SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module

Juhyeng Han, Seongmin Kim, Jaehyeong Ha, Dongsu Han
KAIST

ABSTRACT

A network middlebox benefits both users and network operators by offering a wide range of security-related in-network functions, such as web firewalls and intrusion detection systems (IDS). However, the wide usage of encryption protocol restricts functionalities of network middleboxes. This forces network operators and users to make a choice between *end-to-end privacy* and *security*. This paper presents SGX-Box, a secure middlebox system that enables visibility on encrypted traffic by leveraging Intel SGX technology. The entire process of SGX-Box ensures that the sensitive information, such as decrypted payloads and session keys, is securely protected within the SGX enclave. SGX-Box provides easy-to-use abstraction and a high-level programming language, called SB lang for handling encrypted traffic in middleboxes. It greatly enhances programmability by hiding details of the cryptographic operations and the implementation details in SGX enclave processing. We implement a proof-of-concept IDS using SB lang. Our preliminary evaluation shows that SGX-Box incurs acceptable performance overhead while it dramatically reduces middlebox developer's effort.

CCS CONCEPTS

• **Networks** → **Middle boxes / network appliances**; *Programming interfaces*; • **Security and privacy** → *Firewalls*;

KEYWORDS

Middlebox Security; Intel SGX; Deep Packet Inspection

ACM Reference format:
Juhyeng Han, Seongmin Kim, Jaehyeong Ha, Dongsu Han KAIST . 2017. SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module. In *Proceedings of APNet'17, Hong Kong, China, August 03-04, 2017,* 7 pages.
https://doi.org/10.1145/3106989.3106994

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APNet'17, August 03-04, 2017, Hong Kong, China
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5244-4/17/08…$15.00
https://doi.org/10.1145/3106989.3106994

1 INTRODUCTION

Encryption protocols are becoming a dominant part of today's Internet traffic. Because the usage of encryption protocol ensures end-to-end security, application developers are increasingly adopting encryption protocols, such as SSL/TLS. In fact, HTTPS, one of the most popular encryption protocol, has grown about 40% over the last few years [19].

However, this trend restricts functionalities of network middleboxes because they cannot perform deep-packet inspection (DPI) on encrypted traffic. A network middlebox benefits both users and network operators by offering a wide range of security-related in-network functions, such as web firewalls [4] and intrusion detection/prevention systems (IDS/IPS) [6, 22]. When handling encrypted traffic, middlebox developers face two fundamental challenges. First, a middlebox cannot inspect the packet payloads. This forces network operators and users to make a choice between *end-to-end privacy* and *middlebox processing* [26]. Second, expert knowledge on cryptographic protocols is required. Several studies [21, 22] develop interfaces to reduce the developer's effort but do not support high-level programming abstraction for handling encrypted traffic.

Previous studies take three main approaches to enable middlebox functionality on encrypted traffic: 1) use a man-in-the-middle (MITM) approach to decrypt TLS traffic [10, 14]; or 2) modify SSL/TLS to explicitly include middleboxes during a handshake [20]; or 3) use cryptographic schemes that allow direct inspection on the encrypted traffic [26]. However, each has critical concerns in security or deployment. An insecure interposition of middleboxes (e.g., MITM) breaks the end-to-end encryption [23], which leads to leakage of user's private information when the middlebox is compromised. Modifying the transport layer security (TLS) protocol [20] requires changes to existing middleboxes, server, and client applications. Finally, leveraging new cryptographic schemes, such as searchable encryption [26] has serious performance issues.

