UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

BRUNO EDUARDO DE OLIVEIRA MENEGUELE

SKEEN: AMBIENTE DE EXECUÇÃO SEGURA DO KERNEL COM INTEL SGX

CURITIBA

2022

BRUNO EDUARDO DE OLIVEIRA MENEGUELE

SKEEN: AMBIENTE DE EXECUÇÃO SEGURA DO KERNEL COM INTEL SGX

SKEEN: Secure Kernel Execution Environment with Intel SGX

Dissertação de Mestrado apresentado como requisito para obtenção do título de Mestre em Engenharia Elétrica e Informática Industrial do Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná.

Orientador: Prof(a). Dra. Keiko Verônica Ono Fonseca

Coorientador: Prof. Dr. Marcelo De Oliveira Rosa

CURITIBA

2022

BRUNO EDUARDO DE OLIVEIRA MENEGUELE

SKEEN: AMBIENTE DE EXECUÇÃO SEGURA DO KERNEL COM INTEL SGX

Trabalho de pesquisa de mestrado apresentado como requisito para obtenção do título de Mestre Em Ciências da Universidade Tecnológica Federal do Paraná (UTFPR). Área de concentração: Telecomunicações E Redes.

Data de aprovação: 24 de Outubro de 2022

To my wife for my absence and her, always
present, support.

ACKNOWLEDGEMENTS

This project would not have been possible without the support of several people whose I devote my gratitude. Certainly I will not be able to call out all the names that helped me throughout this project, but all of them must know how thankful I am.

To my family for the affection and full support.

To my tutors Keiko Fonseca and Marcelo Rosa for guiding and helping me on conceiving this project.

For Red Hat coworkers with different ideas and clarifications, mostly: Rafael Aquini who helped me to better understand Linux memory model, Carlos Maiolino for helping with filesystem concerns and Luiz Gonçalves for always being willing to help with all sort of question and concerns I did not know where to start with.

And last, but not least, my laboratory colleagues Douglas Amaral, Alex Vernize and Filipe Lemos for countless bad (and good) ideas.

RESUMO

Intel SGX não é acessível do nível mais privilegiado de execução, conhecido como *anel zero*, onde o núcleo do sistema operacional está localizado. No entanto, é possível separar a responsabilidade de execução entre o núcleo e o espaço do usuário criando uma dependencia entre estes dois níveis que permite dados internos do núcleo de serem armazenados ou processados dentro de enclaves privados do SGX. Neste projeto é apresentado o SKEEN, uma maneira de isolar componentes e estruturas internas do sistema operacional utilizando o Intel SGX, previnindo vazamento de informação para diferentes componentes do mesmo sistema operacional. Uma prova-de-conceito é apresentada para exemplificar o uso desse projeto.

Palavras-chave: sgx; operating system; linux kernel; data privacy; data isolation.

ABSTRACT

Intel SGX is not accessible from the most privileged execution level, known as ring zero, where the operating system kernel is placed. However, it is possible to split the execution responsibility between kernel and userspace by creating a dependency among these two levels that allows internal kernel data to be stored or processed within SGX private enclaves. This project presents SKEEN, an enhanced way to isolate internal operating system components and structures with Intel SGX technology, preventing information leak to different components of the same operating system. A proof-of-concept is provided to exemplify its usage.

Keywords: sgx; sistema operacional; linux kernel; privacidade de dados; isolamento de dados.

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Security of Operating Systems (OS) is a broad subject, mainly when size and complexity of each of its subsystems (MCKUSICK; NEVILLE-NEIL; WATSON, 2014) are considered. OS security has been a trending topic between researchers and conferences throughout the world due to its importance on current business models leaning towards cloud computing solutions (ARNAUTOV *et al.*, 2016).

Cloud computing has its bases on different virtualization technologies, which in turn have their foundation built on top of features provided by the host hardware and the underneath operating system abstraction layers, like hypervisors and containers (Red Hat Inc, 2019). Both hypervisors and containers are susceptible to the underneath OS kernel vulnerabilities, even though their architectures differ: with OS components compromised, internal functionalities are also exposed to the attacker, giving it partial or full control over system resources, including the virtualization layer (European Union Agency for Network and Information Security, 2017).

Considering the intrinsic interaction between subsystems, a single vulnerability offers risk to many different points from the OS. In order to minimize the impact an attacker can have over the entire system, a concept named Trusted Execution Environment (TEE) was conceived. Originally, the TEE concept was built on top of simpler mechanisms and even older resource isolation concepts, like virtual barriers completely isolating user applications from the rest of the system, presenting considerable limitations with relation to its operation and applicability (Sabt; Achemlal; Bouabdallah, 2015).

