A Secure Middlebox Framework for Enabling Visibility Over Multiple Encryption Protocols

Juhyeng Han, Seongmin Kim, Daeyang Cho, Byungkwon Choi, *Member, IEEE*, Jaehyeong Ha, and Dongsu Han, *Member, IEEE*

***Abstract*— Network middleboxes provide the first line of defense for enterprise networks. Many of them typically inspect packet payload to filter malicious attack patterns. However, the widespread use of end-to-end cryptographic protocols designed to promote security and privacy, either inhibits deep packet inspection in the network or forces enterprises to use solutions that are not secure. This article introduces a complete framework for building secure and practical network middleboxes, called EVE, which enables visibility over encrypted traffic. EVE securely processes encrypted traffic using a combination of hardware-based trusted execution and software security technology. For enhanced programmability and security, EVE provides a high-level programming interface based on the Rust language. The high-level APIs of EVE provide security and significantly ease the development effort by hiding the details of cryptographic operations, enclave processing, TCP reassembly, and out-of-band key sharing. Our evaluation shows EVE supports diverse use cases with multiple encryption protocols in a secure fashion while delivering high performance.**

***Index Terms*— Network middleboxes, encryption protocols, trusted execution environment (TEE), deep packet inspection.**

I. Introduction

NETWORK middleboxes serve as the first line of defense for many public-facing Internet applications and enterprise networks. A wide range of security-related in-network functions that inspect network traffic, such as web firewalls [42] and intrusion detection/prevention systems (IDS/IPS) [50], [60] have been deployed by enterprises and organizations. However, the widespread use of encryption protocol, such as SSL/TLS, renders functionalities of these network middleboxes useless because they cannot perform deep packet inspection (DPI) on encrypted traffic. This trend forces network operators and users to make an undesirable choice between *end-to-end privacy* and *security* [57].

Manuscript received June 27, 2019; revised May 27, 2020; accepted August 10, 2020; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor P. Papadimitratos. Date of publication August 24, 2020; date of current version December 16, 2020. This work was supported in part by the Institute of Civil Military Technology Cooperation Center under Grant 18-CM-SW-09. The work of Juhyeng Han was supported in part by National Research Foundation of Korea under Grant NRF-2019-Global Ph.D. Fellowship Program. *(Corresponding author: Dongsu Han.)*

Juhyeng Han, Daeyang Cho, Byungkwon Choi, Jaehyeong Ha, and Dongsu Han are with the School of Electrical Engineering, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea (e-mail: sparkly9399@gmail.com; oceancho2@kaist.ac.kr; nfsp3k@gmail.com; cubeprince@gmail.com; dongsu.han@gmail.com).

Seongmin Kim is with the Department of Convergence Security Engineering, Sungshin Women's University, Seoul 02844, South Korea (e-mail: dallas1004@gmail.com).

Digital Object Identifier 10.1109/TNET.2020.3016785

In an attempt to address the problem, prior works take three main approaches: 1) modify TLS to explicitly include middleboxes during a handshake [44], [45]; or 2) use cryptographic schemes that allow direct inspection on the encrypted traffic [57]; or 3) utilize hardware-based trusted execution environments (TEEs) to securely decrypt encrypted traffic [10], [11], [18], [20], [44], [52], [64]. However, the first two approaches have critical concerns about deployment and performance. Modifying the TLS protocol [45] requires changes to existing middleboxes, server, and client applications. Leveraging new cryptographic schemes, such as searchable [57] or homomorphic encryption [17], is notoriously slow for practical purposes. Thanks to a recent innovation in commodity TEEs, an adoption of hardware-based TEE has probably become the most promising option among the proposed approaches. In particular, Intel Software Guard eXtensions (SGX), x86 ISA extensions for security purpose, delivers native performance and supports multi-threading, which makes it practical for performance-critical applications, such as middleboxes. In fact, a new research trend that leverages Intel SGX on middlebox systems has begun to emerge by several recent studies [11], [18], [20], [34], [44], [52], [58], [64].

However, handling encrypted traffic inside SGX enclaves in a secure fashion involves non-trivial challenges that place a heavy burden on middlebox developers. First, processing encrypted traffic involves reassembling TCP traffic, understanding the details of cryptographic protocols to enable key sharing, in-middlebox decryption, and multiplexing flows within encryption tunnels. A lack of programming abstraction on such operations enforces developers to struggle with the low-level implementations.

Second, existing SGX-enabled middleboxes have limitations in flexibility to embody diverse network functions in various network conditions and encryption protocols. Existing approaches that utilize SGX can be a starting point to implement secure network functions that handle encrypted traffic. However, each of them has several constraints: being specialized for a single protocol and cipher suite only [44], [64]; or relying on a trustworthy gateway [10], [11], [52] for encrypted traffic decryption; or assuming unusual deployment model [18]. Moreover, none of the above approaches can handle nested encryption which is frequently appeared in enterprise network [46], [53] (e.g., TLS-encrypted traffic on top of VPN tunnel) nor datagram-based encryption such as Datagram Transport Layer Security (DTLS) which is gaining

popularity within Internet of Things (IoT) applications [33], [70]. Such limitation leads to significant engineering efforts for implementing a variety of network middleboxes that handle encrypted traffic.

Finally, making the system secure presents another set of challenges. Even though SGX provides a hardware-based memory protection, attackers can still exploit software vulnerabilities (e.g., out-of-bounds access) inside the secure memory region, called an enclave [35], [37], [56]. Securing an enclave program is especially challenging because middlebox developers, like others, often utilize third-party libraries [30], [44], [64] that may contain vulnerabilities. For example, when a middlebox software, running inside an SGX enclave, uses an unpatched version of OpenSSL [47] that has Heartbleed vulnerability [12], TLS session key can be leaked (§VI-D). To address this, recent work [52] leverages a safe language, Rust [40], to mitigate the potential vulnerability in the enclave module. However, it is non-trivial for middlebox developers to port existing third-party libraries to Rust, while preserving the functionality and performance.

To address the issues, we present EVE,[1] a complete platform for building a secure and practical middlebox that handles encrypted traffic. EVE utilizes innovations in hardware-based trusted execution and provides programming abstractions for middlebox developers to handle encrypted traffic in a secure manner. We leverage Intel Software Guard eXtensions (SGX) [41] that protects program's code and data inside a secure container. Unlike existing approaches that utilize SGX, we provide high-level abstractions for enabling visibility over encrypted traffic and relieving engineering efforts of adopting encryption protocols, and ensure SGX-aware memory safety all at the same time. As a result, the system can support diverse use cases involving encryption and allows developers to use third-party libraries in a more secure fashion. Finally, our system ensures the integrity of middleboxes running on a remote platform.

In summary, we make four key contributions:

- EVE's programming interface abstracts away the specifics of encrypted flow processing and in-enclave flow decryption for developers to express operational policies as if they are dealing with plain-text data.
- EVE supports diverse use cases and delivers high performance. EVE programming abstraction allows us to support multiple encryption protocols including TLS, DTLS, VPN tunnels and nested encryption. We extend a user-level TCP stack for high performance and flexibility.
- EVE provides protection against memory safety attacks. To mitigate memory safety attacks on enclave code [62], we use high-level APIs in a safe language, Rust [40], and the state-of-the-art boundary checking mechanism [35].
- EVE coherently integrates large systems, including DPDK [23], mOS [26], SGXBounds [35], SGX-Rust [55], OpenSSL [47], and OpenVPN [48], to achieve the goals.

[1]EVE stands for "Enabling Visibility over Encrypted traffic".

Our evaluations show that EVE supports diverse use cases, provides high performance, and enhances the security of middleboxes.

II. Background and Motivation

A. Background

Intel SGX [41] provides a hardware-based mechanism to ensure the integrity and confidentiality of applications. It allows a developer to protect security-sensitive data (e.g., private keys) and operations inside a secure memory region called *enclave*. An enclave is mapped to the enclave page cache (EPC), which is a hardware encrypted address space in main memory access-controlled by the CPU. The content of EPC is only decrypted inside the CPU package using processor-specific keys. Thus, even the privileged software (e.g., OS and hypervisor) cannot access the enclave content, ensuring isolated execution. SGX also provides remote attestation that allows a service provider to verify the integrity of a remote program running on an SGX CPU [2].

mOS framework: EVE utilizes the mOS [26] framework to perform I/O and TCP processing outside the enclave. mOS is a reusable networking stack for middlebox development that provides high performance by using a user-level TCP stack [27] and Intel DPDK [23]. It abstracts management of TCP state of individual connections and provides an event-driven interface to manage network flows in a flexible manner.

B. Challenges and Requirements

Although SGX offers hardware-based security primitives, building a secure middlebox for handling encrypted traffic involves many challenges. A naïve adoption of SGX is not sufficient for a number of reasons.

High-level abstractions for enabling visibility: Building a secure middlebox module requires expertise in multiple domains, such as software security, cryptography, enclave programming, encryption protocol, and network packet processing. System components must be well designed with appropriate abstractions that encapsulate implementation details. A secure key sharing mechanism must also be in place. While existing works [26], [32] provide useful abstractions for middlebox programming, they do not cover enclave programming and specifics of encrypted traffic decryption, and they are not designed for a strong threat model that EVE supports.

