Felix Kirchengast, BSc

Secure Network Interface with SGX

Master's Thesis
to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Computer Science

submitted to
Graz University of Technology

Institute for Applied Information Processing and Communications

Advisor: Samuel Weiser
Assessor: Stefan Mangard

Graz, May 2019

AFFIDAVIT

Abstract

Edge computing is increasingly used to provide remote access to security-critical infrastructure like industrial equipment, which is placed in a separate local network. Edge computing devices act as a gateway between the local network and the Internet. Therefore, edge computing creates new security threats that have not existed previously. If an edge computing device is compromised by malware, then the security of the entire network is threatened. This raises the challenge of securing edge computing devices against remote attacks.

In this thesis, we use a *Trusted Execution Environment (TEE)* to prevent attacks via an edge computing device. More specifically, we demand that a TEE validates all network packets that are sent to the local network. This provides security at the level of individual packets. Instead of moving a large amount of code into a TEE, we make use of untrusted network stacks by validating the untrusted network stacks at run-time. With this so-called *outsource-and-verify* approach we shrink the trusted code base of our TEE. We describe security validations inside a TEE for several widely used network protocols, including TCP/IP, SNMP, S7COMM+, PROFINET DCP. We use *Intel SGX* as TEE and forward the validated packets via MACSec to the local network. We propose alternative MACSec cipher suites and identify a vulnerability in the MACSec replay protection. Finally, we address the problem of authenticating an SGX enclave towards a trusted I/O device: Firstly, we improve the SGX-USB protocol of Jang [27]. Secondly, we present a protocol that provides mutual authentication between an enclave and a trusted I/O device.

Keywords: Trusted Execution Environments, Network Protocol Validations, Outsource-and-verify, Trusted I/O, Intel SGX, Device Authentication

Kurzfassung

Edge Computing wird verstärkt verwendet um Fernzugriff auf sicherheitskritische Infrastruktur zu ermöglichen, beispielsweise industrielle Ausrüstung welche sich in einem separaten lokalen Netzwerk befindet. *Edge Rechner* fungieren als Gateway zwischen dem lokalen Netzwerk und dem Internet. Deshalb entstehen durch Edge Computing neue Sicherheitsbedrohungen, welche es früher nicht gegeben hat. Wenn ein Edge Rechner von Schadsoftware befallen ist, dann ist die Sicherheit des gesamten Netzwerks gefährdet. Deshalb ist es notwendig, Edge Rechner gegen Netzwerk-Angriffe abzusichern.

In dieser Arbeit verwenden wir ein *Trusted Execution Environments (TEE)* um Angriffe über einen Edge Rechner zu verhindern. Genauer gesagt fordern wir, dass ein TEE jedes Netzwerk-Paket validiert bevor es zum lokalen Netzwerk gesendet wird. Dadurch erreichen wir Sicherheit auf der Ebene von einzelnen Paketen. Anstatt eine große Menge an Code in ein TEE zu verschieben, validieren wir Netzwerk-Stacks welche außerhalb des TEEs laufen, um die korrekte Operation der Netzwerk-Stacks zu erzwingen. Dadurch verkleinern wir die sicherheitsrelevante Codebasis unseres TEEs. Wir beschreiben Sicherheitsvalidierungen innerhalb eines TEEs für mehrere weit verbreitete Netzwerkprotokolle (TCP/IP, SNMP, S7COMM+, PROFINET DCP). Wir verwenden *Intel SGX* als TEE und leiten die validierten Pakete mittels MACSec zum lokalen Netzwerk weiter. Wir schlagen alternative MACSec Cipher Suites vor und identifizieren eine Schwachstelle im MACSec Replay-Schutz. Abschließend behandeln wir das Problem, eine SGX Enclave in Richtung eines externen Geräts zu authentifizieren. Zunächst verbessern wir das SGX-USB Protokoll von Jang [27]. Danach präsentieren wir ein Protokoll welches eine beidseitige Authentifizierung zwischen einer SGX Enclave und einem externen Gerät bietet.

Acknowledgements

I owe several contributors to this thesis my thanks. I thank the IAIK advisors Samuel Weiser and Stefan Mangard. I also thank Sandra Dominikus from Siemens for her support. All of them provided valuable inputs for both the ideas and the implementation of this thesis.

Felix Kirchengast

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the face of Internet of Things applications and industrial communication, there is an increasing need to secure *edge computing devices*. An edge computing device acts as a gateway between a local network and the Internet. Edge computing devices facilitate remote monitoring and remote maintenance for security-critical devices. For instance, security-critical devices are placed in a production network that has been historically separate from the Internet. These devices may be unprotected or vulnerable against remote attacks. Attackers can exploit edge computing devices as entry point to a security-critical local network. An edge computing device is a permanent security threat if it is compromised by malware. Due to the fact that edge computing devices are permanently online, attackers can remotely explore the environment and then launch targeted attacks against industrial equipment.

This raises the challenge of securing edge computing devices against remote attacks. Software vulnerabilities within edge computing devices are a major entry point for remote attacks. The code size and complexity of edge computing devices limit security measures like code reviews, testing or formal verification. One method for coping with software vulnerabilities is to split an edge computing device into a small *trusted part* and a large *untrusted part*. The trusted part is executed within a *Trusted Execution Environment (TEE)*. In this work, we demand that a TEE validates all packets that are sent to the local network. Thereby, a TEE provides security at the level of individual packets. Our solution meets the following design goals:

- **(G1) Security:** The entire traffic of a network interface should run via a TEE. A malicious OS should be prevented from injecting or modifying packets.

- **(G2) Commodity Hardware:** Existing hardware should be usable since edge computing devices are often based on commodity hardware.
- **(G3) Minimal Software:** The trusted code base should be as minimal as possible.

To use a TEE for controlling the entire network traffic, one would normally run entire network stacks inside a TEE. However, doing so increases the size and complexity of a TEE. Instead, we apply an *outsource-and-verify approach* for network stacks. That is, we validate the correct operation of untrusted stacks at run-time. Outsource-and-verify shrinks the trusted code base, which is in line with goal **(G3)**.

We choose *Intel SGX* [26] as a TEE. SGX is integrated within virtually all Intel x86 CPUs since 2016, meeting goal **(G2)**. SGX supports applications with a small trusted code base, meeting goal **(G3)**. Since SGX cannot directly access any peripheral hardware, we need to establish a cryptographic channel between an *SGX enclave* and a network interface **(G1)**. To do so, we choose a *MACSec gateway* as a network interface. MACSec [2] is a protocol for authenticating and optionally encrypting Ethernet packets. We choose MACSec because of the availability of network hardware that already supports MACSec, which is in line with goal **(G2)**. We show that securing a network interface with SGX is possible with a low performance overhead.

Contributions. This work includes the following contributions:

- We propose a concept for securing a network interface with Intel SGX.
- We make use of untrusted network stacks within a TEE, by validating the untrusted stacks at run-time. We describe security validations for several protocols (TCP/IP, SNMP, S7COMM+, PROFINET DCP). We provide a proof of concept implementation.
- We analyze the MACSec protocol and propose alternative cipher suites that are more secure. We identify a vulnerability in the MACSec replay protection.
- We address the problem of authenticating an enclave towards a trusted I/O device: Firstly, we improve the SGX-USB protocol of Jang [27]. Secondly, we present a protocol that provides mutual authentication between an enclave and a trusted I/O device.

The remainder of this work is structured as follows: Chapter 2 provides background information. Chapter 3 discusses related work. Chapter 4 presents our concept for securing a network interface with SGX. Chapter 5 proposes alternative MACSec cipher suites and identifies a vulnerability in the MACSec replay protection. Chapter 6 discusses *SGX embedded remote attestation* and improves the SGX-USB protocol of Jang. Chapter 7 presents a protocol that provides mutual authentication between an SGX enclave and a trusted I/O device. Chapter 8 details how we validate untrusted network stacks, as well as security validations for several protocols. Chapter 9 includes additional implementation details. Finally, we give an evaluation in Chapter 10 and conclude in Chapter 11.

Chapter 2

Background

This chapter provides background information about TEEs, Intel SGX, Intel ME and network protocols.

2.1 Trusted Execution Environments

Software vulnerabilities are still a major issue for system security. The size and complexity of traditional systems make security paradigms like software verification infeasible. One solution for coping with this complexity is to split the software of a system into a “trusted part” and an “untrusted part”. The trusted part is executed inside an isolated environment for security-critical code. The untrusted part may run a traditional OS stack.

A *Trusted Execution Environment (TEE)* is an isolated environment that protects code and data from an untrusted host system. In particular, a TEE should protect code and data in the face of a malicious OS. Typically, a TEE provides the following security guarantees:

- Integrity of code.
- Confidentiality and integrity of data.

On top of that, confidentiality of code might be provided in order to complicate reverse engineering efforts and protect intellectual property. An overview of TEEs is given by Maene et al. [33]. There exist various architectures for implementing an isolated environment, including hardware extensions [10,19,26], hardware-software co-designs [15, 17] and pure software designs [16]. Writing applications for TEEs often involves a refactoring of the application architecture into a trusted part and an untrusted part. Typically, applications that are running within a TEE must be specifically unlocked (signed).

2.2 Intel SGX

Intel Software Guard EXtensions (SGX) [18] is a TEE that is integrated within Intel CPUs. SGX has been introduced with the Intel Skylake Architecture in 2015. In SGX terminology, the protected parts of an application are running within an *enclave*. SGX isolates an enclave from the remaining parts of an Intel system. In particular, the host OS or other enclaves cannot access the memory belonging to an enclave. Enclaves are designed to retain integrity and confidentiality in the face of a malicious OS. An advantage of SGX is the rather small trusted code base since SGX does not require to run an entire OS within an enclave.

Architecture. Each instance of an enclave requires exactly one host process, where the address space of the host process is shared with the enclave. Conceptually, this is similar to a dynamic library that is mapped within a host process. An enclave can access all user-accessible pages within its host process but not vice versa. Enclaves are running within a dedicated CPU mode on regular CPU cores. This CPU mode is an unprivileged user mode, that is, an enclave cannot use any privileged instructions. Moreover, an enclave cannot directly issue any system calls.

At the hardware side, SGX uses an instruction set extension. At the software side, the SGX Platform Software (PSW) provides a few *architectural enclaves* that are signed by Intel. Architectural enclaves perform checks and protocols that are too complex and expensive to be implemented in hardware.

ECALL/OCALL. SGX enclaves must be entered via well-defined functions that are denoted as *ECALLs*. SGX uses the *EENTER* instruction for transferring control from untrusted software to an enclave. An *OCALL* is an untrusted function that is called from an enclave. By definition, an OCALL can be only invoked from within an ECALL.

Sealing. With sealing, enclaves can securely persist sensitive data on untrusted storage. SGX uses *seal keys* for encrypting and authenticating sensitive data. An enclave retrieves seal keys via the *EGETKEY* instruction. A seal key is always bound to a specific Intel SGX CPU, i.e. it is not possible to directly migrate sealed data from one machine to another machine. To do so, one needs to unseal the data, send the data to another machine, and then seal the data again [9]. Typically, a sealed piece of data consists of the following:

- The encrypted data
- An authentication tag
- Meta information on how to retrieve the seal key for authenticating and decrypting the data

Remote Attestation. An essential feature of SGX is *remote attestation.* Remote attestation proves to an external party that a specific enclave is running on an SGX-enabled system. Upon successful remote attestation, a secure channel is established and secrets may be provisioned to an enclave. Remote attestation makes it possible to securely outsource computations to remote parties. The SGX remote attestation protocol involves the generation of a *quote* that attests the identity of an enclave. An external challenger verifies a quote, deciding whether an enclave is trust-worthy. This is done with the help of the *Intel Attestation Service (IAS).*

2.3 Intel ME

The *Intel Management Engine* (Intel ME) [44] is an autonomous subsystem that is integrated in recent Intel chipsets. Since the Intel ME is isolated from the main OS, it can be seen as a TEE. Whereas SGX enclaves run on the main CPU cores, Intel ME is a dedicated microcontroller. Intel ME has unrestricted access to main memory, can send and receive network packets and runs even if the computer is turned off.

The software that is running within Intel ME can bypass the main OS completely. In contrast, SGX enclaves are running in unprivileged user mode. Intel ME is restricted to specific software that is signed by Intel. The *Intel Active Management Technology* [30] provides remote management functionalities that are based on Intel ME.

In the context of SGX, Intel ME is used by the SGX Platform Software. More specifically, Intel ME is the basis of the *monotonic counters* that are provided by the SGX Platform Software. Monotonic counters provide a means for protecting SGX enclaves against rollback attacks.

2.4 Trusted I/O

A *trusted path* represents a secure communication path between a TEE and an external trusted I/O device. Trusted paths provide integrity and confidentiality of I/O data in the sense that the untrusted part of a system cannot reveal or modify the I/O data.

Unfortunately, SGX does not support any generic trusted path. SGX may be used with proprietary trusted paths like Intel Protected Audio Video Path (PAVP), which rely on the Intel Management Engine (ME) [44]. However, these proprietary trusted paths are not generic and not applicable to a network interface. Consequently, additional measures should be taken for securing the communication between an enclave and an external I/O device. A common method for trusted I/O architectures is to establish a cryptographic channel between a TEE and an external I/O device [21, 24, 27, 38]. In this work, we establish a cryptographic channel between an enclave and a so-called *MACSec gateway*.

2.5 Network Protocols

Ethernet [6] is a networking technology that is commonly used for local networks. The *Internet Protocol (IP)* is the principal protocol of the Internet, including the versions IPv4 [41] and IPv6 [20]. The *Address Resolution Protocol (ARP)* [39] resolves IPv4 addresses into MAC addresses in Ethernet networks. The *User Datagram Protocol (UDP)* [40] is a lightweight port layer on top of IP. In contrast, the *Transmission Control Protocol (TCP)* [42] provides reliable, stream-based connections for IP-based networks. A *TCP/IP stack* is a software stack that implements both TCP and IP. The *Simple Network Management Protocol (SNMP)* [32] is a protocol for modifying and collecting information about network devices. *PROFINET* [37] is an open protocol stack that is used for controlling and monitoring industrial equipment. *S7COMM+* [29] is a proprietary protocol stack that is used for the communication between Programmable Logical Controllers (PLCs) and engineering software. *ISO-TCP* [43] is an intermediate protocol layer that is used by S7COMM+.

Chapter 3

Related Work

Several architectures for trusted I/O have been proposed. In this chapter, we specifically discuss architectures that share similarities with this work.

