SafeLib: a practical library for outsourcing stateful network functions securely

Enio Marku*, Gergely Biczók†, and Colin Boyd*
*Dept. of Information Security and Communication Technology, Norwegian Univ. of Science and Technology (NTNU)
{enio.marku, colin.boyd}@ntnu.no
†CrySyS Lab, Dept. of Networked Systems and Services, Budapest Univ. of Technology and Economics (BME)
biczok@crysys.hu

***Abstract*—A recent trend is to outsource virtual network functions (VNFs) to a third-party service provider, such as a public cloud. Since the cloud is usually not trusted, redirecting enterprise traffic to such an entity introduces security concerns. In addition to protecting enterprise traffic, it is also desirable to protect VNF code, policies and states. Existing outsourcing solutions fall short in either supporting stateful VNFs, catering for all security requirements, or providing adequate performance.**

In this paper we present SafeLib, a trusted hardware based outsourcing solution built on Intel SGX. SafeLib provides i) support for stateful VNFs, ii) support for illegal SGX instructions by integrating Graphene-SGX, iii) protection of both packet headers and payload for enterprise user traffic, VNF policies and VNF code, and iv) integration of libVNF for streamlined VNF development. Our performance evaluation shows that SafeLib scales properly for multiple cores, and introduces a reasonable performance overhead. We also outline plans to further improve SafeLib to satisfy even more stringent functional, security and performance requirements.

I. Introduction

Network functions (NFs) are a vital part of modern networks, and are used by enterprises to perform complex tasks. Enterprises have traditionally deployed NFs in dedicated hardware (known as middleboxes), and used them for different purposes such as improving performance (e.g., proxies, caches), security (e.g., intrusion prevention system, firewalls), reliability (e.g., load balancers), and efficiency (e.g., WAN accelerators). Even though hardware middleboxes offer high performance, they are used pervasively, leading to significant maintenance and deployment costs.

In recent years the trend is changing. Increasingly, enterprises are outsourcing middleboxes to a third-party service provider as shown in Figure 1. This network architecture concept is commonly referred to as Network Function Virtualization (NFV) [**?**], and brings the benefits of cost reduction and flexibility. On the other hand, several studies have shown that the cloud is also a target for attackers [**?**], [**?**]; therefore, it is imperative to consider the following security concerns.

User traffic. The user traffic, which may contain sensitive information, is redirected to the cloud provider for processing by NFs.

VNF code and policy. VNFs can be directly developed by the enterprise. For reasons such as cyber-defense strategy and competitive advantage, the enterprise may not want to reveal the code of its VNFs and their policy inputs to the cloud.

978-1-6654-0522-5/21/$31.00 ©2021 IEEE

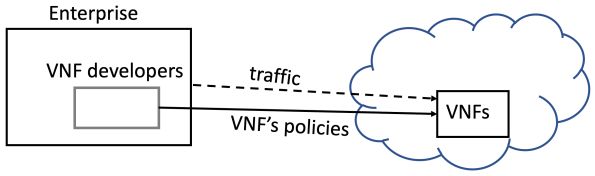

Figure 1: Outsourcing scenario

VNF states. A number of NFs used for complex packet processing tasks maintain internal states. States may contain end-user information such as cached user content and IP addresses. Such information should be hidden from the cloud.

These security issues motivate our work as we try to answer the following question: *How can NFs be securely outsourced to a third-party service provider while providing support for a wide range of NFs and maintaining high performance?*

Existing solutions which address the question above fall into two main categories; i) solutions using a cryptographic approach ([**?**] and its follow-up works), and ii) solutions using a trusted hardware based approach [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. Cryptographic solutions are either limited in functionality (not supporting complex operations needed by sophisticated NFs), or limited in performance (built on Fully Homomorphic Encryption [**?**], not up to the performance requirements of real-world systems). In contrast, a trusted hardware based approach is more flexible and powerful, being able to satisfy stricter functional and non-functional (e.g., security, performance) requirements.

We believe that a hardware-based approach, and in particular Intel SGX [**?**], is an adequate basis for a system that answers our research question above. Solutions using SGX shield the processing of the traffic from service providers by executing NFs within a protected memory region named *enclave*. SGX reduces the attack surface by preventing an administrator with root level privileges, or a malicious OS, from observing data within an enclave. While we are aware that SGX has been shown to be vulnerable to side-channel attacks [**?**], most of these attacks are hard to carry out, and Intel and the SGX community are committed to releasing security patches continuously. Unsurprisingly, there already exist several SGX-based VNF-outsourcing solutions [**?**], [**?**],

[?], [?], [?], [?].

Support for stateful VNFs. Although partially mitigating the security issues stated above, some prominent SGX-based solutions do not support stateful VNFs [?], [?], [?]. This can be a deal-breaking limitation for usage in production environments: NFs with advanced capabilities call for flow-based, stateful traffic processing. Examples of such VNFs are IP Multimedia Subsystem (IMS), Long-Term Evolution Evolved Packet Core (LTE-EPC) and so forth, which are widely used in telco applications.

Integrating a VNF framework. There exist a few solutions [?], [?], [?] with support for stateful VNFs. However, none of these integrate any known VNF library/framework with built-in functionality for developing VNFs. This might prove prohibitive from the usability standpoint: i) developers need to program their VNF from scratch, and ii) porting their VNF to be used within the enclave can be cumbersome. On the other hand, using libVNF results in 50% less code for the same NF [?].