Flexible protocol support: The middlebox framework needs to support a variety of encryption protocols and use cases and be deployable in a variety of network environments. Several studies [1], [61] show that tunneling encryption and end-to-end encryption have different benefits and limitations. End users or network services can combine both of the encryption protocols to enhance their privacy and network security [21]. For example, an end host can generate TLS-encrypted traffic to an enterprise network which uses VPN tunnel service. In this case, the middlebox located in the middle of VPN tunnel should be able to process the traffic in the presence of nested encryption (TLS over VPN tunnel). Also, datagram-based encryption protocols such

as DTLS has become popular for resource-constrained IoT applications [33], [70].

To support multiple use cases, the design should be protocol-independent. For example, a design that extends TLS such as mbTLS [44] would not be applicable to a VPN connection that uses a pre-shared secret or other key agreement protocols. Finally, the cost of deployment should be low. The middlebox deployment should avoid the changes of client system or the legacy protocols (e.g., SSL/TLS) that often take a considerably long time.

SGX-aware memory safety: Enclave programming introduces challenges in security. SGX cannot prevent attacks that stem from vulnerabilities within the enclave code, such as out-of-bounds memory accesses [35], [56]. Such attacks can leak security-sensitive data including plain-text payload or session keys. One may combine SGX with a safe language, such as Rust [40] that does not allow pointer arithmetic and performs boundary checking to prevent out-of-bounds read/write at run-time, similar to SafeBricks [52]. However, it is often difficult to build a complex system without the support of legacy ecosystem that has a large code base. For example, the Rust OpenSSL port [54] only provides Rust APIs internally connected to the original OpenSSL library written in C. In summary, the code is not safe unless all system components running inside the enclave are protected, but porting the entire library is non-trivial.

III. Our Approach

To satisfy the requirements (§II-B), EVE takes three main approaches. First, EVE provides high-level Rust APIs that ease the programming efforts and allow developers to program their own operational policy of middleboxes. It enables developers to build a complex middlebox system without knowing the details of utilizing SGX, flow management, and encryption protocols.

Second, to handle diverse use cases in a variety of network environments such as nested encryption with high performance, we design out-of-band key sharing, flow-key associating module and queue-engine pipeline (§IV-A). Note we design each EVE component to be protocol-independent, so deploying EVE does not require any changes in client system and legacy protocols. This reduces the deployment cost of EVE.

Finally, EVE supports memory safety for SGX enclave. We encourage the use of a safe language inside the enclave by providing Rust APIs. For legacy code, we employ enclave hardening by applying SGXBounds [35], a state-of-the-art software-based address boundary checking mechanism for enclave programs. It mitigates the potential memory vulnerabilities, such as stack smashing and Heartbleed attack [12], in third-party libraries implemented with an unsafe language (e.g., C/C++).

Deployment model and assumption. The EVE framework consists of a controller, a middlebox, and an end server. Figure 1 shows our deployment model. Our basic assumption is that network operators cooperate with servers to deploy their middlebox on an SGX-enabled third-party platform for protecting the servers. The controller manages the middlebox configuration and provisions a ruleset to the middlebox through a secure channel provided by SGX. After the ruleset is loaded in EVE middlebox, the end server verifies the integrity of the middlebox and the ruleset. Detailed attestation and rule provisioning procedure are explained in §IV-E. We assume that the middlebox platform, except for the SGX hardware and the enclave, is untrusted.

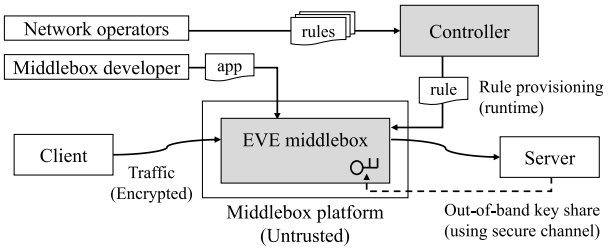

Fig. 1. Deployment model of EVE.

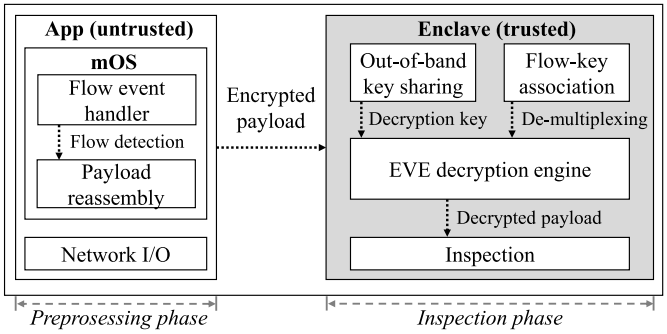

Fig. 2. System components of EVE.

Threat model. Upon our deployment model, we assume attackers whose goal is to obtain the confidential information of a middlebox such as plain-text payloads of encrypted traffic, session secrets, and DPI rulesets. We assume that the attackers are capable to control privileged system software (e.g., OS and hypervisor), and hardware components (e.g., memory and I/O bus), except the CPU package. In addition, the attackers can exploit vulnerabilities in an enclave code (e.g., such as vulnerabilities in the cryptographic protocols [12]) unlike prior systems [30], [44], [58], [63], [71]. However, we do not consider side-channel attacks on SGX [19], [38], [66], [67], [69], including page fault [66], [69], cache [19], [43], branch-prediction [38] and synchronization [67] side-channels. We note that side-channel attacks and mitigation strategies on SGX are an active area of research [7], [15], [22], [56], [59], [71]. Also, we want to prevent attackers from compromising the rulesets to avoid DPI checking during the middlebox operation. Finally, we assume the end server is trusted. Optionally, the server can also utilize SGX, in which case the server assumes the same trust model as the middlebox.

IV. EVE Design

EVE handles encrypted traffic in two phases, as shown in Figure 2. In the preprocessing phase, EVE reassembles

TABLE I
SHARED SESSION CONTEXT

Time of sharing	Shared data
Channel establishment	Cipher list, Certificate, TLS private key
Arrival of TLS flow	Flow tuple (common), Initialization vector, Server random, Premaster secret
Arrival of VPN flow	Flow tuple (common), Pre-shared key

encrypted payloads before packet processing. To support stateful flow-level processing, EVE leverages mOS [26], a framework that extends a user-level TCP stack [27] for middleboxes. The preprocessing phase that handles encrypted payloads, including the mOS framework, is conducted in the untrusted region in an effort to reduce the trusted computing base (TCB) and carefully utilize the limited EPC memory.[2] This does not leak private data because the reassembled payload still has the same security level as end-to-end encryption (e.g., TLS).

In-enclave processing occurs in the inspection phase. The payload is decrypted and the EVE DPI module inspects the payload for middlebox processing (e.g., exact matching of IDS), ensuring that the plain-text never leaves the enclave. EVE's enclave region consists of four components. 1) The out-of-band key sharing module securely retrieves a session key from an end server. 2) The flow-key associating module associates a flow with its decryption context that includes a decryption key and a cipher suite. 3) The decryption engine uses a queue-engine pipeline that performs decryption in multiple stages to handle nested encryption. 4) The inspection module conducts pattern matching based on the rules and performs actions (e.g., send alert) based on the result.

A. In-Enclave Components

We now describe the in-enclave components in detail.

Secure out-of-band key sharing. EVE middlebox retrieves session keys from servers who perform remote attestation and build a secure channel with the EVE middlebox. The SGX remote attestation ensures that EVE middlebox is running on an SGX platform and the integrity of its code is valid. EVE also authenticates the server and checks the validity of the server's certificate. When the attestation and authentication are successful, EVE middlebox and the server establish a TLS session which is terminated inside the EVE enclave to ensure shared secret is not exposed to an untrusted party. The server shares its per-session context shown in Table I,, including the shared secret. If the attestation fails, the end server disconnects the secure channel with the target middlebox without sharing the key.

Flow-key associating module. Note the out-of-band channel is established once for each server, but key sharing occurs for each session asynchronously. EVE middlebox needs to associate the session key upon arrival of encrypted flow. For this, EVE uses the 5-tuple to distinguish incoming flows. EVE stores the flow tuple in the preprocessing phase. Then, the server sends the flow tuple with the session secrets during out-of-band key sharing. Finally, EVE matches the flow tuple to associate the flow with the session key to decrypt it.

[2]The maximum EPC size supported by hardware is less than 128MB.

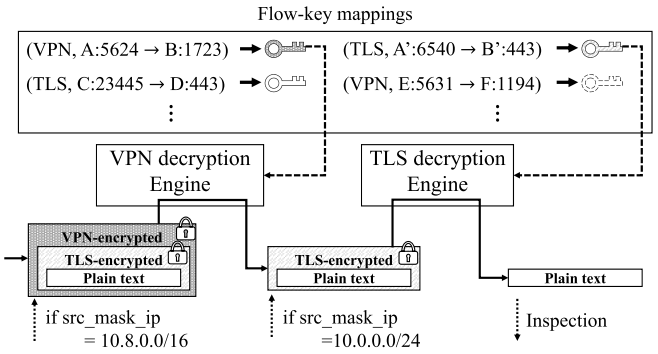

Fig. 3. Handling nested encryption in EVE.