This paper presents SGX-Box, a practical middlebox system that enables secure inspection on encrypted traffic. We

leverage Intel SGX, a commodity TEE technology that protects application's code and data inside a hardware enclave with native speed of a processor. Two key SGX functionalities that SGX-Box uses are memory isolation and remote attestation. First, SGX-Box encloses sensitive data structures (e.g., IDS rules and session keys) and operations such as decryption of encrypted traffic in an SGX enclave. We then employ out-of-band key sharing between the SGX-Box module and an end server to securely share session keys. Second, SGX-Box enables the end server to perform module attestation. Before establishing an SSL/TLS connection for sharing session keys, the end server performs remote attestation to verify the integrity of SGX-Box modules. By applying SGX, SGX-Box prevents the leakage of sensitive data such as shared session keys even from root-privileged attackers that reside in the same middlebox platform or OS. Therefore, SGX-Box ensures stronger trust model compare to state-of-the-art approaches [10, 14, 20, 26] and provides a new feature that enhances security and privacy of users.

However, this involves huge programming effort and requires domain expertise in utilizing SGX, parsing TLS handshake messages, enabling out-of-band key sharing, handling packet reassembly, and performing decryption/re-encryption on middleboxes. To solve this problem, we introduce a new programming language, called SB lang that provides a high-level abstraction to network operators for building an SGX-Box module. The SB lang APIs help developers to define an operational policy of the SGX-Box module explicitly and hide the details of low-level packet processing and cryptographic operations. The SB converter automatically generates the code for all necessary components for secure middlebox processing. This significantly reduces the developer's effort and mitigates potential vulnerabilities inadvertently introduced by mis-implementations (e.g., wrong usage of buffer pointer).

We demonstrate the feasibility of SGX-Box with a proof-of-concept IDS module that performs exact matching on a TCP flow and measure the pattern matching throughput. The results show that SGX-Box incurs moderate performance overhead of 11.9%, while SB lang provides the programming convenience for middlebox developers.

2 BACKGROUND AND MOTIVATION

Intel Software Guard Extensions (SGX): Intel SGX provides a hardware-based mechanism to ensure the integrity and confidentiality of applications. It allows an application developer to protect security-sensitive data (e.g., private keys) and operations inside a secure container 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. SGX also provides remote attestation, which allows a service provider to verify the integrity of a remote program and that the target remote enclave is actually running on an SGX CPU. SGX-Box leverages the two features in designing a secure middlebox.

Limitations of existing approaches: Several studies take the man-in-the-middle approach and use fake certificates for DPI on encrypted traffic [10, 14]. The middlebox terminates a TLS session and builds split TLS sessions with the two end-points. Therefore, the middlebox can inspect on plain traffic between the split sessions. However, this approach does not offer security. When the host of middlebox is compromised, the user's data in the plaintext can be leaked. In addition, a malicious adversary can interpose the connection between an end server and the middlebox to intercept the traffic [14]. In summary, the use of fake certificates creates vulnerabilities and privacy concerns as it breaks end-to-end encryption while users may not be unaware of the situation [23].

Two previous approaches tackle this problem. mcTLS [20] modifies SSL/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. However, making multiple encryption contexts in the end client causes additional overhead compared to the baseline TLS. Also, if a middlebox is not fully granted to decrypt all of the encrypted contexts by the end server, the functionalities of the middlebox are restricted. Blindbox [26] enables direct inspection on the encrypted traffic by using searchable encryption scheme. The system is specifically designed to support two main functionalities: exact matching and regular expression processing. However, its practicality is limited because of its high computational cost (16 times slower than the baseline SSL) and restricted functionality. Finally, both approaches require changes in protocol, middleboxes, and the end-hosts.

2.1 Design Goals

We set our goal as achieving five key properties.

- **Functionality:** a system fully supports deep-packet inspection on encrypted traffic.
- **Performance:** a middlebox performs packet processing on encrypted traffic with an acceptable performance.
- **Security:** a system does not reveal any sensitive data of users, including decrypted payloads and session keys, and securely executes middlebox operations.
- **Programmability:** a system provides abstractions and high-level APIs that help network operators to easily implement middlebox modules for their own purpose.
- **Cost of deployment:** Efforts to deploy the middlebox system (e.g., changes in components) are not costly.