Performance. All SGX-based solutions [?], [?], [?], [?], [?], [?] use the Linux TCP stack, known to cause a significant performance drop [?] for the following reasons: i) system call overheads, ii) lack of connection locality, and iii) inefficient packet processing. In order to provide production-level VNF performance, we believe that a user-level TCP stack is the right choice. Specifically, mTCP [?] is designed to overcome all three limitations. First, mTCP uses a user-level socket API instead of the BSD socket API, resulting in a vastly improved CPU usage efficiency. Second, mTCP implements a per-core accept queue; this way, multi-threaded applications can avoid sharing a single accept queue. Third, mTCP makes use of fast packet I/O libraries (DPDK [?] or netmap [?]) in order to process packets in batches.

Our contribution. In this paper, we introduce *SafeLib*, a library for securely outsourcing VNFs into a third-party service provider while also considering the functional and performance requirements described above. SafeLib follows the trusted hardware approach, shielding the execution of NFs within SGX enclave(s), and thus provides strong security guarantees. In particular, SafeLib builds upon Graphene [?], a practical OS library for applications on SGX. For VNF developers, SafeLib provides a powerful development framework by adapting the libVNF library [?], built on top of mTCP. The libVNF API provides support for building stateful VNFs. SafeLib is designed to overcome SGX's two main performance limitations: i) limited memory size of the enclave page cache (EPC) and ii) illegal SGX instructions [?]. First, SafeLib is carefully partitioned into a trusted and untrusted part. Second, SafeLib executes system calls within the enclave using Graphene, avoiding costly enclave to non-enclave transitions. Furthermore, SafeLib's design improves packet processing performance by using DPDK, and is able to resist Iago attacks [?]. Our performance evaluation shows that SafeLib offers an acceptable performance-security trade-off for both simple and complex VNFs. We also identify the root causes for overheads, and outline a plan to further improve performance.

The rest of this paper is organized as follows. Section II briefly describes related work. Section III introduces our threat model. Section IV presents the architecture of SafeLib. Section V touches upon implementation and deployment issues. Section VI discusses SafeLib's security guarantees. Section VII presents the performance evaluation of our library, and discusses the potential causes of overheads. Finally, Section VIII concludes the paper.

II. Related work

Here we briefly review related work focusing on the trusted hardware approach for securing NFs. Note that cryptographic solutions are currently limited in functionality and performance. More details can be found in [?], [?].

There exist a number of solutions [?], [?], [?], [?], [?], [?] which provide support for different type of applications within SGX enclaves. These solutions are not concerned with a specific scenario (securely outsourcing VNFs), and therefore answer slightly different research questions. These solutions are complementary to, and can be integrated into, our solution.

There exist also other solutions [?], [?], [?], [?], [?], [?], [?] which target outsourcing scenarios similar to ours. SEC-IDS [?], Trusted-Click [?], and S-NFV [?] propose techniques for specific cases. SEC-IDS [?] uses Graphene-SGX for isolating Snort-IDS within enclaves; S-NFV [?] uses SGX for isolation of NF state; and Trusted-Click [?] proposes a proof-of-concept Click element for pattern matching within enclaves. In contrast, our solution proposes a general library which can be applied to a wide range of NFs, and also caters for the interest of the enterprise by protecting its VNF policies and code.

SafeBricks [?] and ShieldBox [?] integrate Click to provide developers a VNF library. However, Click does not support stateful VNFs, and thus SafeBricks and ShieldBox are limited to stateless VNFs. Moreover SafeBricks does not support NFs requiring illegal SGX instructions. ShieldBox uses SCONE [?], and therefore is suitable for such NFs; but unfortunately, it does not protect packet headers.

SGX-BOX [?] and LightBox [?] provide support for securely outsourcing stateful VNFs. However, these solutions integrate neither a VNF library, nor a library OS within the enclave. This means that VNF developers need to spend effort on building their VNFs and porting their VNFs inside the enclave when using either of these two solutions. Added to this, none of these solutions provides support for NFs which might need illegal SGX instructions. Security wise, SGX-BOX [?] does not protect packet headers and VNF code, while LightBox [?] protects headers, but not VNF code. Last but not least, none of these solutions uses a user-level TCP stack, which provides many performance benefits.

To the best of our knowledge, SafeLib is the first secure outsourcing solution that provides i) support for stateful VNFs, ii) support for illegal SGX instructions by integrating Graphene-SGX, iii) protection of both header and payload of user traffic,

VNF policies and VNF code, and iv) integration of a VNF library for streamlined development.

III. Threat model

Intel SGX. The threat model relies on the properties of Intel SGX technology [**?**]. Intel SGX is a set of CPU instructions providing software isolation and attestation capabilities, both remote and local. Software isolation allows a process to request an enclave which can only be accessed by this process. The process can use the enclave to load code and data, and thus provides confidentiality and integrity for said code and data. Access to the enclave is enforced by the processor, so that even a root-level exploit of a corrupted OS cannot tamper with the contents of the enclave. Remote attestation is used to provide a secure connection between the remote verifier and the enclave. This process assures the remote verifier that outsourced VNFs run inside the enclave, and on the latest SGX platform with the corresponding security level. Local attestation is used for a similar purpose, but between two enclaves.

