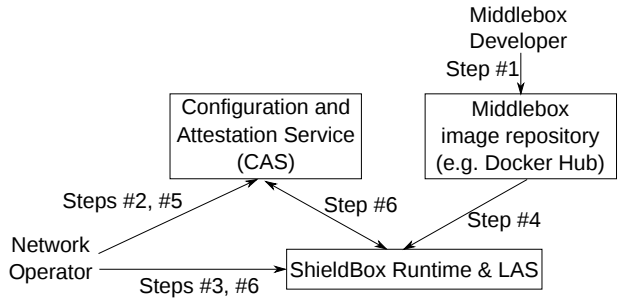

Figure 2: ShieldBox system workflow

of encrypted data packets and system configuration such as cryptographic keys, filtering and classification rules, etc. Furthermore, attackers might try to compromise the integrity of middlebox by subverting its execution.

To circumvent such attacks, we protect against a very powerful adversary even in the presence of complex layers of software in the virtualized cloud computing infrastructure. More specifically, we assume that the adversary can control the entire system software stack, including the OS or the hypervisor, and is able to launch physical attacks, such as performing memory probes.

We rely on Intel SGX to protect against direct memory-reading attacks by the privileged software. This guarantees confidentiality, integrity, and freshness of the SGX-protected memory pages. We also assume the attacker can launch memory safety attacks by forging pointers into trusted memory and pass them to ShieldBox [14, 28, 35].

However, we note that ShieldBox is not designed to protect against side-channel attacks [57], such as exploiting timing and page fault information. Furthermore, since the underlying infrastructure is controlled by the cloud operator we cannot defend against denial-of-service attacks. We also assume that an attacker can arbitrarily reorder or drop packets—we take no particular actions against such attacks. Middlebox developers should protect against these using cryptographic primitives, if necessary.

System workflow. Figure 2 shows the system workflow of ShieldBox. As a preparation for the deployment, developers build middlebox container images, and upload them to an image repository (such as Docker Hub [1]) using the Scone toolchain. A network operator who wants to deploy a middlebox to the cloud should bootstrap a Configuration and Attestation Service (CAS) on a trusted host, and a Local Attestation Service (LAS) on the host that will be running the

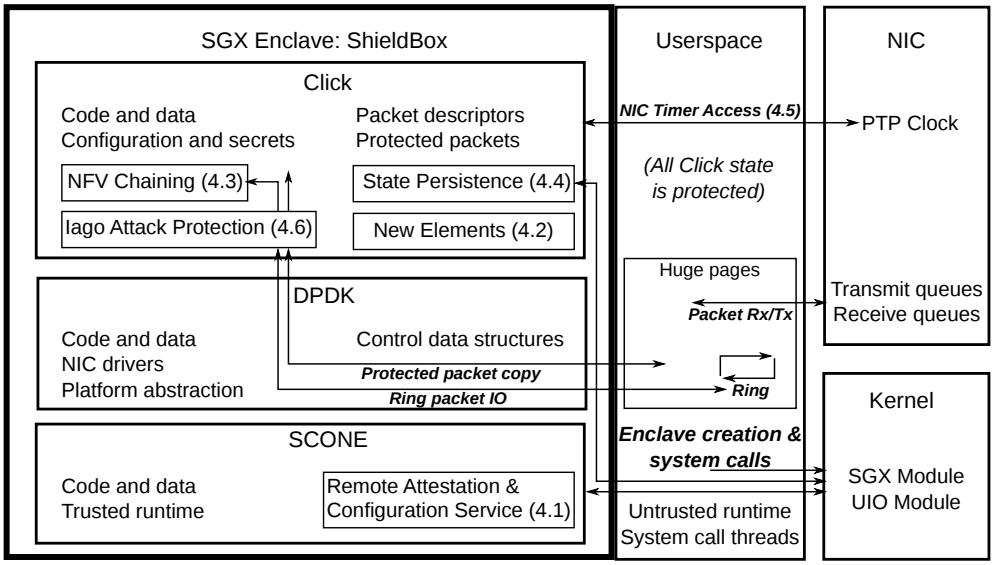

Figure 3: ShieldBox detailed design

middlebox (detailed in §4.1). After this, ShieldBox can be installed on the target machine in the cloud using the container technology—either manually or deployed as a container image from the image repository. Alternatively, it can be installed by transferring a single binary to the target machine.

The ShieldBox framework is bootstrapped using the Configuration and Remote Attestation Service (CAS) (§4.1). The CAS service is launched either inside an SGX enclave of an (already bootstrapped) untrusted machine in the cloud or on a trusted machine under the control of the network operator outside the cloud. Middlebox developers implement the necessary NFs as Click configurations and send them to the CAS service together with all necessary secrets (cryptographic keys, proprietary IDS rules, etc.).

Once the operator launched ShieldBox, it connects to the CAS and carries out the remote attestation (§4.1). If the attestation is successful, the ShieldBox instance receives the configuration and necessary secrets. Thereafter, ShieldBox executes user-defined Click elements, which are responsible for reading packets in the userspace memory directly from NIC, performing network traffic processing, and sending them back to the network. All elements run inside an SGX enclave. Packets that must be processed under SGX protection are copied into the enclave explicitly. We efficiently execute the expensive network I/O operations (to-and-from the enclave memory) by using our high-performance I/O library for shielded execution based on DPDK. To summarize, ShieldBox provides the following benefits:

- **Security:** ShieldBox provides strong confidentiality and integrity for the middlebox execution by leveraging hardware-assisted SGX memory enclaves.
- **Performance:** ShieldBox achieves near-native throughput and latency by building a high-performance network-I/O architecture for shielded execution by optimizing the combination of Scone and DPDK.
- **Generality:** ShieldBox supports a wide range of NFs, as supported in the plain-text network processing, without restricting any functionalities by leveraging Click's simple and generic programming model.
- **Transparency:** ShieldBox provides network operators a portable, configurable, and verifiable architecture for seamless deployment of middleboxes. It builds on the container technology, and therefore, the changes to the software source code and deployment methods are kept at the minimum.