In 2016, Intel released the first version of a x86 hardware-based TEE, the *Software Guard eXtension* (SGX) technology. By design, SGX defines a secure area (named *enclave*) that can be accessed only by the least privileged level processes (in other words, user applications, running on processor ring 3), preventing accesses from any code with higher privilege, like the operating system kernel (Intel Corporation, 2019b).

The overall idea is to prevent a compromised operating system to hijack userspace applications' confidential data running within SGX enclaves, enabling different solutions for today's business model of virtualization and mobile applications. As example we have recent smartphones with auxiliary secure processors implementing different TEE solutions, but with goals close to the defined by Intel SGX. Both Google Android (Android Open Source Project, 2022) and Apple iPhone (Apple Inc., 2022) operating systems make direct use of their processor TEE feature to handle cryptographic keys, securely store personal data and isolate data processing.

However, kernel threads can not process or store sensitive data inside enclaves, depriving any kernel subsystem from protecting its own data in case other subsystem gets compromised. With that in mind, this work describes SKEEN, a **Secure Kernel Execution ENvironment** architecture, that allows kernel subsystems to use the features exposed by Intel SGX. Its architecture offers a generic interface that can be extended by each subsystem to match their own needs. Such architecture has two basic components: (1) a built-in kernel module that defines the interface for internal subsystems, and (2) a userspace program that directly interacts with the SGX technology. The interaction between these two components allows data from the kernel flow through the userspace and then be processed inside SGX enclaves.

With this architecture many different mechanisms can be created aimed at data privacy and isolation, increasing the difficulty for an attacker to gather system information by simply reading runtime memory content or by tracking the data evaluation process. Our experimental evaluation of SKEEN has demonstrated how an internal kernel subsystem can perform a cryptographic operation, like cryptographic key generation, encryption and decryption, within an SGX enclave, despite its limitation of being accessible only by programs in the operating system ring three level. Two artifacts are worthy mentioning as results of this work, being (1) the code repository with all SKEEN's code base based on the Linux Kernel version 5.19[1] and (2) a peer-reviewed paper (MENEGUELE; FONSECA; ROSA, 2020) presenting an overview of the architecture presented in this work.

This work is organized as follows: Section 2 briefly presents projects with similar concepts; Section 3 proposes some use cases for the proposed architecture; Section 4 presents the entire discussion and decisions taken to build SKEEN; Section 4.1 presents few important topics for understanding the project; Section 4.2 describes the threat model and the assumptions made before conceiving this project; Section 4.3 depicts the system architecture; Section 5 presents the results this project achieved with the goals presented in Section 1.2 as benchmarks; Section 5.4 walks through future ideas to enhance the current state of this project, and Section 6 concludes with all achievements and considerations brought by this project.

1.1 Motivation

Looking at the range of applications that Intel SGX can be used with and the security guarantees it offers to common tasks that usual people has to perform everyday made we think about the number of common tasks an operating system has to perform and how we could leverage the same security guarantees that SGX was designed for at kernel level.

When turning kernel subsystems into "normal" users of other subsystems we can mimic the same behavior userspace applications have with the kernel, thus subsystem confidential data should still be confidential to the rest of the operating system.

After some time of research, we found the TresorSGX (RICHTER; GöTZFRIED; MüLLER, 2016) project, which had similar motivations to SKEEN, but with the goal of being a proof-of-concept for the concept of transferring data from the kernel to SGX, not considering thorough transparency or flexibility of its interface.

Hence, SKEEN was conceived from the idea of creating a generic interface, improving usability of SGX for any kernel subsystems. Also, following one of the suggestion from Tre-

[1] https://gitlab.com/radlab-utfpr/skeen-linux-kernel/-/tree/rebase-5.19 - SKEEN code on top of Kernel 5.19.0 tag.

sorSGX discussion topic, we have implemented on SKEEN a communication channel with less overhead by using shared memory instead of the Netlink API used by TresorSGX.

1.2 Objectives

SKEEN's project overall goal is to leverage Intel SGX functionality to operating system kernel subsystems, providing isolated execution and secure storage.

On the other hand, specific goals can be taken from the overall goal, being the established ones as follows:

1. Compare the suggestion brought by a similar project, TresorSGX (Section 2.1), with relation to the interprocess communication channel of choice: instead of using the Netlink API, use a simple shared memory to reduce communication travel time and overall overhead;

2. Suggest a generic interface to allow different kernel subsystems to use SGX enclaves as necessary;

3. Discuss the possibility of making the solution built-in to the kernel, thus allowing its usage on earlier stages of the kernel boot process or with other security related subsystems.

This work focused on creating a stable and flexible architecture where considerations and decisions that ease future works on the same topic were made, allowing additional ideas and features to be smoothly integrated.

2 SIMILAR PROJECTS

Work related to using SGX directly from Kernel code has not been really explored due to the general design of SGX, which is intended for userspace usage instead. However, TresorSGX is worthy mentioning due to its overall design and also the TEE subsystem available into the Linux Kernel mainline project.