In the presence of nested encryption, EVE performs multiple out-of-band key sharing with servers. Each server shares its session key and flow tuple with EVE through the out-of-band secure channel. Then, EVE stores the tunnel information, session key, and the inner flow tuple together, similar to a key-value store. We note that EVE needs to keep the tunnel information to handle nested encryption. Assume that the end-to-end traffic is encrypted with TLS over VPN tunnels. If two flows have the same flow tuple and arrive through different VPN tunnels, EVE cannot associate each flow with the corresponding session key only using a flow tuple from the end server. To address this problem, we modify a VPN server and an end server, allowing EVE to store both inner and outer flow tuples, and session key together. During the connection establishment, the VPN server sends its flow tuple to the end server. The end server then transmits the received tuple with its flow tuple and session key to EVE when performing out-of-band key sharing. This enables EVE to identify unique session key for each flow.

Decryption engine automates the process of decryption and flow management. EVE allows programmers to associate a decryption queue with a flow. When this is done, EVE's flow management directs reassembled payload of the incoming flow into the queue. The decryption engine then fetches the reassembled payload by the unit of encryption and performs decryption using a session key shared from the end server. Decryption queue and its pipeline model are one of the key abstractions of EVE. The queue-engine pipeline model simplifies the process of handling nested decryption as Figure 3 shows. Supporting nested decryption amounts to directing the output of the decryption engine to the next decryption queue (see §IV-B and §VI-A for more details).

Payload inspection. EVE supports both exact string pattern matching and regular expression matching on plain text inside the enclave to protect the plain text from untrusted software running on the middlebox host. EVE provides a programming interface to choose the algorithm among multiple imported pattern matching libraries. By default, we provide DFC [8] pattern matching algorithm that has a small memory footprint and PCRE2 [51] regular expression matching library within the SGX enclave.

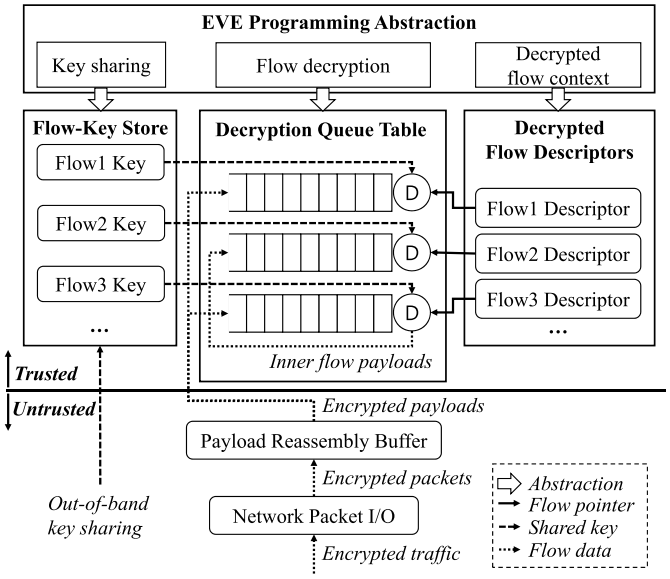

Fig. 4. The overview of EVE abstraction.

B. EVE Programming Abstraction

The key abstraction of EVE is "explicit management of flow context" in the presence of end-to-end encryption. Figure 4 shows the overview of EVE abstraction for encrypted traffic decryption and retrieving decrypted flow context. The EVE abstraction represents a decrypted flow descriptor as a decryption structure (e.g., TLS struct). EVE automatically tracks incoming traffic and stores the per-flow state in a flow context structure after decryption. The flow context structure includes decrypted packet header and payload, cipher suite, and total packet size. Each flow descriptor points to the flow context and a decryption key associated with the flow. Also, EVE decryption queue provides a common interface to receive encrypted payloads from incoming traffic and inner flow. In result, EVE abstraction enables developers to avoid low-level implementations, and handle multiple encryption protocols with a common abstraction. Note the state-of-the-art SGX-based middleboxes that handle encrypted traffic do not consider this. Figure 5 describes step-by-step details for monitoring encrypted traffic in EVE. We now describe how our programming abstraction has advantages in extensibility for each step.

Packet-level processing. EVE internally supports pre-processing on encrypted traffic for both TCP- and UDP-based encryption protocols. To achieve stateful TCP processing which is an essential requirement to handle TCP-based encryption, EVE utilizes mOS [26] that provides a stateful TCP stack. In addition, we extend mOS [26] internal system to support UDP packet. Note that none of existing SGX-based middlebox systems supports UDP-based protocols and some of them [10], [64] even do not provide stateful TCP processing.

Protocol-independent programming abstraction helps in building APIs flexible to handle various encryption protocols. First, we abstract flow reassembly by designing EVE to use the same encrypted payload reassembly buffer, regardless of the type of encryption protocols. We observe that most encryption protocols, such as TLS and VPN, contain record sizes. EVE internally checks if packets are sufficiently gathered in the reassembly buffer by referring the record size in the header.

TABLE II
REUSABLE EVE COMPONENTS AND EXTRA LINES OF CODE (LOC) FOR VPN AND DTLS ADOPTION

Functionality	Reusable component	Extra LoC
Network I/O & packet parsing	DPDK [23] driver mOS [26] internals	None None
Flow management & reassembly	Flow event callbacks EVE reassembly buffer	Less than 100 Less than 100
Payload decryption	Out-of-band key share Flow-key association Port crypto-library Queue-engine pipeline	Less than 30 None Less than 300 Less than 50
Deep packet inspection	DFC [8] pattern matcher PCRE2 [51] regex matcher	None None

Second, our programming abstraction dramatically reduces an effort for sharing session keys. EVE basically establishes a secure channel to receive secrets from a server for building a session decryption context. This is a protocol-independent operation, which means that the corresponding API is reusable. For example, when EVE has an out-of-band key sharing module for TLS secrets, a developer could easily extend it to receive VPN secrets by changing the receiving secret size only. Note that protocol specific key sharing approaches such as mcTLS [45] and mbTLS [44] that rely on handshake message for the key sharing cannot deliver such extensibility.

In addition, the internal enclave code base of EVE reduces implementation effort for further extension. We port widely-used cryptographic and HMAC operations of OpenSSL and OpenVPN libraries in EVE enclave to support TLS, DTLS and VPN protocols, respectively. This lowers the burden of implementing decryption logic and ocall wrappers for SGX-unsupported functions (e.g., system calls) used in cryptographic libraries. Finally, deep packet inspection module is protocol-independent as they perform a pattern matching on plain-text payloads.

Quantifying engineering effort. To quantify the implementation effort when extending EVE to support an additional encryption protocol, we estimate the lines of code while importing VPN and DTLS protocol. Note that we have sequentially expanded EVE to support TLS, VPN and DTLS protocol. We compare the baseline EVE implementation which only supports TLS protocol with VPN-extended version and DTLS-extended version. To build the baseline EVE, we implement 5,450 LoC, including OpenSSL library. To support VPN and DTLS, we extend EVE internals by only adding 314 LoC (5.76% of the baseline) and 317 LoC (5.82% of the baseline), respectively.

Table II summarizes EVE reusable components and additional effort to achieve support for new encryption protocols. Based on our experience of porting VPN and DTLS protocols, to adopt a new encryption protocol, we only need to 1) implement logic of parsing the encryption protocol header

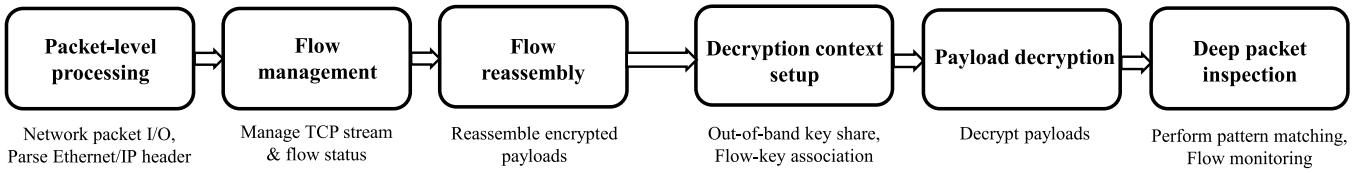

Fig. 5. The process to monitor encrypted traffic for EVE middleboxes.

to extract a record size, 2) change the receiving secret size, and 3) combine cryptographic function in libraries. Note that most EVE abstractions, including network I/O, flow management, payload reassembly, and DPI, can be reused without modifying any EVE internals. This means that EVE has a flexible programming abstraction to support multiple encryption protocols.

C. Programming Model and API

Some middlebox developers might prefer to utilize traditional and well-known programming interface such as Click-based APIs [10], [32], [64]. However, as NetBricks [49] pointed out, a Click module [32] is not flexible for handling diverse use cases, so the developers need to implement new modules or low-level packet processing systems themselves. For example, ShieldBox [64] extends legacy framework, Click router [32], but does not support stateful flow-level packet processing due to the fundamental limitation of its Click modules. Therefore, substantial engineering efforts would be required to support TCP-based encryption protocols on ShieldBox such as implementing TCP networking stack.