Limitation. We note that neither DPDK nor Click have built-in functionality for flow-based stateful traffic. More precisely, it has no functionality to reconstruct flows and process packets in flow context using Click or DPDK— this functionality must be added to the C/C++ core of these applications. This implies that ShieldBox currently supports NFs that work on L2 and L3; as only restricted processing of L4-L6 traffic is supported, which does not require flow reconstruction.

4 Design Details

We next present the design details of ShieldBox. Figure 3 shows the detailed architecture of ShieldBox.

4.1 Configuration and Remote Attestation

To bootstrap a trusted middlebox in the cloud, one has to establish trust in the system components. While Intel SGX provides a remote attestation feature, a holistic system must be built for remote attestation and secure configuration of network appliances [44]. To achieve this goal, we added a

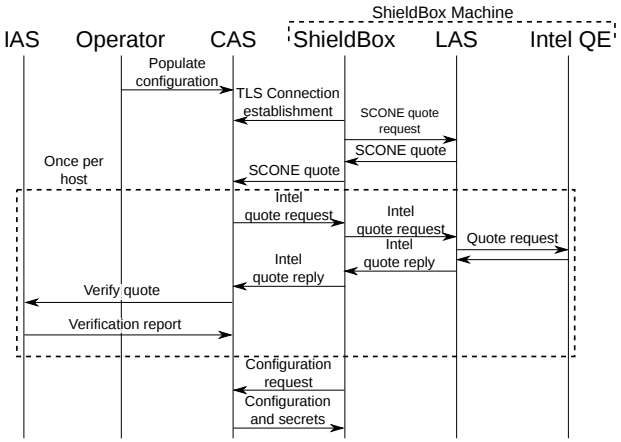

Figure 4: ShieldBox's configuration and attestation

generic remote attestation and configuration framework to Scone. Figure 4 depicts our protocol.

In order to attest an enclave using Intel Remote Attestation, a verifier (operator of a ShieldBox instance) connects to the application and requests a quote. The enclave requests a report from SGX hardware and transmits it to the Intel Quoting Enclave (QE), which verifies, signs, and sends back the report. The enclave then forwards it to the verifier. This quote can be verified using the Intel verification service [3].

Our remote attestation system extends Intel's RA mechanism, and is integrated with a configuration system, which provisions ShieldBox with its configuration in a secure way using a trusted channel established during attestation. This system consists of an enclave startup routine embedded in the Scone library, Local Attestation Service (LAS), and Configuration and Attestation Service (CAS).

- The enclave startup routine takes control before ShieldBox's `main` function is called and interacts with LAS and CAS to carry out remote attestation, and allows setting environment variables, command-line arguments, and keys for the Scone shielding layer securely and confidentially.
- Local Attestation Service is running on the same machine as ShieldBox middlebox. It, eventually, act as the root of trust for remote attestation once CAS trusts LAS. On each host, LAS only has to attest itself one time using Intel RA mechanism to CAS. This decouples our system from Intel's.
- Configuration Attestation Service is running on a single (possibly replicated) node and stores configuration and secrets of the services built with Scone. It builds trust into unknown LAS using Intel Attestation Service (IAS), maintains information about attested LAS instances, and provisions configuration to applications using the startup routine.

`ToEnclave` Transfers a packet to enclave, frees the original packet
`Seal(Key, Security Association state)` Encrypts the packet with AES-GCM
`Unseal(Key, Security Association state)` Decrypts the packet with AES-GCM
`HyperScan(rule database)` High-performance regular expression matching engine
`DPDKRing(Ring name)` Transfers a packet to the DPDK ring structures
`StateFile(Key, path)` Provides settings to the persistent state engine

Table 1: ShieldBox new specialized elements

To bootstrap the system, the operator launches CAS, either on the host under his control or the host in the cloud inside an SGX enclave. Then, the CAS service is populated with configurations and secrets using the REST API or a command-line configuration tool. LAS instances are launched on cloud hosts that will run ShieldBox instances either by the operator or the cloud provider. During startup, Scone's startup routine in each ShieldBox instance establishes a TLS connection to CAS. Simultaneously, it connects to LAS to request a Scone quote that is forwarded to CAS. In case the LAS instance is not yet trusted, CAS uses Intel's RA mechanism to attest it. After LAS is trusted, ShieldBox's Scone quote is verified by CAS proving the binary's integrity and establishing whether it is running under SGX protection. This removes the distribution mechanisms (such as Docker Hub) from the TCB. After that, CAS ensures that the TLS connection is originating from the ShieldBox instance it received the quote of preventing man-in-the-middle attacks. Thereafter, ShieldBox obtains its configuration from the CAS service and transfers control to main ShieldBox code.

4.2 Secure Elements

As described in §3, we designed ShieldBox with the packet-related data structures of DPDK running outside the enclave. Therefore, we needed an efficient way to support the communication between DPDK and the enclave memory region. In particular, we have to consider the overheads of accessing the SGX-encrypted pages from the main memory and copying of the data between the protected and unprotected memory regions. When possible, the data packets with plain-text contents should not be needlessly copied into the enclave, as it will degrade the performance. Therefore, we designed specialized secure Click elements (shown in Table 1) for copying the data packets into/outside the enclave to facilitate efficient communication.

By default, packets are read from NIC queues into the untrusted memory. This reduces the overhead of using SGX when processing packets that are not encrypted and can

be safely treated with fewer security mechanisms involved. Such packets are immediately forwarded or dropped upon header inspection. On the other hand, we must move packets into the enclave memory with explicit copy element. We have implemented such an element (`ToEnclave`), and use it to construct secure packet processing chains.

We have also added support for the commonly used AES-GCM cipher into ShieldBox (`Seal` and `Unseal` elements). This allows us to construct VPN systems that use modern cryptographic mechanisms. These elements were implemented using the Intel ISA-L crypto library. We use CAS to distribute the VPN traffic encryption keys.

To allow the creation of high-performance IDS systems based on ShieldBox, we have created an element based on the `HyperScan` regular expression library. It allows fast matching of multiple regular expressions for the incoming packets, simplifying implementation of systems like Snort [6].

We have also added elements that implement more broad mechanisms: `DPDKRing` (§4.3) for NFV chaining, and `StateFile` (§4.4) for state persistence in network appliances.

