


# SENG, the SGX-Enforcing Network Gateway: Authorizing Communication from Shielded Clients



Fabian Schwarz   
CISPA Helmholtz Center for  
Information Security  
fabian.fsblack@gmail.com

Christian Rossow  
CISPA Helmholtz Center for  
Information Security  
rossow@cispa.de

## Abstract

Network administrators face a security-critical dilemma. While they want to tightly contain their hosts, they usually have to relax firewall policies to support a large variety of applications. However, liberal policies like this enable data exfiltration by unknown (and untrusted) client applications. An inability to attribute communication accurately and reliably to applications is at the heart of this problem. Firewall policies are restricted to coarse-grained features that are easy to evade and mimic, such as protocols or port numbers.

We present SENG, a network gateway that enables firewalls to reliably attribute traffic to an application. SENG shields an application in an SGX-tailored LibOS and transparently establishes an attestation-based DTLS channel between the SGX enclave and the central network gateway. Consequently, administrators can perfectly attribute traffic to its originating application, and thereby enforce fine-grained *per-application* communication policies at a central firewall. Our prototype implementation demonstrates that SENG (i) allows administrators to readily use their favorite firewall to enforce network policies on a *certified* per-application basis and (ii) prevents local system-level attackers from interfering with the shielded application’s communication.

## 1 Introduction

Companies and sovereign institutions aggregate increasing amounts of sensitive digital information, while the number of attacks on them is proliferating steadily at the same time. Attackers regularly infiltrate systems to steal information and disrupt competitors, e.g., using social engineering (phishing) or advanced exploits (watering hole, zero days) [18]. As a response, organizations harden endpoints, deploy network-based attack detection systems, and train their employees. Yet, given the abundance and power of attacks, preventing any kind of information leakage has become practically infeasible, even in highly-secure settings and in absence of internal attackers.

Foremost among these problems is the fact that containing an organization’s incoming and outgoing communication is

almost impossible. On the one hand, network administrators deploy firewalls and Intrusion Detection Systems (IDS) to tightly control and contain information flows. On the other hand, they have to support a vast diversity of applications and access methods and lack a mapping between which application causes which traffic. This enables internal clients to (possibly unknowingly) leak data by executing untrusted or even malicious software. Furthermore, companies opening their servers to partners lack control over which remote client applications are used to access these servers.

One fundamental solution to this problem is a certified attribution of network traffic to its application, which would allow for app-specific communication policies. Existing attempts to attribute traffic fall short in their security guarantees, as they (i) rely on protocol identification and thereby can be evaded by traffic morphing [24], (ii) rely on host-based sensors that can be evaded or manipulated by local attackers, or (iii) are host-based only and cannot be used at central perimeter firewalls. In fact, reliable traffic-to-app attribution is challenging, as attackers can inject code into trusted processes [4] and abuse their identity. For example, if malware injects itself into browsers, it hides its functionality within an otherwise trusted process and thus inherits the browser’s identity and privileges. Lacking a hardware-based trust anchor, existing attribution attempts can be fooled by system-level attackers.

To tackle this underlying core problem, we require a design that (i) shields processes from system-level attackers and (ii) gives stronger integrity protection of processes than just their name or any sort of other loose identifier. In fact, trusted execution environments (TEEs) like Intel SGX [13] ensure such hardware-enforced protections and have been the subject of endeavors to shield client applications [23, 31] and outsourced network services [7, 45, 57]. Library operating systems (LibOSes) tailored for SGX wrap and shield unmodified client and server applications, thus protecting legacy applications out of the box [2, 5, 9]. However, while they do enable transparent shielding and attestation, existing LibOSes fail to provide the following two guarantees. First, they rely on the untrusted host’s network stack, s.t. local system-level ad-

versaries can still manipulate and redirect traffic (e.g., DNS spoofing, IP/TCP header modification). Second, the network gateway is still entirely blind to the concrete application which is sending and/or receiving data. Gateways can therefore neither block unauthenticated, vulnerable senders (e.g., malware, shadow IT) nor restrict communication with security-critical servers to certain trusted client applications.

In this paper, we present SENG, a network gateway service coupled with a client-side runtime library, which aims to solve the above problems. SENG transparently protects the connections of applications that are shielded in an SGX-tailored LibOS to prevent packet manipulation and redirection attacks by local system-level attackers. Technically, SENG automatically establishes attestation-based, trusted DTLS channels between the SGX enclaves and the central network gateway. Traffic from and to an enclave is wrapped in the respective secure tunnel and thus inherits enclave-to-gateway confidentiality and integrity guarantees. Furthermore, this design allows the gateway to link traffic to the trusted application causing it. Consequently, the gateway can distinguish between traffic from shielded and unshielded applications and can ultimately enforce central fine-grained *per-application* policies. We have designed SENG in such a way that shielded apps are wrapped in an SGX-based LibOS without requiring any modifications. This allows us to shield legacy binaries without source code changes and completely independent of the underlying network protocols. We also provide an alternative SENG design, which operates *without* LibOS and provides SENG support for enclaves based on Intel’s SGX SDK [25] instead. While the latter does require application modifications, it outperforms the LibOS variant in terms of performance.

To demonstrate the general feasibility, we have developed SENG in an open-source (cf. Section 12) C++ prototype based on Graphene-SGX [9]. Our proof-of-concept illustrates the security benefits of an SGX-enforcing gateway. To highlight the two most important merits, SENG (i) allows network administrators to readily use their favorite firewall implementation (e.g., Netfilter/iptables [40]) to enforce network policies on a *certified* per-application basis and (ii) prevents local system-level attackers from interfering with the shielded application’s communication.

In summary, we make the following contributions:

- We design SENG, which transparently (i.e., without the need of code rewriting) shields applications to protect and attribute their network traffic.
- SENG enables tight control over network communication at the perimeter and thereby mitigates information leakage by untrusted applications. Consequently, central firewalls can enforce the use of particular trusted applications for traffic entering or leaving their network.
- We implement and release a prototype and thoroughly evaluate its performance based on network- and microbenchmarks as well as a set of real-world client (cURL, Telnet) and server (NGINX) applications.

## 2 Threat Model

Centralized network firewalls (“perimeter firewalls”) are a core security instrument in any network [19]. Network administrators typically segment clients and servers into disjoint subnetworks that are interconnected via a central network gateway—a classical demilitarized zone (DMZ) firewall setup, as shown in Figure 1. They can then specify firewall policies based on source and destination addresses and protocol information to regulate communication between these segments. To retain security guarantees of perimeter firewalls, administrators usually aim to prohibit secondary WAN connections (e.g., 4G/5G) or other bridges that would subvert the gateway’s centralized position.

Unfortunately, perimeter firewalls are restricted to coarse-grained policies. They filter traffic based on host information (IP addresses, port number) and transport protocol (e.g., TCP or UDP). Firewalls cannot filter communication *per application*, as the application source is unknown. Firewalls therefore lack mechanisms to block communication of undesired and/or potentially malicious software. Firewalls have been extended to learn about client programs using host-based sensors [11]. However, these existing app attributions can be undermined when attackers compromise client systems (cf. Section 3), as malware can inject into allowlisted processes [4], or escalate its privileges to subvert host sensors.

This challenging setting is exactly our use case. We aim to provide *app-grained traffic attribution* to organizations with stationary clients that are potentially compromised by malware and/or want to isolate untrusted apps. Identical to the firewall setting (“bastion host”), also in our threat model the firewall and its underlying system is fully trusted. In contrast to firewalls, however, we tolerate a system-level attacker fully controlling the client’s software stack, including its OS and hypervisor(s). That is, we do not mistrust the user or its hardware, but allow its host system to be fully compromised. After compromise, attackers will attempt to leak sensitive host information either directly or indirectly by manipulating the network traffic of shielded applications.<sup>1</sup>

To tackle this problem, we leverage trusted hardware to enable firewalls to rely on app identifications for network traffic. Technically, we shield client apps inside an Intel SGX enclave with a trusted LibOS. Administrators can then maintain a list of trusted apps and use their identifiers to create firewall policies that govern which network resources a given app can access. For ease of discussion, we protect client systems and assume that internal servers are not compromised, while our methodology can also be applied to servers in principle.

For our work, we follow the classical SGX threat model. Denial-of-Service (DoS), side-channel attacks, and physical attacks against the CPU are out of scope [35, 58, 59] and can be tackled by orthogonal work [1, 41, 49, 54]. Similarly,

<sup>1</sup>We refer to related work to mitigate covert channels [8, 60] and focus on stopping explicit and malicious information exchange instead.

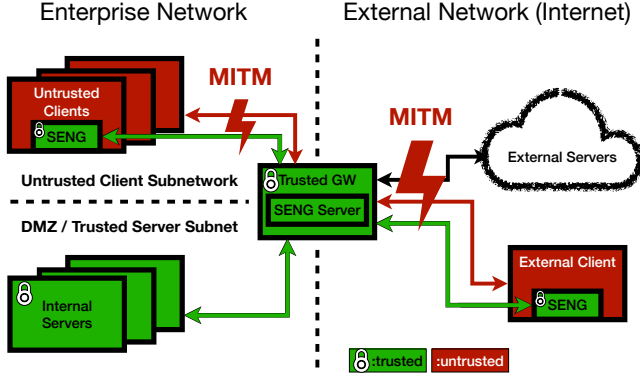


Figure 1: Overview of Network Topology and Threat Model

enclaves are trusted and free of vulnerabilities. Any disk I/O by the application has to be protected (e.g., hashing files and transparent sealing as provided by existing file system shields and SDK functions [2, 9, 25]). Finally, we assume that all locally exposed enclave interfaces are shielded [55] to avoid an oracle-like API access that could be abused for information leaks based on confused deputy attacks.

### 3 Related Work

Table 1 summarizes related work and its deficiencies to cope with our threat model. For the discussion, we consider the following attackers: (a) user-space malware ( $MW_{user}$ ), (b) system-level attackers at the client ( $Sys_{cli}$ ) or middlebox ( $Sys_{mbox}$ ), and (c) on-path MITM attackers ( $mitm$ ). The last four columns rate if an approach fulfills (yes: ●, no: ○, n/a: -) the following requirements: (i) Confidentiality and integrity ( $C+I$ ) of client traffic (incl. IP headers and DNS queries), (ii) traffic authentication ( $TA$ ) of either protected client or host sensor traffic, (iii) secure (client) traffic-to-app attribution ( $Attr$ ), and (iv) protection against information leakage ( $\neg IL$ )—defined as security requirements SR2–SR6 in Section 5.

**Perimeter Firewalls with Host Information.** Perimeter firewalls with client-side sensors are most related to SENG. However, they fail to provide reliable traffic-to-app attribution ( $Attr:○$ ), which is our central design goal. Host sensors like the Cisco Network Visibility Module (NVM) [11] focus on firewall augmentation with per-flow host data, including app identifiers (e.g., hash of binary, process name). Unfortunately, malware can easily bypass such loose, static identifiers by injection into benign processes [4]. Furthermore, a system-level attacker can completely subvert host sensors such as NVM, as they fully rely on the OS. SOCKS [52] proxies and VPN [15] services also control traffic centrally, but, similarly, they cannot reliably link traffic to its applications.