	Blindbox [26]	mcTLS [20]	man-in-the-middle [10, 14]	**SGX-Box**
Functionality	Full	Partial	Full	Full
Performance	Slow	Moderate	Fast	Fast
Security (Module attestation)	Strong (X)	Moderate (X)	Weak (X)	Strong (O)
Programmability	No	No	No	Yes
Cost of deployment (Updated components)	High (S, C, MB)	High (S, C, MB)	Low (None)	Moderate (S, MB)

Table 1: Comparison between SGX-Box and recent research to enable DPI on encrypted traffic. In the cost of deployment field, "S" and "C" denotes a server and a client application; "MB" denotes a middlebox.

However, building a middlebox system satisfying the above properties is non-trivial for a number of reasons. First, cryptographic mechanisms with strong security guarantee (e.g., homomorphic encryption [9]) incurs a high computational cost. Second, developing a secure middlebox system requires domain expertise in cryptography and transport protocols (TCP and TLS) and thus requires time and effort. Finally, modifying the existing TLS protocol is often not an option considering the deployment cost. In fact, none of the recent studies achieve all the properties as shown in Table 1.

SGX-Box takes two main approaches to address the challenges. First, we apply SGX on middleboxes to securely operate a direct DPI computation on the decrypted payload, rather than modifying encryption protocols or relying on heavy cryptographic operations. This allows SGX-Box to perform the middlebox processing in the native speed of a processor while ensuring stronger security model compare to state-of-the-art approaches [10, 14, 20, 26]. Second, we develop a high-level abstraction that reduces huge programming effort to build an operational policy of middleboxes in the SGX-Box module. In addition, it automatically generates every necessary software components for utilizing SGX.

Deployment assumption: Our basic assumption is that network operators cooperate with an end host (e.g., server side) to deploy their middlebox software. For inspecting traffic, the middlebox software receives rulesets from the end server through the secure channel established after remote attestation. The rulesets only reside in the enclave, and SGX-Box seals them using an SGX hardware key when it needs to store them outside the enclave. We assume that the middlebox platform, except for the SGX hardware and the code of middlebox software, is untrusted. In other words, our system allows network operators to use a third-party middlebox platform, equipped with SGX hardware.

Threat model: Following the traditional threat model of SGX [18], we assume that an attacker can compromise any software components, including privileged software (e.g., OS and hypervisor), and hardware components (e.g., memory and I/O bus), except the CPU package. For example, a *system-level adversary* (i.e., an attacker who can obtain full control of the system software on the middlebox platform) can dump the memory layout of middlebox applications or access incoming traffic. Note we do not consider side channel attacks on SGX hardware, denial of service, and attacks using vulnerabilities in the cryptographic protocols.

SB lang API	**Description**
`get_flow_context`	Retrieves packet-level network information.
`store_flow_context`	Stores the flow context.
`match`	Performs DPI on the decrypted payloads.
`forward/drop`	Forwards/drops packets of the traffic.
`seal/unseal`	Securely encrypts/decrypts data in the enclave.

Table 2: The list of SGX-Box APIs

3 HIGH-LEVEL ABSTRACTION

SB lang provides a high-level programming abstraction for building SGX-Box modules. Implementing a middlebox module that requires DPI on encrypted traffic is not easy for network operators. Without a high-level abstraction, network operators and middlebox developers must have expert knowledge of cryptography and network/transport layer protocols to modify the SSL library to inspect encrypted payload in the middlebox module. Therefore, implementing a middlebox that handles encrypted traffic presents a high barrier to entry for ordinary network operators.

Using SB lang, network operators and middlebox developers can easily program a secure SGX-Box module. At a high-level, SB lang specifies the operations that must be performed on decrypted traffic. Then, from an SB lang program, our system automatically generates low-level C/C++ codes. SB lang APIs do not contain any pointer operations to prevent potential vulnerabilities, such as buffer overflow attacks.

High-level abstraction for ease of programming: Basically, developers should implement cryptographic and low-level network operations, including parsing TLS handshake messages, reassembling packets and decryption, to enable middlebox functions. However, they want to focus on the implementation of their operational policies on the decrypted plaintext, rather than considering such details. To address this issue, we design SB lang to provide 1) abstraction of handling encrypted traffic and 2) high-level programming interface to implement operational policies easily.