4.3 NFVs Chaining

Typically NFVs are chained together to build a dataflow processing graph with multiple Click elements, spanning across multiple cores, sockets, and machines [23, 38]. The communication between different cores and sockets happens through the shared memory, and communication across machines via NICs over RDMA/Infiniband. DPDK supports NUMA systems and always explicitly tries to allocate memory from the local socket RAM nodes.

However, unlike normal POSIX applications, SGX enclaves cannot be shared across different sockets. (The future SGX-enabled servers might have support for the NUMA architecture.) As a result, in the current Intel SGX architecture, the users would need to run one ShieldBox instance per each CPU socket. Another important reason for cross-instance chaining is the collocation of middleboxes from different developers that do not necessarily trust each other. In this case, developers would want to leverage SGX to protect the secrets. Therefore, it is imperative for the ShieldBox framework to provide an efficient communication mechanisms between enclaves to support high-performance NFVs chaining.

We built an efficient mechanism for communication between different ShieldBox instances by leveraging existing DPDK features. In particular, DPDK already provides a building block for high performance communication between different threads or processes with its ring API. This API contains highly optimized implementations of concurrent, lockless FIFO queues which use huge page memory for storage. We have implemented the `DPDKRing` element (see Table 1) for ShieldBox to utilize it for chaining. As huge page memory is shared between multiple ShieldBox instances, the ring buffers are shared as well and can be used as an efficient way of communication between multiple ShieldBox processes.

`Seal(StateFile)` Seals elements' state in the `StateFile`
`Unseal(StateFile)` Unseals elements' state from the `StateFile`
`Persist(timer, StateFile)` Periodically persists the state to `StateFile`

Table 2: ShieldBox APIs for state persistence

This solution requires assigning ownership of all shared data structures to a single process. For this, we rely on the DPDK distinction between primary and secondary processes. Primary processes, the default type, request huge page memory from the OS, allocate memory pools and initialize the hardware. Secondary processes skip device initialization and map the huge page memory already requested by the primary process into their own address space. To support NF chaining using multiple processes, we added support for starting ShieldBox instances as secondary DPDK processes.

Depending on the process type, the `DPDKRing` element either creates a new ring (primary process) or looks up an existing ring (secondary process). In ShieldBox, packets pushed towards a DPDKRing element are enqueued into the ring and can be dequeued from the ring in another process for further processing. A bidirectional communication between two processes can be established by using a pair of rings.

4.4 Middlebox State Persistence

Middleboxes often maintain useful state (such as counter values, Ethernet switch mapping, activity logs, routing table, etc.) for fault-tolerance [49], migration [36], diagnostics [56], etc. To securely persist this state, we extend ShieldBox with new APIs (shown in Table 2) for the state persistence. The `Seal` primitive is used to collect the state that must be persisted from the elements, and write it down in encrypted form to disk. `Unseal` reads this state from disk, decrypts it and populates the elements with this state. In order to allow secure cryptographic key generation inside the enclave, we have exposed Scone functions for getting SGX `Seal` keys to the ShieldBox internal APIs.

To configure this functionality, we have added a new configuration element to ShieldBox, called `StateFile` (see Table 1). Its parameters are file to which state should be written and the key that should be used for encryption. Note that this information is transmitted to ShieldBox instance in the configuration string via remote attestation, and is not accessible outside the enclave. We do not use Scone file system shield, but encrypt and decrypt file as a single block instead. This ensures confidentiality and integrity of stored data via the use of AES-GCM cipher. Due to lack of monotonic counters we do not protect against rollback attacks. To overcome this

limitation, we plan to integrate ShieldBox with Pesos [27], a policy-enhanced secure object storage system [4].

We do not attempt to extract the relevant state transparently. Instead, we rely on the programmer providing necessary serialization routines that save only necessary parts of element state. These routines are available in ShieldBox as read and write handlers, and are triggered in the ShieldBox startup procedure after the configuration is loaded, parsed, and initialization of the basic components is finished, or manually via `ControlSocket` of the `StateFile` element. It's also possible to trigger them periodically via a timer.

4.5 NIC Time Source

Timer is one of the commonly used functionalities in middleboxes [33, 38]. It is used for a variety of purposes such as measuring performance, scheduling NFs, etc.

The time measurement can be fine-grained or coarse-grained based on the application requirements. For the fine-grained cycle-level measurements, developers use `rdtscp` instruction, which is extremely cheap and precise. Whereas for the coarse-grained measurements, applications invoke system calls like `gettimeofday` or `clock_gettime`.

However, in the context SGX enclaves, both `rdtsc` and `syscalls` have unacceptable latency to use in middleboxes for the line rate processing. More specifically, the `rdtscp` instruction is forbidden inside the enclave, and therefore, it causes an enclave exit event; whereas, asynchronous system calls in Scone are submitted though a system call queue that is optimized for the raw throughput, but not latency.

To overcome these limitations, we use the on-NIC PTP clock as the clock source for the enclave. This clock can be read inside the enclave reasonably fast (0.9 μsec, which is on the same scale as reading HPET). Moreover, it neither causes enclave exits nor requires submitting system calls. Furthermore, the on-NIC clock is extremely precise since it is intended to use for the PTP synchronization protocol.

We note that this time source is not secure, and can be used as a DoS attack vector by a malicious OS. However, the same is true for the other time sources—a trusted, efficient and precise time source for SGX enclaves remains an unsolved problem that will likely require changes to the hardware [46].

4.6 Memory Safety for DPDK-Specific Iago Attacks

Iago attacks [14] are a serious class of security attacks that can be launched on shielded execution to compromise the confidentiality and integrity properties. In particular, an Iago attack originates through malicious inputs supplied by the untrusted code to the trusted code. In the classical setting, a malicious OS can subvert the execution of an SGX-protected application by exploiting the application's dependency on correct values of system call return values [12].

The decision (§3) to allocate huge pages for packet buffers and DPDK rings has security implications. The fact that packets are passed through rings by reference, and DPDK buffers contain pointers, opens a new attack surface. Attackers with access to this memory region could modify pointers to point into the SGX-protected regions and make the enclave inadvertently leak secrets over the network [28, 35].