**Isolation-Based Traffic Auditing.** Assayer [43] uses a client-side hypervisor to augment app-level data of outbound client traffic with traffic statistics and signs it ( $C+I:○, TA:●$ ).

However, Assayer has no insights into the app identities of annotated traffic (no introspection) and cannot prevent infected or malicious apps from submitting arbitrary traffic for annotation. Thus, Assayer can neither provide traffic-to-app attribution ( $Attr:○$ ) nor prevent leaks by malware ( $\neg IL:○$ ).

Alcatraz [3] establishes secure tunnels between SGX enclaves integrated into network nodes (incl. clients and gateway). Traffic is securely tunneled between enclaves with hop-specific keys to provide traffic confidentiality and integrity as well as path integrity. While Alcatraz shields tunneled IP traffic from MITM attackers and compromised switches, Alcatraz doesn’t protect traffic against client compromise ( $C+I:○$ ). Therefore, Alcatraz’s client enclaves cannot link traffic to apps ( $Attr:○$ ) and do not restrict access to the tunnel, s.t. local attackers can send arbitrary authenticated IP packets ( $\neg IL:○$ ).

EndBox [22] outsources middlebox services to untrusted client systems for scalability. EndBox runs inside an SGX enclave and tunnels all app traffic through a VPN connection ( $C+I:●$ ) to the gateway, which blocks traffic that does not arrive through the enclave-terminated VPN tunnel ( $TA:●$ ). However, similar to Alcatraz, EndBox cannot enforce app-grained policies ( $Attr:○$ ), as all client apps are untrusted.

Container overlay networks like Slim OS [61] or Docker-based networks [14] assign virtual IP addresses to containers enabling per-container firewall policies at virtual switches. However, they cannot protect against system-level attackers, as they trust the client OS, have no HW-based container identifiers, and do not deal with information leakage.

**Client-side Solutions with Host-level Firewalls.** Host-based firewalls enforce policies directly at the client host, but do not provide an enterprise-wide decision and enforcement point. They are often combined with compartmentalization frameworks which confine apps in sandboxes to mitigate system compromises, which lead to direct firewall subversion.

For example, iptables [40] is the de facto standard firewall configuration tool in Linux. A Debian extension allows policies per user and process ID [27], while mandatory access control (MAC) modules [51, 56] allow fine-grained policies (incl. app-grained). However, none of these approaches shares data with a central gateway firewall. While some firewalls support labeled IPsec, which can negotiate MAC contexts as traffic selectors [28], labeled IPsec faces major configuration and key management complexity. ClipOS [12] is a hardened Linux which sandboxes apps and plans to include multi-level compartmentalization support. However, system-level attackers can subvert all aforementioned approaches.

QubesOS [48] uses Xen to sandbox all apps into isolated VMs and provides per-app VM network policies. QubesOS could thus be modeled to enable app-grained, central policy enforcement by setting up separate VPN tunnels for each application VM and enforce rules on the unique per-app VPN IP addresses. However, this would require a complex client setup and requires trust in the hypervisor. In contrast, we want to root our app attribution in hardware and stay fully

Project	Components at...	Trust in...			Attackers	Central?	(SR2/3)	(SR4)	(SR5)	(SR6)
		OS	VMM	CPU			C+I	TA	Attr	¬IL
<b>SENG</b> NVM <i>et al.</i> Assayer Alcatraz EndBox	Client, Gateway	no	no	✓	Sys <sub>cli</sub> , mitm	yes	●	●	●	●
	Client, Gateway	yes	-	✓	MW <sub>user</sub>	yes	○	-	○	○
	Client, MBox, Srv	no	yes	✓	Sys <sub>cli</sub> , mitm	yes	○	●	○	○
	Cli/Srv, MBox, Gw	no	no	✓	Sys <sub>cli+mbox</sub> , mitm	yes	○	●	○	○
	Client, Gateway	no	no	✓	Sys <sub>cli</sub> , mitm	yes	○	●	○	○
iptables MAC	Client	yes	-	✓	MW <sub>user</sub>	no	○	-	●	●
ClipOS	Client	yes	-	✓	MW <sub>user</sub> , mitm	no	○	-	●	●
QubesOS	Client	no*	yes	✓	Sys <sub>cli</sub> , mitm	no	○	-	●	●
SafeBricks	Gateway, MBox	yes	yes	✓	Sys <sub>mbox</sub> , mitm	yes	○	-	-	-
LightBox	Gateway, MBox	yes	yes	✓	Sys <sub>mbox</sub> , mitm	yes	○	-	-	-

Table 1: Related work grouped into perimeter firewalls with host sensors, host-level firewalls, and secure middleboxes. Assessments follow the metrics, symbols and acronyms outlined in Section 3. (\*note: QubesOS trusts OS of admin dom0, though)

compatible with existing gateway firewalls.

**SGX-Protected Middlebox Outsourcing.** Projects such as SafeBricks [45], LightBox [16] and ShieldBox [57] use SGX to protect middlebox services from untrusted cloud or middlebox providers. The approaches mostly differ w.r.t. the focus and implementation. SafeBricks, for instance, uses language-based methods to enforce least privilege on third party middlebox functions and isolation across chained functions, while LightBox [16] focuses on support for stateful full-stack middlebox functions and high-performance. Gkantidis et al. [21] additionally propose a middlebox-aware TLS variant (mbTLS) for secure inspection of encrypted client traffic. In contrast to our threat model, these projects trust the client hosts, and thus fail to provide app-to-traffic attribution (Attr:-) and to mitigate information leakage (¬IL:-). The middleboxes can directly benefit from our desired traffic attribution, as they integrate easily (cf. AR3 in Section 5.1).

## 4 Background

**Intel SGX and Remote Attestation.** TEEs provide an abstraction to run a process isolated from the remaining system. TEEs enforce hardware-based protection of the integrity and confidentiality of the contained code and data and have means to prove it to external entities [13, 44].

In the following, we focus on Intel SGX, which forms the basis for our overall design. SGX’s TEE entities are *enclaves*, which only rely on the security of the CPU. Enclaves provide a dedicated memory region called enclave page cache (EPC) which is isolated and transparently encrypted and authenticated. The enclave app code is limited to user space instructions, s.t. enclaves depend on the cooperation of the untrusted OS for system calls and interaction with hardware devices. Therefore, SGX provides direct access to untrusted memory and the notion of enclave calls (ECALLs) and outside calls (OCALLs), which allow controlled transitions between the trusted and untrusted world. Furthermore, SGX allows to

store data encrypted on the disk via a sealing key derived by the CPU and only accessible to the respective enclave [13].

SGX enclaves can prove their identity and protection to local and remote entities. For local attestation, the CPU creates a cryptographic report of the enclave, which contains a measurement (secure hash) of the initial enclave state. The report is signed by the CPU with the key of the local challenger enclave and can then be passed to the challenger for verification. For remote attestation, the Intel-provided Quoting Enclave (QE) acts as local challenger. The QE then adds the platform state and forwards the resulting quote to a trusted remote attestation service, e.g., Intel Attestation Service (IAS), which checks the platform validity and returns a signed attestation report. Enclaves can bind user data (e.g., keys) to the attestation by embedding custom data into their reports [13, 32].

**Enclave Development and Graphene-SGX.** There are at least three major paradigms to develop TEE-enabled programs. First, applications can be explicitly designed for certain TEEs by using SDKs [25], which abstract the implementation details. SDKs usually provide APIs for attestation and interactions with the untrusted OS, e.g., for sealing files to disk. Second, semi-automated approaches rely on compiler support and developer-provided source code annotations to split code and data into sensitive and non-sensitive parts. The sensitive parts are then moved inside the isolated enclave and connected to the untrusted parts via shielding layers [37, 55]. Finally, as a third approach, SGX library operating systems securely execute unmodified applications inside enclaves [2, 5, 9, 53]. Due to the user space restriction of enclaves, these LibOSes handle system calls on behalf of the apps and transparently provide POSIX abstractions, e.g., multi-threading support. As the underlying OS is untrusted, the frameworks aim to shield system calls against so-called Iago attacks [10], in which the untrusted operating system manipulates system calls and their return values. However, while LibOSes typically provide shielding layers for secure disk I/O and file integrity, they do *not* protect network traffic and rely on the untrusted host



network stack. While SCONE [2] includes transparent TLS proxy support for server apps, it fails to protect client traffic and DNS—both essential requirements of SENG.

In our design, we will follow the third approach, and use the Graphene-SGX LibOS, which is open source and allows us to transparently execute unmodified applications in SGX enclaves [9]. Graphene-SGX transparently emulates some system calls internally, while others are delegated to the untrusted OS. A manifest file specifies the enclave size and number of threads, as well as the application and corresponding dependencies that Graphene-SGX shields. The manifest is part of the enclave identity for attesting the shielded application. While Graphene-SGX provides multi-threading and a file system shield, it provides *no* secure network I/O for apps.

## 5 Design

### 5.1 Requirements

SENG’s high-level goals are twofold: (i) prevent attacks against the traffic of SGX-shielded clients, and (ii) allow a central gateway to govern network access on a per-application basis. From these, we derive six security (*SR*) and three auxiliary (*AR*) requirements of our system, as shown next. These requirements hold equally for internal and external shielded clients. Five of these requirements (SR2–SR6) heavily rely on the new concepts introduced by our design.

**SR1 Code and Data Protection** During execution, the integrity and confidentiality of client code (binary, libs) and data (including files) must be protected.

**SR2 Network Traffic Integrity and Confidentiality** The integrity and confidentiality of network traffic between shielded apps and the gateway is guaranteed, which holds true both for internal *and* external clients.

**SR3 Redirection Prevention** Traffic from shielded clients must be protected against packet header manipulation by local system-level or on-path MITM attackers until it passes the gateway. Furthermore, local and on-path DNS redirection attacks must be prevented.

**SR4 Protection-based Traffic Authentication** The gateway must be able to distinguish between traffic of shielded applications and that of non-shielded ones. This property enables network policies that restrict the access to sensitive subnetworks to shielded apps only.

**SR5 Accountability of Shielded Traffic** The gateway must be able to link shielded traffic back to the respective shielded application to enforce per-app network policies.

**SR6 Information Leakage and Remote Control Prevention** Whenever SENG enforces that only shielded clients may communicate, local system-level and internal MITM attackers must not be able to leak information to external systems. In the opposite direction, attackers must not be able to send information (e.g., malware

commands) from the outside to compromised clients.

**AR1 No Client Code Changes.** To ease adoption and to support closed-source and legacy applications, we seek for a solution that does not require any code changes in the client app and its dependencies.

**AR2 Scalability of Gateway Server** The overhead introduced to the gateway server per shielded app and per network connection must be low to allow for scaling.