Table 2 shows the list of high-level APIs that SB lang provides. [1] The `get_flow_context` API retrieves packet-level

[1] In the current version of SGX-Box, we partially implement several SB lang APIs in the list as a prototype.

network information such as decrypted payload. Basically, the developers should parse flow information depending on the protocol (IP, TCP) from the raw packets without this API. Moreover, they need to know the cipher suite of the encrypted flow and use OpenSSL functions properly to perform decryption themselves. Additionally, they need to perform MAC verification and synchronize MAC hash of each connection with end hosts. `get_flow_context` abstracts all of above functionalities. In summary, developers do not need to care about using a cryptographic library such as OpenSSL to decrypt the encrypted traffic for deep packet inspection.

The `match`, `forward` and `drop` APIs abstract the actual packet processing. Developers can use them to specify operational policies as they intended. Without above APIs, developers need to import a rule matching algorithm into an SGX-Box module and carefully implement the rule matching process to minimize performance overhead. Then, they need to implement flow control codes with low-level socket I/O functions in C/C++ which requires a deep understanding of computer networks and system programming. Developers do not need to implement low-level socket programming themselves because SB lang abstracts it into simple functions.

Developers can easily use sealing/unsealing functionalities of SGX with `seal` and `unseal` APIs. For example, `seal` abstracts sealing functions of SGX that encrypts the target buffer with the enclave's sealing key to store the private data of SGX-Box. With this API, SGX-Box encrypts several kinds of rulesets in the enclave and stores them into the file system of untrusted middlebox platform. Because only the enclave can decrypt the encrypted rulesets using unseal function, anyone cannot see the rulesets in plaintext. Before the rule matching operation, SGX-Box imports one of the rulesets into the enclave and decrypts it by calling unseal function. Note that `seal` and `unseal` protect the integrity of the sealed data, SGX-Box can verify the imported ruleset is not changed. Developers can use these APIs to seal/unseal data conveniently without knowing the usage of SGX SDK.

Automatically generated components: SGX-Box automatically generates C/C++ codes that are semantically same with the programmed SB lang code. For this, we develop SGX-Box converter (SB converter). Figure 1 illustrates the overall process of implementation and compilation of the SGX-Box module. SB converter fills the gap between the high-level abstraction and C/C++ codes by constructing data structures and transforming SB lang APIs into low-level function calls. Specifically, the flow context in SB lang is converted to C/C++ data structure that includes network headers (e.g., Ethernet, IP and TCP header), originally defined in the system library (e.g., "tcp.h" in Linux). SB converter also generates C/C++ function codes to manage per-flow information and perform operational policies such as rule matching.

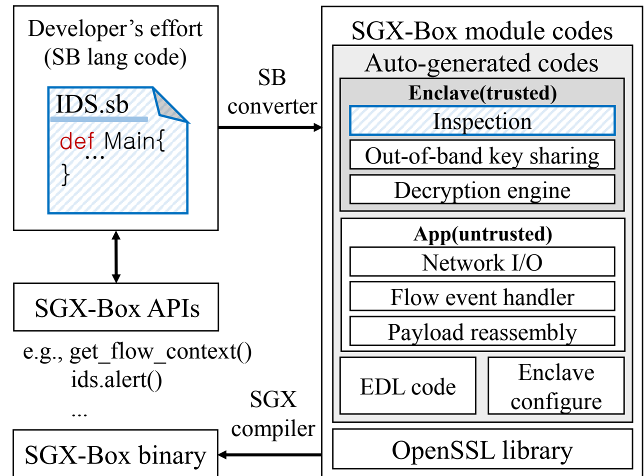

Figure 1: Overview of implementation and compilation process of SGX-Box module.