Threat model. Our threat model is in line with the threat model of Graphene-SGX [**?**], and is similar to the threat model of earlier SGX-based solutions [**?**], [**?**], [**?**], [**?**]. In a third-party service provider domain we consider the following components to be untrusted; i) hypervisor, OS, ii) all user-level applications residing outside the enclave, and iii) hardware components outside of the Intel CPU. On the other hand, we do trust *aesmd*, an enclave provided by SGX SDK and used to verify the creation of the enclave and its signatures. In the enterprise domain, we trust all components such as GW and VNF developers.

We claim that an attacker is unable to tamper with software execution within an enclave due to the security guarantees offered by software isolation. As a result the enclave is secure, as also assumed in all other solutions. However, the security guarantees offered by SGX software isolation are not enough to satisfy all the security requirements mentioned in Section I. Instead, we consider a powerful adversary able to compromise all untrusted components, and able to observe communications between the enclave and the enterprise GW, and also between enclaves. For instance, such a powerful attacker can compromise the OS, and then mount a Iago attack [**?**] during enclave exit by running arbitrary code, and responding incorrectly to system calls. In Section VI, we show how SafeLib protects against such a powerful adversary. Although SGX enclaves are known to be vulnerable to side-channel attacks [**?**], in line with the threat model of existing solutions [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], we consider these attacks out of scope.

IV. SafeLib: Architecture

Here we describe the design of SafeLib and our technology choices for achieving the desired functionality and performance, while protecting against the powerful adversary described in Section III. We follow the guidelines put forward in [**?**].

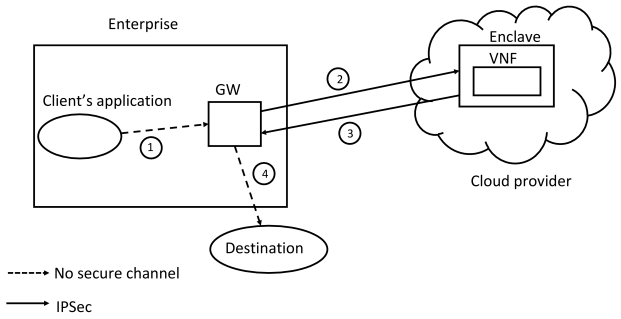

Figure 2: Bounce outsourcing model for SafeLib

A. End-to-end architecture

The current end-to-end design and deployment of SafeLib follows the Bounce outsourcing architecture [**?**]. As shown in Figure 2, SafeLib forwards the client traffic to the GW (1); in turn the GW establishes a set of IPSec tunnels to the enclave in the cloud domain (2). The VNF residing inside the enclave processes the traffic and sends the result back to the GW (3); in turn, the GW forwards the result back to the destination (4). Consistent with our threat model, the GW is a trusted entity, hence the client-to-GW traffic can be sent in the clear. Note that the client could choose to use a secure channel such as SSL, but then the GW would have to intercept the client traffic, decrypt it, then send it through the IPSec tunnel nevertheless (adding a performance overhead).

The Bounce outsourcing model makes it easier to provide our target security and functionality capabilities by keeping the related burden minimal on both the destination and SafeLib. Therefore, Safelib's performance is also evaluated in the Bounce scenario (see Section VII), with the traffic redirection bringing some latency overhead. Note that our library can be adapted to other outsourcing models [**?**], if the necessary trust relationships between domains are pre-established. Moreover, SafeLib can also be used for joint multi-operator service delivery, much discussed in 5G verticals [**?**].

B. SafeLib: high-level design

At a high level, the core of our library is an amalgamation of libVNF over mTCP [**?**] running within an Intel SGX enclave using Graphene-SGX [**?**], while also adding DPDK for batch processing (see Figure 3). Although this high-level view seems reasonably simple, the actual integration is highly non-trivial owing to the two main limitations of SGX.

The first limitation is the limited EPC memory size (128 or 256 MB). When placing DPDK inside the enclave, the memory limitation is surpassed, triggering EPC paging. This is expensive as it requires additional encryption/decryption operations between the enclave and the non-enclave region. Therefore, it became obvious that DPDK should be placed outside the enclave. That is precisely when the second main limitation is met: all instructions that are not allowed to be executed within the enclave result in an enclave exit. Such a transition happens by default via ECALLs (enclave

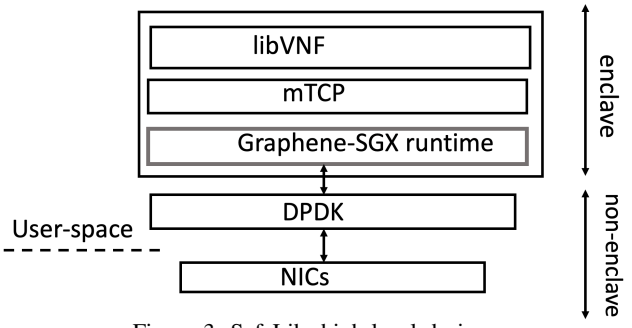

Figure 3: SafeLib: high-level design

enter) and OCALLs (enclave exit) causing overheads due to their synchronous nature. Thus, partitioning the library into enclave and non-enclave regions is a tricky endeavor: we had to reduce the amount of code and data inside the enclave (also yielding a smaller trusted computing base (TCB), a good security practice), while also minimizing the number of enclave transitions.

C. Partitioning

In Figure 4, the detailed architecture of our library is shown. To overcome SGX limitations, we partitioned SafeLib into trusted (enclave) and untrusted (non-enclave) parts. At the bare minimum, the enclave should contain the libVNF code base, the Graphene OS library, and VNF policies (if any). We now explain each component of SafeLib.