**AR3 Compatibility with other Gateway Services** The protection and authentication techniques used by SENG should not interfere with other services on the network gateway, such as middleboxes or firewalls.

### 5.2 Overview

We now provide an overview of the SENG architecture and explain how SENG shields network traffic of unmodified client applications and enables app-grained traffic control.

The SENG architecture consists of two main components: (i) a client-side shielding runtime, and (ii) a SENG server located at the gateway. Figure 2 provides an overview of the SENG components and communication channels. On the client side, the SENG runtime wraps a client application in a library OS (*LibOS*) and combines both in an SGX enclave. The dedicated SENG server is located at the central network gateway. It cooperates with the firewall and the SENG runtime instances to attribute and protect traffic of the shielded apps.

On the client, the LibOS and SENG runtime transparently shield the client applications from local system-level attackers. To this end, the LibOS loads and executes unmodified binary applications inside a hardware-protected SGX enclave. The LibOS transparently handles system calls of the app and shields them against Iago attacks [10] of the untrusted OS. For instance, the LibOS prepares its own file system to protect against disk I/O tampering. The SENG runtime adds to this in that it protects network I/O of shielded apps and establishes trust with the SENG server. Technically, the SENG runtime incorporates a lightweight user space TCP/IP stack to cope with the lack of trust in the host’s network stack. This user-space network stack manages the app’s connections inside SGX and enables secure tunneling of whole IP packets—including the network and transport headers—to the SENG server.

The SENG server has to authenticate client apps and securely forward shielded traffic between SENG runtime and gateway. The SENG runtime and server establish an attested, secure communication channel to tunnel traffic. The SENG server listens for incoming tunnel connections from shielded and trusted client apps. We use SGX’s remote attestation to check the app’s identity and verify that it runs inside a valid SGX enclave with SENG runtime. To this end, the SENG runtime generates a fresh public and private key pair and binds it to the enclave report—inspired by work of Knauth et al. [32]. The SENG runtime then uses the keys to establish a mutually authenticated, end-to-end protected connection to

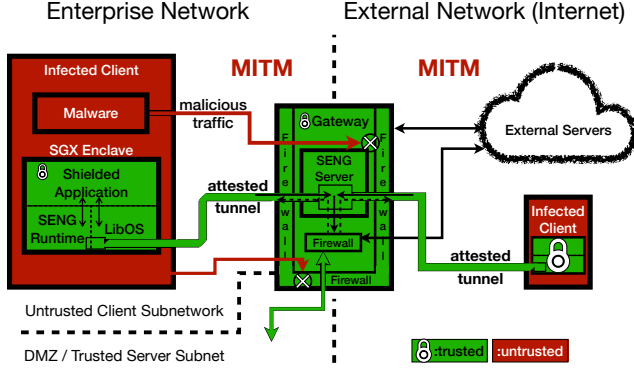


Figure 2: High-level Overview of the SENG Architecture

the SENG server and provides the attestation report during connection setup. Before accepting the connection, the SENG server checks that the attestation report is bound to the connection and belongs to a valid SGX enclave with a shielded application. After tunnel establishment, traffic of the shielded app can be securely tunneled to the SENG server and routed through the gateway (incl. firewall) while being protected from MITM attackers between enclave and gateway.

### 5.3 Application-Grained Firewall Policies

Placing the SENG server on the gateway allows for fine-grained traffic control at the perimeter firewall. With SENG, firewalls can precisely control *which shielded app may communicate where*. This adds a completely new degree of freedom that standard firewalls do not give, as they subsume all applications of a given system into a single address.

The SENG server maintains a central allowlist of trusted applications, which links apps to their trusted attestation reports, and additionally, to an app-specific IP subnetwork. The SENG server assigns a unique IP address from this particular subnet to each shielded enclave instance of a given client app. The enclave-unique addresses make the shielded app’s identifier visible to all gateway services, including firewalls. Firewalls use this mapping to define app-specific policies, which are easily integrated into existing toolchains<sup>2</sup>.

To demonstrate this, we introduce a typical corporate network setup, as shown in Figure 3. The network consists of a central, SENG-enabled gateway which interconnects an untrusted internal client subnetwork, a trusted internal server subnet, a DMZ, and external networks. The DMZ provides typical services for internal and external hosts, including a public web shop and a DNS server. The internal servers are only reachable by internal clients and host an intranet web server, as well as an LDAP and database server. The client workstations run a set of trusted client applications (e.g., browsers, mail

<sup>2</sup>Alternatively, to ease integration, we also implemented a SENG netfilter kernel module and iptables extension that allows to extend netfilter-based firewalls with SENG app identifiers to avoid network fragmentation.

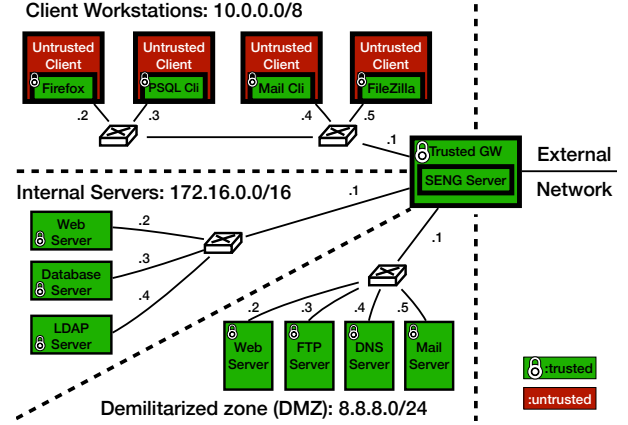


Figure 3: Sample topology of a corporate network consisting of a SENG-enabled gateway, a subnet of untrusted clients with shielded apps, an internal server subnet and a DMZ.

clients) which require access to internal and external servers. The white columns in Table 2 show traditional firewall policies (e.g., configured using iptables) for this setup. Rules 1-2 allow workstations to connect to external hosts, rules 3 and 8-10 grant them connections to internal and DMZ servers, and rules 4-7 allow external clients to connect to servers in the DMZ. Rule 11 allows internal and DMZ servers to connect to external servers. Rule 12 allows all communication of such established connections, and rule 13 is the default policy that rejects any other traffic.

If client hosts are fully compromised by a system-level attacker (cf. Section 2), these traditional policies fall short. First, they allow malware on trusted hosts to communicate to external servers. Second, they do not refine which external clients may use servers in the DMZ. To tackle these shortcomings, SENG grants only trusted apps network access. The gray column in Table 2 shows the policy modifications that SENG requires. Administrators just have to replace the coarse-grained source addresses with app-grained addresses. For example, in rule 1, the firewall can now control that only vetted Firefox clients from the workstation network can access external networks, and any untrusted software is blocked. This minor change significantly hardens the firewall setup. The SENG-enabled policies can be automatically derived when shielded apps specify which endpoints they need for communication.

**Subsumed Enclave Subnetworks.** Optionally, network admins can group shielded apps sharing policies (e.g., all mail clients, or versions of same app) into privilege-based subnets. Table 2 exemplifies both cases: While rule 3 restricts access to an individual mail client version, rule 6 subsumes all FileZilla versions in a subnet. Rule 2 even restricts access to external databases only to PSQL clients configured with SSL mode enabled to protect against external MITM attackers.

**Host IP Addresses.** We override the source IP address with an enclave-unique address to easily integrate SENG into

No.	Source (w/o SENG)	Source (with SENG)	Destination	Dst. Port	State	Action
1	\$WORKSTATIONS	\$WS_FIREFOX_72	\$EXTERNAL	80, 443	NEW	ACCEPT
2	\$WORKSTATIONS	\$WS_PSQL_TLS_ONLY	\$EXTERNAL	5432	NEW	ACCEPT
3	*	\$ANY_THUNDERBIRD_68	\$IMAP \$SMTP	143, 993 465, 587	NEW NEW	ACCEPT ACCEPT
4	\$EXTERNAL	\$EXTERNAL	\$SMTP	25	NEW	ACCEPT
5	*	*	\$DNS	53	NEW	ACCEPT
6	*	\$ANY_FILEZILLA	\$FTPS	989, 990	NEW	ACCEPT
7	*	*	\$WEBSHOP	80, 443	NEW	ACCEPT
8	\$WORKSTATIONS	\$WS_FIREFOX_72	\$INTRANET	80, 443	NEW	ACCEPT
9	\$WORKSTATIONS	\$WS_PSQL	\$DATABASE	5432	NEW	ACCEPT
10	\$WORKSTATIONS	\$WS_ENCLAVES	\$LDAP	389, 636	NEW	ACCEPT
11	\$SERVERS	\$SERVERS	\$EXTERNAL	*	NEW	ACCEPT
12	*	*	*	*	ESTABL.	ACCEPT
13	*	*	*	*	*	REJECT

Table 2: Traditional firewall policies for the corporate sample network (Fig. 3) and their app-grained SENG alternatives (gray column). The variables in column 2 and 4 represent subnets (e.g., \$WORKSTATIONS) or server IP addresses (e.g., \$IMAP). The new variables in the gray column represent SENG enclave subnets (\$WS for workstations, \$ANY for arbitrary IP addresses).

existing firewalls (AR3). Note that the SENG server can still distinguish between enclaves running on different hosts and between enclaves running on different subnets. While rule 6 grants internal *and* external FileZilla enclaves access to the FTP server (DMZ), rule 8 restricts access to intranet web pages to shielded browsers on internal workstations only.

## 5.4 Deployment of SENG

SENG raises questions regarding enclave deployment, key management and update handling, which we discuss next.

**Enclave Deployment.** The SENG runtime and its dependencies are shipped to clients as a container image. Each shielded app needs a configuration file that lists the files the LibOS has to protect, which can be (partially) automated<sup>3</sup>. App bundles can then be offered, e.g., via corporate app stores.

New SENG client devices are enrolled by including their addresses in the SENG policy database. Strong device bindings can optionally be established using orthogonal schemes such as IEEE 802.1X and strict mappings between hosts and IP addresses. Alternatively, one could bind a secret to the client CPU as part of the app installation process.

**Mixed Environments / Gradual Deployment.** SENG can also be deployed in mixed environments, i.e., heterogeneous networks where not all hosts support SGX (and thus SENG). In this case, administrators can use network segmentation to separate SGX-enabled workstations from legacy workstations. Whereas the unprotected subnetwork of legacy clients would be governed by traditional (and possibly more restrictive) firewall rules, the protected network could readily use SENG policies. In fact, given a particular workstation, this setup also

allows to gradually migrate applications to SENG. Shielded apps would belong to the protected subnetwork, whereas all other legacy clients are bound to the unprotected subnetwork.

**Key Management.** SENG requires minimal key management. The SENG server authenticates clients via remote attestation and the client key pair  $(K_{enc}, K_{enc}^{-1})$  is generated on each startup, s.t. no key rollouts are required. The key pair of the SENG server  $(K_{srv}, K_{srv}^{-1})$  must be securely managed and the public key  $K_{srv}$  is shipped to clients as part of the SENG runtime. See Section 7 for respective security considerations.