At the same time, SB converter creates all necessary files for SGX compilation. In addition to C/C++ codes, SGX compiler basically requires an enclave definition language (EDL), a Makefile, and an enclave configuration file to build an SGX program. An EDL has a special syntax that contains prototypes of functions for the trusted/untrusted region. A developer should define every function prototype and the wrapper to handle operations that are not supported in the enclave mode (e.g., system calls). Moreover, the developer needs to specify the enclave configuration file (e.g., setting the enclave size) and Makefile for building an SGX binary. SB converter generates such SGX-specific files automatically. As a result, the code that the developer actually implement is the SB lang code (Developer's effort in Figure 1) with SB lang APIs.

Language security: SB lang allows developers not to care about potential vulnerabilities came from mis-implementations. The current version of SGX SDK [3] only supports C/C++ language to build an enclave program. Basically, C/C++ allows programmers to use direct memory access. However, this makes C/C++ inherently unsafe language that has vulnerabilities such as out-of-bound read/write. Moreover, such attacks are still effective in the presence of SGX hardware because it has limited defense mechanism such as address space layout randomization (ASLR) and attackers can easily know the mapping of an enclave memory [25]. To address this issue, SB lang does not support pointer operations by design. This helps developers focus on implementing the core logics of the SGX-Box module with less concern about secure coding.

4 SGX-BOX COMPONENTS

Figure 2 illustrates the overall architecture of SGX-Box. An SGX-Box module handles encrypted traffic in two phases: a preprocessing phase and an inspection phase. In the preprocessing phase, SGX-Box reassembles the encrypted payload by leveraging mOS [13], a middlebox framework that

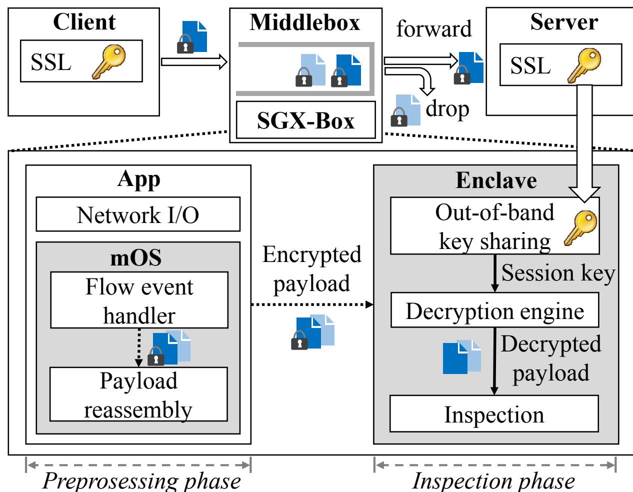

Figure 2: Overall architecture of SGX-Box.

provides built-in events to manage network flows efficiently. Then, at the inspection phase, the payload is decrypted inside an SGX enclave and the SGX-Box module inspects the payload for middlebox processing (e.g., exact matching of IDS). This ensures that the decrypted plaintext never leaves the enclave and is not visible to system software. Finally, the module performs an action (e.g., alert) based on the inspection result as specified by the SB lang program.

Secure out-of-band key sharing: SGX-Box securely receives session keys from the end server using an out-of-band key sharing mechanism to decrypt end-to-end encrypted traffics. The server performs remote attestation to verify the integrity and builds a secure channel with an SGX-Box middlebox. This ensures the SGX-Box module that 1) it is running on the SGX hardware, and 2) the integrity of its enclave code is not modified. When the module successfully passes the attestation, the server shares its session keys through the out-of-band channel. Otherwise, it disconnects the secure channel with the target middlebox, so the middlebox fails to get the session key. The out-of-band key sharing happens on each connection. In contrast, the remote attestation and out-of-band channel establishment are executed only at the initialization of SGX-Box module. Therefore, they are a one-time cost and the SGX-Box maintains the channel during its life cycle. Note that a client application also can verify the integrity of the SGX-Box module by requesting a remote attestation of the middlebox to the end server. Therefore, SGX-Box offers *module attestation* for both users and network operators.