The scenario for Iago attack on DPDK as follows: DPDK maintains a memory buffer associated with each received packet in the unprotected memory. The attacker adds a maliciously crafted memory buffer with an offset or data address pointing to the enclave into the `rte_ring` structure. If NF sends all packets that don't have an IP header to the output, this could leak memory content, and exfiltrate secrets.

To protect against DPDK-specific Iago attacks, we have implemented a pointer validation function. More specifically, the scheme uses an enclave parameter structure that is located inside the enclave memory and defines the enclave memory boundaries. Pointers are validated by checking if they do *not* point into the enclave memory range [*base*,*base*+*enclave_size*). We note that ShieldBox is already protected against the classical syscall-specific Iago attacks through Scone's *shielding* interfaces.

This ensures that no pointers possibly pointing to the secrets stored in EPC are accepted through the unprotected huge page memory. Pointers can still be modified by a malicious attacker, but they can only point to the unprotected memory. However, if they point to the unmapped virtual memory, the operating system will terminate the application. Furthermore, security measures such as `ASLR` also makes it harder for the attackers to find a valid attack vector [48].

As it is possible for an application to enqueue and dequeue arbitrary pointers into DPDK's `rte_ring` structures, it is not easily possible to integrate this pointer check directly into DPDK. Instead, we implemented these pointer checks in the `DPDKRing` and `FromDPDKDevice` (§4.3) elements. If ShieldBox detects a malicious pointer, it assumes an attack, notifies the application operator and drops the packet.

5 Implementation

5.1 Interaction with Scone and Hardware

We use Scone to simplify porting of DPDK and Click. We next describe how we adapted Scone for our system.

System startup. When ShieldBox starts, it performs remote attestation and obtains the configuration. ShieldBox initializes the DPDK subsystem, allocates huge page memory and takes the control over NICs that are available. Then, it starts running the Click element scheduler, which reads packets from the NIC and passes them along the processing chain until they leave the system or are dropped.

System calls. As one of the goals of the ShieldBox is high performance, we minimize the rate of system calls in the fast

path of the application, as this would make it impossible for us to sustain the line rate. On the other hand, systems calls are necessary for the application startup, as it is necessary to do remote attestation, gain access to NIC, etc. This means that the asynchronous system call subsystem is mostly idle after the startup, and causes no runtime overhead.

Memory management. When the Scone runtime starts the application, it automatically places the application code, statically allocated data, and heap (memory allocated via `malloc`, `mmap`) in the SGX-protected EPC memory. This mechanism is in contrast to the way DPDK operates—DPDK by default allocates memory using `x86_64` huge pages, which reduce the TLB miss rate and ensure continuous physical memory layout. Such pages are not supported inside the enclave; besides that, the NIC can only deliver packets to the unprotected memory, and network traffic entering or leaving machine can be modified by an attacker. Therefore, we keep the huge pages enabled in DPDK outside the enclave, *and explicitly copy packets that must be processed with SGX protection into the enclave.* With this scheme, DPDK-created packet data structures are allocated outside the SGX enclave. We support an efficient data transfer between the DPDK and enclave and processing inside the enclave using the new secure Click elements (detailed in §4.2).

Accessing huge pages in DPDK does not require bypassing Scone, because of the specific way DPDK allocates huge pages. In particular, instead of passing `MAP_HUGETLB` flag to `mmap()`, it opens shared memory files in the `hugetlbfs` virtual filesystem and passes those file descriptors to `mmap` call. We configure Scone not to shield these file-to-memory mapping requests, and directly pass them to the OS instead.

Partitioning ShieldBox. Another design aspect that is always present in designing software for Intel SGX is the question of partitioning. One of the components that we could have moved outside of the enclave is DPDK: in the end, NIC cannot deliver data into the enclave, as this would violate SGX security mechanism, which means that a big part of DPDK data is outside of the enclave. This means that DPDK can be easily moved out of the enclave. This would open two possibilities for interaction with enclave: via concurrent queue in shared memory or synchronously via enclave enters/exits. We argue that both approaches are suboptimal from the performance point of view.

If we use synchronous interface, we would have to constantly execute enclave enters and exits, which have extremely high runtime cost. On the other hand, if we use concurrent queue for communication, this leads to another problem: in such partitioning scenario part of the cores would be wasted, because they only read packets from the network into the concurrent queue. Therefore, we conclude that having DPDK inside enclave is the optimal solution for achieving high performance inside SGX enclaves.

5.2 Toolchain

We built ShieldBox's toolchain using DPDK (version 16.11) and Click (master branch commit `0e860a93`). We further integrated it with the Scone runtime to compile ShieldBox. We use `gcc` version 6.3.0 for the compilation process. We used Boost `C++` library (version 1.63) to build a static version of the Hyperscan high performance regular expression matching engine (master branch commit `7aff6f61`) and incorporated it into ShieldBox. We use `WolfSSL` library [8] to implement `StateFile` sealing and packet `Seal/Unseal` elements. The toolchain contains automated scripts for building and deploying middlebox images, and setting up ShieldBox and CAS services (as described in the system workflow in §3).

To make the compilation of ShieldBox work with Scone, some changes to DPDK were necessary. In particular, we need to remove the helper functions for printing stack tracebacks and provide some `glibc`-specific structures, macros, and kernel header files. Click required no adaptions since it is implemented in `C++` mostly using high-level APIs.

The resulting ShieldBox binary is 8.2 MB in size, and around 16 MB including minimal runtime stack and heap allocation. This implies that we could run roughly up to six instances of ShieldBox in parallel on one processor without impacting the performance by EPC paging (94MB).

5.3 Optimizations

We further optimized the data path inside Click, especially for the case of DPDK running inside the enclave, by identifying the performance bottlenecks using the `perf` [7] tool.

Memory pre-allocation. The `FromDPDK` element allocated memory for packet descriptor storage on the stack each time the `run_task` function was called. We pre-allocated this memory once in a constructor.

Branching hints. We inserted GCC-specific `unlikely` / `likely` macros in several `if`-clauses. These get translated to platform-specific instructions to instruct the CPU to always try the given branch first instead of using its prediction.

Modulo operations. We replaced all modulo operations in the data path by cheaper compare-and-subtract operations.

Queue optimization. In the `ToDPDKDevice` Click element we replaced the inefficient implementation of the queue by the `rte_ring` structure provided in DPDK.