**Component Updates.** On each component update (incl. keys, app, libs, SENG and LibOS), the SENG runtime image is rebuilt, and a new attestation report is extracted and inserted into the allowlist. Thus, SENG can identify the exact software bundle of a given enclave (cf. Section 6.1) and allow, e.g., only specific app versions (Table 2, rule 1)—mitigating the risk of outdated software that exposes security vulnerabilities. While SENG provides new reports on each update, LibOSes commonly support dynamic loading [5, 9], s.t. SENG needs to reship *only the modified files*, the (small) configuration and new enclave signature.

**Critical Updates and Key Rollovers.** In case of critical security updates, the compromised reports must be removed from the allowlist to revoke network access. SENG can optionally terminate all established tunnels of such revoked apps, immediately disconnecting revoked apps from other network segments. A special case is the update of SENG’s server key pair  $(K_{srv}, K_{srv}^{-1})$  as part of a periodic or emergency key rollover. As the public key  $K_{srv}$  is pinned by each shielded app and part of their attestation, every app report changes and has to be revoked. However, note that when using a tunnel cipher with (perfect) forward secrecy, their session keys are unaffected by a server key breach  $(K_{srv}^{-1})$ . Thus, *all estab-*

<sup>3</sup>e.g., using <https://github.com/oscarlab/graphene/tree/v1.0.1/Tools>, or an automated build chain for container generation [2]

lished tunnels and associated app connections can continue operation (Table 2, rule 12).

## 6 Implementation

We now provide the details of the SENG architecture in chronological order of the shielded app’s communication. That is, we first detail the setup phase, then how the app’s network traffic is protected, and finally, how the perimeter firewall enforces app-grained communication policies.

### 6.1 Initialization and Tunnel Setup

**Initialization Phase.** Before the SENG runtime can protect a client application, the SGX enclave must be set up. SENG uses the Graphene-SGX LibOS [9], as it supports dynamic loading of unmodified, multi-threaded Linux apps and shields system calls. First, Graphene-SGX sets up an SGX enclave and initializes the shielding layers. After finishing the setup, but before loading the application, the SENG runtime loader is called and launches a dedicated enclave thread for the user space TCP/IP stack and for the tunnel module. The TCP/IP stack is instantiated with the embedded lwIP stack [38], as it is lightweight and modular by design. The tunnel module manages the tunnel to the SENG server and registers itself as network driver for the default interface of lwIP, s.t. lwIP routes all IP packets of the client app through the tunnel module.

On the gateway-side, the SENG server creates a virtual IP-level network interface which it will later use for routing traffic of shielded apps and receiving packets destined for them. Afterwards, the SENG server sets up a welcome socket and waits for incoming tunnel connections by internal or external SENG runtime instances.

**Tunnel Preparation.** After initialization, the SENG runtime generates credentials and the enclave report for the secure tunnel to the SENG server. The tunnel module uses DTLS (RFC 6347), which has well-documented end-to-end protection guarantees. We chose UDP-based DTLS over TLS as it requires less state and is faster, which improves scalability, and as the reliability and ordering guarantees of TLS are not required [20]. For tunneled TCP connections, the TCP/IP stacks of the communication endpoints—namely SENG runtime and target server—already guarantee reliable, in-order packet delivery. For tunneled UDP streams, both communication partners have to resolve packet reordering in the application protocol anyway, and the choice of DTLS thus does not weaken any security guarantees.

To couple remote attestation with the end-to-end protection of DTLS, the tunnel module generates a fresh RSA key pair  $(K_{enc}, K_{enc}^{-1})$  and binds the public key  $K_{enc}$  as user data to the enclave report—following the idea of Knauth et al. [32]. The local Intel Quoting Enclave (QE) transforms the report into a verifiable, signed quote using the attestation key. After receiving the signed remote attestation report via an attestation

service, the tunnel module uses the RSA keys  $(K_{enc}, K_{enc}^{-1})$  to generate an X.509 client certificate and embeds the attestation report with corresponding signature as extra fields.

Note that the tunnel module must not be able to directly communicate with external Attestation Services, e.g., Intel Attestation Service (IAS), to request the signed remote attestation report. Local and on-path adversaries could exploit the unprotected headers of the IAS connection as covert channel and leak information (violating SR6). To solve this dilemma, we can (i) let the enclave send the signed quote to the SENG server, which in turn performs the IAS communication itself, or (ii) operate an internal attestation service in the DMZ, and let the enclave submit the quote to the AS via TLS [50].

**Tunnel Establishment.** The SENG runtime now connects to the SENG server via a mutually authenticated DTLS connection. For server authentication, the runtime uses the pinned server public key  $K_{srv}$ . For client authentication and remote attestation, the SENG server checks the validity and signature of the attestation report and matches the embedded user data with the certificate key  $K_{enc}$ . The SENG server then verifies if the report data belongs to a shielded application in the allowlist. Technically, the enclave measurement contains the Graphene-SGX library and memory-mapped manifest:  $mrenclave \leftarrow \text{measure}_{\text{sgx}}(\text{graphene}, \text{MF})$ . The manifest MF contains secure hashes  $h(\cdot)$  for all dependencies of the SENG runtime and shielded app, including the runtime library, the pinned server key  $K_{srv}$ , the app’s binary and libraries, as well as other protected files:  $\text{MF} := \{h(\text{sengrt}), h(K_{srv}), h(\text{app}), h(\text{lib}_1), \dots\}$ . The file system shield enforces file integrity based on the hashes [9]. The inclusion of the manifest in the measurement results in a unique enclave identity ( $mrenclave$ ) for each bundle of LibOS, SENG, and client app. Therefore, the SENG server can directly link the report to the exact version of the shielded app. If the app was verified, the SENG server knows that the DTLS tunnel is attested and established with a valid SGX enclave. Finally, the SENG server looks up the app-specific IP subnet based on the app’s identity ( $mrenclave$ ) and, optionally, host IP and assigns a unique IP address from the subnet to the SENG runtime instance (cf. Section 5.3). The SENG runtime takes over the reported IP configuration, and Graphene-SGX loads the app and transfers control to it.

### 6.2 Network Traffic Shielding

**Redirecting IP Packets to the Tunnel.** SENG needs to protect the whole network traffic of shielded applications. Graphene-SGX links the client apps against a patched version of the standard C library where syscalls are replaced by calls to LibOS-internal handler functions. This allows us to fully-transparently wrap and shield system calls. The SENG runtime provides own handlers which shadow all network I/O functions, as shown in Figure 4. The SENG handlers transparently redirect all socket API functions of the client app to the respective lwIP functions, s.t. the app can perform



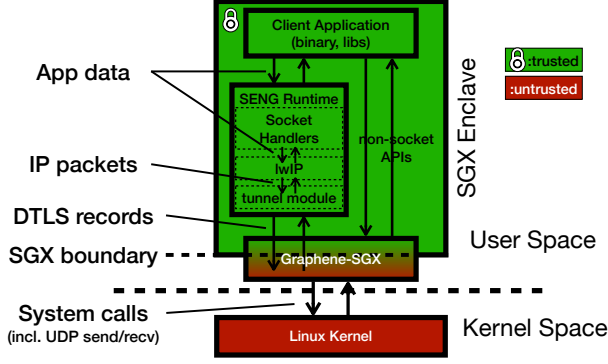


Figure 4: Overview of the SENG Runtime Components

network I/O only through the SGX-internal user space stack. lwIP manages all connections of the app and uses the tunnel module for receiving and sending the associated IP packets.

**Sending Packets.** When the shielded app sends data, lwIP crafts corresponding IP packets and passes them to the tunnel module. The tunnel module wraps the IP packets with DTLS and forwards them through the attested tunnel to the SENG server. For transferring the DTLS records, the tunnel module uses the LibOS to perform the actual UDP send operation via the untrusted OS. Figure 4 shows the app’s data flow and highlights that only the DTLS records cross the SGX boundary. The end-to-end security protection of DTLS prevents attacks by local or MITM attackers. The SENG server receives the DTLS records, decrypts contained IP packets and then passes them through the virtual network interface to the gateway network stack. The gateway then applies app-grained firewall rules (Section 6.4) and routes the packets to the target server.

**Receiving Packets.** For inbound traffic, the SENG server receives the corresponding IP packets from the gateway through the virtual network interface. The SENG server uses the target address to look up the DTLS connection to the respective shielded client app and tunnels them back. The tunnel module receives and decrypts the IP packets and puts them into the lwIP inbox queue. lwIP then processes the packets and passes the contained app data to the shielded app.

### 6.3 DNS Resolution Shielding

Without further precautions, the enclave would fully rely on the host OS to resolve domains. Local system-level attackers could thus launch severe redirection attacks and redirect traffic of shielded apps to IP addresses of their choice. To tackle this problem, SENG shields DNS lookups of client applications via three complementary actions. First, the SENG runtime redirects the respective standard library functions (e.g., `getaddrinfo`) to lwIP and configures lwIP to use a trusted DNS resolver located at the gateway or in the DMZ. The trusted resolver can then securely query internal DNS servers or contact trusted external ones via integrity-protected

DNS variants, e.g., DNSSEC, DNS over TLS (DoT) or DNS over HTTPS (DoH)<sup>4</sup>. Second, we provide trusted versions of configuration files used by third party DNS libraries for looking up information like the name server IP (“`resolv.conf`”) or protocol-specific port numbers (“`/etc/services`”). We leverage the file system shield of the LibOS to protect the integrity of the files. Third, all DNS queries sent via standard resolver functions or third party libraries eventually pass lwIP and are therefore tunneled through the protected DTLS tunnel.

### 6.4 Application-Grained Policy Enforcement

SENG enables the perimeter firewall to apply app-grained network policies whenever shielded traffic is routed through the gateway. App traffic reaches the gateway only through SENG’s virtual network interface and the SENG server forwards traffic to an app tunnel only if it matches the assigned enclave IP. Therefore, the gateway can identify outbound traffic as shielded iff received from SENG’s network interface and routes inbound traffic destined for enclave IPs to the SENG server. In the process, the firewall on the gateway enforces *app-grained policies as network policies on the app-specific enclave IP subnets* (cf. Section 5.3). To prevent impersonation attacks via IP spoofing, the SENG server drops tunneled app traffic with mismatching enclave IP and the firewall drops enclave traffic not arriving through SENG’s network interface.

### 6.5 Shielded Servers

So far, we took it for granted that all shielded apps are clients. However, SENG also supports shielded *server* apps. SENG server sockets work analogously to default server sockets. However, with SENG, the gateway can now fully control (i) if an enclave can expose server functionality, and if so, (ii) which clients are allowed to contact the enclave. Similar to client policies, server policies restrict communication to shielded clients or specific enclaves only (app-grained policies).