Trusted SafeLib. *Graphene LibOS* is a library that provides OS features from within an enclave. By design, SGX does not support OS features such as opening or reading a file. For such an operation the application inside the enclave gives the control back to the host OS for handling it (OCALL). After the host OS handles the operation, it returns control back to the application inside the enclave (ECALL). Having a feature-rich library OS within the enclave eliminates the need for most of these calls, with the potential to greatly increase performance. Contrary to earlier works [**?**], [**?**], Graphene-SGX is able to provide these benefits plus some protection against Iago attacks [**?**], while keeping the TCB of manageable size [**?**].

The RA-TLS library is used to establish a TLS connection between the enclave running on the remote platform (i.e, cloud) and the GW. The enterprise remotely attests the enclave in order to verify the integrity of the enclave. Finally, the enterprise sends the cryptographic keys to inside the enclave via the secret provisioning library (*Secret Prov lib*). The key is used to decrypt configuration/policy files, which the enterprise classifies as sensitive. These configuration files mainly consist of VNF policies.

IPSec endpoints. Simply excluding DPDK from the enclave is not a practical choice. VNFs (built using libVNF over mTCP) access the packet buffers that are allocated by DPDK without performing any copy operation. This means that DPDK will have access to the packets after decryption. To overcome this issue we implemented IPSec endpoints within the enclave (see Listing 1). We modified function (1) to pop pointers of

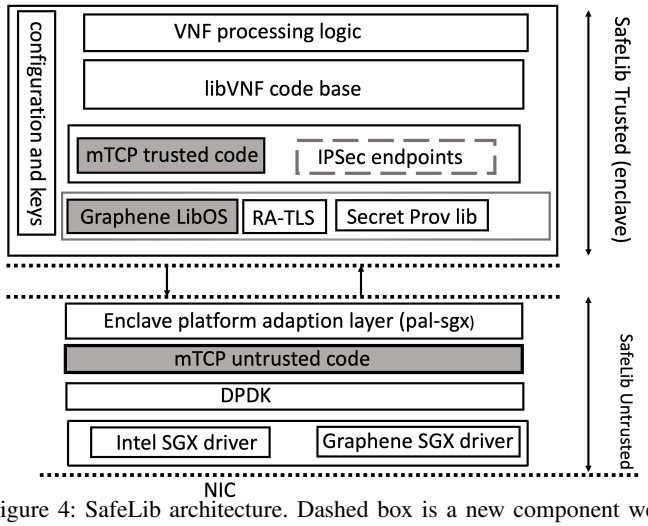

Figure 4: SafeLib architecture. Dashed box is a new component we developed, shadowed boxes are components we modified.

.

encrypted packets inside the enclave from the shared queue on the border of enclave and non-enclave regions; then we use function (2) to decrypt them once inside the enclave. After processing, function (3) encrypts the packets; then function (4) pushes the pointers of encrypted packets to the shared queue.

```
1)rte_ring_dequeue_burst(dpc->rmbufs[ifidx].rw_ring,
(void **)dpc->pkts_burst, MAX_PKT_BURST, NULL)
#receives the pointers of encrypted packets
2)esp_decrypt_aes_cbc_128_hmac_sha1_96(mtcp->mb_mgr,
rte_pktmbuf_mtod(m, char*),m->pkt_len,
cipherkey, authkey)
#performs decryption within enclave
3)esp_encrypt_aes_cbc_128_hmac_sha1_96(mtcp->mb_mgr,
rte_pktmbuf_mtod(m, char*), m->pkt_len,
cipherkey, authkey)
 #performs encryption within enclave
4)rte_ring_dequeue_burst(dpc->wmbufs[ifidx].rw_ring,
(void **)pkts_burst, MAX_PKT_BURST, NULL)
 #Dequeues encrypted pointers and sends them
    outside enclave
```

Listing 1: IPSec endpoints

Trusted mTCP. We divided mTCP into a trusted and an untrusted part. The trusted part consists of TCP packet processing and a trusted I/O interface used for interacting with *untrusted mTCP code* for packet I/O invocation (see Section V-A for details).

The libVNF code base includes libVNF programming and state abstractions. The libVNF library [**?**] is lightweight, therefore we decided to place the whole library inside the enclave.

VNF processing logic is the VNF code base outsourced by the enterprise, and built over libVNF.

Configuration and keys includes VNF policies and other configuration files considered sensitive. The actual configuration files are specific to the VNF scenario, but always include the IPSec configuration file and IPSec and TLS keys. IPSec keys are used for encrypting/decrypting the packets reaching the enclave, while the TLS key is used for decrypting VNF policies.

Untrusted SafeLib. The *pal-sgx loader* is used to initialize the enclave by calling SGX drivers. During the initialization

Table I: OCALLs interface between trusted and untrusted SafeLib

OCALLs	Description
`ocall_set_net_env`	Gets CPU and memory info, and calls `rte_eal_init()` to initialize `dpdk_eal_env`
`ocall_dpdk_load_module`	Loads and configures dpdk device and deploys Rx/Tx queues for dpdk packet I/O
`ocall_dpdk_init_handle`	Creates Tx/Rx ring buffers
`ocall_dpdk_destroy_handle`	Releases resources allocated by `ocall_dpdk_init_handle`

phase, the enclave contains a shielding library, VNF code, Graphene LibOS, standard C libraries, and a manifest file which specifies all binaries of the enclave libraries. The manifest includes integrity measurements of all libraries residing within the enclave. The shielding library will open these libraries only if their computed SHA-256 hash matches the manifest. The shielding library continues to load other libraries inside the enclave after the initialization phase, given that their hash value is correct. (Note that the shielding library and manifest are part of trusted SafeLib but we omit them in Figure 4 for readability. For more details please refer to [**?**]).