Handling encrypted traffic: When encrypted traffic arrives, SGX-Box reassembles the encrypted byte stream and passes it to the decryption engine (Figure 2). Decryption and processing on the plaintext payload are performed within an SGX enclave. The memory region of an SGX application is divided into a secure enclave memory and an untrusted application memory. We enclose the security-sensitive operations of SGX-Box, instead of putting the every SGX-Box code, including the packet preprocessing, to avoid performance overhead of using SGX[2]. Here, the reassembled payload still has the same level of security guarantee with the SSL/TLS protocol. Security-sensitive operations include payload decryption, out-of-band key sharing, and applying the application logic (e.g., rule matching of IDS) to decrypted traffic. In the enclave, the module decrypts encrypted payloads using the session key obtained through out-of-band key sharing. SGX-Box does not store or copy decrypted plain payloads buffers into untrusted memory. Therefore, our system does not leak privacy of users in the presence of middleboxes, even when the system software of SGX-Box platform is compromised.

Updated components (cost of deployment): To deploy SGX-Box, network operators need to implement SGX-Box module and update their server application. Note that network operators can easily implement SGX-Box module using SB lang APIs described in §3, and we provide modified OpenSSL library to send session keys through an out-of-band secure channel at the end server. Network operators do not need to change existing protocols such as SSL/TLS. SGX-Box receives a cipher suit and a session key through an out-of-band secure channel, and it automatically adapts to existing encryption protocol that OpenSSL library supports. In addition to TLS protocol, SGX-Box can be extended to process network traffic that uses other encryption protocols such as IPSec.

Implementation: To reassemble packets, SGX-Box leverages mOS [13], a framework that extends a user-level TCP stack [15] and supports stateful flow-level processing on middleboxes. It provides an event-based abstraction to build middleboxes, which reduces engineering effort that comes from low-level network processing. SGX-Box uses built-in events of mOS that are triggered when the flow condition is changed. Specifically, SGX-Box uses `MOS_ON_CONN_NEW_DATA` event to detect and collect incoming encrypted payloads. Then, it requests an enclave to perform decryption and packet inspection in the trusted region. We implement and evaluate SGX-Box using Intel SGX Linux SDK [3] and OpenSSL 1.1.0e [5].

5 PRELIMINARY EVALUATION

5.1 Case study: a proof-of-concept IDS

We implement a proof-of-concept IDS as an SGX-Box module using SB lang (Figure 3). The operational policy of the IDS is divided into two procedures. One retrieves the flow context to check whether the flow is using TCP protocol and the other performs exact matching with ET-PRO ruleset [2]. As the Figure 3 shows, a developer can use convenient high-level APIs (e.g., `net.get_flow_context()` and `ids.match()`) that SB lang provides. Note that SB lang

[2]It incurs additional encryption/decryption overhead and context switch overhead while entering/leaving the enclave.

```
import net, ids

def IDS_module {
  net.interface = "eth0"
  net.ip = "143.248.56.14"
  net.port = 8080
  ids.ruleset = unseal(ids.ET_PRO)
  ids.log_path = "/var/log/sgxbox.log"
  /* Retrieve flow information */
  flow = net.get_flow_context()
  /* When a TCP flow comes in, match it to IDS rules */
  if (flow.proto_type == net.TCP) {
    if (ids.match(flow) == True) {
      msg = "Alert! "+flow.src_ip+":"+flow.src_port
      ids.alert(msg) /* alert logging */
    }
  }
}
```

Figure 3: The SB lang code of a proof-of-concept IDS.

Component	SB lang	Naive implementation
SB lang	15 lines of code	-
Application	-	718 lines of C++
Enclave	-	1,458 lines of C++
EDL	-	38 lines of edl
Makefile	-	174 lines of code
OpenSSL	-	204 lines of C
Total	15 lines of code	2592 lines of code

Table 3: Comparison of lines of code.

does not support any direct pointer arithmetics or memory operations that might occur potential vulnerability such as out-of-bound read/write caused by mis-implementation of developers. Therefore, SB lang reduces the attack surface of the SGX-Box module and makes developers focus on the core logic of middlebox module. The total SB lang codes to implement the IDS is only 15 lines.