Once created, SENG server sockets are reachable through the gateway under the assigned enclave IPs. Recall that enclaves can either have public (globally routable) or private (RFC 1918) IP addresses. In case of public addresses, the enclave’s server socket is immediately exposed. If the enclave’s IP is private, yet should be reachable from external clients, the gateway uses destination NATing to expose the service.

## 7 Security Analysis

We now discuss how adversaries could attempt to attack SENG. Table 3 summarizes the attacks and respective defense mechanisms. We discuss why the protection from the above adversaries implies the fulfillment of the security goals of Subsection 5.1 and therefore solves the initial challenges.

<sup>4</sup>RFC 4033, RFC 8484 and RFC 7858

Target / Goal	Attack	Adversaries	Defense Mechanisms	Secure?
Shielded App	A01: Code/Data Tampering	Sys	SGX Enclave	✓
	A02: File Tampering	Sys	File System Shield	✓
	A03: LibOS Modification	Sys	Attest + Allowlist	✓
SENG's Tunneling and Access Control	A04: Fake/Custom Enclave	Sys	Attest + Pinning + Allowlist	✓
	A05: Client Impersonation	Sys, $M_{c2gw}$	Key Binding + Traffic Auth.	✓
	A06: Server Impersonation	Sys, $M_{c2gw}$	Pinning + DTLS	✓
	A07: Attacking SENNG's Keys	Sys, $M_{c2gw}$	SENG's Key Management	✓
	A08: Tunnel Tampering	Sys, $M_{c2gw}$	DTLS + Trusted TCP/IP Stack	✓
App Connections	A09: DNS Spoofing	Sys, $M_{c2gw}$ , $M_{gw2srv}$	SENG's DNS Shield	✓
	A10: Internal Conn. Tampering	Sys, $M_{c2gw}$	DTLS Tunnel + DMZ	✓
	A11: External Conn. Tampering	$M_{gw2srv}$	(Enforce Apps w/ Sec. Comm.)	(✓)
Information Leaks and Remote Control	A12: Direct Info Leak	Internal	SENG's Shielding and Policies	✓
	A13: Direct Remote Control	External	SENG's Shielding and Policies	✓
	A14: Covert Channel (Header)	Internal + External	SENG's Tunneling + DTLS	✓
	A15: Covert Channel (Timing)	Internal + External	(Adopt Time Masking)	(✓)
App Interfaces	A16: Steering Shielded Apps	Sys	(Secure I/O + Caller IDs)	(✓)
SENG's Policies	A17: Privilege Escalation	Malicious Enclave	Traffic Auth. + Policies	✓
Central Gateway	A18: Gateway Compromise	Sys <sub>gw</sub>	(TEE-protected Srv+FW+NIC)	(✓)

Table 3: Assessment of attacks on SENG and its respective countermeasures, following the attacker models defined in Section 7.

**Adversary Types.** With reference to Figure 1 (see page 3), SENG faces several types of adversaries: (i) a system-level attacker (“Sys”), which fully controls the enclave’s OS interactions (including traffic), (ii) MITM attackers in the internal or external client subnetwork (depending on the client’s location), who can fully control the traffic between the client and SENG server (“ $M_{c2gw}$ ”), (iii) MITM attackers on the path between the gateway and the server (either internal or external) (“ $M_{gw2srv}$ ”), (iv) an internal attacker inside the organization who aims to leak sensitive data (“Internal”), and finally, (v) an external attacker outside of the organization who aims to sneak data (or malware commands) into the network (“External”). We will use these attacker models to discuss how SENG protects against 18 security-critical attacks.

**A01: Code/Data Tampering (SR1).** Sys may aim to hijack the shielded app code, tamper with the runtime data or leak sensitive information like tunnel keys. The hardware-enforced protection of Intel SGX blocks all unauthenticated access to enclave memory and therefore prevents such attacks.

**A02: File Tampering (SR1).** Furthermore, the file system shield uses the manifest MF to check the integrity of the SENG runtime, pinned SENG server key  $K_{srv}$ , application binary and all its dependencies (e.g. libs, config files), such that any attempt to tamper files is detected and blocked.

**A03: LibOS Modification (SR2-4).** Patching the LibOS binary or its manifest to replace loaded files, e.g., the client app, or the pinned SENG server key  $K_{srv}$ , is possible, but results in deviating enclave identities (mrenclave). During remote attestation, the SENG server will thus refuse the tunnel, as the unknown enclave is not in the allowlist.

**A04: Fake/Custom Enclave (SR4).** An adversary could

try to establish a tunnel to the SENG server directly, or from within a custom enclave. As the SENG server expects a valid, correctly-signed attestation report, it will refuse direct connections with attacker-crafted fake reports. When the adversary contacts the SENG server from within a custom enclave, the attestation report will be valid, but not in the allowlist. Therefore, the SENG server will refuse the connection by the unknown enclave as in the previous attack (A03).

**A05: Client Impersonation (SR4+SR5).** Attackers could try to impersonate a trusted client application. First, attackers could intercept an allowlisted attestation report and embed it into their own client certificates. However, the report will not be bound to the certificate and the SENG server will detect the mismatch and deny access. Second, attackers could spoof an IP from a trusted enclave subnetwork. However, the SENG-enabled gateway can identify the non-tunneled traffic as unauthenticated and drop the packets (see Section 6.4).

**A06: Server Impersonation (SR2).** The attacker can also try to impersonate the SENG server by intercepting connection attempts. If successful, the adversary could gain access to all connections of the shielded application, including unprotected legacy traffic. However, the SENG runtime pins the valid SENG server key  $K_{srv}$  and checks it during the DTLS handshake to detect such impersonation attacks.

**A07: Attacking SENG Keys (SR2).** SENG performs secure key management to prevent multiple attacks against the tunnel security: (i) Rollback attacks against SENG’s server public key  $K_{srv}$  do not exist, as  $K_{srv}$  is not sealed to disk and is integrity protected (A02). A rollback of the whole app bundle (incl.  $K_{srv}$ , LibOS and all dependencies) results in a deprecated, blocked report (A03). (ii) If a private key of the SENG

(or attestation) server is breached, SENG blocks all vulnerable reports and thus enclaves with stolen keys (cf. Section 5.4). As DTLS supports ciphers with perfect forward secrecy, established tunnels are not affected by a breach of the SENG server key  $K_{srv}^{-1}$ . (iii) The client RSA key pair  $(K_{enc}, K_{enc}^{-1})$  is freshly generated for every new enclave instance and the *private key*  $K_{enc}^{-1}$  *never leaves* the enclave, s.t. it is protected against attackers (cf. A01).

**A08: Tunnel Tampering (SR2).** Tampering with established tunnel connections is not possible, because of the end-to-end security guarantees of DTLS. An adversary can reorder or drop tunnel packets, which is explicitly supported by DTLS. However, tunneled UDP connections do not expect reliable or in-order delivery and the endpoint network stacks still ensure reliability and ordering guarantees for TCP packets (Sec. 6.1).

**A09: DNS Spoofing (SR3).** An attacker can try to leak information by redirecting connections of shielded apps via DNS reply spoofing. SENG shields DNS traffic via multiple complementary methods as discussed in Subsection 6.3. First, spoofing the results of untrusted resolver functions is prevented by redirecting the function calls to lwIP. Second, DNS redirection to attacker-controlled nameservers via modification of system configuration files is prevented by providing versions with trusted IP addresses and port mappings. The LibOS ensures the integrity of the files via the file system shield. Third, Sys and both types of MITM attackers ( $M_{c2gw}$ ,  $M_{gw2srv}$ ) can try to attack unprotected DNS traffic directly. Direct attacks are prevented by securely tunneling DNS traffic through the DTLS tunnel to trusted, internal resolvers which follow integrity-protected DNS protocols for name resolution (e.g. DNSSEC, DoH, DoT).

**A10: Attacking Connections to Internal Servers (SR2+SR3).** Attacking the communication between shielded apps and internal servers (incl. DMZ) is not possible. The traffic is protected from Sys and  $M_{c2gw}$  attackers by SENG’s DTLS tunnels between the shielded apps and the gateway. As the internal servers are located in trusted networks, there are no  $M_{gw2srv}$  attackers between them and the trusted gateway.

**A11: Attacking Connections to External Servers (SR2+SR3).** SENG cannot protect the traffic between gateway and external servers. However, SENG enables network administrators to grant access to external networks only to shielded applications that securely establish end-to-end protected connections (e.g. Table 2, rule 2). If required, the file system shield can protect app-specific configuration files that define the security level of the shielded app. Therefore, SENG can indirectly enforce protection against  $M_{gw2srv}$  attackers.

**A12: Direct Information Leakage (SR6).** SENG enables the gateway to identify and block traffic coming from non-shielded senders, such as malware. Attackers cannot modify the behavior of shielded apps to leak information (A01–A03). They cannot get access to attested tunnel connections to authenticate malicious traffic for homecalling either (A04–A05, A07–08). Leaking non-encrypted traffic of shielded apps to

the external network or to attacker-controlled external servers via DNS- or header-based redirection attacks are prevented as well (A09–A11). As a result, adversaries can neither connect to external servers, nor encode sensitive data in shielded traffic, nor redirect internal, shielded traffic to external networks.

**A13: Direct Remote Control (SR6).** SENG enforces access control also for incoming connections, which blocks direct connections from external adversaries to internal malware. Sneaking data into the internal network by attacking external shielded clients is prevented analogously to attacks against internal apps (see A12).

**A14: Header-based Covert Channels (SR6).** Any attempts to establish a covert channel via header manipulations is prevented by SENG. Information leakage by internal attackers via tunnel header manipulation is prevented, as the SENG server strips the headers at the gateway. Remote commands that external attackers may inject by manipulating communication headers is likewise prevented, as the gateway strips the link layer headers and the SENG server securely tunnels the IP packets to the shielded applications. Therefore, adversaries cannot observe information encoded in the internal headers.

**A15: Timing-based Covert Channels (SR6).** Attackers may aim to create side channels based on packet timings (e.g., encoding information by delaying packets). While we excluded such covert channels from our threat model, SENG could adopt techniques to mask timing channels [8, 60].

**A16: Steering Shielded Programs for Info Leaks (SR6).** Attackers could try to abuse shielded applications to exfiltrate data. Consider a shielded browser. Its interactive interface lets users navigate (e.g., enter URLs). While we trust the user, a system-level attacker could intercept keyboard input and inject malicious commands into the shielded app. This way, adversaries control network traffic even of shielded apps. Non-interactive interfaces allow for similar attacks. For example, if users click on links displayed in a shielded mail client, the mail client calls a non-interactive interface to steer a browser to open the link. Attackers can intercept or use the interface to control the browsing targets and query strings. The general underlying problem is that shielded applications have to verify if their inputs stem from shielded applications.