Untrusted mTCP code. This component has two main tasks. It defines the DPDK thread for the ring buffer, and interfaces with the *trusted mTCP code*. We give more details in Section V-A.

DPDK resides outside the enclave, and communicates with the enclave via the interface in *mTCP untrusted code*. DPDK is a fast packet I/O library which processes packets in batches. mTCP supports both DPDK and netmap; our current SafeLib implementation relies on DPDK. Integrating DPDK into our scenario is not straightforward: we had to modify DPDK to work with encrypted packets, since packets are only decrypted once they reach the enclave.

V. Implementation and Bootstrapping

Here, we provide some details on SafeLib's implementation and bootstrapping.

A. Packet I/O

Excluding DPDK from the enclave results in two interfaces between trusted and untrusted parts of SafeLib. The first interface is a collection of four OCALLs (see Table I) used for starting and stopping DPDK. This interface is called only during initialization and termination. Hence, the invocation of this interface does not result in performance overheads.

The second interface (see Figure 5) is used to handle every send and receive operation (packet I/O invocation) asynchronously, and is the only active interface during run-time. We designed N DPDK threads, where each thread receives and sends pointers from/to the NIC in bursts. Then each thread pushes the pointers of packets to an RX ring, and pops pointers of packets from the TX ring. We designed N mTCP threads running inside the enclave, used to pop the available batch of

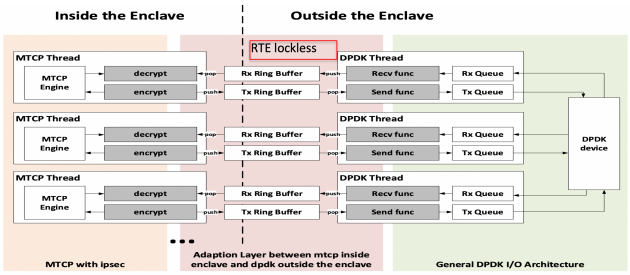

Figure 5: Packet I/O via RTE lockless ring

pointers from the RX ring from within the enclave, and, after processing, to push the pointers of the packets to the TX ring from within the enclave. This way, there is no need for enclave transition for each send and receive operation.

RX and TX rings are implemented as RTE lockless rings, and are used as a shared memory between enclave and non-enclave regions. RX/TX rings contain only the pointers of packet buffers, processing them in place without any copying. We use a pair of RX/TX rings per mTCP thread because each mTCP thread has its own bidirectional network flow. Note that the procedure shown in Figure 5 is motivated by [**?**]; however, SafeLib works with encrypted packets and supports generic stateful VNFs.

B. Illegal SGX instructions and Iago attacks

System calls and instructions leading to enclave exit (e.g., `rdtsc`) are not allowed within the enclave. Our library ensures that no system call results in an enclave exit. SafeLib inherits this property for three main reasons: i) mTCP eliminates system calls used for event handling with batched function calls, and also uses user-level socket API [**?**]; ii) Graphene-SGX provides support for most used Linux system calls from within the enclave (e.g., `open`, `close`, `read`, `write`, `fstat`, `fchmod`, `poll`, etc.); and iii) for other system calls Graphene-SGX provides "exitless" features (executing them within the enclave) by using RPC thread.

Although we carefully designed and implemented the I/O interface, initial experiments showed prohibitive performance overheads. Further investigation revealed that `clock_gettime()` was the main culprit: this system call is resolved in user space by `rdtsc`, and is not supported by Graphene-SGX; this resulted in an enclave exit for each invocation. Therefore, we adopted an existing technique [**?**], and introduced a "secure clock" thread inside the enclave. This thread uses dummy code which emulates `rdtsc` to resolve this system call within the enclave.

On a further note, we have decreased the possibility of Iago attacks [**?**] in multiple ways by i) using an asynchronous interface (Figure 5), ii) integrating Graphene LibOS to provide intra-enclave Linux system calls [**?**], and iii) emulating `rdtsc` inside the enclave.

C. VNF chaining

We provide basic support for VNF chaining by running all VNFs on separate enclaves. One can use the *local attestation procedure* to attest the enclaves, and then establish a TLS

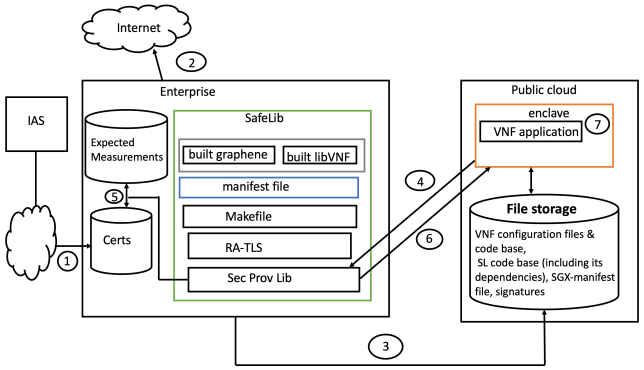

Figure 6: Deployment procedure of SafeLib; the orange box refers to the enclave (detailed in Figure 4), the green box refers to the SafeLib code base and its dependencies, the double arrow between enclave and file storage denotes the dynamic loading process of the enclave.

channel between enclaves. Support for VNF chaining is provided by the combination of libVNF and Graphene by forking a child enclave and copying the content of the parent enclave (except VNF binaries and policies, of course) via message passing [**?**].