2.1 TresorSGX

SKEEN was inspired on TresorSGX (RICHTER; GöTZFRIED; MüLLER, 2016) project, which, from an external perspective, has the same goal. However, the architecture implementation differs significantly when compared to the chosen interprocess communication mechanism, simultaneous request handling, UMH (*Usermode Helper*) interface usage, architecture design, interface extensibility, and other internal details.

These differences may vary depending on the case or workload being used for testing. However, it is worth to mention a major difference: the interprocess communication protocol with Netlink.

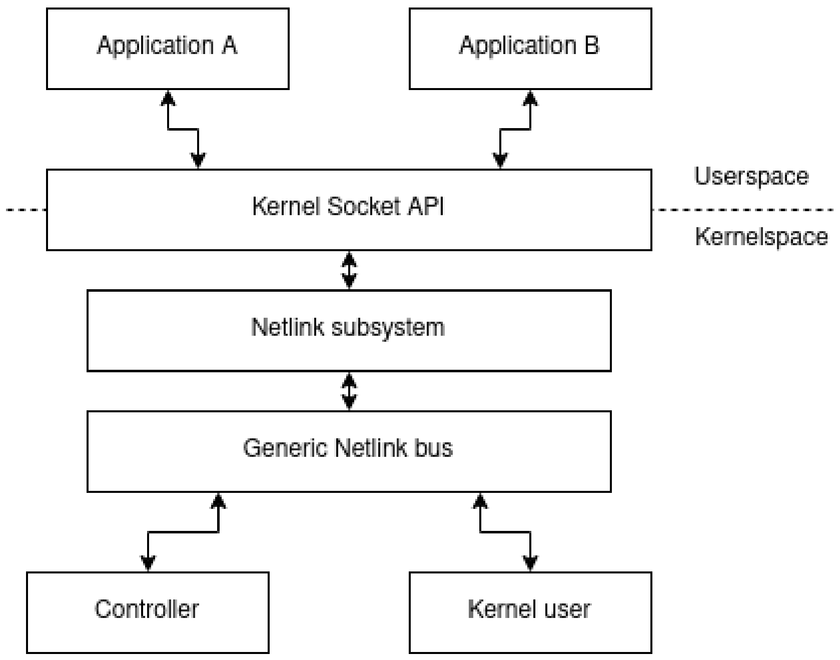
2.1.1 Netlink

To the IPC (*Interprocess Communication*) layer TresorSGX made use of the well-known kernel Generic Netlink interface, which from userspace the *libnl* (Netlink Community, 2022) was used as library. Netlink is a generic protocol that allows domain-specific IPC protocols to be created as needed, allowing multiple level of message headers, payloads and attributes in a single transmitted message. It was specially designed to support communication between userspace applications, user to kernelspace and event/notification driven notifications from kernel to userspace (Thomas Graf, 2011). Figure 1 depicts the Generic Netlink stack: the *controller* is part of the Netlink itself and is considered an special user responsible for dynamically allocating other Generic Netlink communication channels and performs different management tasks.

However, Netlink's flexibility also increases its usage complexity and overall protocol overhead due to the number of checks and possible choices that interface may have to do in order to retrieve the requested data. The overall performance hit might not be noticeable in scenarios where high data rate is not a concern, but when transferring data of arbitrary size to an encrypted block volume, as tested by TresorSGX (RICHTER, 2016), data throughput decreases significantly.

From 2013 to 2016 the Linux Kernel had an specific Netlink interface rooted in the *mmap* system call instead of the default BSD socket data transmission method, which allowed subsystems requiring bigger data throughput to use it. However, due to difficulties on maintaining both

Figure 1 – Generic Netlink API stack. Based on a Kernel documentation (Kernel.org, 2017a).



Source: Own authorship (2022).

Netlink interfaces compatible and also further improvements on the original Netlink, the variant using *mmap* operations was deprecated and later removed from Linux[1].

2.2 Linux TEE subsystem

A specific TEE subsystem was integrated into Linux Kernel mainline (Kernel.org, 2017b) project to officially support different TEE solutions across the industry, but only those based on a Trusted Application (TA) running on top of a trusted operating system.

Two different solutions are currently supported: the OP-TEE and the AMD Secure Processor. The first is an architecture on top of the ARM TrustZone (ARM, 2021) that specifies how userspace, kernel and the TA communicate between them. The second is the proprietary solution used on AMD CPUs with a separate chip holding the whole secure environment: both the trusted OS, firmware within the chip's memory, and the TA loaded at runtime by a kernel driver.

SGX, on the other hand, enables userspace applications to create private and encrypted memory regions for both code and data, which ran in the same processor chip directly, without an intermediate component.