To mitigate these attacks, we can rely on trusted I/O for interactive applications in addition to the shielded interfaces we specified in our threat model (cf. Section 2). We regard the adoption of secure I/O in the form of upcoming HW extensions [34] or dongles [17, 29] as realistic for critical business environments which already deploy HW authentication dongles. The LibOS can leverage trusted I/O to use attested, secure I/O paths between enclave and I/O devices [17, 29]. The LibOS can then verify that user input comes from a trusted device before forwarding input to the shielded app. Shielded interfaces based on local attestation, like SGX-based RPC calls [55], allow shielded apps to securely interact and thereby protect non-interactive interfaces (e.g., trustworthy path from mail client to browser). Problems still persist, how-



ever, if the caller has different (lower) app-grained privileges than the callee. To avoid the resulting confused-deputy attacks, the callee would have to forward the identifier of the caller to the SENG server—a significant research endeavor we leave open to future work.

**A17: Privilege Escalation by Backdoored or Compromised Enclaves (SR6).** We now discuss a relaxed threat model, where attackers can gain control over shielded apps, e.g., via backdoors or runtime compromises. Once compromised, attackers can send malicious traffic through the app’s attested tunnel as long as the traffic matches the app’s policies. If the policies are restrictive and allow communication to few vetted destinations only (e.g., shielded mail clients may only contact the local mail server), the resulting harm is limited. Any attempt of the compromised enclave to spoof its IP addresses, e.g., to join a more privileged subnetwork, will fail, because the SENG server detects unauthenticated traffic (A05) and restricts tunneled traffic to the assigned enclave IP (cf. Section 6.4). Perspectively, the app-grained traffic separation enables app-specific classification models for network intrusion detection systems, which further ease the detection of anomalous behavior of shielded apps upon compromise.

**A18: SENG Bypass via Gateway Compromise (SR2-3, SR4-6).** Our threat model fully trusts the central gateway, following the widely popular “bastion host” setting of network firewalls. If system-level attackers gain full control over the SENG server, firewall and network card (NIC), they obtain full access to the network traffic (breaking SR2+SR3) and can bypass the firewall (breaking SR4-6). While one could move the SENG server and firewall into user-level TEEs (e.g., SGX enclaves) to protect the decrypted enclave traffic and firewall integrity, this approach can only protect enclave-to-enclave communication (breaking SR2+SR3). Yet as system-level attackers control the hardware, they can still bypass the firewall and tamper with the communication.

To tackle this extended threat model, the gateway could rely on a system-level TEE, which is isolated from the compromised OS and can additionally claim exclusive ownership of the network card. We regard TrustZone-assisted TEE systems, e.g., OP-TEE<sup>5</sup>, a reasonable choice for the SENG gateway. TrustZone extends CPUs, memory and devices with the notion of a normal and secure mode (resp. “world”) and allows HW-enforced access control based on the current CPU mode [44]. OP-TEE runs the regular OS and apps in the normal world and a HW-isolated trusted kernel inside secure kernel mode together with trusted applications (TAs) in secure user mode. For SENG, the trusted kernel gets exclusive ownership of the NIC and includes a trusted network stack and firewall. The NIC access policy blocks direct access by normal-world system-level attackers (SR6) and enables the trusted kernel to force all network I/O through its “system calls” (complete mediation). On each network operation, the trusted kernel

can guarantee firewall enforcement on all traffic (SR4+SR5). The SENG server (including the policy database) runs as a trusted application to be isolated from the attackers and interacts directly with the trusted kernel for secure network I/O (SR2+SR3). To allow trusted policy administration, a secure bootstrapping phase can register trusted credentials (e.g., public keys) and a policy TA can commit authenticated policy update requests. Secure boot and SW- or TPM-based remote attestation can be used to further enhance trust into the gateway. We leave a full system implementation of the protected gateway open to future work and thus stay in line with the common bastion host assumption of firewalls.

## 8 Prototype Implementation

We have implemented a prototype for the SENG Runtime and SENG Server, as well as an alternative, library OS-independent runtime SDK based on Intel’s SGX SDK [25].

**SENG Client Runtime (with LibOS).** Our client-side component is written in C/C++ and consists of Graphene-SGX<sup>6</sup> [9] and our SENG runtime library. As enclave exits cause huge performance overhead [42], we use experimental support for exitless syscalls in Graphene-SGX [33]. The runtime is implemented in about 2400 lines of code<sup>7</sup> and uses lwIP 2.1.2 [38], OpenSSL 1.0.2g and an adapted version of the sgx-ra-tls attester code<sup>8</sup> [32]. We only included the IPv4 modules of lwIP to minimize the code base, and patched the definitions in the header file to be compatible with POSIX/Linux. We chose OpenSSL as it is well-known and fast. If a smaller code base is preferred over performance, we can easily replace it with lightweight alternatives like mbedTLS. For the tunnel, we use DTLS 1.2 with the ECDHE\_RSA\_WITH\_AES\_256\_GCM\_SHA384 cipher suite.

The SENG runtime is integrated as a middle layer between Graphene-SGX and the shielded app via the preloading functionality of the internal linker. The runtime exposes a socket API to the app which shadows the one of Graphene and forwards calls to lwIP. We configured Graphene-SGX and lwIP to use two distinct file descriptor ranges, s.t. we can distinguish between calls of the app and those of the tunnel module.

In our current version, the tunnel module directly communicates with the IAS and embeds the attestation report inside the X.509 client certificate. However, note that the attestation variants described in Subsection 6.1 could be easily integrated. While the tunnel module thread handles DTLS packet receipt, the lwIP thread handles the decrypted IP packets. For increased parallelization and syscall reduction, we currently use one DTLS socket per direction and replaced lwIP-internal locks with spinlocks.

<sup>6</sup>commit: 58cb88d2c187358aad428b100d1ff444173e1a2b

<sup>7</sup>according to <https://github.com/AlDanial/cloc>

<sup>8</sup>commit: 10de7cc9ff8ffaebc103617d62e47e699f2fb5ff

<sup>5</sup><https://www.op-tee.org/>



### SENG Client Runtime Without LibOS (SENG-SDK).

Our standard client runtime uses a LibOS, which adds to the client app’s complexity and overhead to ease SENG integration. In certain settings, it may be desired to deploy SENG for client apps that cannot sacrifice performance or memory overhead. We thus designed an alternative client-side runtime SDK that adds support for apps based on Intel’s SGX SDK [25]. This so-called SENG-SDK does not include a library OS, which makes it more lightweight and enables flexible integration into other frameworks [55]. Furthermore, by dropping the LibOS, the SDK trades legacy support (AR1) in for higher performance (cf. Section 9.5) and support for native SGX apps with trusted-untrusted split design.

The SENG-SDK is fully compatible with the SENG server and all SGX SDK-based toolchains. While SENG-SDK cannot remove the effort of porting apps to SGX, the toolchain integration makes porting enclaves to the SDK straightforward. Furthermore, the SDK provides a single init function which handles the whole setup (network stack, tunnels, threads) and afterwards exposes a secure POSIX-style socket and DNS API for trusted enclave code. SENG-SDK is written in about 2300 lines of C/C++ code and uses lwIP, adapted sgx-ra-tls attester code, SGX SSL<sup>9</sup> v2.2 and the SGX SDK v2.7.1. We added timeout support to condition variables of SGX SDK for lwIP, included the SSL stack into SGX SSL and added O/E-CALLs for the DTLS tunnel management. We use switchless OCALLs to accelerate the tunnel socket I/O.

**SENG Server.** Our server prototype is an event-based, single-threaded DTLS server written in C/C++ based on libuv 1.9.1 [36], OpenSSL 1.0.2g and the challenger code of sgx-ra-tls. The core functionality consists of ~1300 lines of code, and support for SENG server sockets adds ~1500 lines. The server uses a TUN device as IP-level virtual network interface to the gateway. The SENG server configures the TUN device as the default gateway for connected SENG runtime clients and links each DTLS tunnel to the client’s enclave IP address.

## 9 Evaluation

We now evaluate our prototype implementation regarding efficacy and overhead. We use iPerf3 [26] to measure the network throughput, and then show how the results transfer to real-world client (cURL, Telnet) and server (NGINX) applications. We then provide microbenchmarks to measure the setup phase of the SENG runtime. Afterwards, we revisit SENG’s NGINX performance and significantly improve it by porting NGINX to the SENG-SDK. We conclude with a discussion on the SENG server scalability under an increasing number of enclaves and according tunnels.

In our experiments, the SENG server runs on a workstation with an Intel® Core™i5-4690 CPU with 4 cores, 32 GB of

<sup>9</sup>Intel’s SGX port of OpenSSL

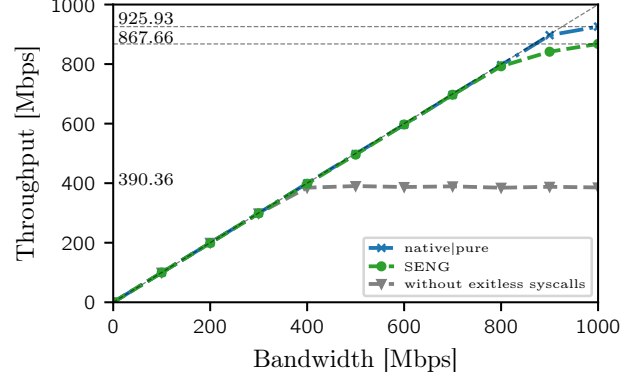


Figure 5: iPerf3 Throughput of a Single TCP Connection

memory and Debian 9 with a 4.9 Linux kernel. The SGX-enabled client system has an Intel® Core™i7-6700 CPU with 8 cores, 64 GB of memory and runs the SGX enclaves inside a Ubuntu 16.04.4 LTS docker container with a 4.15 Linux kernel. The underlying host runs Ubuntu 18.04.2 LTS. Both systems are connected to the local network via 1 Gbps NICs (Intel I217-LM/I219-LM). We route the client’s traffic via the SENG server to ensure that traffic from and to our SGX client system passes our virtual network gateway.

We take the native execution of the applications (“native”) as baseline for our evaluation and compare it with the performance of Graphene-SGX (“pure”) and of SENG (“SENG”). This way, we can attribute the overhead to either Graphene-SGX or the additional latency and overhead introduced by the SENG runtime and SENG server components.

### 9.1 Network Performance

We first report on the maximum downlink throughput of a single TCP connection using iPerf3. iPerf3 sends TCP packets to another iPerf3 instance and measures the resulting throughput. We generate the traffic on the gateway and receive traffic inside the enclave on the client system. We keep the default configuration of iPerf3 which calculates the average over 10 s and we step-wise increase the bandwidth of the work load.