3.1 Trusted I/O with Hypervisors

Trusted I/O architectures can be based on a secure hypervisor and trusted VMs, where a hypervisor has to enforce the exclusive assignment of hardware to a corresponding VM. Zhou et al. [51] showed how to construct generic trusted paths using VMs on commodity computers. SGXIO [49] is a generic trusted I/O architecture which combines trusted VMs and SGX enclaves by binding their trust domains together. Zhou et al. [52] used an untrusted USB protocol stack within a trusted VM, by validating the USB protocol stack at run-time. Our approach does not use any hypervisor or trusted VM. This has the advantage that we avoid the implementation complexity of hypervisor-based security solutions.

3.2 Trusted I/O with Cryptographic Channels

Another line of research establishes a cryptographic channel between a TEE and an external I/O device. Bumpy [38] and BitE [36] are trusted I/O architectures for securing user input. Bumpy and BitE use the Flicker TEE [35] that provides a subset of the SGX functionality for older hardware. SGX-USB [27] uses USB dongles as a cryptographic gateway between an enclave and keyboards/displays. Building on top of SGX-USB and Bumpy, Fidelius [24] is an SGX-based trusted

I/O concept for securing web applications. A core component of Fidelius is a *web enclave* that communicates with a browser extension. Our concept uses a MAC-Sec gateway instead of USB dongles. Whereas Fidelius involves user interaction with trusted displays, keyboards and notification LEDs, our concept does not involve any user interaction at all. Instead, our focus is on trusted I/O for network interfaces.

ProximiTEE [21] proposed a *distance bounding protocol* which provides a distance bounding guarantee between an SGX-enabled machine and a trusted I/O device. The regular SGX remote attestation protocol does not provide any distance bounding guarantee. Furthermore, ProximiTEE proposed a *boot-time initialization protocol* that establishes a shared secret between a trusted I/O device and an enclave. In this work, we propose a protocol that achieves a similar goal with a symmetric key.

Chapter 4

Concept for an SGX-secured Network Interface

This chapter presents our concept for an SGX-secured network interface. The main use case is to restrict a network interface to a set of well-defined actions. The high level concept is depicted in Figure 4.1. Trusted components are highlighted with a green color.

The *host system* is an SGX-enabled machine that runs an untrusted OS. We assume that the host system acts as an edge computing device. The *local network* contains potentially unsecured or vulnerable devices. The *enclave* is running within the host OS and implements application-specific functionality for accessing the local network. The untrusted host OS shall still be able to run uncritical functionality like network discovery, but all network protocols are validated or white-listed by the enclave.

Use cases include reading sensor data or status reports from the local network. Other use cases include remote maintenance tasks like the deployment of firmware updates to devices in the local network. For some protocols, it suffices to implement security validations inside the enclave while keeping application logic outside. For others protocols, the enclave itself generates application-specific data like TCP streams. An SGX-secured network interface shall be transparent from an application's perspective, e.g., by representing it as a virtual network interface within the host OS. Thereby, an SGX-secured network interface supports untrusted legacy applications that rely on legacy protocol stacks.

Since an enclave can access a network interface only via the (untrusted) host OS, we need to establish a cryptographic channel between an enclave and a network interface. In our case, a so-called *MACSec gateway* acts as a network interface.

MACSec is an industry standard protocol for authenticating and optionally encrypting Ethernet packets. The MACSec gateway can be a switch or router which is supporting MACSec. Alternatively, the MACSec gateway can be an embedded device that is integrated within the host system. Regardless of how the MACSec gateway is implemented in hardware, the prerequisite is to establish a cryptographic channel between the MACSec gateway and the enclave.

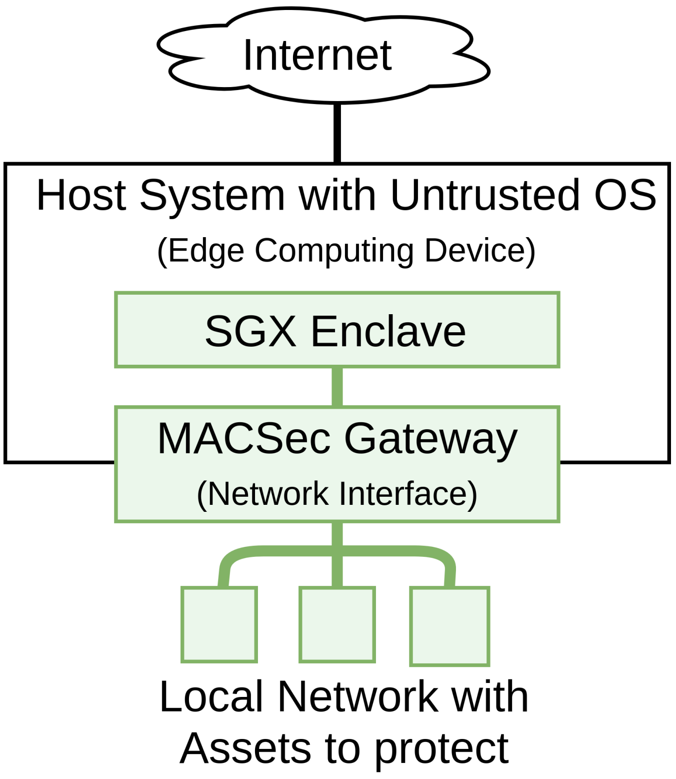


Figure 4.1: High level concept for an SGX-secured network interface.

4.1 Packet Flow

This section describes the overall packet flow, independent of the protocol and use case. Incoming and outgoing packets are not treated symmetrically.

Outgoing Packet Flow. *Outgoing packets* are directed from the host system to the local network, traversing the enclave and the MACSec gateway. The host OS passes an outgoing packet to the enclave. The enclave executes a sequence of protocol-specific validations. If these protocol-specific validations succeed, then the enclave transforms the packet into a cryptographically authenticated MACSec packet. The MACSec gateway verifies the authenticity of an outgoing MACSec packet. Then the MACSec gateway transforms the MACSec packet back into its original packet representation and forwards the packet to the local network.

Incoming Packet Flow. *Incoming packets* are directed from the local network to the host system, traversing the MACSec gateway. The MACSec gateway transforms incoming packets into MACSec packets and forwards them to the host OS. The host OS may or may not forward incoming MACSec packets to the enclave. However, the enclave only accepts incoming MACSec packets whose authenticity can be verified. If the optional packet encryption is enabled, then the host OS has to forward all incoming MACSec packets to the enclave in order to perform any meaningful action. The enclave then decrypts incoming MACSec packets and keeps the payload hidden from the host OS.

4.2 Threat Model

The asset to protect is a local network with potentially unsecured or vulnerable devices. At the host side, we follow a standard threat model of SGX: We consider the host OS as distrusted and compromised. Consequently, all outgoing packets must be either directly generated or vetted by the enclave. We consider the packets that are originating from the local network as trusted. The enclave should be able to distinguish between packets from the local network and packets that are injected by a malicious OS.

We consider two different scenarios: Either OS-generated traffic is vetted by the enclave, or the enclave itself generates network traffic. The security of the packet validation is protocol and application dependent. We require an initial trust phase for the initial installation of MACSec keys, similar to ProximiTEE [21]. Side-channel attacks are an orthogonal problem that we do not address in this work [45]. Physical attacks are out of scope since an attacker can physically bypass the MACSec gateway in any case. Denial-of-service attacks are out of scope for the host, but we do protect the local network against denial-of-service attacks.

4.3 Authentication and Key Exchange

Our concept can be implemented either with *one-way authentication* or with *mutual authentication*. With one-way authentication, we refer to the enclave authenticating itself towards the MACSec gateway, but not vice versa. If mutual authentication is used, then both the MACSec gateway and the enclave authenticate each other. In this section, we discuss both authentication options with respect to key exchange protocols and security implications.

4.3.1 One-way Authentication

In our context, one-way authentication means that only the enclave is authenticated, whereas the MACSec gateway is not authenticated. Therefore, a malicious OS can simulate a faked MACSec gateway. Nevertheless, one-way authentication is still sufficient for protecting a local network against malicious packets. For example, this ensures that remote management functionalities can be only used via an authenticated enclave.

Key Exchange. Authenticating an enclave towards a MACSec gateway can be done either with SGX remote attestation or with pre-shared keys. SGX remote attestation drops the initial trust assumption, but it is unable to achieve the expected security. More specifically, SGX remote attestation is unable to identify the physical machine that a MACSec gateway is connected to. Therefore, a malicious OS can redirect the SGX remote attestation protocol to an attacker-controlled platform. We discuss SGX remote attestation in Chapter 6 and pre-shared keys in Chapter 7.

4.3.2 Mutual Authentication

Mutual authentication provides both authenticity and confidentiality for both outgoing and incoming MACSec packets. With mutual authentication, an enclave can verify whether incoming MACSec packets are indeed originating from the associated MACSec gateway. This supports the retrieval of authenticated status information from the local network, for example.

Key Exchange. In general, mutual authentication requires more effort to implement than one-way authentication. One possibility for implementing mutual authentication is to rely on asymmetric cryptography and deploy certificates to a MACSec gateway. However, we consider a pre-shared (symmetric) key setting instead. In Chapter 7, we present a protocol that provides a symmetric mutual authentication between a MACSec gateway and an enclave. As with one-way authentication, we consider SGX remote attestation as insufficient for authenticating an enclave towards the MACSec gateway.

4.4 Outsource-and-verify

A major goal of our concept is to minimize the trusted code base. *Outsource-and-verify* is an approach where a TEE relies on untrusted code for its operation, and the correct operation of the untrusted code is verified at run-time. If the verification code is significantly smaller than the outsourced code, then outsource-and-verify shrinks the trusted code base.

In Chapter 8, we describe an outsource-and-verify approach for an untrusted TCP/IP/ARP stack. We demonstrate that TCP/IP stacks can be validated with fewer than 500 lines of C-code. The result is a *trusted socket API* that can be used by application-specific enclave code. This approach involves a few wrapper functions around (untrusted) socket system calls, as well as a packet validation interface for both incoming and outgoing packets. Afterwards, we describe TEE security validations for SNMP and PROFINET DCP. Finally, we describe an outsource-and-verify approach for the S7COMM+ protocol stack.

Chapter 5

MACSec

MACSec [2] is a protocol for authenticating and optionally encrypting Ethernet packets. In this chapter, we conduct a security analysis of MACSec that is specifically targeted for an SGX-like setting. We discuss the risk of *nonce reuse attacks* against MACSec. Nonce reuse attacks are especially relevant in an SGX-like setting, where a malicious OS can influence the generation of outgoing MACSec packets. We propose alternative cipher suites that are not only suitable for an SGX-setting, but also more secure in general. Finally, we identify an integer overflow attack that breaks the MACSec replay protection.

5.1 Background

This section describes the fields of a MACSec packet as outlined in Table 5.1.

Table 5.1: Description of a MACSec packet.

Length in bytes	**Field**	**Comment**
6	Destination MAC	Part of Ethernet header
6	Source MAC	Part of Ethernet header
2	Ethertype	= 0x88 0xE5, indicates the MACSec protocol
1	TCI-AN	*TAG Control Information* and *Association Number*
1	Short Length	Ignored for our purposes
4	Packet Number	Used for replay protection and for the ICV computation
8	SCI	*Secure Channel Identifier*
variable, at least 2	Payload	Payload of the original packet, including the original Ethertype, optionally encrypted
16	ICV	*Integrity Check Value*, authenticates the entire packet

TCI-AN. The tag control information (TCI) is a 6-bit field that includes various MACSec flags. The association number (AN) is a 2-bit field that selects one out of four MACSec keys that may exist simultaneously.

Packet Number. The packet number is a unidirectional 4-byte-counter that is incremented for each sent packet. The purpose of the packet number is two-fold: Firstly, the packet number enables an optionally configurable replay protection. Secondly, the packet number is an input for the computation of the Integrity Check Value (ICV).

Secure Channel Identifier. The SCI is an 8-byte-identifier that uniquely identifies a unidirectional secure channel within a network.

Integrity Check Value. The ICV authenticates an entire MACSec packet. A cipher suite specifies how the ICV is computed. The mandatory default cipher suite is based on GCM-AES-128 (Galois Counter Mode - Advanced Encryption Standard - 128-bit key size). Listing 5.1 illustrates the default cipher suite:

```
NONCE      := (SCI || Packet Number)
KEY        := Secure Association Key
ASS.DATA   := MACSec packet without trailing ICV
PLAINTEXT  := null
ICV        := AES-GCM(NONCE, KEY, ASS.DATA, PLAINTEXT)
```

Listing 5.1: MACSec default cipher suite (authentication-only).

Secure Association Key. A MACSec key is denoted as *secure association key.* Using MACSec for bidirectional communication requires at least two secure association keys (one of them for each direction).

5.2 Standard Amendments

The original MACSec standard from 2006 has been amended multiple times. Those amendments culminated into a new MACSec revision [8].

2011 - GCM-AES-256. This amendment includes 256-bit cipher suites as an alternative option to GCM-AES-128 [3].

2013 - Extended Packet Numbering. This amendment provides 64-bit packet numbers [4]. However, only the least significant 32 bits of the packet number are explicitly encoded in MACSec packets. The receiver must recover the 32 most significant bits of the packet number by using an algorithm that is specified in the standard amendment.

2017 - Ethernet Data Encryption Devices. This amendment specifies *Ethernet Data Encryption Devices (EDE)* [7]. An EDE is a packet forwarding device that has exactly two physical interfaces. One interface receives and transmits unprotected packets, whereas the packets on the other interface are protected by MACSec. An EDE is captured by our notion of a MACSec gateway.

5.3 Nonce Reuse Attacks

This section discusses nonce reuse attacks against MACSec, threatening key recovery attacks and the forgery of MACSec packets. We propose alternative cipher suites that are resilient against nonce reuse attacks. This is especially relevant if MACSec is used in an SGX-like setting. Finally, we discuss how nonce reuses can be prevented with session-based MACSec keys.

5.3.1 Susceptibility to Nonce Reuses

We define a nonce reuse as a set of two different MACSec packets whose ICV is computed with the same pair of `(KEY, NONCE)`. For a given MACSec packet, the nonce is determined by the SCI and by the packet number.

The most obvious possibility for a nonce reuse is an overflow of the 4-byte packet number at the side of the MACSec sender. Therefore, MACSec requires the sender to use a new key after at most $(2^{32}-1)$ sent packets. MACSec does not specify how keys should be switched. This leaves the implementation of key switching open to the vendor or operator of a MACSec device. If key switching is not properly implemented or configured, then an overflow of the packet number may occur.

Apart from packet number overflows, nonce reuses can also happen in the case of other misconfigurations: For instance, there could exist multiple associations with the same key and the same SCI. Especially in SGX setups, there is an increased risk of nonce reuse attacks: A malicious OS may rollback an enclave to a previous configuration, triggering a nonce reuse.