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d1b4c689d4130bcfd3532680b64db562300716b6 - Commit removing Netlink through MMAP operations.

2.3 Library Operating Systems and Unikernels

Library Operating Systems (LibOS) and, also, Unikernels are part of a class of operating systems where the application is directly linked with the operating system functionality, being that the application and the kernel are deployed and executed as a single unit (binary), removing many different usual kernel features that are not used by the application and, thus, increasing data throughput and decreasing deployment delay.

LibOSes and Unikernels, although quite similar, they handle application execution privilege differently: an application running with a LibOS has the needed kernel features running at userspace mode (ring 3), while on Unikernels the application is linked and runs at kernel mode (ring 0) (TAN *et al.*, 2020). With that, applications on Unikernels are not suitable for SKEEN, since there is no userspace for accessing SGX enclaves.

At Intel SGX SDK website (Intel Corporation, 2022) many different LibOSes are mentioned as possible solutions for an application willing to use Intel SGX feature. However, these OSes follow the standard SGX application specification, where the application confidential data completely bypasses the underlying kernel and are stored and processed directly into an SGX enclave. With that, these solutions are not considered as possible alternatives to the SKEEN architecture. However, using SKEEN with a LibOS is an interesting path for future work.

3 USE CASES

With the ability of both processing and storing information within SGX enclaves, many different kernel subsystems may employ such mechanism to secure their sensitive data.

3.1 Firmware TPM

Trusted Platform Modules (TPM) are known as secure processors to store, process, and also generate cryptographic sensitive information such as user asymmetric key pairs. These platforms are deployed in different range of systems, from embedded devices to personal and server computers. In general, there are three different modes of implementation for TPMs (RAJ *et al.*, 2016):

1. Dedicated: a real and small dedicated hardware implementation, delivered as a single microchip usually soldered to computer's motherboard;
2. Integrated: part of a different component on the motherboard, such as the *chipset* (also known as *Platform Controller Hub*, PCH), or built-in to the processor's silicon dice;
3. Firmware: software implementation into platform's TEE.

Although a TPM is usually meant to be used as the root of trust for the entire system - from its bootstrap to its runtime, defining the system *chain of trust* - and in *firmware* mode it would only be available once the TEE is fully initialized, it still has a valid usage beyond the platform source of trust: cryptographic operations ranging from hashing to digital signing and data sealing (Trusted Computing Group, 2019).

TPM features are used by internal kernel subsystems, such as *Integrity Measurement Architecture* (IMA) and *Extended Verification Module* (EVM) (IMA Project, 2020), through the concept of *encrypted* and *trusted* keys (Kernel.org, 2020) from the *Key Retention Subsystem* (KRS), to ensure system integration during normal operations at runtime.

3.2 Kernel Internal Structures

Some kernel structures are referenced only in specific moments and may contain sensitive information, like cryptographic keys managed by the *Key Retention Subsystem*. These structures could be stored and/or processed within the TEE, without exposing its real content to userspace or kernel, noticeably increasing the difficulty for an attacker to get access to the data. Also, using SKEEN with the *Audit* subsystem and the internal kernel structures stored into an SGX enclave, enables the possibility to self-audit the kernel at different times during system's life cycle and, thus, increasing the system chain of trust.

4 METHODOLOGY

This work implements a new Linux mechanism to achieve the objectives presented in Section 1.2, being that most - but not all - tools used for building it were already present into kernel's current version[1] code.

In the following sections both the needed background and the architecture itself are presented focusing on the overall concepts and ideas. Detailed information on their internals are depicted in the Appendix A.

4.1 Background

This section presents the two most important concepts needed to understand SKEEN's overall architecture. Minor concepts are presented alongside their mention throughout Section 4.3.

4.1.1 Software Guard Extension

Intel Software Guard Extension (SGX) is a new instruction set with a conjunction of architectural data structures that allow userspace applications to ensure the confidentiality and integrity of sensitive data, even if any privileged software (operating system, hypervisor, or BIOS) is compromised (ARNAUTOV *et al.*, 2016). The two guarantees offered by the SGX are:

- *Confidentiality*: any data or process state handled within the trusted environment cannot be observed by another system component; only the input and output are observable.
- *Integrity*: system components, external to the trusted environment, cannot change internal process behavior or content.

The protection against irregular accesses, from malicious privileged software to standard direct memory access (DMA), is guaranteed by hardware-assisted memory access control mechanisms in conjunction with several metadata stored in different architecture data structures dedicated to the SGX functionality. This control creates secure areas in the main memory known as *enclaves*.

Enclaves are stored within the userspace application virtual memory, restricting the ownership of each enclave to a single process. An enclave holds a variable number of memory pages (to store user application trusted data and code) in a structure called Enclave Page Cache (EPC). Although each page within EPC has a fixed size of 4kB and initially is allocated within the system cache, they may be evicted to the main memory as any other regular memory page. Therefore, applications demanding a large amount of pages are not restricted to the EPC maximum size.