In contrast, EVE supports flexible "event-driven" programming model with helpful middlebox APIs. As mOS [26] demonstrates useful event systems for monitoring TCP flows, EVE simplifies the processing of various encrypted flows. The execution of EVE module is determined by two core events: 1) a new arrival of encrypted flow (*flow-in event*) and 2) completion of a decryption unit (*decryption-done event*). For each event, developers can provide user-defined callbacks to implement their own logic using a combination of EVE APIs. EVE provides a total of 11 data structures and 38 EVE APIs to handle flow processing and payload inspection policies. Table III shows the key data structures and EVE APIs that manipulate them, classified into four categories.

Flow management API provides flow context information, such as packet header, count, payload. Also, it supports convenience functions for filtering flows that match certain conditions based on the flow context information. This helps developers to easily specify their per-flow policies in a flow information-aware fashion.

Decryption API provides an interface to the decryption queue. Using `set_queue` API, a developer can associate a decryption queue with a specific decryption scheme. EVE then automatically collects the payloads of the associated flow and retrieves the shared secret from end servers. Then, the corresponding decryption engine deciphers the payload. In addition, developers can retrieve the decryption scheme of both tunnel and inner flows in the presence of the nested

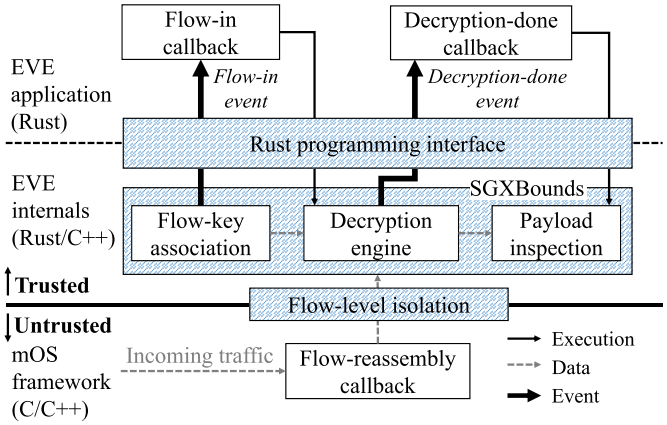

Fig. 6. The interaction between EVE modules.

encryption (TLS over VPN tunnels), using `get_prev` and `get_queue_type` APIs. This enables developers to manage in-tunnel flows with different trusted level. Based on the queue association policy, EVE internally creates a queue-engine pipeline, and EVE runtime drives the pipeline.

Payload inspection API. EVE provides APIs to perform payload inspection on plain-text. One can state the execution of exact string or regular expression matching with `rule_match` API. In addition, we provide an extensible set of rule matching algorithms (e.g., DFC [8] and PCRE2 [51]) in the enclave.

Configuration API is used to specify the middlebox configuration and callback registration. A developer can specify the ruleset for packet inspection with `load_ruleset` API. When the ruleset is requested by the API, EVE internally performs rule attestation with the controller and loads the ruleset if it successfully passes (§IV-E). The `set_callback` API registers event callbacks of each built-in event. Finally, `define_event` API enables developers to define a custom event that is triggered when a user-defined filter condition satisfied. For example, a developer can create an event that is triggered only when the destination port number is 443.

D. Enclave Memory Safety

As shown in §II-B, EVE is required to mitigate the leakage of enclave memory contents by exploiting pointers in C/C++ codes. To thoroughly explore possible enclave code vulnerabilities, we classify EVE enclave code into three parts: 1) high-level APIs, 2) third-party libraries code, and 3) low-level implementations of flow/packet processing. We design EVE security components by considering the characteristics of each enclave code part. Figure 6 shows the interaction between

TABLE III
REPRESENTATIVE DATA STRUCTURES AND EVE APIS SUMMARIZED BY FOUR CATEGORIES

Name	Type	Description
`FlowInfo`	Structure	Stores the flow information.
`match_src/dst`	Flow management API	Matches the source/destination IP and port of the flow.
`get_pkt_count`	Flow management API	Gets the total number of packet of the flow.
`get_payload`	Flow management API	Gets the decrypted payload.
`DecQueue`	Structure	Stores the decryption queue and its assigned decryption engine.
`set_queue`	Decryption API	Assign a decryption engine to the queue.
`get_queue_type`	Decryption API	Returns the current decryption state (e.g., VPN or TLS).
`get_prev`	Decryption API	Returns the previous queue data structure.
`DPIEngine`	Structure	Stores the rule matching algorithm and matching information (e.g., ruleset).
`rule_match`	Payload inspection API	Performs rule matching on the decrypted payload.
`load_ruleset`	Configuration API	Retrieves a ruleset from the EVE controller.
`Event`	Structure	Stores the event information.
`set_callack`	Configuration API	Registers callback function for an event.
`define_event`	Configuration API	Sets an event with a user-defined filter.

EVE modules and the location of EVE security components. We explain each security component.

Secure programming interface. EVE framework supports secure and flexible programming interface by providing secure APIs implemented with Rust [40]. Rust guarantees language-level safety, such as strict out-of-bounds access checking for array indexing and use-after-free prevention. Also, EVE APIs hide the unsafe access to C/C++ structures, but provide parsed flow data in complete Rust structures. Therefore, EVE programming interface helps the developers with implementing their secure EVE application.

Hardening library code. EVE uses several C/C++ code for its enclave code including third-party libraries which may contain vulnerabilities themselves. For example, the OpenSSL [47] contains about 244 k lines of code, which means that TCB might have potential vulnerabilities, such as out-of-bounds memory read/write, caused by mis-implementations.

A straight forward approach to mitigate such vulnerabilities is to convert the libraries into Rust. However, it is often impractical to convert the entire code base. One of the reasons is porting performance-critical code is especially non-trivial. For example, many OpenSSL APIs (e.g., `ssl3_read_bytes`) use low-level pointer arithmetics, but converting them to Rust involves redesigning data structures and rewriting the algorithms.

Instead of converting existing code base to Rust, EVE hardens the enclave code implemented in unsafe languages (e.g., C/C++). This allows middlebox developers to utilize the legacy code base in a more secure manner. For this, we integrate SGXBounds [35], the state-of-the-art memory protection scheme for SGX, that uses tagged pointers to prevent out-of-bounds access and exploits hardware features of SGX for efficiency. To integrate SGXBounds, we provide wrappers for each EVE's in-enclave C/C++ components that takes pointers as parameters, including libc, OpenSSL, and OpenVPN libraries, in a reusable fashion. Specifically, we provide 7 memory-related wrappers, 3 network-related wrappers, and 12 decryption-related wrappers that EVE uses. The underlying SGXBounds run-time then prevents out-of-bounds memory accesses.

Flow-level memory isolation. EVE isolates memory for a flow processing by providing a dedicated enclave thread. To maximize CPU utilization, middleboxes generally utilize multiple threads to concurrently process incoming flows. However, utilizing an enclave thread pool like Talos approach [4], demands delicate thread synchronization code for accessing shared in-enclave structure, which increases the complexity of program and the risk of mis-implementation. For example, concurrent access to shared TLS buffer without correct thread locking might incur the corruption or leakage of decrypted flow data.

Mitigating the above problem, EVE preserves run-to-completion threading model [26] and provides per-core structures for secure in-enclave threading. EVE associates each flow with a specific core. Each core then processes the associated flow utilizing per-core structures, such as TLS/VPN decryption queues, which are allocated for the core. This guarantees flow-level memory isolation because each concurrent flow processing uses separated memory space. In addition, the flow-level memory isolation is beneficial for core scalability compared to the thread-pool approach [4] as it does not require enclave spinlocks for thread synchronization (§VI-B).

E. Rule Provisioning and Deployment

Rule provisioning. To prevent attackers from modifying the rules, EVE performs secure rule provisioning and allows an end server to verify the integrity of ruleset. As we illustrated in Figure 1, an EVE middlebox, an end server, and a controller participate in the rule attestation. Before EVE middlebox requests a ruleset from the controller, the controller performs remote attestation to check the code integrity of the middlebox. If it successfully passes module attestation, the controller chooses the requested ruleset and provisions it to the middlebox. Then, both the middlebox and the controller calculate the hash of the ruleset and send it to an end server through an out-of-band secure channel. Finally, the end server

TABLE IV
LINES OF CODE FOR EVE

Component	Lines of code (LoC)	% Changed
EVE (Enclave)	3,891 lines of C/C++	- (3,891)
EVE (Untrusted)	985 lines of C/C++	- (985)
EVE sample code	63 lines of Rust	- (63)
mOS	35,824 lines of C	0.25% (88)
OpenSSL	243,785 lines of C	0.10% (249)
OpenVPN	1,302 lines of C	6.84% (89)
DFC	2,271 lines of C/C++	0.97% (22)
SGXBounds	280 lines of C/C++	- (280)
Rust APIs	465 lines of Rust	- (465)
Enclave Definition Language.	49 lines of code	- (49)
Scripts	287 lines of code	- (287)
Total	253,378 lines of code	2.52% (6,380)

compares the hash to verify the integrity of the ruleset. If hashes are equivalent, the rule attestation is done and the middlebox is ready to perform packet inspection. This ensures that the EVE middlebox securely retrieves the unmodified ruleset from the controller. To keep the rules private, EVE keeps them in the enclave. When it needs to store them to the file system of the middlebox platform, EVE seals them using the SGX hardware key using the SGX sealing feature [2].