Figure 5 shows the average receive throughput over five iterations. The throughputs of all three approaches scale linearly with increased iPerf3 bandwidths, and SENG shows no overhead for bandwidths up to ~800 Mbps. The native and pure Graphene-SGX setups both reach a maximum throughput of 925.93 Mbps, whereas SENG’s peak average throughput is 867.66 Mbps (~6% lower). Our 10 s measurements include TCP’s slow start, and we observed higher temporal throughputs of ~933 Mbps for native and pure, as well as ~899 Mbps for SENG, reducing the peak loss to 3–4%. The slightly lower peak throughput of SENG is caused by the additional latency added by the SENG-internal TCP/IP stack and the DTLS tunnel. We included the results of SENG with enclave exits on every syscall (~390 Mbps) to highlight that exitless designs

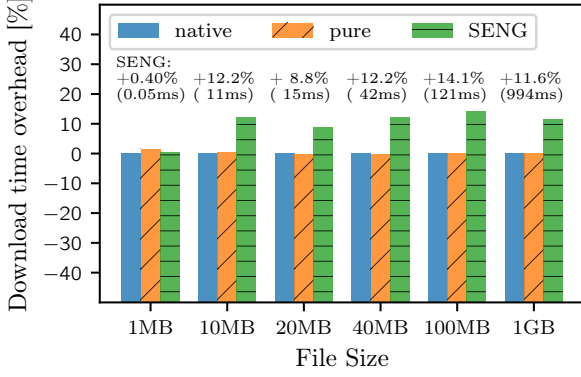


Figure 6: Time differences from cURL Benchmark

are a key-enabler for I/O-intense enclaves [2, 42].

We conclude that the reduced throughput peak (3–7%) is acceptable, especially as clients and/or remote parties are typically bound to lower bandwidths, which showed no overhead.

## 9.2 Client Applications

**cURL.** cURL is a popular tool/library to transfer data via several common protocols. In our setting, an external partner could use cURL to exchange files with internal servers. We have chosen cURL to check if SENG readily supports and scales to real-world client apps. To this end, we set up an Apache web server and measured how long cURL takes to download files via HTTP. Apache runs on the local gateway to capture the overhead with minimal impact from network jitter, analogous to iPerf3. We used the built-in measurements of cURL and took the 30 % trimmed mean over 50 iterations for each file size as a robust estimator [2].

Figure 6 shows the observed download time overhead relative to native execution. Graphene-SGX is again on par with the baseline as it shares the untrusted kernel network stack. For a file size of 1 MB, SENG shows minimal overhead due to the short download time. As the file size increases, SENG faces overhead of 8.8–14.1% which is higher than the one reported for iPerf3, but still reasonable. We observed TCP segmentation for every cURL payload, which was not present during iPerf3 and adds reassembly load and delay on lwIP as it cannot use HW offloading and has a lightweight design.

We conclude that SENG also shows reasonable performance for real-world client apps. Note that exitless syscalls in Graphene-SGX are still experimental and future versions might stabilize and further reduce the network overhead.

**Telnet.** Telnet (RFC 854) is widely used for remote terminal access and serves as our representative for remote login tools. SENG’s built-in DTLS tunnel protects plaintext Telnet against local system-level *and* on-path attackers within the organization network. Furthermore, SENG can restrict remote

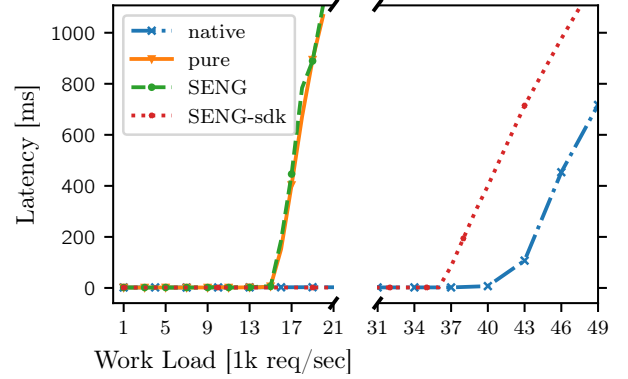


Figure 7: Average Request Latencies of NGINX

access to trusted, TLS-based login clients and shield them from local user- or system-level attackers (e.g., hooks).

We used a Telnet server on a local workstation and measured over 10 iterations the average time it takes for a Telnet client to log in, execute a set of Bash commands for entering a directory, list the contained files, and finally, display the content of a 1 kB document. Telnet takes 269.38 ms during native execution and faces 0.17 % overhead for Graphene-SGX and 0.09 % for SENG, which is practically negligible.

## 9.3 Server Application (NGINX)

We next evaluate a server setting where we aim to shield an internal server from internal MITM and system-level attackers. We chose NGINX as a demonstrator which is a wide-spread event-based HTTP server. NGINX runs on the client host inside SGX and uses a single, poll-based worker thread to serve the 612 Byte demo page via HTTP. We used the wrk2 benchmark tool from an internal workstation to issue HTTP requests under step-wise increasing request frequency. For each workload, wrk2 spawned two threads with 100 connections and calculated the mean reply latency over ten seconds.

Figure 7 shows the average latencies over five iterations. Graphene-SGX and SENG can handle ~15 k requests per second with a per-reply latency of 1.5–2.5 ms before performance degrades. Native execution clearly outperforms “pure” and SENG with ~40 k. This is no surprise and follows the observations of Tsai et. al [9], because Graphene-SGX currently only supports synchronous syscalls, which cannot effectively overlap computation and I/O. We inspected the CPU utilization of NGINX under different loads and revealed that in the “pure” and “SENG” setting, the NGINX thread saturates the CPU via continuous polling and Graphene’s I/O overhead.

In conclusion, SENG cannot yet compete with native NGINX, but is on par with Graphene-SGX while providing more security guarantees and features on top of it. Furthermore, the bottleneck can be attributed to Graphene-SGX rather than to SENG and we therefore expect better performance under future asynchronous or batched I/O support. In

Microbench	Time [ms]	StdDev [ms]
Spawn lwIP thread	38.13	$\pm 0.53$
OpenSSL init	<b>710.98</b>	$\pm 10.16$
RSA key gen (2048)	84.55	$\pm 66.25$
get SGX quote	35.67	$\pm 2.20$
get IAS report	<b>639.05</b>	$\pm 16.46$
gen X.509 Cli-Cert	1.59	$\pm 0.13$
DTLS Tunnel setup	19.86	$\pm 1.22$
Spawn Tunnel thread	42.64	$\pm 1.20$
Total SENG Runtime	1578.03	$\pm 68.12$
Without SSL Init	867.05	-
Without SSL Init + IAS	228.00	-
(a) LibOS init (default)	868.00	$\pm 12.64$
(b) LibOS init (reduced)	728.27	$\pm 8.06$
(c) LibOS init (minimal)	274.27	$\pm 1.67$

Table 4: Client Setup Times of SENG and Graphene-SGX

Section 9.5, we will revisit this claim and show that we can significantly improve the performance of NGINX by porting it to the SENG-SDK (cf. “SENG-sdk” in Figure 7).

## 9.4 Setup Microbenchmark

We now measure the initialization overhead that the SENG runtime adds to Graphene-SGX, excluding the prototype-specific socket API handlers. As the setup time of Graphene-SGX depends on the enclave configuration, we measured the time for three configurations: (a) default values of LibOS-internal tests, (b) with reduced stack, heap and thread number, and (c) with minimal accepted size.<sup>10</sup> For SENG, we measured the different setup phases of the runtime.

Table 4 breaks down the average setup times over ten iterations. The total startup overhead of the SENG runtime is 1578.03 ms, i.e. it adds about 182 % overhead on top of the Graphene-SGX initialization under default configuration. However, the vast majority of this overhead stems from two steps: (i) the init routine of the OpenSSL library (710.98 ms) and (ii) the IAS communication (639.05 ms). The high OpenSSL startup time is partially attributable to the default seeding of the random number generator. It could be reduced by switching to the RDRAND engine to approach a setup time of 867.05 ms, which is comparable to the default LibOS time (a). As discussed in Sec. 6.1, the remote attestation could be handled by an internal AS server with caching support instead. Thus, the total startup time could be further reduced to ideally 228 ms, i.e. *about 26 % of the default LibOS time (a)*.

We conclude that SENG adds a reasonable startup overhead which could be optimized to become comparable to that under reduced (b) or minimal (c) SENG runtime configurations.

<sup>10</sup>default: 256MB size, 32MB heap, 4MB stack, 4 threads; reduced: 4MB heap, 256KB stack, 2 threads; min.: 128MB size + reduced; all: 2 rpc threads

## 9.5 Accelerating NGINX using SENG-SDK

We next revisit the NGINX results of Section 9.3 and show that SENG performs significantly better when replacing Graphene-SGX with a faster primitive. SENG performed on par with “pure” Graphene-SGX for NGINX with  $\sim 15$  k requests per second, but got clearly outperformed by the native baseline of  $\sim 40$  k (cf. Figure 7). To show that SENG can overcome the bottleneck caused by Graphene-SGX, we dropped the LibOS and instead ported NGINX<sup>11</sup> to our SENG-SDK. We ported only NGINX’s platform-specific code to preserve comparability with previous results and added about 1100 lines of code for enclave setup and missing syscalls.

Figure 7 shows that SENG-SDK (“SENG-sdk”) reaches  $\sim 36$  k request per second with a per-reply latency of 1.5–2.0 ms. SENG-SDK significantly outperforms the Graphene-based SENG runtime by factor 2.4 and reaches up to 90 % of native performance. Compared to Graphene-SGX, SENG-SDK provides more efficient OCALL interfaces tailored for the DTLS tunnel I/O and benefits from the more lightweight abstractions of Intel’s SGX SDK. However, note that SENG-SDK loses legacy support and drop-in deployment (AR1).

We conclude that SENG can significantly benefit from performance improvements of the underlying primitives, letting it handle complex apps like NGINX with small overhead. Our rudimentary port to SDK-SENG achieved 90 % of native performance and could be further improved by adding NGINX-specific optimizations and an efficient file system shield. We are confident that the SENG runtime will likewise benefit from upcoming improvements of Graphene-SGX.

## 9.6 Server Scalability and Memory Overhead

We now discuss how the SENG server scales w.r.t. the number of clients and connections. The server has a small static memory footprint of which the TUN interface accounts for at most 750 kB under a full transmit queue<sup>12</sup>. The dynamic memory overhead is largely determined by the send and receive buffers of the per-enclave DTLS tunnels. In common settings, these would consume 8 KiB to 256 KiB per enclave and direction, plus about 32 KiB for the SSL frame buffer, but can be tuned to lower values. When considering the upper range, this still means that we could handle about 2000 clients per 1 GiB memory, with a huge potential for swapping large parts of the typically unused buffers. For SOCKS servers, the memory overhead increases with the number of connections they have to perform on behalf of the clients. In contrast, the SENG server is oblivious to the tunneled client connections and therefore faces constant per-client overhead.

The limiting performance bottleneck of the SENG server is the computational overhead of de- and encryption of DTLS packets and the general network I/O. In our experiments, the

<sup>11</sup>in single-process mode

<sup>12</sup>default length stores maximum 500 packets

server easily coped with any client bandwidth, and given its 1 Gbps network card we cannot test higher loads. The CPU utilization (around 65% on a single core, including waiting time) at maximum bandwidths suggests that the non-optimized server implementation will scale to 6+ Gbps on our hardware. This performance could be further optimized by improving the server code (e.g., using vectored sending, replacing the tunnel device with DPDK kernel NICs, etc.).