[1] This work was built on top of the Linux kernel version 5.19.

Whichever EPC page is evicted to memory, it will be encrypted by the Memory Encryption Engine (MEE), ensuring the confidentiality and integrity of that data from any read or write attempt (MC-KEEN *et al.*, 2013).

The memory access control is done by the processor with some additional information contained into the Enclave Page Cache Map (EPCM): each entry on this structure has an attribute mapping to a single page within EPC. These additional information are: page type, access (read, write and execute) permissions, validity, and so forth, all data are used as filters to the access control engine.

The SGX instruction set, added to 6th Intel processors generation and onward, is divided in three main mnemonics, with several underneath leaf functions. These main mnemonics are: ENCLS, ENCLU and ENCLV (Intel Corporation, 2019b). As can be seen from above, the core **ENCL** three suffixes are:

1. **S**: stands for *supervisor*, meaning that all leaf function can be executed by privileged software, like the operating system kernel, generating an software exception in case it is issued by any other software beyond ring 0 privileged level;

2. **U**: stands for *user*, giving the right to the user applications to issue any leaf function under this category. In case any privileged software issues a function belonging to this set a software exception is raised, blocking further execution;

3. **V**: stands for *virtualization*, being used as support for VTX technology, Intel processor virtualization extension (Intel Corporation, 2019a).

In short, supervisor functions are restricted to privileged software and are used to manage the underlying enclave control structures, regarding enclave creation, initialization and maintenance. User functions are related to memory access within enclaves by user applications.

That is the reasoning behind the general goal of this paper: to create an architecture that enables ring 0 software to *indirectly* interact with Intel SGX feature, allowing sensitive data to be held and processed within such enclaves to protect OS sensitive data from its own internal components.

4.1.2 Linux Usermode Helper

A user program can interact with internal kernel functionalities via system calls, which have their own special meanings and calling arguments. Sometimes a call from inside the kernel to a userspace program is needed, for example, when a new device is attached to the machine and the kernel requests a specific userspace application to load a device driver as soon as the device gets recognized.

This process is done through the *usermode helper* API (UMH) (M. Jones, 2010), which is a kernel API that enables kernel code to invoke userspace applications on demand. There are a couple of ways to actually execute userspace programs:

1. **Direct path**: the simplest way is to call a binary located in a well-known path, possibly passing some program and environments arguments through API specific structures;
2. **In-kernel binary**: the binary is physically located into kernel memory, which was statically compiled during kernel building time and executed as a user process when requested.

Caution must be taken when using the second approach: the binary may be executed before any real filesystem was effectively mounted in the system, preventing any dynamic linkage of shared libraries on such binary. Because of that and other possible side effects the in-kernel binary must be statically compiled.

The user process created to run the program receives superuser privileges, thus it has full control over system configuration.

By default, the *in-kernel binary* approach creates a interprocess communication channel between the kernel and the user program using pipes. Section 4.3 details about the UMH interface implementation, which was enhanced with shared memory handling code in order to improve overall performance.

4.2 Threat Model and Assumptions

The primary software assumption is that the overall system is not trusted and is possibly compromised by a malicious user - including the operating system kernel (consequently, the SKEEN itself) and the entire userspace environment - and therefore all components are treated as hostiles. In case of any kernel subsystem gets compromised, any subsystem data or algorithm implementation already stored within a SGX enclave must not be accessible.

Although the runtime kernel is not trusted, its compilation and code are considered sane and trustworthy. Consequently, no known security holes or backdoor are intentionally added to the code.

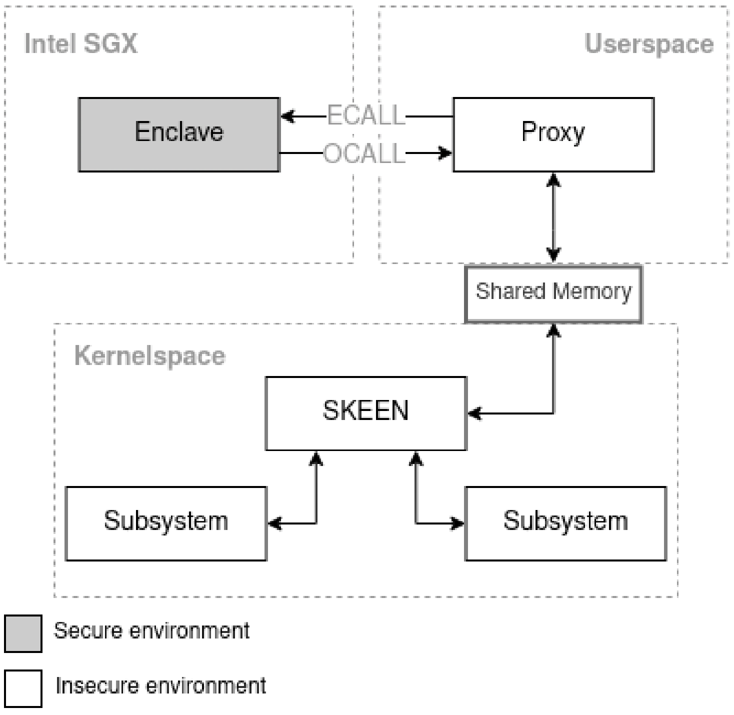
Deny of Service (DoS) attacks can be performed in many different ways, preventing any SKEEN service to run. With that said, handling DoS attacks is out of scope for the current state of this project.