Updated components. In addition to the middlebox, EVE requires server updates to support out-of-band key sharing. To streamline this process, we provide modified OpenSSL/OpenVPN libraries. Existing applications can simply relink our updated library, and applications that use OpenSSL/OpenVPN can readily benefit from EVE. Note EVE does not change legacy protocols (e.g., TLS, DTLS and VPN) and client applications. Thus, the cost of deployment to adopt EVE is relatively small compared to existing approaches [18], [44], [52].

V. IMPLEMENTATION

We develop EVE by using Intel SGX Linux SDK 2.5 [24] and Rust-SGX-SDK [55]. DPDK version is 18.02. Table IV shows the lines of code (LoC) for each EVE component and 6.38K LoC are changed to implement our system in total. We extend mOS [26] internal system to detect a UDP packet arrival event and parse the UDP header. We modify the SSL and OpenVPN libraries to enable out-of-band key sharing and support both dynamic key agreement and static shared key. To support dynamic key agreement, we modify the TLS handshake implementation in OpenSSL. For example, we modified `ssl_fill_hello_random` in EVE middlebox to use nonces provided by the end server. EVE then generates TLS context that has the same session key with the end server during the TLS handshake procedure. For sharing VPN static keys, we modify OpenVPN. During initialization, EVE receives the static key to enclave memory through the out-of-band secure channel with the VPN server.

We port OpenSSL-1.0.2l [47] and OpenVPN-2.4.3 [48] into an SGX enclave instead of using the Intel SGX-SSL library [25]. The Intel SSL library offers crypto APIs but not the TLS. For example, it does not provide TLS APIs

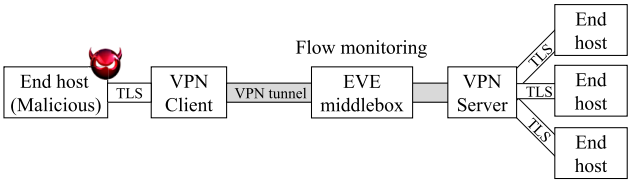

Fig. 7. Use case for nested encryption (TLS over VPN tunnel).

```
// User-defined Callback for flow-in-event
fn cb_flow_in(flow:&mut FlowInfo, queue:&mut DecQueue,
engine:&mut DPIEngine) {
    if flow.proto == Protocol::TCP {
        if flow.dst_port == 1194 {
            queue.set_queue(QueueType::VPN);
        } else {
            queue.set_queue(QueueType::TLS);
}}}
// User-defined Callback for decryption-done-event
fn cb_decryption_done(flow:&mut FlowInfo, queue:&mut
DecQueue, engine:&mut DPIEngine) {
    match queue.get_queue_type() {
        QueueType::TLS => {
            engine.get_DPI_engine()
                .rule_match(MatchAlgo::DFC, flow, queue.get_core());
            queue.forward(); },
        _ =>queue.forward(),
};}
```

Fig. 8. EVE callback functions for DPI on nested encryption.

(e.g., `SSL_read` and `SSL_write`). In contrast, our library offers full TLS and DTLS features and currently supports 8 different cipher suites in TLSv1.2 and a VPN cipher suite that uses 2048-bit static key.

Finally, we port the DFC [8] exact string matching algorithm and a PCRE2 [51] regular expression matching library into the enclave code. We modify about 0.97% (22 out of 2271 LoC) of the original DFC code to port it to the EVE module. We release our SGX-OpenSSL and SGX-DFC port as open source.[3]

SGX Optimizations. We modify SGX SDK library functions to enhance the decryption performance of EVE. Basically, EVE frequently calls `memmove` during TLS decryption to shift payload data in the TLS record buffer, following the current OpenSSL implementation. As previous study [68] pointed out the performance issues raised by the naïve implementation of stdlib functions in SGX SDK, the provided `memmove` function significantly degrades the decryption performance.[4] To address this, we port the recent glibc `memmove` that uses Advanced Vector Extensions (AVX) [14] in the EVE enclave. In result, EVE improves TLS decryption throughput by up to 400.80% in our evaluation (§VI-C).

VI. EVALUATION

Our evaluation answers five questions:

- Does EVE API provide useful abstractions and support diverse use cases?

[3]Each code is available at https://github.com/sparkly9399/SGX-OpenSSL and https://github.com/sparkly9399/SGX-DFC.

[4]Intel provides a quick patch in SGX SDK 1.7 version to deal with the issue, but some functions such as `memmove` were not updated.

```
1 pub fn rust_main() {
2     let rule_name = String::from("ET_PRO.rules");
3     let mut event = Event::new();
4     let mut engine = DPIEngine::new();
5     engine.load_ruleset(MatchAlgo::DFC, rule_name);
6     event.set_callback(EventType::OnFlowIn, cb_flow_in);
7     event.set_callback(EventType::OnDecryptDone,
        cb_decrypt_done);
8 }
```

Fig. 9. The callback registration code of EVE.

```
1 // User-defined Callback for flow-in-event
2 fn cb_flow_in( flow:&mut FlowInfo, queue:&mut DecQueue,
  engine:&mut DPIEngine) {
3     if flow.dst_port == 1194 {
4         queue.set_queue(QueueType::VPN);
5     }
6     else if queue.get_prev().get_queue_type() == QueueType::VPN {
7         let curr_time = rcall_get_time();
8         engine.get_stat().save_flow(flow, curr_time);
9         let cnt = engine.get_stat().search_flow_dst_within(flow.dst_ip,
            curr_time, 1);
10        if  cnt ¿ 10 {
11            log_warn("Port scanning detected");
12 }}}}
```

Fig. 10. Port scanning detector.

- What is the performance overhead of EVE? How does it compare with existing approaches?
- What is the impact of EVE on end hosts?
- How does each EVE component affect performance?
- What types of attacks are mitigated by EVE?

We use Quad core Intel Xeon E3-1280 v6 3.90GHz CPU machines running Linux 4.4.0. We run clients, servers, and an EVE middlebox on different machines connected at 10 Gbps. The servers and clients generate multiple TLS connections. We use long-lived TLS connections unless otherwise noted.

A. Writing EVE Application

EVE API is flexible enough to support diverse use cases. It allows developers to dynamically adapt their operational policies. To write an EVE application, developers need to implement callback registrations and callback functions for flow-in and decryption-done events. We show three realistic EVE applications as examples that demonstrate diverse use cases. Note, existing approaches, such as SafeBricks [52], mbTLS [44] and ShieldBox [64], do not provide such a common set of abstractions for in-middlebox processing of encrypted traffic.

DPI on nested encryption: We present a DPI middlebox that handles TLS traffic over VPN tunnel. The middlebox decrypts outer and inner flow sequentially and performs pattern matching upon the plain-text. As Figure 7 illustrates, we assume the end-to-end traffic is TLS encrypted over an OpenVPN tunnel. Figure 8 and Figure 9 show 25 lines of code that achieves this, demonstrating the power of our high-level abstraction. Note, without EVE APIs, this requires significant programming effort.

Figure 8 shows two callback functions. Upon the arrival of a new flow, `cb_flow_in` is invoked by the EVE runtime. A VPN tunnel is identified by the destination port,

```
1 // User-defined Callback for flow-in-event
2 fn cb_flow_in(flow:&mut FlowInfo, queue:&mut DecQueue,
  engine:&mut DPIEngine) {
3     if flow.proto == Protocol::TCP {
4         queue.set_queue(QueueType::TLS);
5     } else if flow.proto == Protocol::UDP {
6         queue.set_queue(QueueType::DTLS);
7 }}
8 // User-defined Callback for decryption-done-event
9 fn cb_decryption_done(flow:&mut FlowInfo, queue:&mut
  DecQueue, engine:&mut DPIEngine) {
10    match queue.get_queue_type() {
11        QueueType::TLS — QueueType::DTLS => {
12            engine.get_DPI_engine()
13                .rule_match(MatchAlgo::DFC, flow, queue.get_core());
14            queue.forward(); },
15        _ => queue.forward(),
16 };}
```

Fig. 11. EVE callback functions for DPI on both TLS and DTLS traffic.

and a VPN queue is assigned to the flow (line 5). When a VPN queue is assigned, successive packets of the flow will be automatically inserted to the queue, and the flow will be decrypted by the queue-engine pipeline. When a VPN payload is deciphered, decryption-done event is raised and `cb_decryption_done` callback is invoked (line 10). Also, upon the decryption of the VPN flow, EVE would discover that there is an inner flow. EVE then raises the flow-in event that invokes `cb_flow_in` again. In the `cb_flow_in` callback, a TLS queue is assigned to an inner flow of interest (line 7). The flow then is decrypted by the TLS decryption engine. In the `cb_decryption_done` callback (line 10 $\sim$ 17), the plain-text payload of the inner flow is matched against a set of patterns. The callbacks and ruleset are registered as shown in Figure 9.