To show the programming convenience of SB lang, we naively implement the same middlebox application in the presence of SGX CPU. Table 3 shows the comparison of lines of codes to build a simple IDS module between SB lang and the naive approach. Without SB lang, middlebox developers need to implement an EDL and other SGX-related files. In contrast, SB converter of SGX-Box automatically generates such files, ready to be compiled by the Intel SGX SDK. Moreover, a porting effort of the SSL library (e.g., OpenSSL) to use it inside an enclave is not required. As a result, the lines of code to implement the IDS module is reduced by 99.4%.

5.2 Performance Overhead Characterization

For evaluation, we use a Quad core Intel i7-6700 3.4GHz CPU machine running Linux 4.4.0. We use Intel SGX Linux SDK 1.8 [3] to compile and build an SGX-Box module. To measure the overhead in a CPU-bound setting, we execute a client, a server, and an SGX-Box module in the same host and run the SGX-Box module as a single thread.

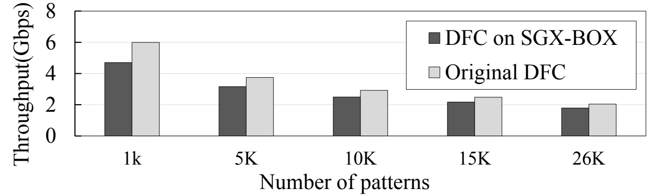

Figure 4: Comparison of pattern matching throughput.

Performance overhead of SGX-Box: To measure the overhead of SGX-Box, we evaluate the throughput of SGX-Box module that performs pattern matching. Here, we use DFC [8] as pattern matching algorithm with patterns collected from ET-PRO ruleset [2]. Figure 4 shows the performance of SGX-Box and the baseline DFC by increasing the number of patterns from 1k to 26k. The result is calculated by averaging 30 runs. Here, we extract the first part of patterns to generate a various set of patterns. [3] As the number of patterns increases, the overhead of SGX-Box decreases because the pattern matching time dominates the context switching overhead when entering/leaving an enclave. The result shows that SGX-Box incurs 11.9% overhead with 26k patterns, which is moderate. We believe SGX-Box has an acceptable performance while providing a strong security guarantee, without relying on heavy cryptographic operations.

Out-of-band key sharing: We characterize the throughput of out-of-band key sharing by measuring the number of shared key-blocks because capable connections of SGX-Box is bounded to it. For this, we run an SGX-Box to receive session-key-blocks from the end server using the out-of-band channel. Here, we use a cipher suite of TLS v1.2. The result shows that the server can share 59,287 session-key-blocks with SGX-Box per second. we compare the result with the connections per second (CPS) of the commodity middlebox devices for the enterprise data center. Note that Cisco ASA 5585-X with SSP-10, one of the commodity firewall product, serves 50,000 CPS [1]. Therefore, the throughput of the middlebox is not throttled by out-of-band key sharing.

6 RELATED WORK AND FUTURE WORK

Systems that apply Intel SGX: SGX technology is motivated by the cloud environment to protect the user's data and code from untrusted platform [7, 24]. Haven [7] adopts SGX in the cloud platform to securely execute applications without modification. VC3 [24] proposes MapReduce computations while ensuring the correctness and completeness of results using SGX. Ryoan [11] studies a distributed sandbox to keep the secret data from the potentially malicious parties, by leveraging SGX enclave. Starting from the cloud, many recent studies explore security implications of leveraging SGX on networking [16, 17, 27]. Kim et al. [17] suggest new design choices

[3] ET-PRO ruleset has 26k patterns by default.

and functionalities of adopting SGX on software-defined inter-domain routing and the anonymity network based on SGX emulator [12]. Also, SGX-Tor [16] enhances the security and privacy of Tor network. S-NFV [27] adopts SGX to protect the state information of network function virtualization (NFV).