Note that such a mechanism would not satisfy our security requirement on user traffic, as TLS does not protect packet headers between enclaves. Our current implementation circumvents the issue by redirecting the traffic from the parent enclave through the GW to the child enclave over IPSec; this naive approach introduces large overheads. We outline a potential real solution in Section VIII.

D. Bootstrapping the system

To make use of SafeLib, the enterprise has to request an enclave and, after creation, verify the enclave running in the cloud. This process involves two parties, enterprise and public cloud, as shown in Figure 6. The first preliminary step is for the enterprise to retrieve SGX-related information from the Intel Attestation Service (IAS) (1). The second preliminary step is for VNF developers to download[1] and build SafeLib together with its dependencies (e.g., Graphene-SGX, libVNF, mTCP, etc.) (2). Our manifest file lists all necessary binaries, but VNF developers should take at least two actions before building SafeLib: i) generate their own cryptographic key, and encrypt configuration files (VNF policies plus their own IPSec config), and ii) load these configuration files into the manifest. Now they can build SafeLib, and then load VNF-specific executables in the manifest file. The process of building SafeLib generates enclave signatures and an SGX-specific manifest file defining the integrity measurements of the VNF's configuration files and executables plus SafeLib's executables. Next, VNF developers ship all these files to a file storage on the untrusted host in the cloud (3). Then, SafeLib is executed on the untrusted host creating an enclave as depicted in Figure 4; in this phase, the dynamic loading process of the enclave and integrity checks are performed. Meanwhile, VNF developers start the secret provisioning library on the trusted host. Using RA-TLS (4), the trusted and untrusted hosts establish a TLS connection, and VNF developers verify the enclave running on the untrusted host (5). If the verification is successful, VNF developers send their cryptographic keys to the untrusted host via TLS (6). With the cryptographic keys provisioned, the untrusted host can start executing the VNF application (7), and decrypting the configuration files within the enclave.

Afterwards, the GW configures a set of IPSec tunnels: one per client application thread. To load balance flows on the cloud side we use Receive Side Scaling (RSS) based on port number. RSS makes sure that packets from the same tunnel end up at the same core.

Table II: An overview of security properties provided by SafeLib.

Property	**Security**
VNF execution	Integrity
VNF state (flows streams)	Integrity and confidentiality
VNF policies	Integrity and confidentiality
VNF code	Integrity (confidentiality: planned)
User traffic	Integrity and confidentiality

VI. Security Guarantees

As mentioned in Section III, we are only concerned with an attacker able to observe the communication between enclaves and between the GW and the enclave. (Recall that side-channel attacks against SGX are out of scope.) Here we describe how SafeLib protects against the powerful adversary considered in the threat model. Table II summarizes all security properties that our library offers.

Confidentiality and integrity of VNF's execution and states. Since the VNF's processing is done within the enclave, the integrity of execution is provided by SGX. *As of now*, we store all VNF states within the enclave, so confidentiality and integrity of VNF states are also provided by SGX.

Confidentiality of VNF policies and code. We use RA-TLS and secret provisioning libraries to allow VNF developers to encrypt configuration files (e.g., VNF input policies) using their own cryptographic key, and then ship this key to inside the enclave via TLS. The key is then used to decrypt the configuration files, once they are loaded within the enclave. The key is sent via TLS, and then stays inside the enclave: the confidentiality of VNF policies is guaranteed. As of now, SafeLib does not protect the confidentiality of the VNF code. Although the attacker has to invest significant effort to reverse-engineer the binaries sent to the non-enclave region of the untrusted host, and with questionable results, this threat is not fully mitigated. On a positive note, the developers of Graphene[2] will provide support for VNF code confidentiality soon; we plan to integrate this feature to SafeLib promptly.

Integrity of VNF policies and code. To provide integrity protection, we rely on the shielding library of Graphene-SGX which compares the original and the to-be-deployed hashes,

[1]SafeLib's source code is available at https://github.com/eniomarku/SafeLib

[2]Private communication with Dmitrii Kuvaiskii (Intel), December 2020

and loads configuration files and VNF binaries into the enclave only if hashes match.

Confidentiality and integrity of user traffic between GW and enclave. The main benefit of our library is that we protect the confidentiality and integrity of user traffic. SafeLib protects both headers and payload using IPSec in tunnel mode. Moreover, we have implemented IPSec endpoints within the enclave as explained in Section IV-C: traffic is protected all the way between the GW and the enclave. Also, if an attacker in the cloud attempts a Iago attack [**?**], such as modifying queues or packet buffers from outside the enclave, this will be detected by the authentication mechanism of IPSec. Furthermore, we transfer the IPSec keys to the enclave via TLS established using the RA-TLS library in the setup phase. Therefore, SafeLib also protects IPSec keys. Moreover, the integrity of the traffic is guaranteed by IPSec. Note that SafeLib does not protect traffic metadata.

Confidentiality and integrity of traffic between enclaves. As mentioned in Section V-C, if TLS is used for communication between enclaves then our library provides confidentiality and integrity for the payload. If IPSec is used then our library provides confidentiality and integrity for both payload and header, albeit with considerable performance overhead owing to multiple traffic re-directions.