Timer event scheduler optimization. In the Click timer event scheduler, we have optimized the code to reduce the number of `clock_gettime` system calls.

6 Evaluation

6.1 Experimental Setup

Testbed. We evaluated ShieldBox using two machines: (1) load generator, and (2) SGX-enabled machine. The load generator is a Broadwell Intel Xeon D-1540 (2.00GHz, 8 cores, 16 hyperthreads) machine with 32GB RAM. The SGX machine

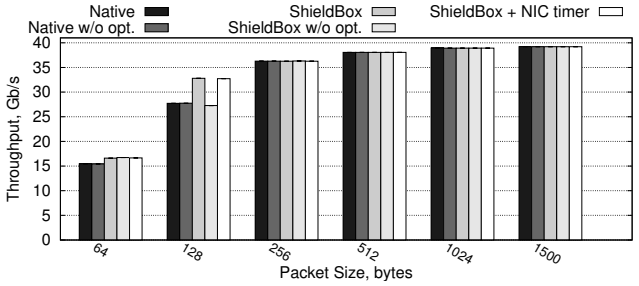

Figure 5: Throughput: Wire w/ varying packet size

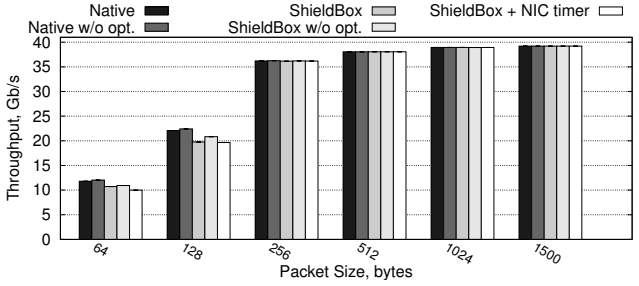

Figure 6: Throughput: EtherMirror w/ varying packet size

under test is Intel Xeon E3-1270 v5 (3.60GHz, 4 cores, 8 hyperthreads) with 32GB RAM running Linux kernel 4.4. Each core has private 32KB L1 and 256KB L2 caches, and all cores share an 8MB L3 cache. The load generator is connected to the test machine using 40 GbE Intel XL-710 network card. We use `pktgen-dpdk` for throughput testing. The load generator saturates the link starting with 128-byte packets.

Applications. For the micro-benchmarks, we use three basic Click elements: (1) `Wire`, which sends the packet immediately after receiving; (2) `EtherMirror`, which sends the packet after swapping the source and destination addresses; and (3) `Firewall`, which does packet filtering based on PCAP-like rules.

For the case-studies, we evaluated ShieldBox using two applications: (1) a multiport `IPRouter`, and (2) an `IDS`.

Methodology. For the performance measurements, we consider several cases of our system:

- Native (Non-SGX) w/ and w/o generic optimizations.
- SGX-enabled ShieldBox w/ and w/o optimizations.
- SGX-enabled ShieldBox w/ the on-NIC timer.

We use native Click as the evaluation baseline since it is the worst-case scenario for us. Lastly, unless stated otherwise, `ToEnclave` element is not used in the benchmarks.

6.2 Throughput

We first report ShieldBox's throughput with varying packet size running on four cores. Figure 5, Figure 6, and Figure 7 present the throughput for `Wire`, `Ethermirror`, and `Firewall`, respectively.

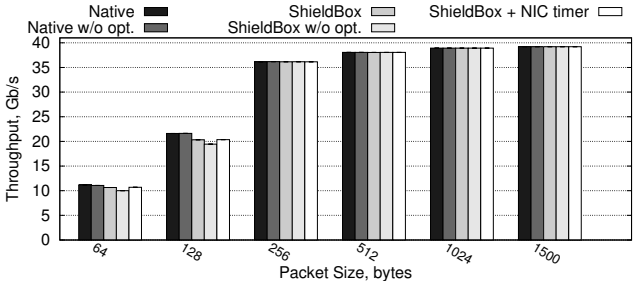

Figure 7: Throughput: Firewall w/ varying packet size

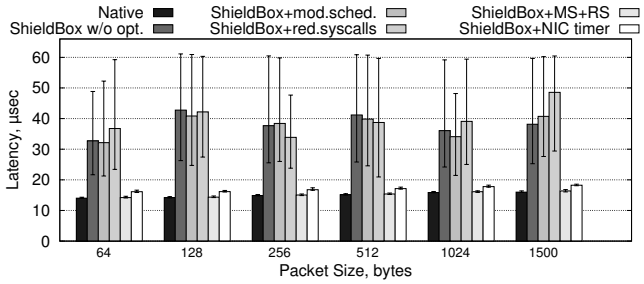

Figure 8: Latency: EtherMirror measurements

The results show that the performance of ShieldBox matches the performance of Click. In the case of Wire application with the packet sizes smaller than 256 bytes, ShieldBox is better than the native version. This is explained by the fact that Click timer event scheduler optimization is missing in the native Click, which removes some system call overhead from the `Wire` application. The impact is smaller with other applications, because they contain elements that reduce the relative overhead of Click scheduler. We also see that ShieldBox achieves line rate at 512 byte packets.

6.3 Latency

We have also measured the packet processing latency using the following scheme: the load generator continuously generates a UDP packet and waits for its return from the enclave. We study packet round-trip time measured at the load generator. On the ShieldBox instance, we are running the `EtherMirror` application. (We omit the results for other applications due to the space constraint.) For this measurements, we did not perform any latency-specific tuning of the environment other than thread pinning, which is enabled by default in DPDK. We expect that a production system with stringent requirements for low latency will use SCHED_FIFO scheduler and have isolated cores.