Port scanning detector: With the same scenario of Figure 7, we present a simple port scanning detector that inspects traffic inside a VPN tunnel when a malicious end host tries to scan the opened ports of other benign hosts in the same private network. It is hard for third-party middleboxes to detect port scanning inside a VPN tunnel because the scanning packets are encrypted. The EVE abstraction allows developers to program port scanning detection of in-tunnel flows as if they were unencrypted.

Figure 10 shows a code snippet of a middlebox that logs a warning message when the number of scanned ports exceeds 10 in a second. The detector first decrypts VPN flows by assigning a decryption queue (line 4). After the decryption, the detector checks whether a flow comes from the VPN queue (line 6). Then, the detector keeps the current time and the flow tuple of the inner flow to track the number of scanned ports (line 8). The detector counts the number of flows that have the same IP address of destination within a second (line 9). Finally, the detector notifies a warning message when the count exceeds the threshold (line 10 $\sim$ 12).

DPI on Internet of Things (IoT) environment: Recent studies [16], [28] present that network IDS is necessary for securing IoT devices. EVE middlebox is applicable to such use cases that require flow monitoring on both TLS-encrypted traffic from a mobile application and DTLS-encrypted traffic from IoT devices. Figure 11 shows EVE callback functions

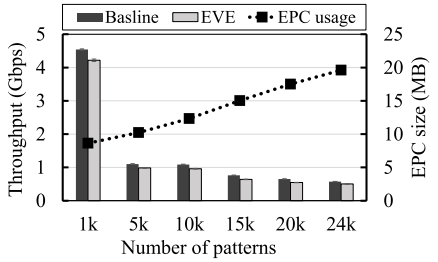

Fig. 12. Pattern matching throughput and EPC usage.

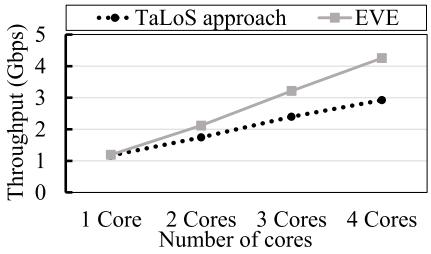

Fig. 13. Core scalability.

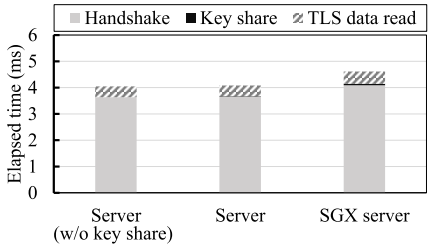

Fig. 14. Flow completion time.

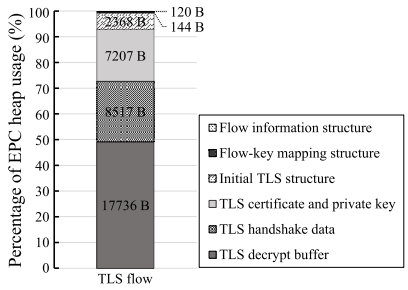

Fig. 15. EPC heap usage for decrypting a TLS flow.

to achieve the goals. In the `cb_flow_in` callback, different decryption queue is assigned to incoming flow by retrieving the L4 protocol of the flow (line 3 $\sim$ 7). When each flow is decrypted, `cb_decryption_done` callback performs DPI on the decrypted plain-text (line 9 $\sim$ 16).

In summary, we show EVE's high-level abstraction is useful for handling encrypted traffic with various encryption protocols and use cases.

B. EVE Middlebox Performance

We now evaluate the performance overhead of the EVE applications and compare them with existing approaches.

Deep packet inspection: We measure the throughput of a middlebox that runs deep packet inspection on TLS-encrypted traffic. For pattern matching, we use the DFC [8] and a PCRE2 [51] regular expression matching engine with a commercial ET-Pro ruleset [13]. We use randomly generated traffic as input while varying the number of patterns from 1k to 24k. Figure 12 shows the throughput of EVE compared with a version that does not have any security features of EVE, including SGX. The EVE middlebox incurs only up to 16.16% of degradation compared to the baseline. The EVE middlebox also efficiently manages EPC memory. As we increase the number of patterns, the EPC heap usage increases as the dotted line shows. Nonetheless, EVE middlebox only uses 19.6 MB of EPC memory with 24 k rules at its peak. This effectively reduces EPC paging that incurs serious performance overhead.

TCB size comparison with existing approaches: We compare the TCB size of EVE which has about 3.30 MB of code size to ShieldBox [64] and SafeBricks [52]. Because the codes of both systems are not publicly available, we use the numbers presented in the papers. ShieldBox puts the most of DPDK [23] code except for the packet buffer and the core parts of CLICK [32] which has about 6 MB of code inside the enclave [64]. In contrast, EVE puts DPDK and mOS [26] code in the untrusted region as we separate the security-insensitive parts of EVE to minimize TCB.

SafeBricks [52] contains less than 1 MB code, which is smaller than that of EVE. However, this gap comes from that SafeBricks supports only a specific IPSec decryption [52]. EVE, in contrast, supports various cipher suites such as TLS, DTLS and VPN protocols by containing OpenSSL [47] and OpenVPN [48] code in its enclave. Therefore, EVE is applicable to diverse use cases with its TCB code be larger than that of SafeBricks but still be reasonably small.

Multi-core scalability: EVE delivers multi-core scalability by using per-core thread in the enclave. We measure EVE's performance while increasing the number of cores and compare it against an implementation that uses TaLoS-like [4] thread pool. We tune the number of threads in its pool to give the best performance. Figure 13 shows the performance of EVE scales linearly to the number of cores. This is because EVE uses efficient per-core lock-free queues. In contrast, thread pool approach suffers from inter-core communication and locking overhead.

Out-of-band key sharing overhead: For each out-of-band key sharing, an end server encrypts and transmits 104 bytes of secrets to EVE. We measured the maximum rate of out-of-band key sharing using a single core. The result shows that EVE can handle 94 k out-of-band key transfer per second per core. This is comparable to a high-end hardware firewall [9] capable of handling 50 k to 350 k connections per second.

We also evaluate how out-of-band key sharing affects flow completion time. We use a vanilla TLS as a baseline. We measure the time between connection establishment and the arrival of the first 64 B TLS payload. Then, we break the time into TLS handshake, key share and TLS data read. Figure 14 shows out-of-band key sharing incurs minimal latency overhead (0.49% of TLS handshake). Finally, we apply SGX to

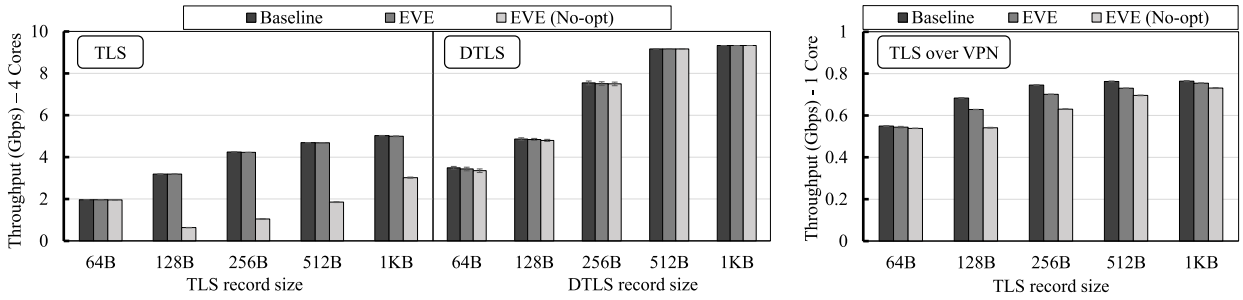

Fig. 16. Comparison of decryption throughput with various encryption protocols.

the TLS server to perform out-of-band key-sharing and TLS operation in the enclave and measure the same latency. The result shows the total latency overhead of SGX is moderate (12.99%) and the key sharing at the SGX-enabled server incurs 1.49% of additional latency overhead compared to its TLS handshake. Note, applying SGX to the server is optional if the operator trusts the server and its platform.

After the out-of-band key sharing, EVE flow-key associating module stores a flow tuple and initialized decryption context pairs in memory-efficient key-value structure. To quantify the memory overhead of the flow-key association, we measure EPC heap memory usage for a TLS flow decryption. Figure 15 shows the distribution of EPC heap memory usage for decrypting a TLS flow. The flow-key associating module only takes about 0.4% of the memory usage for mapping a flow-key pair, while more than 99% of the usage is occupied by TLS-related structures. In addition, to lower the EPC usage for TLS decryption, we can reduce the size of TLS record buffer which is more than 16 KB by OpenSSL default [47].

C. Microbenchmark Evaluation

We now characterize the performance overhead of EVE's components.

Decryption throughput: We measure the decryption throughput of EVE with various encryption protocols by increasing record size. The baseline is EVE version without any security components (SGX, Rust and SGXBounds). We compare the complete EVE and EVE without SGX optimization (`EVE(No-opt)`) (§V) with the baseline. We measure decryption throughput without DPI in this scenario.