In case of the mandatory default cipher suite GCM-AES-128, a nonce reuse is particularly catastrophic. An attacker can extract the so-called *authentication key* with only a single nonce reuse [28]. The authentication key is derived from the actual AES-GCM key. The authentication key cannot be used to decrypt existing messages but to authenticate arbitrary malicious data [13].

5.3.2 Proposal for Alternative Cipher Suites

We propose one of the following cipher suites in order to fix the nonce reuse problem.

Authentication-only. If encryption is not required, then we propose the cipher suite in Listing 5.2. This cipher suite is based on AES-CMAC instead of AES-GCM.

```
NONCE     := (SCI || Packet Number)
KEY       := Secure Association Key
ASS.DATA  := MACSec packet without trailing ICV
ICV       := AES-CMAC(KEY, NONCE || ASS.DATA)
```

Listing 5.2: Proposal for an authentication-only cipher suite

Authentication + Encryption. If encryption is required, then we propose to use either a nonce misuse-resistant authenticated encryption scheme or random nonces that are sufficiently large.

We propose to use a cipher suite that is based on AES Synthetic Initialization Vector (AES-SIV) [25]. If a "nonce reuse" happens with AES-SIV, then an attacker may only tell whether the same plaintext has been reused or not. In contrast to the default cipher suite, however, a key recovery is not possible.

Alternatively, one could change the default cipher suite by appending random nonces to each packet. The length of random nonces should be 256 bits to reach a security level of 128 bits. 256-bit-nonces render *birthday attacks* infeasible (i.e. a malicious OS cannot trigger random nonce collisions within any conceivable running time). This, however, would increase the space overhead of MACSec packets.

5.3.3 Session-based MACSec Keys

In our setup, a malicious OS can rollback an enclave to a previous state. Therefore, it is not easily possible to maintain a persistent state for preventing nonce reuses, e.g. maintaining a persistent packet counter. The *monotonic counters* that are provided by the SGX Platform Software are too slow. According to ROTE [34], incrementing a monotonic counter takes 80-250 milliseconds. This is far too slow for sending network packets.

Instead of relying on a persistent state, it is possible to prevent nonce reuses with a session-based protocol where a new session key gets established after each enclave launch. One possibility is to use *MACSec Key Agreement (MKA)* [5]. For instance, an enclave can request a fresh key from an MKA server after each enclave launch.

5.4 Replay Protection

Whereas nonce reuse attacks should be prevented by a MACSec sender, replay protection is mainly a concern of a MACSec receiver. A replay attack injects previously captured MACSec packets into an unexpected context. The practical impact of a replay attack is highly dependent on the application. In this section, we first describe the replay protection that is standardized by MACSec. Then we discuss how an effective replay protection can be achieved in an SGX-like setting. Finally, we identify an integer overflow attack against the MACSec replay protection.

A particularly devastating replay attack is possible against the TCP/IP stack validation that we describe in this work. With this attack, a malicious OS can craft arbitrary malicious TCP payloads. The attack works as follows: Given a fixed victim device in a local network, the malicious OS establishes and closes multiple benign TCP connections to the victim device. In the context of those benign TCP streams, the malicious OS assembles a set of TCP packets with only one byte payload and a carefully chosen TCP sequence number. This is possible since a malicious OS can choose both the initial sequence number and the packet boundaries of benign TCP connections. The malicious OS captures the MACSec packets that contain the previously crafted TCP packets with only one byte payload. By replaying those MACSec packets in the right order, a malicious OS can send a chosen payload to the victim device.

Due to this strong attacker model of SGX, we consider replay protection as an essential ingredient of an SGX-secured interface. Even for environments where replay attacks are less likely to be exploited, we still consider replay protection as sensible.

5.4.1 Replay Protection Window

MACSec configures the size of a so-called *replay protection window*. All packets whose packet number is below the replay protection window are dropped. A non-zero replay protection window copes with situations where packets are reordered by the network. Algorithm 1 illustrates the replay protection that is specified by MACSec. `next_pn` represents the packet number that is expected for the next incoming MACSec packet.

Algorithm 1 MACSec replay protection

```
    Input: uint32 $pn \leftarrow$ packet number of incoming MACSec packet
    Input: uint32 $window\_size \leftarrow$ window size configuration
    Global state: uint32 $next\_pn \leftarrow 1$
1: if $pn + window\_size >= next\_pn$ then
2:     if $pn >= next\_pn$ then
3:         $next\_pn \leftarrow pn + 1$
4:     Accept incoming MACSec packet
5: else
6:     Drop incoming MACSec packet
```

Only a replay protection window of zero provides an effective protection against replay attacks. If the replay protection window is configured to any value larger than zero, then it is already possible for an online attacker to instantly replay any freshly sent packets. Configuring the replay protection window to zero boils down to the simplified replay protection in Algorithm 2.

Algorithm 2 MACSec replay protection with zero window size

```
    Input: uint32 $pn \leftarrow$ packet number of incoming MACSec packet
    Global state: uint32 $next\_pn \leftarrow 1$
1: if $pn >= next\_pn$ then
2:     $next\_pn \leftarrow pn + 1$
3:     Accept incoming MACSec packet
4: else
5:     Drop incoming MACSec packet
```

Replay Protection for SGX. As already argued in Section 5.3.3, an enclave cannot easily rely on a persistent state for the replay protection, e.g. the persistent variable `next_pn` that is specified by Algorithm 2. However, replay protection for incoming packets can be achieved by establishing a new session key after each enclave launch. Thus, replay attacks across different enclave launches can be prevented.

5.4.2 Integer Overflow Attack

We identify an *integer overflow attack* that breaks the replay protection of MACSec. The integer overflow of `next_pn` can be easily spotted in Algorithm 2 in line 2. The attack requires to capture a MACSec packet with the maximum packet number (0xFFFFFFFF). An attacker that has captured such a MACSec packet can trigger an integer overflow of `next_pn`, which resets `next_pn` to zero. Resetting `next_pn` to zero allows an attacker to replay a previously captured MACSec packet since any packet number gets accepted. Moreover, an attacker can trigger multiple subsequent overflows of `next_pn`.

From an isolated viewpoint, this integer overflow is not necessarily a vulnerability since the MACSec standard requires the sender to switch to a new key once the maximum packet number is reached. Normally, one would expect that switching to a new key prevents replay attacks. However, due to subtle details in the way that MACSec keys are switched, even a system that fully conforms to the MACSec standard can be vulnerable to this integer overflow attack. The reason is that the MACSec standard facilitates the simultaneous existence of multiple keys. By using at least two keys simultaneously, a new key can be deployed to a production environment while the old key is still active. A MACSec receiver is not guaranteed to withdraw the old key within a defined time frame. Therefore, the integer overflow can be exploited during the time frame where the old key is still active at the side of the MACSec receiver. Depending on the implementation, this time frame might be indefinitely long. We confirmed experimentally that the attack works against the Linux-MACSec implementation in kernel version 4.15. This is not a bug of the Linux-MACSec implementation since the update of `next_pn` is specified by Algorithm 1.

Remedies. The attack can be prevented either at the sender side or at the receiver side. The sender can switch to a new key before the packet number would reach the maximum value, e.g. the sender can switch to a new key once the packet number reaches the value `(0xFFFFFFFF - 1)`. The receiver can explicitly check for an overflow of `next_pn` and invalidate the associated key immediately if an overflow would happen. If a new key is not distributed well ahead of time, then either solution breaks availability if there is no strict synchronization between sender and receiver.

Chapter 6

SGX Embedded Remote Attestation

As mentioned in Chapter 4, SGX remote attestation is one of the methods for authenticating an enclave towards a trusted I/O device. A trusted I/O setting is significantly different from the traditional remote attestation setting where a *client enclave* attests its identity to a *service provider* via the Internet. Instead, a trusted I/O device is either close to or integrated within the computer where the enclave is running. Therefore, we introduce the term *embedded remote attestation*. Embedded remote attestation is the process of establishing a cryptographic channel between an enclave and a trusted I/O device, where the enclave proves its identity to the trusted I/O device.

However, SGX remote attestation is unable to identify the physical machine that a trusted I/O device is connected to, which is a major security issue in an embedded remote attestation setting. Furthermore, SGX remote attestation involves interaction with the Intel Attestation Service (IAS), which raises reliability concerns.

In this chapter, we discuss the following protocols for embedded remote attestation:

- SGX-USB from Jang [27]. We reveal that SGX-USB is susceptible to a replay attack.
- SGX remote attestation without a service provider, preventing the replay attack against SGX-USB.
- SGX remote attestation with a service provider acting as a trusted third party, also preventing the replay attack against SGX-USB.

6.1 Regular SGX Remote Attestation Protocol

Before going into modified *embedded remote attestation protocols*, we describe the regular SGX remote attestation protocol.

As a prerequisite for SGX remote attestation, an *attestation key* needs to be provisioned to an SGX-enabled device. Attestation keys are used for signing so-called *quotes*. A quote attests the identity of an enclave. Attestation keys are provisioned by Intel servers using the *EPID join protocol* [14]. To successfully complete the EPID join protocol, an SGX-enabled device must prove the possession of a valid *provisioning key*. Provisioning keys are burned into CPUs at manufacturing time.

Once the attestation key is provisioned, the regular SGX remote attestation protocol consists of the following steps as outlined in Figure 6.1:

1. The client enclave sends `msg0` and `msg1` to the service provider, including a public Diffie-Hellman parameter `g_a`.
2. The service provider retrieves the signature revocation list `SigRL` from the Intel Attestation Service (IAS).
3. The service provider sends `msg2` to the client enclave, including a public Diffie-Hellman parameter `g_b`. Service provider and client enclave establish a shared secret `g_a_b`.
4. The client enclave generates a quote via the quoting enclave. The quote attests the identity of the client enclave. The quote is signed with the attestation key. `SigRL` is used as input for the quote computation in order to compute a *non-revocation proof*. The client enclave sends the quote to the service provider within `msg3`. `msg3` is authenticated via the shared secret `g_a_b`.
5. The service provider sends the quote to the IAS. The IAS returns a *quote report* that is signed by Intel. A quote report attests whether a quote has been generated by a genuine Intel SGX system. Furthermore, the IAS verifies whether the most recent version of `SigRL` has been used during the generation of the quote.
6. The service provider inspects and verifies both the quote report and the quote itself. The quote report is verified with a public key that is provided by Intel. The quote itself is verified with application-specific policies. Typically, the service provider verifies whether the enclave has been signed by a specific vendor. If the service provider decides to accept both the quote and the quote report, then `msg4` may contain secrets that are provisioned to the

client enclave. Otherwise, `msg4` should contain an error message. In contrast to the other messages, the format of `msg4` is not specified. `msg4` may be encrypted via the shared secret `g_a_b`.

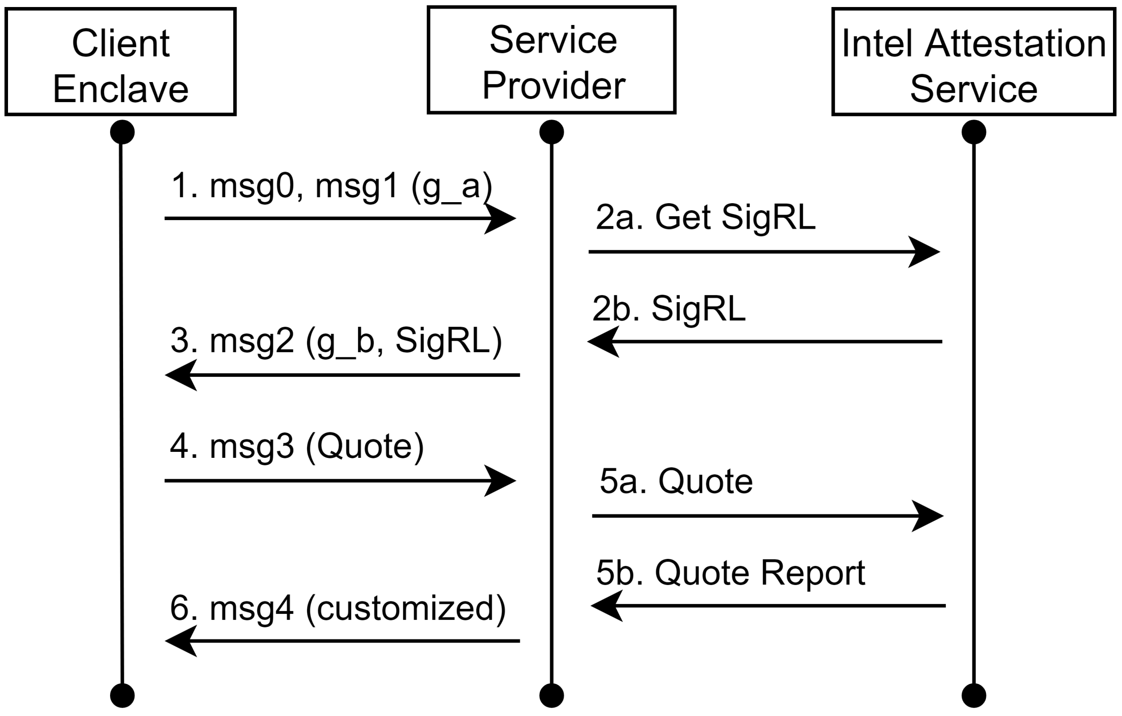
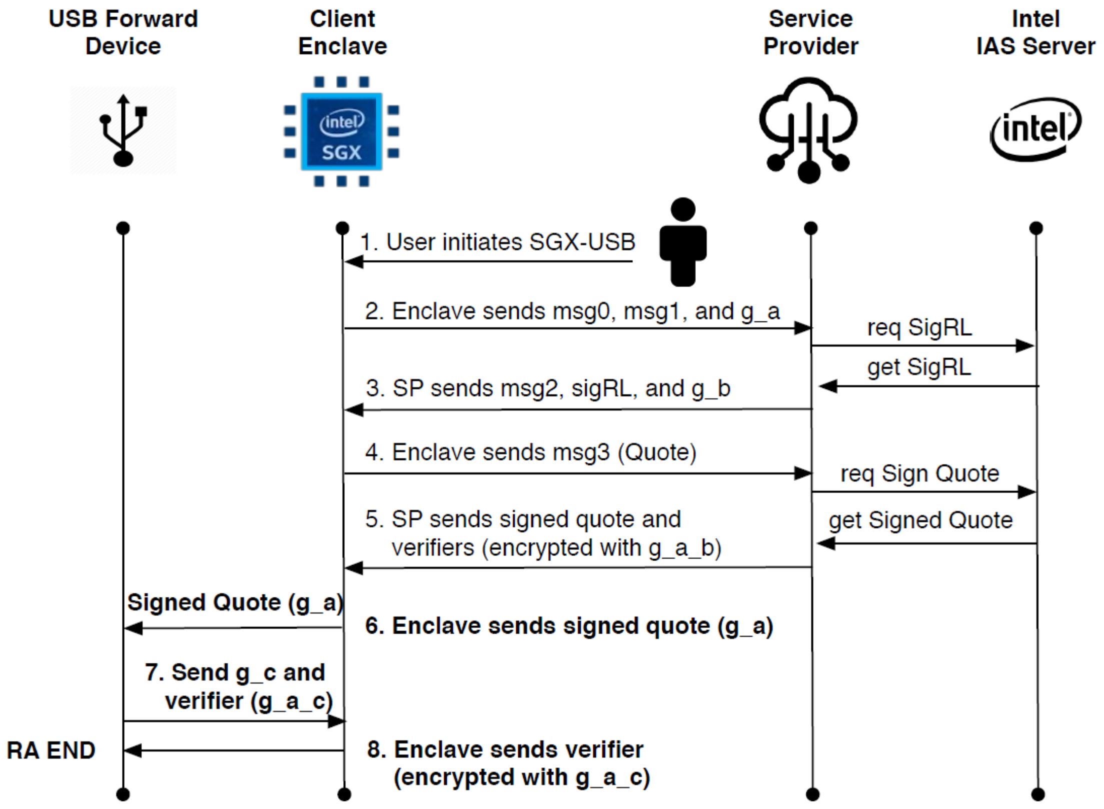


Figure 6.1: Regular SGX remote attestation protocol.