Figure 8 presents the latency measurements for `EtherMirror` with varying packet size. The low performance of ShieldBox without optimizations is explained by the fact that ShieldBox executes `clock_gettime` system calls in the timer event scheduling code. Scone system calls are optimized for raw throughput with a large number of threads, but not for low

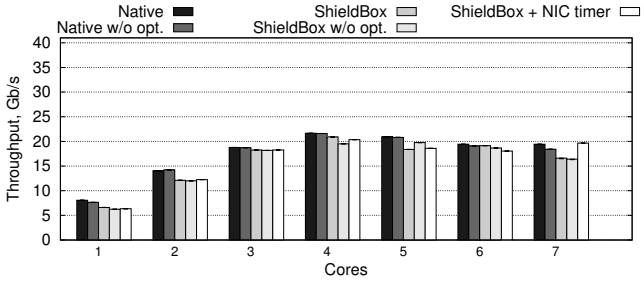

Figure 9: Scalability: Firewall w/ increasing cores

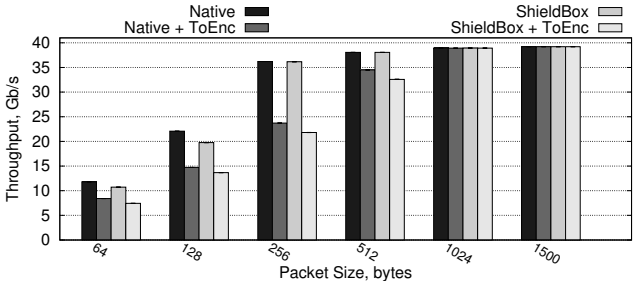

Figure 10: ToEnclave Throughput: EtherMirror measurements

latency; this makes the latency measurement result 3× worse than the native execution. We have considered the following latency optimizations:

- Reduced system call rate for immediately-scheduled timer events. It removes one system call round-trip from the packet latency.
- Modified scheduler that prioritizes immediately-scheduled events and allows to remove a system call from scheduler if there are no periodic timer events.

One of the surprising results that we have is that each of these optimizations does not have a statistically significant influence when applied individually, which can be explained by the fact that once the system call thread has left the back-off mode, it will execute system calls with low individual overhead. On the other hand, when applied simultaneously, they return the latency to almost-native levels—influence of SGX and SCONE on the latency is extremely small.

We consider using NIC timer as a separate optimization. One can see that reading NIC timer is a costly operation; it happens twice per packet in our measurements, adding approximately $0.9*2=1.8\mu$sec to the total latency. On the other hand, it is much faster than executing clock-reading system calls, and can further improve system timeliness when combined with other optimizations.

6.4 Scalability

We next evaluate ShieldBox's scalability with increasing number of cores. (Our latest SGX-enabled server has maximum of 4 cores / 8 HT. Recently released Intel X-Series will consist of 18 cores.) Figure 9 presents the throughput for `Firewall`. (We omitted the throughput measurements for other applications due to the space constraint.) The scalability of both ShieldBox and CLICK is limited. We can see that the performance for both native and ShieldBox peaks at four cores. This is due to the fact DPDK and ShieldBox work best with hyperthreading disabled. This is also confirmed by the poor scalability of native CLICK.

6.5 ToEnclave Overheads

Throughput. We next measure the throughput of the new secure `ToEnclave` element added in ShieldBox, which is used to copy the packet data inside SGX enclave protected memory. We evaluate the impact of this extra data copy by measuring the throughput scaling with varying packet size. Figure 10 shows the results for `EtherMirror`. (We have omitted other applications due to the space constraint.)

We can see that the overhead of the extra memory copy peaks with small packet sizes. This is because for each received packet, operations with rather high overhead must be executed to allocate the packet. One way to reduce this cost would be to batch the memory allocation for all packets. Note that the overhead of ShieldBox compared to the native execution is relatively small: ShieldBox with `ToEnclave` is running within 88% of the native version with extra memory copy in the worst case of small packet sizes, and within 60% of the native CLICK without `ToEnclave` element.

Latency. The latency impact for the `ToEnclave` element is as follows: at 64 byte packets (median, 95th percentile) latency changes from (14.25,15.04) to (14.51,15.24) μsec, at 1500 byte packets it changes from (16.39,17.39) to (17.49,18.24) μsec.

Phase	Average Duration, µsec
Attestation	19467
CAS communication	19301
LAS communication	1474
Configuration	825.6
Total time	26368

Table 3: Overheads of configuration and attestation

6.6 Configuration and Attestation

We next evaluate the overheads of the configuration and attestation service in ShieldBox. The measurement results are presented in Table 3. The results show that remote attestation has a negligible effect on ShieldBox's startup time. Furthermore, even though TLS session establishment is a costly operation, it is performed once per instance start-up, allowing an operator to use a single CAS node for thousands of ShieldBox instances.

6.7 NFVs Chaining

To measure the throughput of the NFV chaining scheme, we have implemented a chaining application. The chaining application implements packet communication between two