The result in Figure 16 shows EVE achieves high decryption throughput over TLS (4.99 Gbps), DTLS (9.33 Gbps) and TLS over VPN (0.75 Gbps) traffic with 1 KB record size. The performance overheads from EVE security components are up to 0.60%, 1.57%, and 8.04%, respectively. We observe that degradation caused by the original SGX SDK (`EVE(No-opt)`) is dominant, and our optimization dramatically improves the throughput, except for DTLS that does not frequently call `memmove` during the decryption. In particular, EVE achieves much higher TLS decryption throughput when the record size is more than 64 B because our `memmove` port supports vectorized instructions (AVX).

Overhead breakdown: To characterize the performance overhead of security components, we evaluate four versions of EVE, each without one or more security components: 1) SGX, 2) Rust and 3) Bounds check using SGXBounds. To test EVE without Rust, we implement the same API function in C/C++. We evaluate the EVE throughput when it decrypts TLS traffic over VPN tunnel and performs DPI on the decrypted plain-text while increasing TLS record size from 64 B to 1 KB. We use TLSv1.2 AES256-GCM-SHA384 cipher suite.

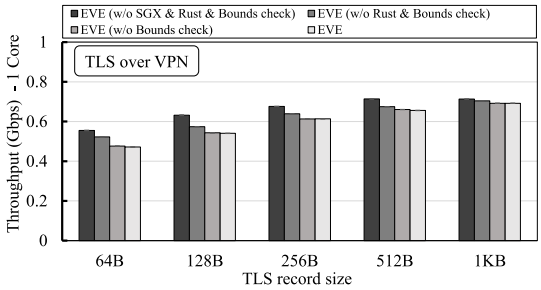

Fig. 17. Comparison of DPI throughput on TLS over VPN encrypted traffic with the four versions of EVE.

Figure 17 shows the DPI throughput of each version. Note the overhead of security components of EVE becomes minimal as the record size increases because the record decryption time dominates 1) the enclave transition overhead, 2) the flow data copying overhead from C/C++ structures to Rust APIs and 3) the runtime boundary checking for accessing record buffers. The performance overhead of all three security features on decryption is at most 14.99% (64 B TLS record size) and as small as 3.00% (1 KB TLS record size).

D. Security Analysis

Recent SGX studies [29], [37] emphasize the seriousness of enclave software vulnerabilities. For example, COIN attack [29] shows that a number of well-known SGX systems are vulnerable to concurrent enclave accesses, and Dark-ROP [37] presents return-oriented programming attacks exploiting enclave code vulnerabilities.

EVE mitigates such attacks by providing enclave memory safety components. EVE Rust APIs prevent middlebox application code from directly handling C/C++ pointers and use-after-free. Also, EVE internals and integrated third-party libraries, which are implemented with C/C++, reduce software-driven memory vulnerabilities such as buffer overflow by memory boundary checker [35]. Finally, flow-core association and per-core structures achieve flow-level isolation

that alleviates the potential vulnerabilities from concurrent flow processing. Though we do not address micro-architectural SGX side-channel attacks as we mentioned in our threat model, we achieve fine-grained enclave memory safety against software vulnerabilities.

VII. Related Work

SGX for middlebox: Inspired by the pioneering studies [20], [31], [58], researchers have explored the possibility of adopting SGX on network functions. mbTLS [44] extends the TLS handshake protocol to support interoperability with legacy TLS endpoints and protects session key data using Intel SGX. However, mbTLS is TLS-specific and does not support diverse use cases, such as nested encryption and VPN support. Also, mbTLS does not support a strong threat model that provides defense against memory safety attacks on enclaves.

ShieldBox [64] leverages SCONE [3], a shielded execution framework, to execute existing network functions inside SGX enclaves without modification. ShieldBox delivers high performance using kernel bypass by utilizing Intel DPDK. It exposes a generic programming interface based on Click [32]. However, ShieldBox does not natively support visibility on end-to-end encrypted traffic nor handle stateful flow management. Trusted Click [10] and ENDBOX [18] also utilize Click with SGX enclaves to explore the feasibility of performing network functions. However, both of them has limitations on supporting stateful processing. In addition, ENDBOX suffers from high deployment cost because network operators need to change all client systems to be equipped with SGX-enabled hardware due to its de-centralized system model.

SafeBricks [52] is a system for securely outsourcing network functions to an untrusted cloud environment. SafeBricks assumes the presence of trusted client gateways to enable the decryption of end-to-end encrypted traffic. SafeBricks decrypts the incoming traffic that is encrypted by the trusted client gateway with the fixed encryption protocol and cipher suite. SEC-IDS [34] combines SGX with Snort IDS [60] by leveraging Graphene-SGX [65]. It protects the integrity of Snort system but does not handle encrypted traffic. AirBox [6] applies SGX to address security concerns for edge clouds.

We believe EVE is superior to existing frameworks in building diverse network functions, as it provides high-level abstractions for flexible encryption protocol support.

Handling encrypted traffic. mcTLS [45] modifies TLS to explicitly include middleboxes during a handshake. It encrypts a packet into multiple encryption contexts and grants read-only or read/write access permissions to middleboxes using different partial keys. PlainBox [39] proposes a session key sharing procedure between end client and middleboxes without changes in encryption protocols by utilizing Attribute-Based Encryption [5]. BlindBox [57] uses a searchable encryption scheme to support string matching and regular expression matching on encrypted traffic. Embark [36] enhances the performance of BlindBox by introducing a new faster encryption scheme. Compared to the above approaches, EVE can be deployed on top of existing encryption protocols without any modification of end client system.

VIII. Conclusion

We propose EVE, a complete platform for building network middleboxes that provide visibility on encrypted traffic to enable packet inspection. For programmability, EVE provides high-level abstractions for middlebox programming and supporting multiple encryption protocols. It allows developers to easily implement EVE applications without in-depth knowledge of cryptographic libraries and low-level packet processing. EVE mitigates the leakage of the user private data because every security-sensitive operation is protected by an SGX enclave. Also, it uses a safe language, Rust, and state-of-the-art memory boundary checking technology to harden the enclave program. Our evaluation shows that EVE is a practical system that covers diverse use cases and introduces moderate performance overhead.

References

[1] A. Alshalan, S. Pisharody, and D. Huang, "A survey of mobile VPN technologies," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 2, pp. 1177–1196, 2nd Quart., 2016.

[2] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. HASP*, 2013, pp. 1–7.

[3] S. Arnautov *et al.*, "SCONE: Secure Linux containers with intel SGX," in *Proc. OSDI*. Berkeley, CA, USA: USENIX Association, 2016, pp. 1–16.

[4] P.-L. Aublin *et al.*, "TaLoS: Secure and transparent TLS termination inside SGX enclaves," Imperial College London, London, U.K., Tech. Rep. 2017, vol. 5, 2017.

[5] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, May 2007, pp. 321–334.

[6] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, "Fast, scalable and secure onloading of edge functions using AirBox," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, Oct. 2016, pp. 14–27.

[7] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with Déjá Vu," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 7–18.

[8] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han, "DFC: Accelerating string pattern matching for network applications," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX, 2016, pp. 551–565.

[9] *Cisco ASA 5585-X Stateful Firewall Data Sheet*. Accessed: Jun. 2019. [Online]. Available: https://www.cisco.com/c/en/us/products/collateral/security/asa-5500-series-next-generation-firewalls/datasheet-c78-730903.html

[10] M. Coughlin, E. Keller, and E. Wustrow, "Trusted click: Overcoming security issues of NFV in the cloud," in *Proc. ACM Int. Workshop Secur. Softw. Defined Netw. Netw. Function Virtualization (SDN-NFVSec)*, 2017, pp. 31–36.

[11] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren, "LightBox: Full-stack protected stateful middlebox at lightning speed," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 2351–2367.

[12] Z. Durumeric *et al.*, "The matter of heartbleed," in *Proc. Conf. Internet Meas. Conf. (IMC)*, 2014, pp. 475–488.

[13] *ET Pro Ruleset*. Accessed: Jun. 2019. [Online]. Available: https://www.proofpoint.com/us/threat-insight/et-pro-ruleset

[14] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel AVX: New frontiers in performance improvements and energy efficiency," Intel, Mountain View, CA, USA, White Paper 20, vol. 19, no. 20, 2008.

[15] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, "SGX-LAPD: Thwarting controlled side channel attacks via enclave verifiable page faults," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*, 2017, pp. 357–380.

[16] A. A. Gendreau and M. Moorman, "Survey of intrusion detection systems towards an end to end secure Internet of Things," in *Proc. IEEE 4th Int. Conf. Future Internet Things Cloud (FiCloud)*, Aug. 2016, pp. 84–90.

[17] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc.41st Annu. ACM Symp. Theory Comput.*, 2009, pp. 169–178.

[18] D. Goltzsche *et al.*, "EndBox: Scalable middlebox functions using client-side trusted execution," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, vol. 18, Jun. 2018, pp. 386–397.

[19] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel SGX," in *Proc. 10th Eur. Workshop Syst. Secur. (EuroSec)*, 2017, pp. 1–6.