Cache timing attacks that can affect SGX, for instance, L1TF (L1 Terminal Fault) (Intel Corporation, 2018), are not checked for their presence, but we assume their respective mitigation are applied if necessary. Other side-channel attacks (SPREITZER *et al.*, 2018), like power analysis or any other with hardware access level are also out of scope of this project.

4.3 Architecture

Figure 2 presents an overview of the SKEEN architecture proposed in this project and used as the basis for further discussion. This figure can be exploded in order to observe the layered design of the architecture as shown in Figure 3. This design was used to accommodate differences between each client subsystem, allowing specific behavior handling in all three execution environments: kernelspace, userspace and SGX enclave. Another aspect to be noted is the common core, which behaves as the arbitrator for the whole architecture.

Figure 2 – In the insecure side of the platform the SKEEN kernel module is used as the interface for the underneath kernel subsystems to interact with the userspace program, that is launched to every new subsystem request, through a shared memory IPC scheme. The data is then insecurely forwarded to secure SGX enclaves where the data is finally protected against eavesdropping and malicious modification.



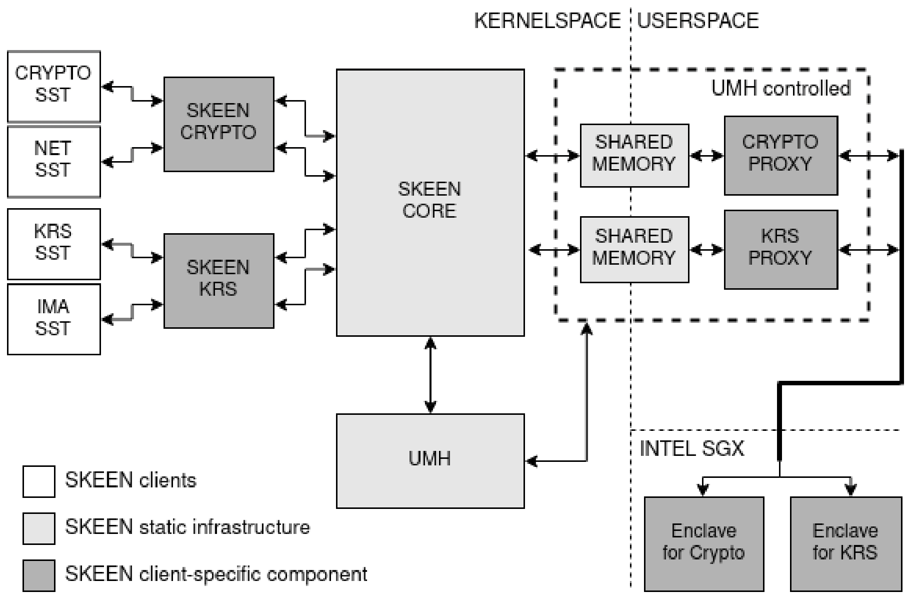
Source: Own authorship.

In Figure 3 the client subsystems can be described as Cryptography (Crypto), Network (NET), Key Retention System (KRS) and Integrity Measurement Architecture (IMA), whose purposes and behaviors are out of this document's scope.

To further clarify what is presented in Figure 3, NET subsystem makes directly use of the cryptographic subsystem, while IMA makes heavy use of KRS subsystem; being the reason why *Crypto* and *KRS* were chosen as the abstraction layers on SKEEN side. With that, when the abstraction layer is called from within kernel's subsystems a reference is maintained in order to listen the responses coming from the userspace program.

SKEEN's data flow will be further explored and explained in Section 4.3.5.

Figure 3 – Layered view from clients in the kernelspace to the Intel SGX enclaves, where the core acts as the main component and communication arbitrator between kernel and userspace. Each client-specific component has its relative counter part in both userspace and within Intel SGX, creating the code and data isolation between clients.



Source: Own authorship (2022).