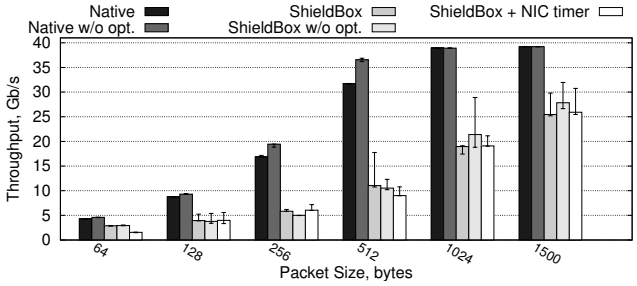

Figure 11: NFV chaining application throughput

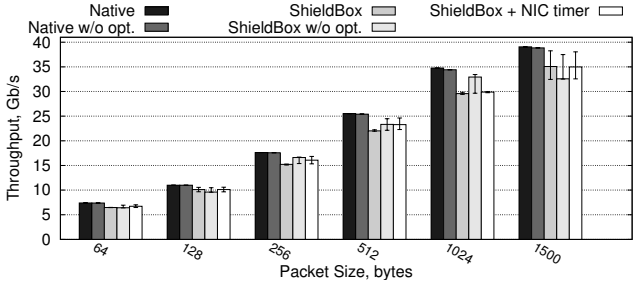

Figure 12: Seal application throughput

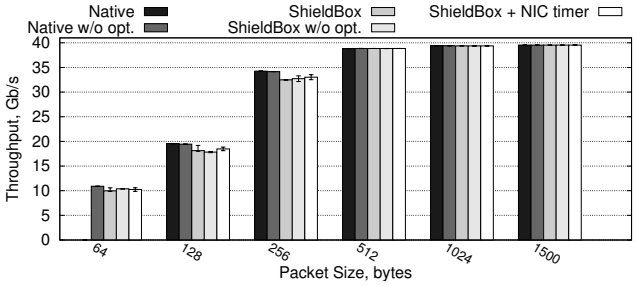

Figure 13: IPRouter: Throughput measurements

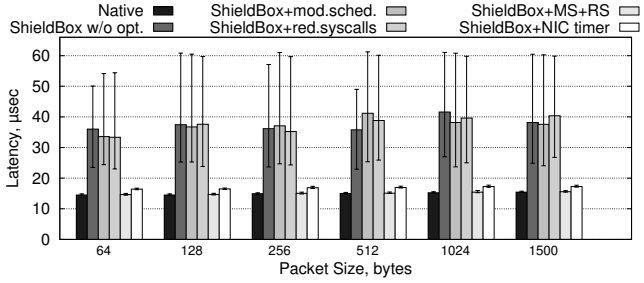

Figure 14: IPRouter: Latency measurements

ShieldBox instances running on the same machine through a DPDK packet ring. One instance contains an application that receives packets from the network and sends them to the other instance via the `DPDKRing` element. The second instance receives packets from the ring and sends them back through a different `DPDKRing` element. These packets are received by the first ShieldBox forming a circular ring. Thereafter, the packets are transmitted back to the load generating node. Note that the packets cross the rings twice. The chaining application showcases the worst-case scenario for us since the NF elements are not performing any computation.

Figure 11 presents the throughput with varying packet size for the NFV chaining application. The results show that using the ring communication causes a substantial performance drop for ShieldBox independent of the optimizations. This is mostly related to the way Scone runs enclaves—it must allocate a constantly-running thread for the service threads created by ShieldBox and DPDK. Due to this, there is interference between the service threads and processing cores, which decreases the throughput and also increases the variance.

Importantly, note that our experiment for the NF chaining across multiple enclaves shows the scenario where two middleboxes are operated by different network operators, who may not necessarily trust each other. Whereas, the performance of NF chains within a single enclave would still be comparable to the native execution.

6.8 Packet Sealing Performance

We next evaluate throughput of the `Seal/Unseal` secure elements. In particular, we use our AES-GCM encryption code running inside the SGX enclave. Figure 12 presents the throughput of the `Seal` element with varying packet size. The result shows that the code inside SGX enclave runs within 88% of the native performance irrespective of the optimizations applied. This is explained by the fact that most of the application CPU time is spent doing the encryption. The difference between the native and SGX version can be explained by different thread scheduling strategies used by Scone and native `POSIX`. In `POSIX`, threads are pinned to the real CPU cores, while in Scone, the userspace threads inside enclave are pinned to the in-enclave kernel threads. This makes thread pinning non-deterministic—sometimes two threads that are to be pinned to different cores are pinned to sibling hyper-threads.

6.9 Case Studies

We next evaluate ShieldBox's performance with the following two case-studies: (1) `IPRouter`, and (2) `IDS`.

IPRouter. `IPRouter` application is an adaptation of a multi-port router Click example application to our evaluation hardware. This application first classifies all packets into three categories: ARP requests, ARP replies, and all other packets. ARP requests are answered. ARP replies are dropped. Other packets are passed to a routing table element that sends them to the NIC output port. Figure 13 shows the throughput of the `IPRouter` application with varying packet size. We can see that ShieldBox has the same performance as Click with packet sizes bigger than 256 bytes, and performs within 90% of Click with smaller packets.

We also measured the latency of the `IPRouter` application as presented in Figure 14. We can see that even if the number

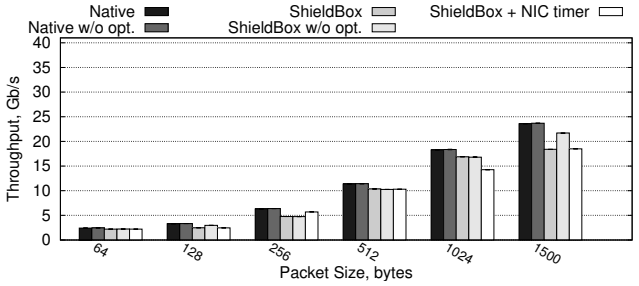

Figure 15: IDS: Throughput measurements

of elements in the application increases, the latency of the application remains the same as the native execution.

Intrusion Detection System (IDS). `IDS` application implements NF that is commonly found in the enterprise network. It pushes the traffic through the firewall, and then performs traffic scanning with the `HyperScan` element. Traffic that does not match any pattern is sent to the output, while matching traffic passes through a counter and then dropped.

ShieldBox performs as close to the native Click execution with a slight performance drop. This drop comes from the general SGX overhead for memory accesses. Due to the space constraint, we omit the latency measurements result for IDS.

7 Related Work

Middleboxes. Click's [26] modular architecture has been leveraged to build many useful software-based middleboxes [11, 13, 22, 22, 31, 33]. Our work also builds on the Click architecture, but unlike the previous work, ShieldBox focuses on securing the Click architecture on the untrusted hardware.

Most Click-based network appliances operate at the L2-L3 layer, with the notable exception of CliMB [30]. To support flow-based abstractions, many state-of-the-art middleboxes [9, 10, 20, 32, 47] support comprehensive applications and use-cases. Since both Click and DPDK are geared toward L2-L3 network processing, our current architecture does not support L4-L7 NFs. As part of the future work, we plan to integrate a high-performance user-level networking stack [21] in the Scone framework to support the development of secure higher layer network appliances.

Secure middleboxes. APLOMB [50] is one the first systems to showcase that it is a viable alternative, performance- and cost-wise, to outsource middleboxes from the enterprise environment to the cloud. However, APLOMB did not consider the security implications of outsourcing in the cloud. To overcome the limitation of APLOMB, the follow-up systems, namely Embark [29] and BlindBox [51], advocate network data processing over the encrypted traffic. In particular, BlindBox [51] proposes an encryption scheme based on garbled circuits to support string matching operations over encrypted traffic. However, Blindbox supports only a restricted type of functionalities, such as NFs for DPI. To overcome this limitation, Embark [29] extends BlindBox to support a wider range of NFs. However, Embark suffers from prohibitively low performance as it involves complex cryptographic computations over the encrypted network traffic. In contrast, ShieldBox supports a wide range of NFs (same as plain-text), and achieves a near-native throughput and latency.