[20] J. Han, S. Kim, J. Ha, and D. Han, "SGX-box: Enabling visibility on encrypted traffic using a secure middlebox module," in *Proc. 1st Asia–Pacific Workshop Netw.*, Aug. 2017, pp. 95–105.

[21] *HTTPS vs VPN Makes No Sense*. Accessed: Aug. 2020. [Online]. Available: https://www.expressvpn.com/internet-privacy/https-vs-vpn

[22] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement.* Berkeley, CA, USA: USENIX, 2016, pp. 533–549.

[23] *Intel Data Plane Development Kit (DPDK)*. Accessed: Aug. 2020. [Online]. Available: http://dpdk.org/

[24] *Intel Software Guard Extensions SDK for Linux* OS*. Accessed: Jun. 2019. [Online]. Available: https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf

[25] *Intel Software Guard Extensions SSL*. Accessed: Jun. 2019. [Online]. Available: https://github.com/intel/intel-sgx-ssl.

[26] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mOS: A reusable networking stack for flow monitoring middleboxes," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX, 2017, pp. 113–129.

[27] E. Jeong *et al.*, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX, 2014, pp. 489–502.

[28] K. Kalkan and S. Zeadally, "Securing Internet of Things (IoT) with software defined networking (SDN)," *IEEE Commun. Mag.*, vol. 56, no. 9, pp. 186–192, Sep. 2018.

[29] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "COIN attacks: On insecurity of enclave untrusted interfaces in SGX," in *Proc. 25th Int. Conf. Architectural Support for Program. Lang. Operating Syst.*, Mar. 2020, pp. 971–985.

[30] 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 *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX, 2017, pp. 145–161.

[31] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han, "A first step towards leveraging commodity trusted execution environments for network applications," in *Proc. 14th ACM Workshop Hot Topics Netw. (HotNets)*, 2015, p. 7.

[32] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[33] T. Kothmayr, C. Schmitt, W. Hu, M. Brünig, and G. Carle, "DTLS based security and two-way authentication for the Internet of Things," *Ad Hoc Netw.*, vol. 11, no. 8, pp. 2710–2723, Nov. 2013.

[34] D. Kuvaiskii, S. Chakrabarti, and M. Vij, "Snort intrusion detection system with intel software guard extension (Intel SGX)," 2018, *arXiv:1802.00508*. [Online]. Available: http://arxiv.org/abs/1802.00508

[35] D. Kuvaiskii *et al.*, "SGXBOUNDS: Memory safety for shielded execution," in *Proc. 12th Eur. Conf. Comput. Syst.*, Apr. 2017, pp. 205–221.

[36] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, "Embark: Securely outsourcing middleboxes to the cloud," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2016, pp. 255–273.

[37] J. Lee *et al.*, "Hacking in darkness: Return-oriented programming against secure enclaves," in *Proc. USENIX Secur. Symp.* Berkeley, CA, USA: USENIX, 2017, pp. 523–539.

[38] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *Proc. USENIX Secur. Symp.* Berkeley, CA, USA: USENIX, 2017, pp. 16–18.

[39] C. Liu, Y. Cui, K. Tan, Q. Fan, K. Ren, and J. Wu, "Building generic scalable middlebox services over encrypted protocols," in *Proc. IEEE INFOCOM-IEEE Conf. Comput. Commun.*, Apr. 2018, pp. 2195–2203.

[40] N. D. Matsakis and F. S. Klock, "The rust language," in *Proc. ACM SIGAda Annu. Conf. High Integrity Lang. Technol. (HILT)*, vol. 34, 2014, pp. 103–104.

[41] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *Proc. 2nd Int. Workshop Hardw. Architectural Support Secur. Privacy (HASP)*, 2013, pp. 1–8.

[42] *ModSecurity*. Accessed: Aug. 2020. [Online]. Available: https://www.modsecurity.org/

[43] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," 2017, *arXiv:1703.06986*. [Online]. Available: http://arxiv.org/abs/1703.06986

[44] D. Naylor, R. Li, C. Gkantsidis, T. Karagiannis, and P. Steenkiste, "And then there were more: Secure communication for more than two parties," in *Proc. 13th Int. Conf. Emerg. Netw. Exp. Technol.*, Nov. 2017, pp. 88–100.

[45] D. Naylor *et al.*, "Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS," in *Proc. SIGCOMM*. ACM, 2015, pp. 199–212.

[46] *Nested Tunnels*. Accessed: Aug. 2020. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.halz002/ipsecurity_cfg_models_nest_tunnel.htm

[47] *OpenSSL-1.0.2l*. Accessed: Jun. 2019. [Online]. Available: https://www.openssl.org

[48] *OpenVPN-2.4.3*. Accessed: Jun. 2019. [Online]. Available: https://community.openvpn.net/openvpn

[49] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2016, pp. 203–216.

[50] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, nos. 23–24, pp. 2435–2463, Dec. 1999.

[51] *Perl Compatible Regular Expressions Library (PCRE2)*. Accessed: Jun. 2019. [Online]. Available: https://ftp.pcre.org/pub/pcre/

[52] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, "SafeBricks: Securing network functions in the cloud," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX, 2018, pp. 201–216.

[53] M. Rossberg and G. Schaefer, "A survey on automatic configuration of virtual private networks," *Comput. Netw.*, vol. 55, no. 8, pp. 1684–1699, Jun. 2011.

[54] *Rust-Openssl*. Accessed: Jun. 2019. [Online]. Available: https://github.com/sfackler/rust-openssl

[55] *Rust SGX SDK*. Accessed: Jun. 2019. [Online]. Available: https://github.com/baidu/rust-sgx-sdk

[56] J. Seo *et al.*, "SGX-shield: Enabling address space layout randomization for SGX programs," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017.

[57] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "BlindBox: Deep packet inspection over encrypted traffic," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*, 2015, pp. 213–226.

[58] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-NFV: Securing NFV states by using SGX," in *Proc. ACM Int. Workshop Secur. Softw. Defined Netw. Netw. Function Virtualization*, 2016, pp. 45–48.

[59] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–16.

[60] *Snort Intrusion Detection System*. Accessed: Aug. 2020. [Online]. Available: https://snort.org

[61] R. Stanton, "Securing VPNs: Comparing SSL and IPsec," *Comput. Fraud Secur.*, vol. 2005, no. 9, pp. 17–19, Sep. 2005.

[62] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 48–62.

[63] B. Trach, A. Krohmer, S. Arnautov, F. Gregor, P. Bhatotia, and C. Fetzer, "Slick: Secure middleboxes using shielded execution," 2017, *arXiv:1709.04226*. [Online]. Available: http://arxiv.org/abs/1709.04226

[64] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "ShieldBox: Secure middleboxes using shielded execution," in *Proc. Symp. SDN Res.*, Mar. 2018, p. 2.

[65] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *Proc. USENIX Annu. Tech. Conf. (ATC)*. Berkeley, CA, USA: USENIX, 2017, pp. 645–658.

[66] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proc. USENIX Secur. Symp.* Berkeley, CA, USA: USENIX, 2017, pp. 1041–1056.

[67] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves," in *Proc. Eur. Symp. Res. Comut. Secur. (ESORICS)*. Cham, Switzerland: Springer, 2016, pp. 440–457.

[68] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 81–93.
[69] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 640–656.
[70] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao, "A survey on security and privacy issues in Internet-of-Things," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1250–1258, Oct. 2017.
[71] W. Zheng, A. Dave, G. J. Beekman, R. A. Popa, E. J. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX, 2017, pp. 283–398.

Juhyeng Han received the B.S. degree from the School of Computing, KAIST, in 2016, and the M.S. degree from the School of Electrical Engineering, KAIST, in 2018, where he is currently pursuing the Ph.D. degree. His research interests include network systems and network security.

Seongmin Kim received the B.S. and M.S. degrees from the School of Electrical Engineering, KAIST, in 2012 and 2014, respectively, and the Ph.D. degree from the Graduate School of Information Security, KAIST, in 2019. He is currently an Assistant Professor with the Department of Convergence Security Engineering, Sungshin Women's University. His research interests include system security, especially trusted computing and network security.

Daeyang Cho received the B.S. and M.S. degrees from the School of Electrical Engineering, KAIST, in 2018 and 2020, respectively, where he is currently pursuing the Ph.D. degree. His research interests include network systems and network security.

Byungkwon Choi (Member, IEEE) received the B.S. degree from the School of Information and Communication Engineering, Inha University, Incheon, South Korea, in 2014, and the M.S. degree from the School of Electrical Engineering, KAIST, in 2016, where he is currently pursuing the Ph.D. degree. His research interests include enhancing performance of various networked and distributed systems. More details can be found at http://ina.kaist.ac.kr/ brad.

Jaehyeong Ha received the B.S. degree from the School of Electrical Engineering, KAIST, in 2017. His research interests include network systems and network security.

Dongsu Han (Member, IEEE) received the B.S. degree in computer science from KAIST in 2003 and the Ph.D. degree in computer science from Carnegie Mellon University in 2012. He is currently an Associate Professor with the School of Electrical Engineering and the Graduate School of Information Security, KAIST. His research interests include networking, distributed systems, and network/system security. More details about his research can be found at http://ina.kaist.ac.kr.