Future work: In addition to SSL/TLS protocol, SGX-Box can be extended to cover existing encryption protocols (e.g., IPSec and DNSSEC) and the case that uses multiple encryption schemes (e.g., SSL/TLS encryption on IPSec encrypted channel). In this paper, we partially implement SB lang APIs to demonstrate a prototype IDS module. We believe providing diverse and flexible high-level SB lang APIs can further enhance the programmability of SGX-Box. To show the feasibility of SGX-Box, we would like to evaluate the end-to-end performance of various middlebox applications with real network traffic. In addition, this paper has some open questions with the evaluation of SGX-Box such as performance overhead for worst case scenario. We leave these as future works.

7 CONCLUSION

In this paper, we propose SGX-Box, a new middlebox system that provides visibility on encrypted traffic to enable packet inspection by leveraging SGX technology. SGX-Box does not leak the private data of users because every security-sensitive operation is protected by an SGX enclave. We believe that SGX-Box is the first system that explores the design of middleboxes running on the real SGX hardware. To support a convenient programming interface, we develop SB lang that provides a high-level abstraction for middlebox programming. It allows network operators to easily implement SGX-Box modules without in-depth knowledge of the usage of cryptographic libraries and the low-level networking stack. Our preliminary evaluation, a proof-of-concept IDS, shows that SB lang and high-level APIs provide the programming convenience for middlebox developers, and SGX-Box is a practical system which incurs moderate performance overhead.

8 ACKNOWLEDGMENT

We would like to thank our shepherd Marco Canini and anonymous reviewers for their helpful feedback. This work was supported in part by IITP grants funded by the Korea government (MSIP) (No.R-20160222-002755 and No.2015-0-00164), and Office of Naval Research Global (ONRG).

REFERENCES

[1] Cisco ASA 5585-X Spec. http://www.cisco.com/c/en/us/products/collateral/security/asa-5500-series-next-generation-firewalls/datasheet-c78-733916.html.

[2] ET Pro. https://www.proofpoint.com/us/threat-insight/et-pro-ruleset.

[3] Intel(R) Software Guard Extensions SDK for Linux* OS. https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

[4] ModSecurity. https://www.modsecurity.org/.

[5] OpenSSL-1.1.0. https://www.openssl.org/.

[6] Snort Intrusion Detection System. https://snort.org.

[7] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proc. USENIX OSDI*, 2014.

[8] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han. DFC: Accelerating String Pattern Matching for Network Applications. In *Proc. NSDI*. USENIX, 2016.

[9] C. Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proc. STOC*, 2009.

[10] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing Forged SSL Certificates in the Wild. In *Proc. IEEE S&P*, 2014.

[11] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan:A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proc. OSDI*. USENIX, 2016.

[12] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. OpenSGX: An Open Platform for SGX Research. In *Proc. NDSS*, 2016.

[13] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proc. NSDI*. USENIX, 2017.

[14] J. Jarmoc. SSL/TLS Interception Proxies and Transitive Trust. *Black Hat Europe*, 2012.

[15] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. NSDI*. USENIX, 2014.

[16] 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. NSDI*. USENIX, 2017.

[17] 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. HotNets*. ACM, 2015.

[18] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proc. HASP*, 2013.

[19] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste. The Cost of the S in HTTPS. In *Proc. CoNEXT*. ACM, 2014.

[20] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proc. SIGCOMM*, 2015.

[21] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. OSDI*, 2016.

[22] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer networks*, 31(23), 1999.

[23] R. Sandvik. Security vulnerability found in Cyberoam DPI devices (CVE-2012-3372). 2012.

[24] 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 *Proc. IEEE S&P*, pages 38–54, 2015.

[25] 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 *Proc. NDSS*, 2017.

[26] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proc. SIGCOMM*, 2015.

[27] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV States by Using SGX. In *Proc. ACM International Workshop on Security in SDN-NFV*, 2016.