VII. Performance Evaluation

We have evaluated SafeLib in several VNF scenarios; we only present two of them due to the lack of space. Note that SafeLib has performed within the same range in all cases. Scenarios we show here involve VNF chains of two, where a single VNF is outsourced to a public cloud. We run micro-benchmarks, and compare the end-to-end performance of the SafeLib-enabled outsourced scenario to vanilla libVNF; it is shown that libVNF itself does not introduce measurable overheads [**?**].

Setup. We use two Intel(R) Core(TM) i7-8809G CPU@3.10GHz servers with 8 cores, and with an enclave size of *128MBs*. Our computers are interconnected via a 1Gbps link. Note that in both scenarios the outsourced VNF does not run inside a VM, but directly on the physical machine. The reason for this is, while using DPDK, we could not run mTCP inside a VM; we found that mTCP could not interpret the multiple queues exposed by the BESS software switch.

Scenario AB. This scenario is a very simple service chain, and consists of only two VNFs: VNF A, implemented over *pthreads* as a multithreaded C++ application, and VNF B, built over libVNF, and securely outsourced using SafeLib in order to be executed within the SGX enclave. Here, we focus on worst-case throughput: VNF B operates close to capacity. In order to saturate VNF B, VNF A generates multiple requests to VNF B in a closed loop fashion. For each received request, VNF B performs a CPU-intensive computation such as updating a value, and then replies back to VNF A. The performance of VNF B is evaluated by means of end-to-end throughput (number of completed requests per second). Figure 7 reports an average of 5 runs of 120 seconds (with minimal variation between runs, standard deviation is omitted).

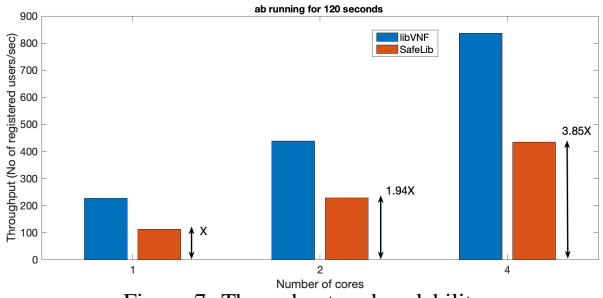

Figure 7: Throughput and scalability

The results offer two takeaways. First, SafeLib's throughput scales linearly with the number of cores used. This is crucial: SafeLib preserves the scalability advantage of cloud computing. Second, the worst-case overhead is around 50%, and is independent of the number of cores used. While this may seem a little steep, recall that this VNF is operating close to capacity. Modern cloud management platforms offer auto-scaling mechanisms that are able to adjust available resources on the fly; thus, such unfavorable operating points could be avoided [**?**]. In fact, results show much lower overheads under lighter loads (not shown due to the lack of space).

Scenario LTE-EPC. Evolved Packet Core (EPC) is a framework used to handle converged data and voice on a Long-Term Evolution (LTE) network. The three main components of EPC are: i) Mobility Management Entity (MME) used for handling the registration of mobile users in the network, among other tasks , ii) Packet GateWay (PGW) is mainly used to manage quality of service (QoS), and acts as an interface between LTE and other packet data networks; and iii) Serving GateWay (SGW) is used for routing the packets through the access network. Our scenario is similar to the scenario used in [**?**] with a few changes: i) we implemented SGW and PGW over TCP, ii) we make use of HSS for initial attachment and authentication, and iii) SGW, HSS and PGW run over the Linux kernel stack. Our EPC framework is simplified, handling only data path setup, registration/de-registration of mobile users and data transfer. MME is built with the libVNF API, and then outsourced to the server machine using SafeLib in

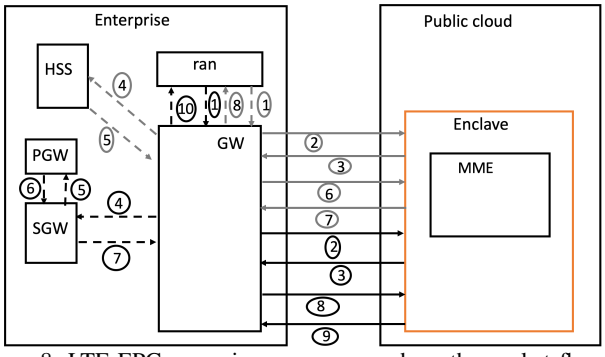

Figure 8: LTE-EPC scenario; gray arrows show the packet flow for *initial attach and authentication* phase, black arrows show the packet flow for *session setup and detach* phase

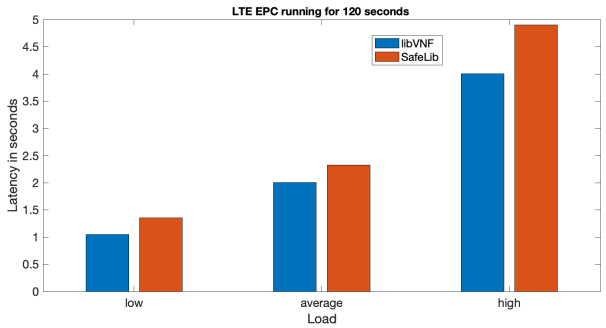

Figure 9: Latency

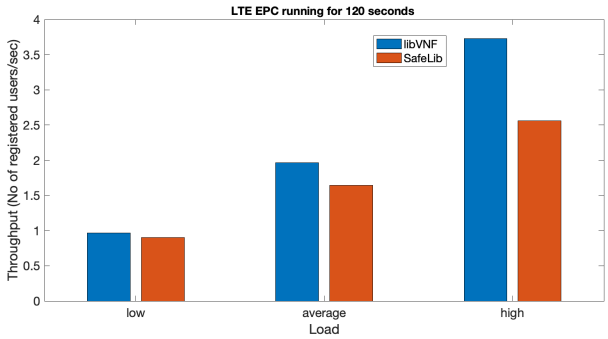

Figure 10: Throughput

order to be executed within the enclave. All other components including a load generator reside in the enterprise (see Figure 8). The load generator is a C++ application used to generate data and control plane traffic in a closed loop fashion.