6.2 SGX-USB from Jang

To our knowledge, *SGX-USB* from Jang [27] is the first *embedded remote attestation protocol* for SGX. SGX-USB establishes a shared secret between an enclave and a *USB forward device*. The USB forward device is captured by our more general notion of a trusted I/O device. Figure 6.2 illustrates SGX-USB from Jang.



USB Forward Device
Client Enclave
Service Provider
Intel IAS Server
1. User initiates SGX-USB
2. Enclave sends msg0, msg1, and g_a
req SigRL
get SigRL
3. SP sends msg2, sigRL, and g_b
4. Enclave sends msg3 (Quote)
req Sign Quote
get Signed Quote
5. SP sends signed quote and verifiers (encrypted with g_a_b)
Signed Quote (g_a)
6. Enclave sends signed quote (g_a)
7. Send g_c and verifier (g_a_c)
RA END
8. Enclave sends verifier (encrypted with g_a_c)

Figure 6.2: SGX-USB from Jang [27], with permission obtained from the author.

SGX-USB is a slight modification of the regular remote attestation protocol in Figure 6.1. Until step 4, the protocols are exactly the same. Starting from step 5, SGX-USB deviates from the regular remote attestation protocol. SGX-USB denotes an IAS quote report as *signed quote*. SGX-USB uses an IAS quote report for authenticating the client enclave towards the USB forward device. The USB forward device verifies the IAS quote report and establishes a shared secret `g_a_c` with the client enclave.

Inconsistency. Step 6 is not consistent with the way SGX remote attestation works. SGX-USB requires to inform the USB forward device of a valid `g_a`. However, a quote cannot hold `g_a` since a quote only contains 64 bytes of *report data* that can be freely chosen by an enclave developer. 64 bytes are not sufficiently large for storing the Diffie-Hellman parameter `g_a`. The SGX SDK puts the following default value into the 64 bytes of report data:
`SHA-256(g_a || g_b || VK) || {0}^32`
`VK` is a so-called *verification key* that is derived from the shared secret `g_a_b`.

To fix this problem and inform the USB forward device of a valid `g_a`, we propose to modify step 5 and step 6 such that the triple `(g_a, g_b, VK)` is sent to the USB forward device. This triple does not reveal any secrets and can be sent in clear. Then the USB forward device would verify `g_a` by recomputing the value `SHA-256(g_a || g_b || VK)` and comparing this value with the quote.

6.3 Replay Attack against SGX-USB

A security issue in Figure 6.2 is the susceptibility to a replay attack. Starting with step 6, a malicious OS can replay an old quote report, along with an old Diffie-Hellman parameter `g_a`. If a malicious OS knows the private Diffie-Hellman parameter `a`, then it can fully impersonate the client enclave.

Knowing the private Diffie-Hellman parameter `a` is a strong assumption that is not possible without a vulnerable enclave or SGX platform. Unfortunately, this assumption is not too far-fetched since SGX has been already broken completely by the Foreshadow attack in 2018 [47]. By exploiting Foreshadow, an attacker can read arbitrary enclave memory and extract attestation keys. Subsequently, an attacker could generate a spoofed quote with attacker-chosen `g_a` and `a`. Next, an attacker was able to send a spoofed quote to the IAS and obtain a valid quote report. Nowadays, obtaining such a quote report is not possible anymore since the IAS rejects platforms that are not patched against Foreshadow. However, since the USB forward device does not verify the freshness of a quote report it is still possible to replay an old quote report with an attacker-chosen `g_a` and `a`. On top of that, it is possible that a similarly severe SGX vulnerability will be discovered in the future.

This replay attack is a direct result of the protocol modification in step 6. The regular SGX remote attestation protocol prevents replay attacks since both the client enclave and the service provider generate new randomness (`g_a`, `g_b`) and this randomness is signed by the respective opposite party at each protocol invocation. To counteract the replay attack, we present two alternative protocols.

Attestation without Service Provider. To solve the replay problem of SGX-USB, the most straightforward option is to let the USB forward device act as service provider. In that case, all relevant SGX-messages need to be processed by the USB forward device and there does not exist a real service provider anymore.

The only deviation from the regular remote attestation protocol in Figure 6.1 is that the communication with the IAS needs to be tunneled via the (untrusted) host OS of the enclave. Nevertheless, this tunneling is not a security issue since IAS messages are secured via two independent mechanisms: Firstly, the communication to the IAS is secured via TLS. Secondly, IAS quote reports are signed with a private key that is only known by Intel.

Attestation with Service Provider. Another option for solving the replay problem of SGX-USB is to offload more responsibility to the service provider. Once the service provider has completed the regular remote attestation flow, the service provider could act as a trusted third party, establishing a shared secret between an enclave and a USB forward device. This option could be implemented with the following steps:

1. Upon successful remote attestation, establish a secure channel between an enclave and the service provider.
2. Establish a secure channel between a USB forward device and the service provider.
3. Run some key exchange protocol between an enclave and a USB forward device, relying on the service provider as a trusted third party.

6.4 Limitations of SGX Embedded Remote Attestation

Regardless of the protocol modifications, SGX embedded remote attestation has several limitations with the current incarnation of SGX. We discuss these limitations in this section, including potential future solutions.

6.4.1 Security

SGX remote attestation does not achieve the level of security that we expect for a trusted I/O setting. Specifically, SGX remote attestation is susceptible to *relay attacks* and *emulation attacks* [21]. Those attacks are out-of-scope for the regular usage of SGX remote attestation, but they are relevant for embedded remote attestation. A relay attack redirects the remote attestation protocol to an unintended (attacker-controlled) platform. An emulation attack forges arbitrary remote attestation responses, by using a leaked attestation key that is not yet revoked. Although emulation attacks rely on a strong assumption, leaked attestation keys have been already discovered in the wild [47].

As a mitigation, ProximiTEE proposed a *distance bounding protocol* that extends the regular SGX remote attestation protocol [21]. However, the distance bounding protocol only protects against relay attacks, but not against emulation attacks. The underlying issue with emulation attacks is the lack of *fault tolerance*. With fault tolerance, we refer to the desired security property that the existence of a compromised platform should not break the security of a non-compromised platform. For instance, a remote platform that is compromised by Foreshadow [47] should not threaten a trusted I/O device that is attached to a non-compromised platform.

We speculate that a solution against emulation attacks could be introduced to future SGX versions by leveraging the Intel Management Engine (Intel ME). Intel ME might be able to extend the already existing local attestation mechanism to trusted I/O devices. The Platform Services Enclave (PSE) links the trust domain of SGX to Intel ME. For example, Protected Audio Video Path (PAVP) is a feature that utilizes Intel ME for a trusted I/O setting [44]. Nevertheless, in Chapter 7, we prevent both attacks with a protocol that provides mutual authentication based on pre-shared keys.

6.4.2 Reliability

Reliability is another requirement that should be achieved by an embedded remote attestation protocol. Therefore, it is problematic to depend on the Intel Attestation Service (IAS) for setting up trusted I/O devices. Although we do not assume that the IAS faces any service outages, it is not guaranteed that a trusted I/O device acting as a service provider is able to connect to the IAS. In many scenarios, it might even be necessary to setup trusted I/O devices without Internet connectivity on the host machine.

Technically, SGX remote attestation depends on quotes that are generated by the *quoting enclave*. Quotes are signed with the EPID group signature scheme. Normally, it would suffice to verify a quote signature with an EPID group public key. However, Intel encrypts the quote signatures with a largely undocumented chain of crypto operations. According to Aumasson et al. [11], the quote generation involves a random key that is encrypted with some 2048-bit public RSA key. This encryption forces users and vendors to rely on the IAS. Thereby, the IAS is able to swiftly exclude broken platforms that are affected by newly discovered SGX vulnerabilities.

We consider the inability to locally verify remote attestation quotes as a drawback of SGX. As a conclusion, we urge Intel to open up the quote verification to independent software vendors (ISVs). ISVs should have the choice of whether they want to rely on the IAS or perform the quote verification themselves.

Chapter 7

SGX Mutual Authentication

In Chapter 4, we already discussed why *mutual authentication* between an enclave and a MACSec gateway is desirable. In general, pre-shared keys provide mutual authentication between two or more parties. However, this is not necessarily true in an SGX setting since a malicious OS can repeat a key installation at any time. In a pre-shared key setting, mutual authentication can be only achieved if enclaves are protected against *fake key installations*. A fake key installation is an attack where a malicious OS attempts to install self-chosen keys to an enclave.

In this chapter, we discuss monotonic counters as a protection method against fake key installations. Finally, we present a protocol that provides a strong protection against fake key installations with the help of an external signing facility.

7.1 Initial Key Installation

Figure 7.1 sketches a simplified scheme for the installation of pre-shared MACSec keys. A trusted entity generates MACSec keys and installs them to both the enclave and the MACSec gateway (e.g. a vendor or system administrator). The initial key installation requires an initially non-compromised environment, as already proposed by ProximiTEE [21].

Upon successful key installation, an enclave *seals* the MACSec keys in an encrypted and authenticated file. SGX uses so-called *seal keys* for persisting secrets in distrusted storage. A seal key can only be retrieved within an enclave and should never leave an enclave.

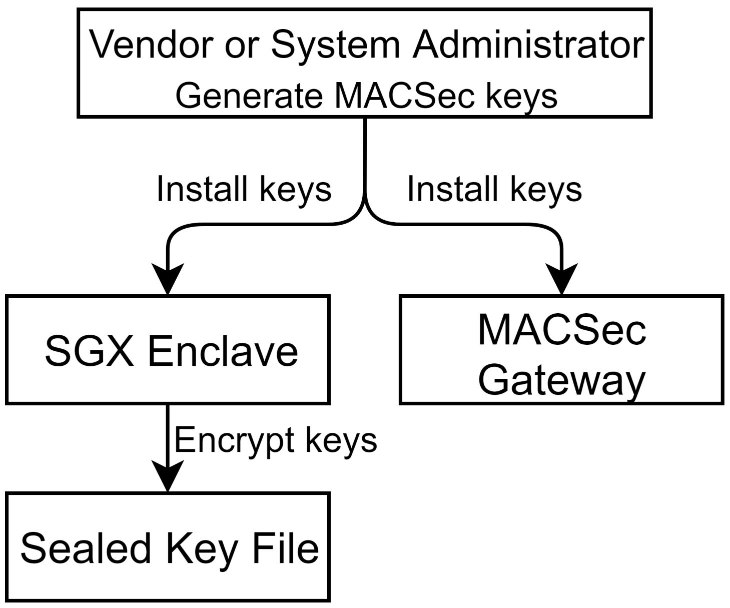


Figure 7.1: Installation of pre-shared MACSec keys.

7.2 Fake Key Installations

Sealed files do not provide any protection against fake key installations. Unless specific countermeasures are deployed, a malicious OS can easily perform a key installation towards the enclave with an attacker-chosen key. A successful fake key installation breaks the mutual authentication property that we usually expect from pre-shared keys. Once mutual authentication is broken, a malicious OS can simulate a MACSec gateway. Thereby, fake key installations undermine the confidentiality of outgoing data if packet encryption is enabled.

Even if mutual authentication is not strictly required, a fake key installation can help a malicious OS to exploit additional vulnerabilities within an enclave. For instance, a malicious OS could leverage a fake key installation for injecting arbitrarily crafted malicious MACSec packets into an enclave. Under normal circumstances, these malicious packets would be dropped immediately by the MACSec verification code without ever reaching any protocol-specific code. These malicious packets could lead to the exploitation of a potential buffer overflow vulnerability within the enclave, which in turn could lead to the extraction of a seal key. Once a malicious OS has extracted a seal key, it can decrypt the real MACSec keys.

7.3 Key Installation with Monotonic Counters

If an initially installed key can be protected against rollback attacks, then a malicious OS cannot perform a fake key installation. One method for protecting an enclave against rollback attacks are the *monotonic counters* that are provided by the SGX Platform Software. Listing 7.1 shows how monotonic counters can be used as a primitive countermeasure against fake key installations.

```
IF (monotonic_counter_exists(CONSTANT_KEY_ID)) THEN
    reject_keys()
ELSE
    install_keys()
    monotonic_counter_create(CONSTANT_KEY_ID)
```

Listing 7.1: One-time key installation based on montonic counters

From a security perspective, monotonic counters add a considerable amount of code and complexity to the trusted code base, namely the Intel Management Engine (Intel ME) and the Platform Services Enclave (PSE). The Intel ME stores monotonic counters on chipset flash storage that is not accessible for a malicious OS. Large portions of the Intel ME are undocumented and subject to reverse engineering. To avoid monotonic counters, we propose a protocol that relies on an external signing facility instead.

7.4 Key Installation with Signing Facility

In this section, we present a one-time key installation protocol that establishes a shared secret between an enclave and a trusted I/O device. By preventing a malicious OS from repeating a key installation, our protocol provides mutual authentication based on a symmetric key.

The basic principle is that an enclave only accepts *sealed keys* that are signed by an external signing facility. Signing sealed keys instead of plaintext keys provides two additional security properties: Firstly, the signing facility never sees a key in plaintext. Secondly, we achieve fault tolerance for the case that a key gets stolen from a remote machine. Since we bind the signatures to sealed keys instead of plaintext keys, each signature is only valid for exactly one SGX-enabled CPU. This prevents a malicious OS from installing a key that has been stolen from a remote machine.