The recently published workshop papers [16, 18, 25] have elaborated the challenges and potential usages of SGX in the network applications. In the domain of network-intensive applications, SGX-Tor [24] is one of the first systems to use SGX to enhance the security and privacy of Tor. In a similar vein, CBR [39] leverages SGX to support privacy-preserving routing. Likewise, the ShieldBox project builds the first comprehensive system using Intel SGX to secure middleboxes.

The two other concurrent research projects also investigated secure deployment of NFs: First, SafeBricks [40] is a system for outsourcing NFs to the untrusted cloud. It has high isolation properties stemming from Rust type system, and implements least privilege principle for NFs. Secondly, mbTLS [34] presents a modification to TLS v1.2 protocol that allows seamless and secure integration of middleboxes into connections between two peers. It leverages Intel SGX to authenticate the middlebox, and has a high level of backward compatibility with legacy peers.

Shielded execution. Shielded execution provides strong security guarantees for legacy applications running on untrusted platforms [12, 28, 37, 42, 45, 52, 55]. Our work leverages shielded execution based on Intel SGX. It is worth noting that unlike the prior usage of shielded execution for commonly used services like HTTP servers or KV stores, we need to adapt the shielded execution to process the network traffic at line rates. To achieve this, ShieldBox is the first system that integrates a high-speed packet I/O framework [2, 19, 41] with shielded execution.

8 Conclusion

In this paper, we presented the design, implementation, and evaluation of ShieldBox—a secure middlebox framework for deploying high-performance network functions (NFs) on untrusted commodity servers. ShieldBox exposes a generic interface based on Click to design and implement a wide-range of NFs using its out-of-the-box elements and C++ extensions. To securely process data at line rate, ShieldBox integrates a high-performance I/O processing library (Intel DPDK) with a shielded execution (Scone) framework based on Intel SGX. We have also added several new useful features, and optimizations for secure end-to-end network processing. Our evaluation using a wide-range of NFs and case-studies show that ShieldBox achieves near-native throughput and latency.

Acknowledgements. We thank our shepherd Aurojit Panda for the helpful comments. This project was funded by the European Union's Horizon 2020 program under grant agreements No. 645011 (SERECA) and No. 690111 (SecureCloud).

References

[1] Docker Hub. https://hub.docker.com/. Last accessed: February, 2018.

[2] Intel DPDK. http://dpdk.org/. Last accessed: February, 2018.

[3] Intel Software Guard Extensions Remote Attestation End-to-End Example. https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example. Last accessed: February, 2018.

[4] Kinetic Disks. https://www.openkinetic.org/. Last accessed: February, 2018.

[5] New approaches to network fast paths. https://lwn.net/Articles/719850/. Last accessed: February, 2018.

[6] Snort. https://www.snort.org/. Last accessed: February, 2018.

[7] `perf`: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Last accessed: February, 2018.

[8] Wolf SSL Library. https://www.wolfssl.com/. Last accessed: February, 2018.

[9] A. Alim, R. G. Clegg, L. Mai, L. Rupprecht, E. Seckler, P. Costa, P. Pietzuch, A. L. Wolf, N. Sultana, J. Crowcroft, A. Madhavapeddy, A. W. Moore, R. Mortier, M. Koleni, L. Oviedo, M. Migliavacca, and D. McAuley. FLICK: Developing and Running Application-Specific Network Services. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2016.

[10] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2012.

[11] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming Slick Network Functions. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, 2015.

[12] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[13] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2016.

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

[15] V. Costan and S. Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, 2016.

[16] M. Coughlin, E. Keller, and E. Wustrow. Trusted Click: Overcoming Security Issues of NFV in the Cloud. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks Network Function Virtualization (SDN-NFVSec)*, 2017.

[17] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. Edge-centric computing: Vision and challenges. *SIGCOMM CCR*, 2015.

[18] J. Han, S. Kim, J. Ha, and D. Han. SGX-Box: Enabling Visibility on Encrypted Traffic Using a Secure Middlebox Module. In *Proceedings of the First Asia-Pacific Workshop on Networking (APNet)*, 2017.

[19] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the 2010 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[20] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[21] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[22] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless Network Functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2015.

[23] G. P. Katsikas, G. Q. Maguire Jr., and D. Kostic. Profiling and Accelerating Commodity NFV Service Chains with SCC. *Journal of Systems and Software*, 2017.

[24] S. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[25] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, 2015.

[26] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 2000.

[27] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2018.

[28] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBounds: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[29] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[30] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi. CliMB: Enabling Network Function Composition with Click Middleboxes. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMIddlebox)*, 2016.

[31] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[32] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2014.

[33] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[34] D. Naylor, R. Li, C. Gkantsidis, T. Karagiannis, and P. Steenkiste. And then there were more: Secure communication for more than two parties. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2017.

[35] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *CoRR*, abs/1702.00719, 2017.

[36] V. A. Olteanu and C. Raiciu. Efficiently Migrating Stateful Middleboxes. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[37] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[38] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[39] R. Pires, M. Pasin, P. Felber, and C. Fetzer. Secure Content-Based Routing Using Intel Software Guard Extensions. In *Arxiv*, 2017.

[40] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, 2018.

[41] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[42] S. Arnautov et al. SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[43] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[44] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proceedings of the 21st USENIX Conference on Security Symposium (USENIX Security)*, 2012.

[45] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[46] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Arxiv*, 2017.

[47] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *In the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[48] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[49] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[50] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[51] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[52] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[53] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[54] B. Trach, A. Krohmer, S. Arnautov, F. Gregor, P. Bhatotia, and C. Fetzer. Slick: Secure middleboxes using shielded execution. *CoRR*, abs/1709.04226, 2017.

[55] C.-C. Tsai, D. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[56] W. Wu, K. He, and A. Akella. PerfSight: Performance Diagnosis for Software Dataplanes. In *Proceedings of the 2015 Internet Measurement Conference (IMC)*, 2015.

[57] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (Oakland)*, 2015.