4.3.1 Userspace Program (*Proxy*)

To allow better isolation between subsystems operating with SKEEN, each initialization request made launches a new *proxy* application with a unique process memory and a unique shared memory region (not shared to any other subsystem). Once all transactions are finished, the client can request to SKEEN to terminate - by freeing and zeroing any memory region allocated - the proxy program and other structures held within the architecture used to manage each operation context.

The proxy program is statically compiled against any external library, thus dynamic linkage attacks are not feasible. At the same time, the program is placed directly within the kernel image.

Also, the proxy has a reactive behavior, where it only responds (send data to kernel) when it is requested to (via request coming from the kernel). Every proxy response is tied to a single kernel request.

4.3.2 Interprocess Communication

Shared memory was the chosen approach to be used on SKEEN IPC mechanism, employing near-zero overhead in its raw format. However, this approach has two issues: data synchronization and access control, demanding architectural solutions to manage them, which are shown in the sections below.

Data Synchronization

When using shared memory for exchanging data among different processes a synchronization model must be implemented. Our strategy consisted on not sharing the same memory region among different subsystems, preventing management logic for different subsystem's requests. Also, each half of this memory is strictly used by each flow direction: upwards for data coming from the kernel to the userspace, denoted by *request*, and downwards for data in the other direction, denoted by *response*. These data objects have fixed size and are composed of different fields[2].

The memory division and the fact that the proxy program operates reactively to kernel requests, help to mitigate data concurrency issues. Thus, each side of the channel cares only to not exceed the buffer size and to signal which message was already handled. Also, requests and responses are handled sequentially, meaning that data ordering is kept. In the current state of this project, the communication protocol does not support parallelism.

Proxy Access Control

The shared memory mechanism itself does not imply any access control to the memory, the proxy program benefits from the memory space isolation and ownership already imposed by the default memory management behavior, however, any concurrent kernel thread with access to the list of running processes and their respective structures can have access to the shared memory. In the current state of this project, it is still unknown the mechanism to control memory access from concurrent kernel threads. This topic is revisited in Section 5.4.

At userspace level, a specific character driver was implemented to handle the shared memory mapping and to verify the validity of process' access request: only processes created through SKEEN's UMH mechanism has access to their respective shared memory, any other external process trying to dig or traverse the memory will be denied.

4.3.3 Kernelspace Module

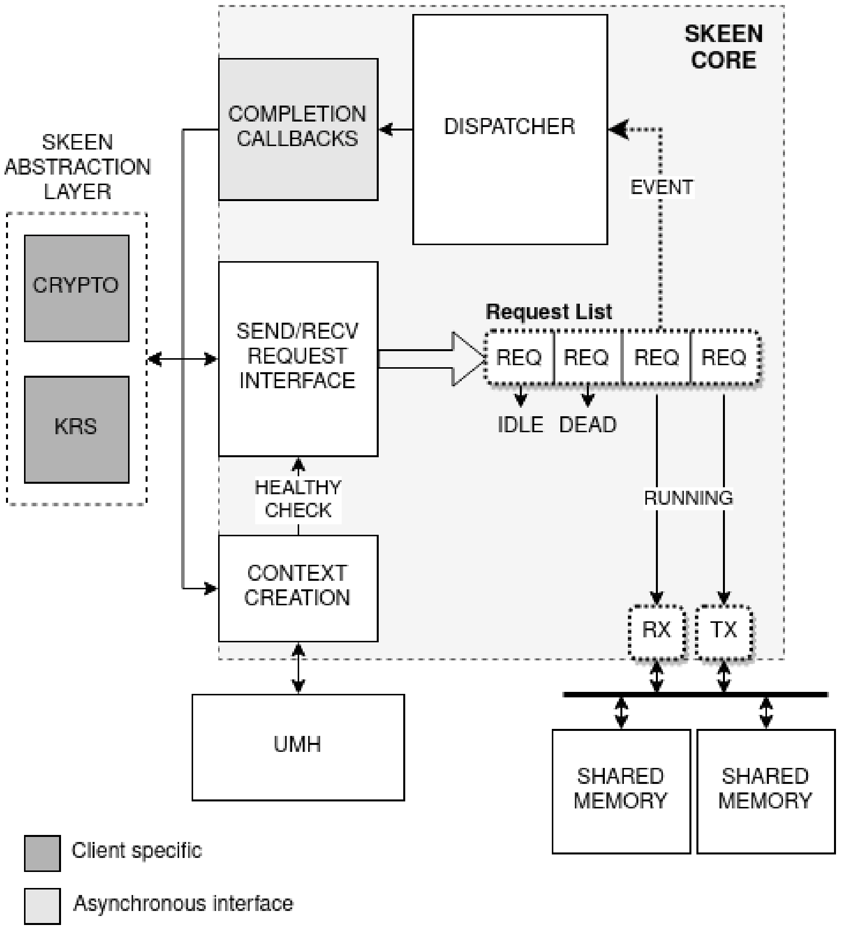
The kernel module contains separate components that are responsible for enabling and launching the userspace program, the communication channel and the interface exposed to subsystems willing to use SGX features.