Protocol Steps. The following steps describe the one-time key installation protocol as illustrated in Figure 7.2. All steps are done during an initial trust phase within a controlled environment. The signing facility must not be publicly accessible from the Internet unless additional security measures are employed. We expect that the signing facility is controlled by the vendor of the enclave.

1. The enclave generates a fresh key `skey` and installs `skey` to the trusted I/O device. The trusted I/O device must enforce that a malicious OS cannot repeat a key installation, e.g. by keeping a flag in persistent memory or requiring physical access for a key installation.
2. The enclave seals `skey` to a file that we denote as `seal`. The enclave sends `seal` to the signing facility.
3. Using its private key, the signing facility computes a signature `sig` over `seal` and sends `sig` to the enclave.

During normal operation, the enclave verifies `sig` with the (hardcoded) public key of the signing facility. Then the enclave unseals `seal`, yielding `skey`. Using `skey`, the enclave establishes a secure channel to the trusted I/O device. The verification of `sig` and the unsealing are done at each enclave launch.

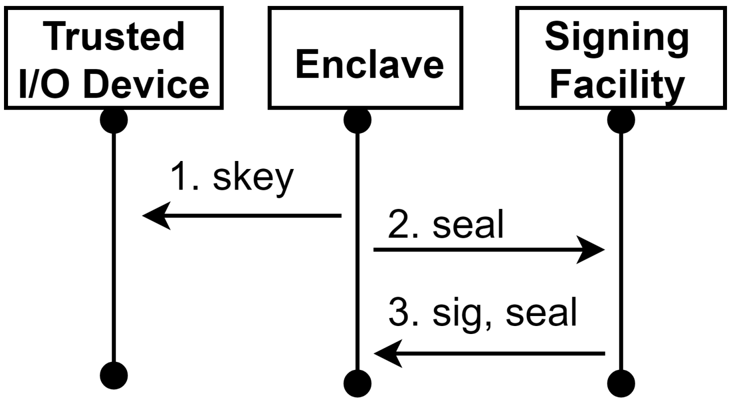


Figure 7.2: Key installation with signing facility.

Security Considerations. We enforce the physical proximity of the trusted I/O device to the enclave by using an initial trust phase, as proposed by ProximiTEE [21]. Therefore, the major security issue of our protocol is the size and complexity of the trusted code base during the initial key installation. To mitigate this issue, two techniques from ProximiTEE can be reused for our protocol: Firstly, a small and custom-tailored OS can be used for the initial key installation. Secondly, a generic boot enclave supports a dynamic deployment of new enclaves without having to repeat a key installation.

Robustness of One-time Key Installations. Storing sealed keys only locally might be problematic if a sealed key gets lost after an OS re-installation. This issue can be mitigated with an online backup service for sealed keys. Moreover, by leveraging an online backup service for sealed keys, vendors can ship trusted I/O devices along with Intel PCs without shipping any pre-installed OS image.

Chapter 8

Outsource-and-verify for Network Protocol Stacks

Outsource-and-verify is an approach where a TEE relies on untrusted code for its operation, and the correct operation of the untrusted code is verified at run-time. If the verification code is significantly smaller than the outsourced code, then outsource-and-verify shrinks the trusted code base. Beside of shrinking the trusted code base, outsource-and-verify may improve the compatibility of a TEE with existing legacy software, e.g. legacy software that relies on a specific protocol stack that can be verified in a TEE.

In this chapter, we present a run-time validation approach for ARP as well as TCP/IP stacks. We demonstrate that TCP/IP stacks can be validated with fewer than 500 lines of C-code. In contrast, even lightweight implementations of TCP/IP have several thousand lines of code [1, 23].

Moreover, we describe packet validations for SNMP, PROFINET DCP and S7COMM+. We apply a stateless packet validation whenever it is feasible for our intended protocol and use case. Our packet validations follow a strict whitelisting approach: All unexpected protocols or protocol subsets are denied.

8.1 ARP/IP Validation

We apply the outsource-and-verify approach for an (untrusted) ARP implementation. Therefore, the TEE needs to validate outgoing ARP packets. Specifically, we intend to prevent *ARP spoofing* and *IP spoofing*. With ARP spoofing, a malicious OS can reroute or sniff IPv4 packets in a local network by sending forged

ARP replies. IP spoofing denotes IP packets with a forged source IP. A malicious OS might combine IP spoofing with ARP spoofing to perform man-in-the-middle attacks in a local network.

Both attacks resort to a malicious OS misusing another's IP address rather than its own. A TEE can prevent both attacks by knowing its only valid host IP. We assume that the TEE knows a host IP that is either pre-configured or retrieved from a trusted source. In many cases, it suffices that the TEE only knows the host bits of the host IP (the least significant bits). If this assumption is met, then a TEE can easily enforce the following security restrictions:

- The source IP of outgoing ARP packets must match the host IP.
- The source IP of outgoing IP packets must match the host IP.

These simple checks are sufficient for preventing ARP-based attacks in our setup. Hence, the outsource-and-verify approach for ARP is very efficient in terms of code size and code complexity.

Figure 8.1 illustrates the packet validation flow for untrusted legacy applications. MACSec packets are highlighted with yellow arrows. Trusted components are highlighted with a green color.

1. Untrusted legacy apps communicate with the kernel via socket system calls.
2. In response to socket system calls, the kernel generates ARP packets and IP packets that are outbound from the virtual network interface. We forward these packets to the packet validator, via the support library.
3. The packet validator enforces that outgoing packets conform to a known host IP as well as other protocol-specific checks. If all checks succeed, then the TEE transforms a packet into a MACSec packet and forwards it to the local network (via the support library and the MACSec gateway).

We forward incoming ARP packets to the virtual network interface without performing any ARP-specific security checks.

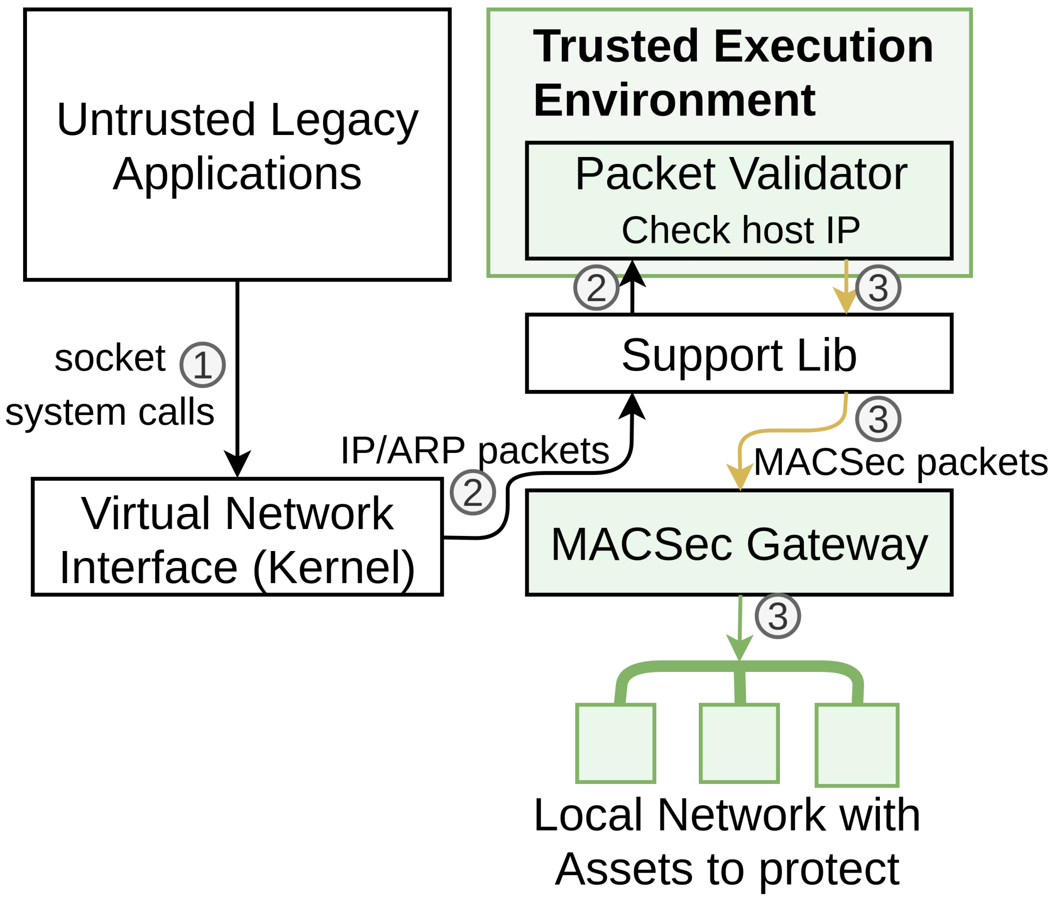

Untrusted Legacy Applications
Trusted Execution Environment
Packet Validator
Check host IP
2
3
Support Lib
socket
1
system calls
IP/ARP packets
2
3
MACSec packets
Virtual Network Interface (Kernel)
MACSec Gateway
3
Local Network with Assets to protect

Figure 8.1: Packet validation for untrusted legacy applications.

8.2 TCP/IP Stack Validation

In this section, we detail our approach for outsourcing a TCP/IP stack and validating TCP packets within a TEE. The result of this section is a *trusted socket API* that can be used to establish trusted TCP connections to a local network. The trusted socket API creates and maintains a *shadow state* for each open TCP connection. We use the shadow state for the stateful validation of TCP packets. We describe both the data flow of incoming and outgoing TCP traffic as well as the validation of individual TCP packets. Furthermore, our approach supports an optional encryption of TCP payload.

8.2.1 Outgoing Data Flow

The data flow for outgoing TCP traffic is depicted in Figure 8.2. Trusted components are highlighted with a green color. MACSec packets are highlighted with yellow arrows. Figure 8.2 comprises the following steps:

1. App-specific TEE code calls the functions `connect()` and `send()` that are provided by our trusted socket API. The TEE is assumed to know the target IP and port.
2. The trusted socket API inserts information into the TCP shadow state. For the `connect()` function, the trusted socket API instantiates a new shadow state. For the `send()` function, the trusted socket API copies the payload data into an already existing shadow state.
3. The trusted socket API calls OS-specific socket system calls via the support library (e.g. the system calls `socket`, `connect`, `send` for POSIX-compliant systems).
4. The virtual network interface receives the data from the `send` system call and creates a sequence of TCP packets. These TCP packets are forwarded to the packet validator (via the support library).
5. The packet validator executes a sequence of TCP-specific security checks that are based on the shadow state. Packets that do not pass the checks are dropped.
6. The packet validator transforms TCP packets into MACSec packets. The MACSec packets are sent to the local network, via the support library and the MACSec gateway.

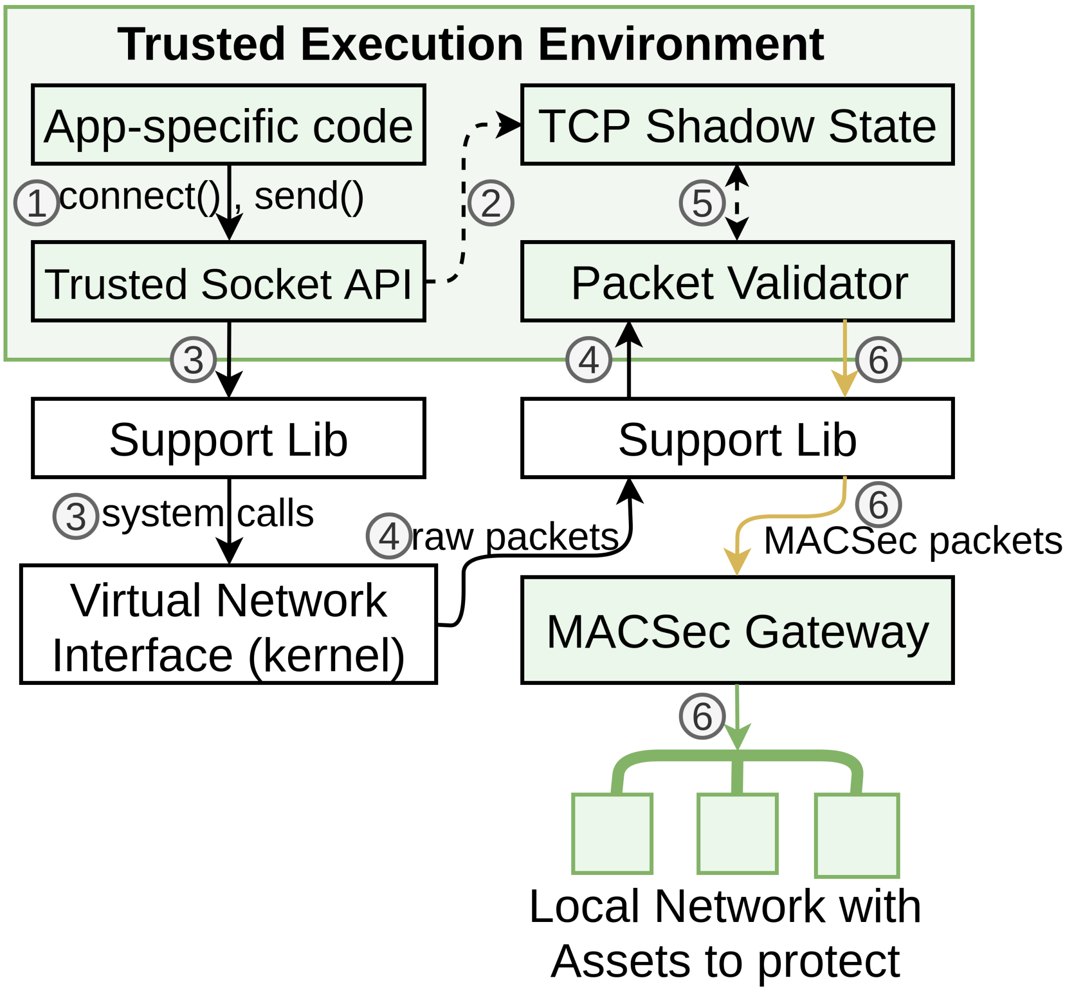


Figure 8.2: Data flow for outgoing TCP traffic.

Outgoing Packet Validation. Our TCP shadow state contains the following entries for the validation of outgoing packets:

- Initial outgoing sequence number
- SYN-INIT flag
- Send buffer

We use this shadow state for matching the payload of an outgoing TCP packet with the TCP stream that was previously stored in the send buffer. More specifically, the validation of an outgoing TCP packet comprises the following steps:

1. Stateless TCP/IP header validation.

2. If the TCP packet does not have a payload and `SYN` is set to zero, then go to the last step. For example, this allows `ACK` packets to pass without any further security checks.
3. Find an associated shadow state based on the following packet entries: `(destination IP, source port, destination port)`.
4. If the packet is a `SYN` packet, then synchronize the shadow state if it is not yet synchronized (i.e. store the initial outgoing sequence number into the shadow state and set the SYN-INIT flag to 1). If the synchronization succeeds, then go to the last step. All unexpected `SYN` packets are dropped.
5. Check whether the payload of the TCP packet matches the shadow state send buffer. This check is based on the sequence number, determining the offset within the shadow state send buffer.
6. Transform the TCP packet into a MACSec packet, authenticating the packet with an Integrity Check Value (ICV).