Again, we compare the performance of SafeLib to vanilla libVNF. Performance is measured in terms of end-to-end latency (the time it takes to successfully register a mobile user) as shown in Figure 9 and end-to-end throughput (number of mobile users registered successfully per second) as shown in Figure 10. We report averages of 5 runs of 120 seconds each under three different load settings, where high load is close to saturation, while low and medium settings roughly corresponds to 25% and 50% load, respectively (with minimal variation between runs, standard deviation is omitted.)

Our main observation is that overheads are clearly load-dependent. Regarding throughput, SafeLib introduces $\approx 7\%$, 13% and 27% overhead for low, medium and high load settings, respectively (latency values are similar). There is a clear proportional relationship between load and overhead. Note that we have also evaluated SafeLib for longer time spans, and have not observed any increase in overheads.

Discussion on performance. Although SafeLib introduces reasonable overheads, especially under lighter loads, we believe there are ways to improve, so that our library may perform close to line rate speed. Such improvements could be particularly important when running outsourced VNFs close to capacity; this may happen when the Service Level Agreement (SLA) between enterprise and cloud contains forgiving performance guarantees in exchange for lower prices [**?**]. Based on our investigations, our current understanding of the main culprits for overheads introduced by SafeLib are the following.

EPC size. In our setup, EPC size is 128 MBs. This means that the procedure of EPC paging is triggered more often compared to if we were to use brand new Intel CPUs with an EPC size of 256 MBs. By switching to brand new Intel CPUs, we have a straightforward solution to this issue.

Bounce architecture. Depicted in Figure 8, we are evaluating end-to-end throughput while redirecting the traffic back to the GW. Such setup introduces additional overheads, especially with two-way IPSec communications. To overcome this issue, we plan to provide support for the Direct outsourcing model [**?**], by using a GW in the cloud domain, instead of redirecting the traffic back to the enterprise.

States. The current version of SafeLib keeps all states of stateful VNFs inside the enclave. This inflates the size of TCB, and may cause additional EPC paging. In fact, we plan to implement a more efficient state management in the future. We will divide states into active and inactive, and keep only active states inside the enclave. We will encrypt inactive states before putting them into the main memory. This approach also requires a timer to keep track of states, retrieve them into the enclave upon activation, and decrypt them before modifying.

Note that while there could be other potential performance issues, utilizing Graphene-SGX does not introduce significant overheads. We ran performance evaluations for the AB and IMS scenarios [**?**] with modifications. We used the Bounce architecture, but i) without IPSec tunnels between enterprise and cloud, and ii) without using mTCP and DPDK, relying on the Linux kernel TCP stack. In such a stripped setup we measured overheads of around 2% for high loads; consequently, Graphene LibOS itself is not a performance bottleneck for SafeLib.

VIII. Conclusion

In this paper we have presented SafeLib, a practical solution for outsourcing general purpose network functions securely. SafeLib is the first secure outsourcing solution that provides i) support for stateful VNFs, ii) support for illegal SGX instructions by integrating Graphene-SGX, iii) protection of both header and payload of user traffic, VNF policies and VNF code, and iv) integration of libVNF for streamlined development. Our performance evaluation shows reasonable performance overheads compared to vanilla libVNF, with room to improve under high load conditions.

Limitations and future work. First and foremost, we plan to implement the security (confidentiality of VNF code, see Section VI) and performance enhancements as discussed above. Adding to these, we are working on adding support for stateless VNFs relying on libVNF's API (recall that SafeLib's support for stateful VNFs comes from the integration of libVNF), making SafeLib the first solution for *really* general-purpose VNFs.

Second, SafeLib's support for VNF chaining (Section V-C) is rudimentary. While SafeBricks [**?**] supports multiple NFs within the same enclave, Graphene-SGX does not offer such a feature. Instead, it uses TLS between NFs in separate enclaves; this would break our security guarantees for user traffic. Hence, SafeLib uses multiple IPSec-based re-directions to

overcome this issue, hurting end-to-end performance. We are currently investigating ways of using IPSec directly between enclaves. One approach we are closely monitoring is SGX-LKL [**?**] which sets up a set of etap devices and a virtual interface in the enclave serving as an IPSec endpoint.

Third, SafeLib has limitations owing to being based on SGX. The limited enclave memory size renders the secure outsourcing of NFs with very large computing power impractical. Note that future versions of SGX may provide a larger enclave-dedicated memory. Another inherent limitation of SGX is its vulnerability to side-channel attacks; we rely on on Intel updates to mitigate such attacks. Moreover, SGX is a feature of Intel CPUs, so our library can only be used with such processors. Last but not least, our library cannot currently be deployed into a container (e.g., Docker) owing to an inherent limitation of mTCP and DPDK, which may change in the near future.

ACKNOWLEDGEMENTS

We would like to thank Priyanka Naik and Mythili Vutukuru (libVNF), and Dmitrii Kuvaiskii (Graphene-SGX) for providing helpful guidance.