## 10 Discussion

We conclude with a discussion on upcoming improvements and directions to overcome limitations of our prototype.

**Overcoming Memory Limitations of Enclaves.** TEEs like SGX face two common challenges in practice: (i) performance impacts of context switches and (ii) limited secure memory. In Section 9.1 and Section 9.5, we have already presented that careful switchless designs and improvements in existing LibOS primitives (incl. upcoming ones like Occlum [53]) can significantly increase SENG’s performance for complex apps like NGINX. In the following, we focus on the memory bottleneck (ii). SGX currently limits EPC memory to 128 MB (of which around 90 MB are useable by apps) and does not support memory sharing across enclaves. Thus, running many enclaves in parallel stresses memory and triggers expensive paging. We see multiple ways to overcome this in SENG: (a) Intel CPUs now support dynamic memory management for SGX [39]<sup>13</sup> which decreases memory pressure via lazy loading and page unloading. In fact, recent studies on library debloating [46, 47] have shown that apps only use small fractions of the loaded code (incl. libraries) and tools like RAZOR [46] trim over 70% of bloated binaries. With widespread dynamic paging support, SENG can integrate compiler- and loader-based schemes into the LibOS to reduce the enclave footprint. (b) SENG could follow the idea of Panoply by splitting the SENG runtime library and other shared libraries into separate SGX enclaves that are shared by all shielded apps and used for attested RPC calls. [55] (c) Upcoming LibOSes like Occlum [53] apply HW-isolation mechanisms together with SW-based fault isolation to efficiently and securely run multiple processes in a single enclave. By integration of SENG inside Occlum rather than Graphene-SGX, multiple shielded apps with same privileges could directly share common libraries inside SGX. While the memory bottleneck of SGX right now indeed poses a major challenge to LibOSes and SENG, we conclude there are several mid-term and long-term directions for improving the number of concurrent apps.

**Frequent Measurement Updates.** Any change to an app will cause a change to the enclave report and identity, too. While alternative designs limit the number of updates by including only a loader inside the measurement [5], we highlight

<sup>13</sup><https://github.com/ayeks/SGX-hardware#hardware-with-sgx2-support>

that our choice roots the app identity directly in the HW. We thus can directly specify app-grained policies on the exact app identity and do not need additional, potentially vulnerable, SW-based authentication schemes. As discussed in Section 5.4, we also regard integration of measurement updates into today’s continuous build chains as practical and have shown in Section 5.3 that SENG is flexible enough to group multiple app versions into shared enclave IP subnetworks. A future direction might include exploration of shared “library enclaves” (“micron” in Panoply [55]) to compartmentalize enclaves while keeping HW-based identification.

**Other TEEs and Improvements.** While our current design uses SGX, it relies on common properties of other TEEs, namely trusted execution and remote attestation. Therefore, we can likely transfer SENG to other TEEs [6, 30]. We chose SGX, as it is widely available on commodity systems, and poses challenges due to its restriction to user space code.

**Prototype Limitations.** Our current prototype does not support all system calls yet. We miss `fork` and `exec` in particular, which could be extended like in other LibOSes [9, 55]. Furthermore, we have not yet integrated a database.

## 11 Conclusion

Network administrators have lost control over which client apps communicate in their sensitive networks. Not being able to centrally, *precisely and reliably* govern network accesses regularly results in data exfiltration by malware or exploitation attempts against vulnerable client software. Unfortunately, existing attempts to prevent such incidents (anti-virus, malware sandboxes, IDS, etc.) are susceptible to evasion. SENG’s ability to specify app-grained policies enables for fine-grained and *application-aware* traffic control concepts. Moreover, SENG provides strong security guarantees that are rooted in hardware and even withstand system-level attackers. SENG thus fills a need that has existed since the introduction of firewalls: per-app attribution of network traffic.

## 12 Artifacts

The prototype of SENG is available as an open source project at <https://github.com/sengsgx/sengsgx>.

## Acknowledgments

We thank our anonymous paper and artifact reviewers and our shepherd Adrian Perrig for their valuable feedback. Also, we thank Cas Cremers for his feedback on the initial SENG design, and Giorgi Maisuradze for his paper draft review.



## References

- [1] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVATE: A data oblivious filesystem for intel SGX. In *Network and Distributed System Security Symposium (NDSS)*, 2018.
- [2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [3] D. E. Asoni, T. Sasaki, and A. Perrig. Alcatraz: Data Exfiltration-Resilient Corporate Network Architecture. In *International Conference on Collaboration and Internet Computing (CIC)*, 2018.
- [4] T. Barabosch and E. Gerhards-Padilla. Host-based code injection attacks: A popular technique used by malware. *Proceedings of IEEE International Conference on Malicious and Unwanted Software (MALCON)*, 2014.
- [5] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI’14)*.
- [6] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Network and Distributed System Security Symposium (NDSS)*, 2019.
- [7] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Middleware Conference (Middleware)*, 2016.
- [8] S. Cabuk, C. E. Brodley, and C. Shields. IP Covert Timing Channels: Design and Detection. In *Conference on Computer and Communications Security (CCS)*, 2004.
- [9] C. che Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.
- [10] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
- [11] Cisco. NVM. [https://www.cisco.com/c/dam/global/en\\_au/assets/pdf/anyconnect-network-visibility.pdf](https://www.cisco.com/c/dam/global/en_au/assets/pdf/anyconnect-network-visibility.pdf).
- [12] The CLIP OS Project, 2020. <https://clip-os.org/en/>.
- [13] V. Costan and S. Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.
- [14] Docker networking. <https://docs.docker.com/network/>.
- [15] J. A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In *Network and Distributed System Security Symposium (NDSS)*, 2017.
- [16] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren. LightBox: Full-Stack Protected Stateful Middlebox at Lightning Speed. In *Conference on Computer and Communications Security (CCS)*, 2019.
- [17] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. G. Jr., E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting User Secrets from Compromised Browsers. In *IEEE Symposium on Security and Privacy (SP)*, 2019.
- [18] FireEye. M-Trends 2019. <https://content.fireeye.com/m-trends/rpt-m-trends-2019>.
- [19] FireMon’s State of the Firewall, 2019. [www.firemon.com/2019-state-of-the-firewall-report/](http://www.firemon.com/2019-state-of-the-firewall-report/).
- [20] S. Gallenmüller, D. Schöffmann, D. Scholz, F. Geyer, and G. Carle. DTLS Performance - How Expensive is Security? 2019. <https://arxiv.org/pdf/1904.11423.pdf>.
- [21] C. Gkantsidis, T. Karagiannis, D. Naylor, R. Li, and P. Steenkiste. And Then There Were More: Secure Communication for More Than Two Parties. Technical Report MSR-TR-2017-24, July 2017.
- [22] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P. Aublin, P. Cosa, C. Fetzer, P. Felber, P. Pietzuch, and R. Kapitza. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2018.
- [23] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. R. Pietzuch, and R. Kapitza. TrustJS: Trusted Client-side Execution of JavaScript. In *Workshop on Systems Security (EuroSec’17)*.
- [24] A. Houmansadr, C. Brubaker, and V. Shmatikov. The Parrot Is Dead: Observing Unobservable Network Communications. In *IEEE Symposium on Security and Privacy (SP)*, 2013.
- [25] Intel. SGX SDK. <https://software.intel.com/sgx/sdk>.
- [26] iPerf3. <https://iperf.fr/>.
- [27] iptables Application level firewalling, 2005. [debian-administration.org/article/120/Application\\_level\\_firewalling](http://debian-administration.org/article/120/Application_level_firewalling).

- [28] T. Jaeger, D. H. King, K. R. Butler, S. Hallyn, J. Latten, and X. Zhang. Leveraging IPsec for Mandatory Per-Packet Access Control. In *2006 Securecomm and Workshops*.
- [29] Y. Jang. *Building trust in the user I/O in computer systems*. PhD thesis, 2017.
- [30] Keystone Enclave, 2019. <https://keystone-enclave.org/>.
- [31] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.
- [32] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij. Integrating Remote Attestation with Transport Layer Security. *CoRR*, abs/1801.05863, 2018.
- [33] D. Kuvaiskii. Add exitless system calls (pr 405). <https://github.com/oscarlab/graphene/pull/405>.
- [34] R. Lal and P. Pappachan. An architecture methodology for secure video conferencing. *Conference on Technologies for Homeland Security (HST)*, 2013.
- [35] 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 *USENIX Security Symposium*, 2017.
- [36] libuv. <https://libuv.org/>.
- [37] J. Lind, C. Priebe, D. Muthukumaran, D. O’Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eysers, R. Kapitza, C. Fetzer, and P. Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.
- [38] lwIP. <https://savannah.nongnu.org/projects/lwip/>.
- [39] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel SGX Support for Dynamic Memory Management Inside an Enclave. In *Hardware and Architectural Support for Security and Privacy (HASP)*, 2016.
- [40] netfilter, 2019. <https://www.netfilter.org/>.
- [41] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX Annual Technical Conference (USENIX ATC)*, 2018.
- [42] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *European Conference on Computer Systems (EuroSys)*. ACM, 2017.
- [43] B. Parno, Z. Zhou, and A. Perrig. Using Trustworthy Host-based Information in the Network. In *Workshop on Scalable Trusted Computing (STC)*. ACM, 2012.
- [44] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.*, 51(6), Jan. 2019.
- [45] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.
- [46] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security Symposium*, 2019.
- [47] A. Quach, A. Prakash, and L. Yan. Debloating Software through Piece-Wise Compilation and Loading. In *USENIX Security Symposium*, 2018.
- [48] The Qubes OS Project, 2020. <https://www.qubes-os.org/>.
- [49] S. Sasy, S. Gorbunov, and C. W. Fletcher. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *Network and Distributed System Security Symposium (NDSS)*, 2018.
- [50] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski. Supporting Third Party Attestation for Intel® SGX with Intel® Data Center Attestation Primitives, 2018.
- [51] SELinux, 2019. [http://selinuxproject.org/page/NB\\_LSM](http://selinuxproject.org/page/NB_LSM).
- [52] shadowsocks. <https://shadowsocks.org/en/index.html>.
- [53] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, and Y. Xia. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020.
- [54] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium*, 2017.
- [55] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Network and Distributed System Security Symposium (NDSS)*, 2017.
- [56] Smack (LSM), 2019. <http://schaufler-ca.com/>.
- [57] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure Middleboxes Using Shielded Execution. In *Symposium on SDN Research (SOSR’18)*. ACM.
- [58] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bind-schaedler, H. Tang, and C. A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Conference on Computer and Communications Security (CCS)*, 2017.
- [59] Y. Xiao, M. Li, S. Chen, and Y. Zhang. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Conference on Computer and Communications Security (CCS)*, 2017.
- [60] J. Xing, A. Morrison, and A. Chen. NetWarden: Mitigating Network Covert Channels without Performance Loss. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.
- [61] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.