Send Buffers. We are dealing with two different send buffers: A send buffer that is maintained by the host kernel and a TEE send buffer that is used by the packet validator. In order to perform a reliable payload validation, we need to guarantee that all the data in the host kernel's send buffer is also present in the TEE send buffer. We enforce this guarantee by configuring the size of the host kernel's send buffer to the same size as the TEE send buffer. For POSIX-compliant systems, the (maximum) send buffer size can be set with the `SO_SNDBUF` socket option.

Deallocating Shadow States. Since memory is a constrained resource a TEE needs to deallocate TCP shadow states at some point of time. Our trusted socket API deallocates a shadow state right after calling the (untrusted) `close` system call. Therefore, we require that the `close` system call blocks until all data in the send buffer has been acknowledged by the receiving end (or until a timeout is reached). For POSIX-compliant systems, the blocking behaviour of the `close` system call can be configured with the `SO_LINGER` socket option.

8.2.2 Incoming Data Flow

The data flow for incoming TCP traffic is depicted in Figure 8.3. This works as follows:

1. App-specific TEE code calls the function `recv()` that is provided by our trusted socket API. A valid shadow state and an open TCP connection must already exist, that is, `connect()` must have been called beforehand.

2. The trusted socket API calls the blocking `recv` system call via the support library. The support library passes a dummy buffer as the target buffer for the `recv` system call.
3. In the meantime, the MACSec gateway transforms incoming TCP packets into MACSec packets and forwards them to the packet validator.
4. The packet validator verifies an incoming MACSec packet and transforms it into a TCP packet. Then the packet validator attempts to find a valid shadow state.
5. The shadow state includes an associated TEE receive buffer. The packet validator copies the payload of the TCP packet into the associated TEE receive buffer.
6. The TCP packet is forwarded to the virtual network interface via the support library.
7. The virtual network interface reassembles incoming TCP packets and copies the payload into the dummy receive buffer. Since the TEE gets authenticated data directly from the packet validator, this dummy buffer can be discarded.
8. Eventually, the `recv` system call returns and hands back control to the trusted socket API.
9. Let n be the return value of the (untrusted) `recv` system call. n represents the number of received bytes. The trusted socket API attempts to fetch and remove the oldest n bytes of payload data from the shadow state.
10. The trusted socket API passes the n bytes of payload data to the app-specific TEE code and hands back control to the app-specific TEE code.

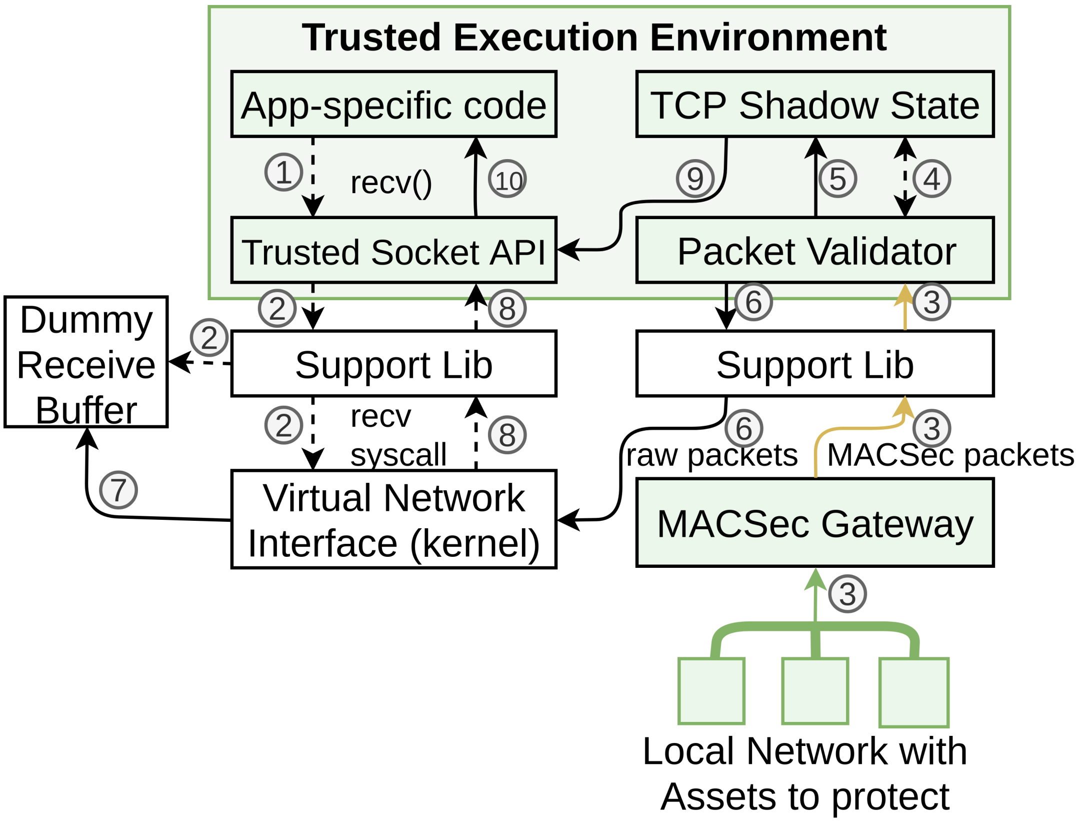


Figure 8.3: Data flow for incoming TCP traffic.

Incoming Packet Validation. Although there exists only one shadow state for each open TCP connection, the following shadow state entries are specifically destined for the validation of incoming TCP packets:

- Initial incoming sequence number
- SYN-ACK-INIT flag
- Record of received bytes
- Receive buffer

Using this shadow state, we copy the payload of an incoming TCP packet into the receive buffer. More specifically, the validation of incoming TCP packets comprises the following steps:

1. Verify the MACSec packet and transform it into a TCP packet.
2. Stateless TCP/IP header validation.

3. If the TCP packet does not contain a payload and `SYN` is set to zero, then go to the last step. For instance, this forwards incoming `ACK` packets without any further checks.
4. Find an associated shadow state based on the following packet entries: `(source IP, source port, destination port)`.
5. If the packet is a valid `SYN ACK` packet, then synchronize the shadow state (i.e. store the initial sequence number into the shadow state and set the SYN-ACK-INIT flag to 1). All non-conforming `SYN` packets are discarded.
6. Copy the payload of the TCP packet into the receive buffer that is associated with the shadow state. The receive buffer offset is determined by the sequence number.
7. Record the number and the position of the received bytes in the shadow state.
8. Forward the TCP packet to the virtual network interface of the host kernel.

Receive Buffers. We are dealing with two different receive buffers: A TEE receive buffer and a receive buffer that is maintained by the host kernel. The *TCP window size* ensures that a receiver is able to fit incoming TCP data into a receive buffer. However, we cannot directly control the TCP window size since it is controlled by the host kernel. Instead, we configure the receive buffer of the host kernel to the same size as the TEE receive buffer. For POSIX-compliant systems, the (maximum) receive buffer size can be set with the `SO_RCVBUF` socket option.

Note that a TEE receive buffer should be allocated even before calling the `recv` system call for the first time. Otherwise, we might run into a race condition where an incoming packet arrives while a TEE receive buffer does not exist yet.

Record of Received Bytes. When the kernel returns from the untrusted `recv` system call, our trusted socket API verifies that indeed all data has been received. Therefore, the shadow state keeps a record of the received bytes.

We implement this record with a simple counter of the received bytes. Our implementation enforces that incoming TCP packets must conform to the expected sequence number without any gaps in between. If an incoming sequence number is higher than expected, then this is most likely the result of a packet loss. In case of a packet loss, not only the lost packet needs to be re-transmitted, but also all packets that have a higher sequence number than the lost packet. Therefore, enforcing the correct sequence number does not degrade the performance since a re-transmission needs to happen in any case.

A more heavy-weight record of the received byte would be a fully-fledged TCP reassembly buffer [22]. By using a fully-fledged reassembly buffer, the packet validator can benefit from the performance of *Selective Acknowledgements* (SACK), if SACK is used by the host kernel.

8.2.3 TCP Header Validation

For incoming packets, the validation of TCP headers is not security critical since incoming packets are already authenticated via MACSec. For outgoing packets, the validation of TCP headers is security critical since a malicious OS can create outgoing TCP packets with malformed or unexpected TCP headers. This subsection details how we validate the individual fields of the TCP header, with the main focus on outgoing TCP packets. Table 8.1 includes a complete list of the TCP header fields and how they are handled by our packet validator.

Table 8.1: TCP header validation.

Field	**Treatment**
Source port	Shadow state
Destination port	Shadow state
Sequence number	Shadow state
Acknowledge number	Shadow state
Data offset	Sanitized
Reserved	Must be zero
NC Flag	Ignored, only performance-relevant
CWR Flag	Ignored, only performance-relevant
ECE Flag	Ignored, only performance-relevant
URG Flag	Must be zero
ACK Flag	Ignored for Non-`SYN` packets
PSH Flag	Ignored for Non-`SYN` packets
RST Flag	Ignored for Non-`SYN` packets
SYN Flag	Shadow state, special treatment
FIN Flag	Ignored for Non-`SYN` packets
Window size	Ignored, not security-relevant
Checksum	Ignored, not security-relevant
Urgent pointer	Ignored, since URG must be zero
Options	Whitelisting
Padding	Must be zero

Source Port, Destination Port. In combination with the IP addresses, the packet validator uses the ports for finding a matching shadow state for both incoming and outgoing TCP packets. When connecting to a socket, the source port is not yet known since it is chosen by the untrusted TCP/IP stack. When a shadow state gets synchronized with an outgoing `SYN` packet, then the packet validator stores the source port in the shadow state.

Sequence Number. We first distinguish between `SYN` packets and Non-`SYN` packets. When a `SYN` packet is sent or received, we use it for synchronizing a shadow state. That is, the packet validator stores the sequence number in the shadow state. If the shadow state is already synchronized, then we only allow a `SYN` packet if the sequence number matches the shadow state (i.e. we only allow a re-transmission of a `SYN` packet if it is unchanged). A `SYN` packet must not contain a payload.

For Non-`SYN` packets, we distinguish between packets that contain or do not contain a payload. For Non-`SYN` packets that do not contain a payload, the sequence number is semantically ignored by TCP. For (Non-`SYN`) packets that contain a payload, the packet validator uses the sequence number for determining the correct payload offset in the TEE shadow state buffer (either in the shadow state send buffer or in the shadow state receive buffer).

Acknowledgment Number. We ignore the acknowledge number for outgoing packets. By sending a spoofed acknowledge number, a malicious OS could either force a TCP re-transmission or the loss of packets. Both only impacts availability but not security. Independent of the acknowledge numbers, we keep a record of the bytes that have been received in the shadow state.

However, there exists an edge case where the acknowledge number of an *incoming packet* is security-relevant. Specifically, a malicious OS can send multiple `SYN` packets that only differ in the sequence numbers, by initiating and closing multiple subsequent TCP connections. Further, a malicious OS can selectively choose which outgoing and incoming packets reach their target. This may lead to a situation where the sequence numbers are not in sync, that is, our shadow state and the remote end in the local network do not agree on the same initial sequence numbers. If the sequence numbers are not in sync, then a malicious OS might be able to inject a legitimate payload into the context of a different TCP connection with the same tuple of (`source port`, `destination port`, `source IP`, `destination IP`).

To prevent this kind of attacks, we combine the MACSec replay protection with a stateful validation of handshake packets. We enforce that for new TCP connections a valid `SYN-ACK` packet must be received before allowing any outgoing TCP packets with a payload. We enforce that the acknowledge number of the `SYN-ACK` packet must match the previously sent `SYN` packet.

Data Offset. Data offset holds the size of the TCP header in 4-byte words. This is necessary to determine the size of the options field. We enforce that data offset is greater or equal to 5, which corresponds to the minimum TCP header length of 20 bytes.

Reserved. As required by TCP, we enforce that the reserved bits are zero.

Control Flags. The *control flags* are: `NC`, `CWR`, `ECE`, `URG`, `ACK`, `PSH`, `RST`, `SYN`, `FIN`. We ignore the flags `NC`, `CWR`, `ECE` since they are only relevant for performance. We enforce that `URG` must be zero.

For the remaining flags, we distinguish between `SYN` packets and Non-`SYN` packets. The `SYN` flag synchronizes a sequence number with the receiving end, which is the prerequisite for sending a TCP payload and performing a stateful validation of the TCP payload. Therefore, we treat `SYN` packets as a special case and perform a stateful validation of `SYN` packets. We accept a `SYN` packet if and only if there exists an associated shadow state that has not yet been synchronized, or if the `SYN` packet is a valid re-transmission for a shadow state that is already synchronized. Moreover, we enforce the following:

- `PSH`, `RST`, `FIN` must be zero for all `SYN` packets.
- `ACK` must be zero for outgoing `SYN` packets and 1 for incoming `SYN` packets.

For Non-`SYN` packets, we ignore the flags `ACK`, `PSH`, `RST`, `FIN` since we do not need to track these flags for the validation of outgoing TCP payload. Normally, a malicious OS could use the `RST` flag for *TCP reset attacks* [48]. A TCP reset attack destroys legitimate TCP connections between (unrelated) peers. However, as already detailed in the section about ARP spoofing and IP spoofing, we prevent TCP reset attacks on other peers by enforcing a known and valid source IP address. Thus, a malicious OS can only destroy its own TCP connections with reset attacks. The remaining security implication is that a malicious OS can send Non-`SYN` packets that do not fit into the current context, e.g. sending a `FIN` packet although there does not exist an open TCP connection. Nevertheless, this is not a security issue since the TCP state machine drops bogus `FIN` packets or other unexpected Non-`SYN` packets.

Window Size, Checksum, Urgent Pointer. These fields are not security relevant for our setting and hence, ignored.

Options. *Options* is a variable-sized field in the TCP header that is used for various options. There exist a few widely used options that are implemented by more sophisticated TCP/IP stacks. Since we want to prevent a malicious OS from sending malformed or unexpected options, we enforce a strict whitelisting approach for the options field. Our TCP validation code accepts the following options:

- `WINDOW SCALE` (RFC 1323)
- `TIMESTAMP` (RFC 1323)
- `MAXIMUM SEGMENT SIZE` (RFC 879)
- `SACK PERMITTED` (RFC 2018)
- `SACK` (RFC 2018)

The outsource-and-verify approach takes advantage of these options without implementing them inside a TEE. Since these options are not security-relevant by itself we merely enforce that there are no unexpected or malformed options.

8.2.4 Payload Confidentiality

If MACSec packet encryption is enabled, then the outsource-and-verify approach should be adapted in order to keep the TCP payload confidential. To do so, we temporarily replace the payload of TCP packets with a dummy payload.

For outgoing TCP traffic, we replace the payload with a dummy payload in step 3 in Figure 8.2. In step 5, we replace the dummy payload with the payload that is stored in the shadow state, before encrypting the TCP packet via MACSec. For incoming TCP traffic, we replace the payload with a dummy payload in step 6 in Figure 8.3. The dummy payload is discarded anyway in step 7. All other steps remain unchanged.

8.3 PROFINET DCP Validation

PROFINET Discovery and Configuration Protocol (DCP) is a part of the PROFINET protocol stack [37]. DCP discovers PROFINET devices in a local network. DCP also supports the assignment of *device names* and IP addresses to devices.

In our scenario, we only allow the subset of DCP that is required for the discovery functionality (read-only functionality). Thus, we want to prevent the assignment of device names and IP addresses. More specifically, our enclave only allows outgoing DCP packets that are *DCP Identity Requests* directed to the multicast address. This effectively prevents a malicious OS from tampering with PROFINET devices, while still enabling PROFINET discovery functionality. Existing software that relies on DCP can be used without modifications.

8.4 SNMP Validation

The *Simple Network Management Protocol (SNMP)* [32] is a protocol for modifying and collecting information about network devices. In our scenarios, we want to restrict a malicious OS to the subset of SNMP that is required for collecting information (a restriction to read-only commands). More specifically, we only allow outgoing SNMP packets that are either `GetNextRequest` or `GetRequest` packets. This effectively prevents a malicious OS from tampering with SNMP devices, while still allowing to collect device information via SNMP.

An SNMP device is expected to respond to a `Get(Next)Request`
with a `GetResponse`, including a set of *variable bindings*. These variable bindings provide status information that is used by untrusted legacy software. SNMP is based on UDP (User Datagram Protocol). Therefore, we also implement a validation for UDP headers. SNMP uses the well-known UDP port 161. Hence, we only accept outgoing UDP packets that are directed to port 161.

8.5 S7COMM+ Stack Validation

In this section, we describe an outsource-and-verify approach for the S7COMM+ protocol stack. After describing the protocol stack we detail our security validations. Finally, we outline the validation of the ISO-TCP protocol, which acts as an intermediate layer between TCP and S7COMM+.

8.5.1 Protocol Stack Description

S7COMM+ [29] is a proprietary protocol that is used for communication between Siemens PLCs (e.g. S7-1200, S7-1500) and industrial software (e.g. TIA Portal). S7COMM+ uses protocol data units (PDUs) that start with the byte `0x72`. S7COMM+ runs on top of *ISO-TCP* [43], which listens on the well-known TCP port 102. On top of S7COMM+, Siemens PLCs use the *Object Management System Plus (OMS+)*. OMS+ is an architecture that provides means for creating, manipulating and deleting objects and attributes of objects, whereas the objects are placed within a (nested) tree structure. The full protocol stack is depicted in Table 8.2.

Table 8.2: S7COMM+ protocol stack.

OMS+
S7COMM+
ISO-TCP
TCP
IPv4
Ethernet

8.5.2 S7COMM+ Security Validations

S7COMM+ exposes security-critical functionality like PLC program downloads. A remote attacker can use S7COMM+ to disrupt or manipulate the operation of industrial equipment [12, 31, 46].

A typical protection against threats via S7COMM+ is to use a firewall that blocks TCP port 102. In this work, we follow a more elaborate security approach with S7COMM+. We perform security checks for the S7COMM+ protocol stack such that only *signed PLC firmware updates* are permitted, whereas all other functionality of S7COMM+ should be blocked. This can be used in the context of an edge computing device that implements *remote firmware updates*.

Due to the complexity of the S7COMM+ protocol stack, this validation is only possible with a stateful approach. In contrast to the TCP/IP stack validation, we do not provide an intra-enclave API for interacting with S7COMM+ devices. Instead, we generate the S7COMM+ shadow state on the fly based on the packets that are validated. Our packet validation for S7COMM+ includes the following steps (simplified):

- We extract S7COMM+ PDUs out of a TCP stream that is headed to port 102 (via the ISO-TCP layer). This includes the reassembly of *inner fragments* to complete S7COMM+ PDUs. After the reassembly, we enforce that the S7COMM+ headers are well-formed.
- We only allow the following *functions* via S7COMM+: `SetVariable`, `CreateSession`, `Explore`, `SetVarSubstreamed`, `GetVarSubstreamed`, `SetMultiVariables`, `DeleteObject`.
- Depending on the function, we only allow specific *object ids*. For example,

the functions `CreateSession` and `DeleteObject` must be only used for the creation and deletion of a so-called *session object*.

- Depending on the object id's, we only allow specific *object attributes*. This includes the object id's and attributes that are necessary for deploying a firmware update.

8.5.3 ISO-TCP

ISO-TCP [43] is used as an intermediate layer between TCP and S7COMM+. ISO-TCP on its own is not security-critical. Thereby, we perform a white-listing to enforce that a well-formed ISO-TCP stream is sent to port 102. We extract ISO-TCP chunks out of the TCP stream and pass those chunks to the higher-level validation code for S7COMM+.

More specifically, ISO-TCP splits a TCP stream into chunks that are denoted as *Transport Protocol Data Units (TPDUs)*. One or more subsequent TPDUs form a larger chunk that is denoted as *Transport Service Data Unit (TSDU)*. We perform the following steps for the validation of an ISO-TCP stream:

1. We enforce that an *ISO Connect Request* is sent right after the establishment of a new TCP connection. This is the normal protocol flow and any deviations are not expected.
2. We extract TPDUs out of the TCP stream that is headed to port 102.
3. We reassemble TSDUs out of the extracted TPDUs. Each TSDU contains a S7COMM+ PDU that is passed to the higher-level validation code.

Chapter 9

Implementation Details

Our implementation is entirely written in C and requires a Linux system along with the SGX Driver and the SGX Platform Software. For systems where SGX is not available, our implementation supports a simulation mode for testing purposes. We tested our implementation for interoperability with the MACSec implementation of the Linux kernel.

9.1 Component Description

A layer representation of the system architecture is depicted in Figure 9.1. Trusted components are highlighted with a green color. This section provides implementation details about the components and the communication between them.

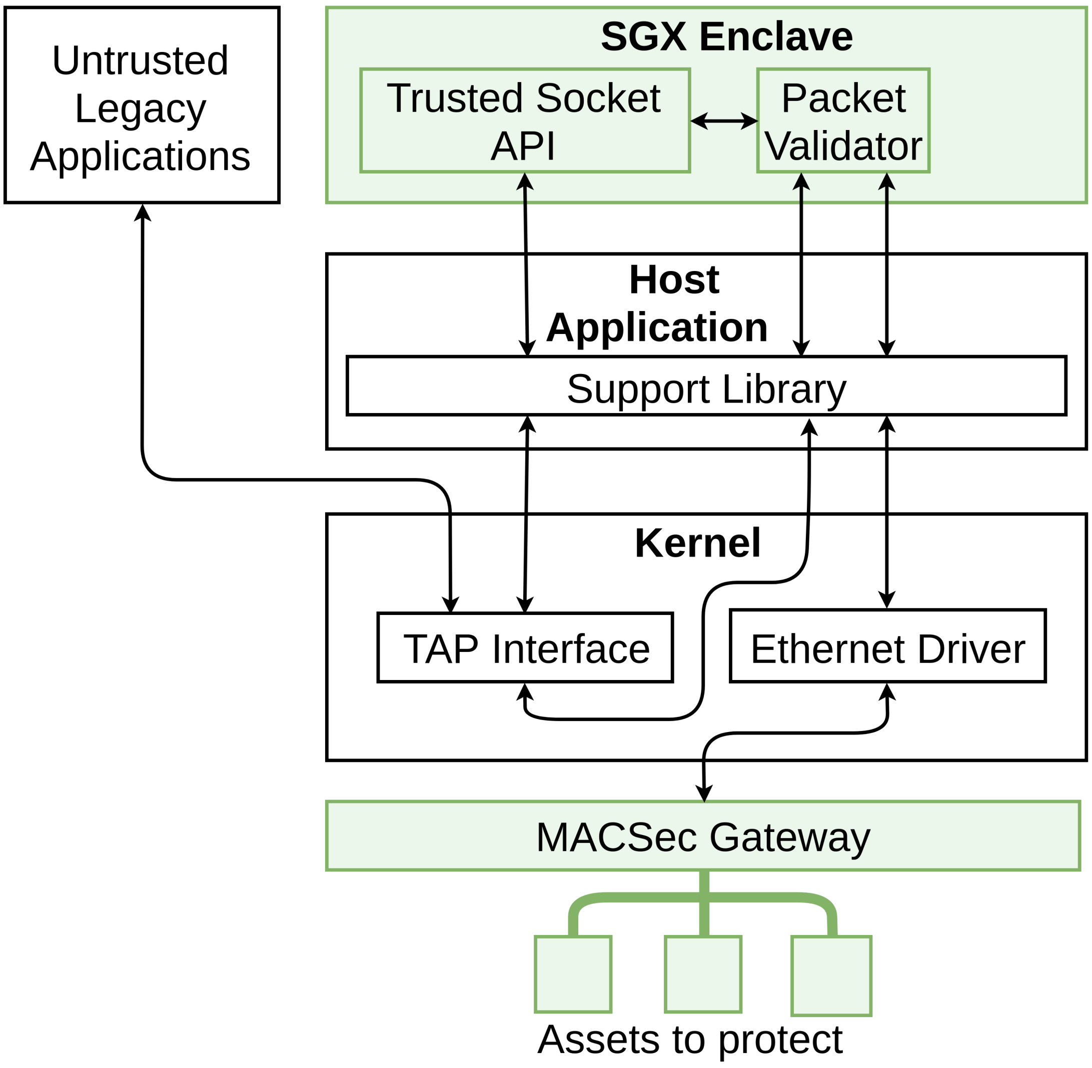


Figure 9.1: Layered system architecture.

SGX Enclave. Beside of protocol validation code, the enclave may provide application-specific logic for accessing the protected network.

Trusted Network Library. The trusted network library implements all the network security functionality that is described in this work. This includes the packet validator and the trusted socket API. The trusted socket API provides the following TCP socket functions for app-specific enclave code: `connect()`, `send()`, `recv()`, `close()`. The trusted network library consists of mostly self-contained C code. The only dependencies are the trusted libraries that are shipped with the SGX SDK. These trusted libraries implement a small subset of the C standard library, as well as a few cryptographic and SGX-specific functions.

Host Application. The host application is a regular Linux process that hosts the enclave of an SGX-secured network interface. At startup, the host application performs the following tasks:

- Launch the enclave.
- Load the support library. The support library configures the SGX-secured network interface and launches the *packet forwarding threads*.
- Load sealed MACSec keys into the enclave.

TAP Interface. A *TAP interface* is a virtual Ethernet interface that is implemented by a Linux kernel driver. From an application perspective, a TAP interface behaves like a regular network interface. If an application uses a TCP socket via system calls, then the data traverses the kernel networking stack and the resulting raw Ethernet packets may be routed to the TAP interface. Hence, even untrusted legacy applications can open sockets whose traffic is outbound from the TAP interface. From a kernel perspective, each TAP interface must be bound to exactly one process that holds a special file descriptor for sending and receiving raw Ethernet packets. In our case, the TAP interface is linked to the support library, which forwards packets to packet validator.

Support Library. The support library acts as a bridge between the trusted network library and the operating system. The support library holds two *raw socket file descriptors*: One for the TAP interface, and one for the MACSec gateway. Two dedicated threads forward the packets between the TAP interface and the MACSec gateway, traversing the packet validator. We statically link the support library within the host application.

9.2 TCP Optimizations

Our trusted socket API implements additional optimizations that are not described in Chapter 8. The shadow state holds a *TEE send buffer* and a *TEE receive buffer* for each open TCP connection. Normally, one would use a data structure like a ring buffer or a linked list buffer for implementing dedicated send buffers and receive buffers. However, our trusted socket API does not implement any dedicated send buffer or receive buffer. Instead, our shadow state points to the original buffers that were passed by the user of the trusted socket API.

This optimization improves the performance by avoiding an additional copy of the data. On the other hand, this optimization has the following implications on the trusted socket API: The buffer that is passed to the `send()` function must be kept valid and unaltered until one of the following events occur:

- The `close()` function returns after closing the socket.
- The receiving end responds to the sent data, acknowledging the receipt of the data at the application level.

Moreover, we require that the TEE pre-allocates a single receive buffer of sufficient size that is used during the entire live time of a TCP connection. This pre-allocated buffer must be kept valid until the `close()` function returns. Chapter 10 evaluates this optimized version of the trusted socket API.

Chapter 10

Evaluation

This chapter evaluates our implementation with respect to code size, performance and usage scenarios.

10.1 Security

We use a MACSec gateway for preventing a malicious OS from directly accessing a local network. MACSec protects packets between an enclave and the MACSec gateway. We achieve mutual authentication between a MACSec gateway and an enclave by installing pre-shared keys. We prevent fake key installations in the enclave by signing sealed MACSec keys via a trusted third party. The security of outsource-and-verify critically depends on the correct validation. For instance, our TCP validation combines a stateless validations with a stateful validation of payload. The payload is given by a higher-level protocol, whose security should be evaluated separately.

Code Size. A major goal of the outsource-and-verify approach is to shrink the trusted code base. In Table 10.1, we compare the code size of our TCP/IP/ARP validation with the code size of `picotcp` [1]. `picotcp` is a small-footprint, modular TCP/IP stack. `picotcp` includes separate modules for TCP/IPv4/ARP. We only count the code of the TCP/IPv4/ARP modules, instead of the complete `picotcp` stack. For the outsource-and-verify approach, we only count code that is running inside a TEE. All line counts are done with the `cloc` tool. Comments and blank lines are excluded; all other lines are counted.

Table 10.1 clearly shows that outsorce-and-verify is effective in shrinking the trusted code base. Our TEE code is an order of magnitude smaller than the comparable modules of `picotcp`. A smaller trusted code base leads to a higher level of security confidence.

Table 10.1: Code size comparison between outsource-and-verify and picotcp [1].

Protocol	**Outsource-and-verify**	**picotcp stack** [1]
TCP	333	2.653
IPv4	69	1.379
ARP	27	448

Table 10.2 lists the code size for all the functionality that is implemented by our trusted network library. One can see that outsource-and-verify is feasible in practice.

Table 10.2: Code size of our trusted network library.

Functionality	**Lines of code**
Trusted socket API	53
TCP validation	333
IPv4 validation	69
ARP validation	27
SNMP validation	105
PROFINET DCP validation	27
ISO/TCP validation	246
S7COMM+ validation	125
OMS+ validation	158
SGX packet validation interface	90
SGX-MACSec protocol	163
SGX-MACSec key installation (sealing)	51
SGX-MACSec key loading (unsealing)	43
Utility functions	130

10.2 Performance

The goal of this section is to evaluate the individual performance overhead that is introduced by the following aspects of our implementation:

- TAP forwarding: Overhead of using a TAP interface.
- SGX forwarding: Overhead of the ECALLs for copying packets from and to an enclave.
- Trusted socket API: Overhead of the OCALLs and ECALLs that are introduced by our trusted socket API.
- MACSec: Overhead of our SGX-MACSec implementation and the MACSec gateway.

To perform this evaluation, we implemented a hierarchy of test modes that range from a regular Ethernet interface to the full SGX-secured interface that we introduced in this work.

Test Setup. We conduct this performance evaluation with two machines that are placed in the same local network. Both machines run Ubuntu 18.04 as OS. One machine is the SGX-secured machine; the other machine does not use SGX. The SGX-secured machine is a SIMATIC IPC427E featuring an Intel Core i5-6442EQ CPU (an Industrial PC). The other machine is a Fujitsu Lifebook featuring an Intel Core i5-2520M (without SGX support).

The SGX-secured machine represents an edge computing device; the other machine represents a device in a security-critical local network. The SGX-secured machine acts as a client (connecting to TCP ports). The other machine acts as a server (listening on TCP ports). The server simulates the MACSec gateway with the Linux kernel implementation of MACSec.

Benchmark Tests. We run the following three benchmark tests for all test modes:

- UDP round-trips: Evaluating the round-trip latency of UDP packets.
- HTTP requests: Evaluating the performance of short-lived TCP connections.
- Bulk data transfer: Large amount of data via a single TCP connection.

For each combination of test mode and benchmark test, we repeat the benchmark test 10 times to estimate the standard deviation of the measured execution times. All benchmark results are given in seconds.

UDP Round-trips. This benchmark measures the latency of UDP packet transmissions. The client sends a UDP packet to the server, and the server responds with a UDP packet. We repeat this round-trip 15 000 times. The client generates the UDP packets independently of the enclave. The enclave only performs the validation of UDP packets. Table 10.3 shows that overheads for the TAP interface, for the SGX ECALLs and for MACSec are approximately 15%. As expected, the stateless validation of UDP packets has a negligible overhead within the standard deviation.

Table 10.3: 15 000 UDP round-trips benchmark results.

Test mode	**Mean execution time**	**Standard deviation**	**Diff**	**Relative diff**
Regular interface	1.987	0.092	-	-
+ TAP forwarding	2.278	0.079	0.291	14.64 %
+ SGX forwarding	2.610	0.090	0.332	16.73 %
+ UDP validation	2.616	0.099	0.006	0.28 %
+ MACSec	2.919	0.064	0.303	15.26 %

HTTP Requests. This benchmark consists of 7 500 sequential HTTP requests. The server responds to those requests with a constant, small response. Due to the frequent TCP handshakes and tear-downs, this benchmark measures the round-trip latency of packets. For the trusted socket API, we generate the TCP stream inside the enclave. Table 10.4 shows that the execution time increases according to the hierarchy of the test modes. The overhead of our trusted socket API for TCP/IP stacks is only 5.7%. The largest overhead stems from the SGX ECALLs (22.54%).

Table 10.4: 7 500 HTTP requests benchmark results.

Test mode	**Mean execution time**	**Standard deviation**	**Diff**	**Relative diff**
Regular interface	2.337	0.054	-	-
+ TAP forwarding	2.665	0.066	0.327	14.00 %
+ SGX forwarding	3.192	0.053	0.527	22.54 %
+ Trusted socket API	3.325	0.057	0.133	5.70 %
+ MACSec	3.679	0.050	0.354	15.14 %

Bulk Data Transfer. The client sends a 50MB chunk to the server via a single TCP connection. The 50MB chunk is transferred within TCP packets of approximately 1500 bytes size. Therefore, this benchmark measures the *throughput* instead of the round-trip latency. Table 10.5 shows that the execution time increases according to the hierarchy of the test modes. It appears that the TAP interface has a small overhead for the throughput of packets (4.06%). The overhead of our trusted socket API is expensive (42.13%). A part of this overhead is necessary anyways because we generate the entire TCP stream within the enclave before copying it to untrusted memory. Moreover, we suspect SGX memory latency as reason for the slowdown since we store and compare the entire 50MB chunk within enclave memory [50].

Table 10.5: 50MB bulk data benchmark results.

Test mode	**Mean execution time**	**Standard deviation**	**Diff**	**Relative diff**
Regular interface	1.067	0.030	-	-
+ TAP forwarding	1.110	0.041	0.043	4.06 %
+ SGX forwarding	1.256	0.050	0.146	13.68 %
+ Trusted socket API	1.706	0.067	0.449	42.13 %
+ MACSec	2.032	0.066	0.326	30.58 %

10.3 Benefits of Outsource-and-verify

Beside of shrinking the trusted code base, outsource-and-verify may improve the compatibility of a TEE with existing legacy software. For instance, we applied an existing tool for deploying a firmware update through our S7COMM+ validation. With regard to TCP/IP stacks, the Linux ecosystem provides a rich facility of network configuration and network statistics tools. The outsource-and-verify approach takes advantage of these tools without moving the entire ecosystem into the trusted computing base.

Chapter 11

Conclusion

We presented a concept for binding a network interface to an SGX enclave. We demonstrated how outsource-and-verify for protocol stacks reduces the size and complexity of TEE implementations. We validate all network packets inside a TEE. Our validation code is significantly smaller than lightweight implementations of protocol stacks. We validate TCP/IP with fewer than 500 lines of code. Our evaluation indicates that the performance overhead of this approach is feasible. For example, our UDP benchmark has an overhead of 31.65% for the SGX packet validation and 15.26% for MACSec. Furthermore, we described the adoption of MACSec for SGX and identified security issues in MACSec. Finally, we addressed the problem of authenticating an enclave towards a trusted I/O device: Firstly, we improved the SGX-USB protocol of Jang [27]. Secondly, we presented a protocol that provides a symmetric mutual authentication for an SGX environment.

Bibliography

[1] PicoTCP. https://github.com/tass-belgium/picotcp. (accessed 2019-02-04).

[2] Ieee standard for local and metropolitan area networks: Media access control (mac) security. *IEEE Std 802.1AE-2006* (Aug 2006).

[3] Ieee standard for local and metropolitan area networks–media access control (mac) security amendment 1: Galois counter mode–advanced encryption standard– 256 (gcm-aes-256) cipher suite. *IEEE Std 802.1AEbn-2011 (Amendment to IEEE Std 802.1AE-2006)* (Oct 2011).

[4] Ieee standard for local and metropolitan area networks—media access control (mac) security amendment 2: Extended packet numbering. *IEEE Std 802.1AEbw-2013 (Amendment to IEEE Std 802.1AE-2006)* (Feb 2013).

[5] Ieee standard for local and metropolitan area networks – port-based network access control amendment 1: Mac security key agreement protocol (mka) extensions. *IEEE Std 802.1Xbx-2014 (Amendment to IEEE Std 802.1X-2010)* (Dec 2014).

[6] Ieee standard for ethernet. *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)* (March 2016).

[7] Ieee standard for local and metropolitan area networks–media access control (mac) security - amendment 3:ethernet data encryption devices. *IEEE Std 802.1AEcg-2017 (Amendment to IEEE Std 802.1AE-2006 as amended by IEEE Std 802.1AEbn-2011 and IEEE Std 802.1AEbw-2013)* (May 2017).

[8] Ieee standard for local and metropolitan area networks-media access control (mac) security. *IEEE Std 802.1AE-2018 (Revision of IEEE Std 802.1AE-2006)* (Dec 2018).

[9] Alder, F., Kurnikov, A., Paverd, A., and Asokan, N. Migrating sgx enclaves with persistent state. In *2018 48th Annual IEEE/IFIP Interna-*

tional Conference on Dependable Systems and Networks (DSN) (2018), IEEE, pp. 195–206.

[10] ARM, A. TrustZone. https://developer.arm.com/technologies/trustzone. (accessed 2019-02-01).

[11] Aumasson, J., and Merino, L. Sgx secure enclaves in practice. *Black Hat USA* (2016).

[12] Beresford, D. Exploiting siemens simatic s7 plcs. *Black Hat USA 16* (2011), 723–733.

[13] Böck, H., Zauner, A., Devlin, S., Somorovsky, J., and Jovanovic, P. Nonce-disrespecting adversaries: Practical forgery attacks on gcm in tls. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)* (2016).

[14] Brickell, E., and Li, J. Enhanced privacy id: A direct anonymous attestation scheme with enhanced revocation capabilities. In *Proceedings of the 2007 ACM workshop on Privacy in electronic society* (2007), ACM, pp. 21–30.

[15] Champagne, D., and Lee, R. B. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on* (2010), IEEE, pp. 1–12.

[16] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review 42* (2008), 2–13.

[17] Chhabra, S., Rogers, B., Solihin, Y., and Prvulovic, M. Secureme: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing* (2011), ACM, pp. 108–119.

[18] Costan, V., and Devadas, S. Intel sgx explained. *IACR Cryptology ePrint Archive 2016* (2016), 1–118.

[19] Costan, V., Lebedev, I. A., and Devadas, S. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium* (2016), pp. 857–874.

[20] Deering, S. E., and Hinden, R. M. Internet protocol, version 6 (ipv6) specification. RFC 2460, RFC Editor, December 1998. `http://www.rfc-editor.org/rfc/rfc2460.txt`.

[21] Dhar, A., Puddu, I., Kostianen, K., and Čapkun, S. Proximitee: Hardened sgx attestation and trusted path through proximity verification. *IACR Cryptology ePrint Archive* (2018).

[22] Dharmapurikar, S., and Paxson, V. Robust tcp stream reassembly in the presence of adversaries. In *USENIX Security Symposium* (2005), pp. 65–80.

[23] Dunkels, A. Design and implementation of the lwip tcp/ip stack. *Swedish Institute of Computer Science 2* (2001).

[24] Eskandarian, S., Cogan, J., Birnbaum, S., Brandon, P. C. W., Franke, D., Fraser, F., Garcia Jr, G., Gong, E., Nguyen, H. T., Sethi, T. K., et al. Fidelius: Protecting user secrets from compromised browsers. *IEEE Symposium on Security and Privacy* (2018).

[25] Harkins, D. Synthetic initialization vector (siv) authenticated encryption using the advanced encryption standard (aes). Tech. rep., 2008.

[26] Intel. Software guard extensions programming reference, revision 2, 2014.

[27] Jang, Y. J. *Building trust in the user I/O in computer systems.* PhD thesis, Georgia Institute of Technology, 2017.

[28] Joux, A. Authentication failures in nist version of gcm. *NIST Comment* (2006).

[29] Kleinmann, A., and Wool, A. Accurate modeling of the siemens s7 scada protocol for intrusion detection and digital forensics. *Journal of Digital Forensics, Security and Law 9* (2014).

[30] Kumar, A. *Active platform management demystified: unleashing the power of intel VPro (TM) technology.* Intel Press, 2009.

[31] Lei, C., Donghong, L., and Liang, M. The spear to break the security wall of s7commplus. *Black Hat USA* (2017).

[32] Levi, D., Meyer, P., and Stewart, B. Simple network management protocol (snmp) applications. STD 62, RFC Editor, December 2002. `http://www.rfc-editor.org/rfc/rfc3413.txt`.

[33] Maene, P., Götzfried, J., De Clercq, R., Müller, T., Freiling, F., and Verbauwhede, I. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers 67* (2018), 361–374.

[34] Matetic, S., Ahmed, M., Kostiainen, K., Dhar, A., Sommer, D., Gervais, A., Juels, A., and Capkun, S. Rote: Rollback protection for trusted execution. In *USENIX Security Symposium* (2017), pp. 1289–1306.

[35] McCune, J. M., Parno, B. J., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review* (2008), vol. 42, ACM, pp. 315–328.

[36] McCune, J. M., Perrig, A., and Reiter, M. K. Bump in the ether: A framework for securing sensitive user input. In *Proceedings of the annual conference on USENIX'06 Annual Technical Conference* (2006), pp. 17–17.

[37] Neumann, P., and Poschmann, A. Ethernet-based real-time communications with profinet io. *WSEAS Transactions on Communications 4* (2005), 235–245.

[38] Perrig, J. M. M. A., and Reiter, M. K. Safe passage for passwords and other sensitive data. In *Proceeding of the 16th Annual Network and Distributed System Security Symposium* (2009).

[39] Plummer, D. C. Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware. STD 37, RFC Editor, November 1982. `http://www.rfc-editor.org/rfc/rfc826.txt`.

[40] Postel, J. User datagram protocol. STD 6, RFC Editor, August 1980. `http://www.rfc-editor.org/rfc/rfc768.txt`.

[41] Postel, J. Internet protocol. STD 5, RFC Editor, September 1981. `http://www.rfc-editor.org/rfc/rfc791.txt`.

[42] Postel, J. Transmission control protocol. STD 7, RFC Editor, September 1981. `http://www.rfc-editor.org/rfc/rfc793.txt`.

[43] Rose, M., and Cass, D. Iso transport service on top of the tcp version: 3. STD 35, RFC Editor, May 1987.

[44] Ruan, X. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine.* Apress, 2014.

[45] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2017).

[46] Spenneberg, R., Brüggemann, M., and Schwartke, H. Plc-blaster: A worm living solely in the plc. *Black Hat Asia 16* (2016).

[47] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security Symposium* (2018), pp. 991–1008.

[48] Watson, P. Slipping in the window: Tcp reset attacks. *Technical Whitepaper* (2004).

[49] Weiser, S., and Werner, M. Sgxio: generic trusted i/o path for intel sgx. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (2017), ACM, pp. 261–268.

[50] Weisse, O., Bertacco, V., and Austin, T. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *ACM SIGARCH Computer Architecture News* (2017), vol. 45, ACM, pp. 81–93.

[51] Zhou, Z., Gligor, V. D., Newsome, J., and McCune, J. M. Building verifiable trusted path on commodity x86 computers. In *IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 616–630.

[52] Zhou, Z., Yu, M., and Gligor, V. D. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 308–323.