SKEEN core component acts as an arbitrator between the different subsystems creating requests and then listening for userspace's program responses with the data processed inside SGX enclaves. This in-between component, with structure as presented in Figure 4, was designed to be as transparent as possible, hiding the entire bookkeeping, data tracking, and IPC mechanism that guarantees that data reaches its destination. Also, considering that each clients

[2] Implementation details regarding the communication protocol is presented in the Appendix A.1.3

can make use of different structures and expects different data types, the core exposes an extendable interface that supports an additional subsystem-specific abstraction layer as a plugin. Therefore the proposed solution is flexible enough to allow each clients specificity to be handled and maintained as a separated module.

Figure 4 – The *core* coordinates different instances of clients-specific components and also manages the data flow from both up and downwards directions with a work queue holding operation requests in different states. clients wait on completion callbacks that are triggered by the dispatcher upon request state change events.



Source: Own authorship (2022).

The only object that is shared among core and clients-specific components is the *context*, which is the structure used to perform all bookkeeping and the aforementioned tracing. This object is initialized before any request is created and lives until the clients explicitly destroys it when the responses to the requests are sent from userspace. A more detailed explanation of the data flow throughout the architecture will be given in Section 4.3.5

Request List

The **Request List** shown in Figure 4 is a standard *linked list* holding the request created by kernelspace's subsystems willing to start a communication with userspace. This list is maintained by the SKEEN *core* and is used by the *dispatcher* to keep track of requests' life cycle.

Request Life Cycle

Each request keeps an internal state to indicate in which phase of its life it currently is. There are five different states:

1. **DEAD**: The request can be completely deleted from the request list by the dispatcher;
2. **IDLE**: The request was just created and is awaiting any action from the kernelspace subsystem;
3. **RUNNING**: The request was built and is ready to be sent or receive a response for it: it is said that the data flow is running;
4. **RECVD**: A response was received from userspace;
5. **SENT**: The request was sent to the userspace.

On every state change, an event is generated to wake up the dispatcher thread, which is responsible for taking the next action on the respective request. Since the dispatcher is a single thread handling all requests in the request list, the transient states **IDLE** and **RUNNING** are important to allow the costumer to perform any action before effectively executing any action with the request; it is also helpful on multiprocessor scenarios, where the number of requests may increase rapidly alongside the number of state changes.

Dispatcher

The dispatcher is a simple and dormant component within the SKEEN core, acting upon events caused by request state changes. Whenever a request changes its state, the dispatcher is responsible for checking the actual request state and execute the required action on that request. Currently, the dispatcher has only three actions to perform, depending on the state, being them:

1. Execute the costumer subsystem's abstraction layer callback responsible for handling requests just sent (`sent()`);
2. Execute the costumer subsystem's abstraction layer callback responsible for handling responses received from the userspace (`recvd()`);

3. Delete the request from the request list once the costumer subsystem has finished with it (by calling `destroy()`).

The whole request list is checked whenever the dispatcher wakes up by an event. However, in case a new event is generated while the dispatcher is already awaken, it will only restart the state verification once the current list checking is finished.

4.3.4 SGX Enclave

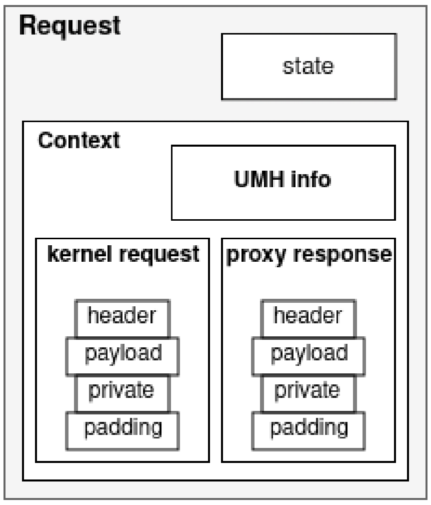
The trusted program, running within the SGX enclave, is built alongside the proxy program, since it is client-specific, and because both must be aware of each other existence and the interface being exposed. The proxy program makes *ECALLs* to functions from the trusted side, while *OCALLs* are performed in the other way.

Likewise the proxy program, the trusted portion is also built-in to the Kernel image and loaded at runtime into SGX's enclave, ensuring its validity and integrity from build time.

4.3.5 Data Flow

Due to the layered architecture, the data objects are also required to be wrapped in layers to keep it transparent throughout the processing chain. Figure 5 depicts the core data components from the subsystem operation request to the highest level of abstraction.

Figure 5 – Kernel module layered data view. The deeper inside the data is, the closer to the subsystem requesting data process on Intel SGX.

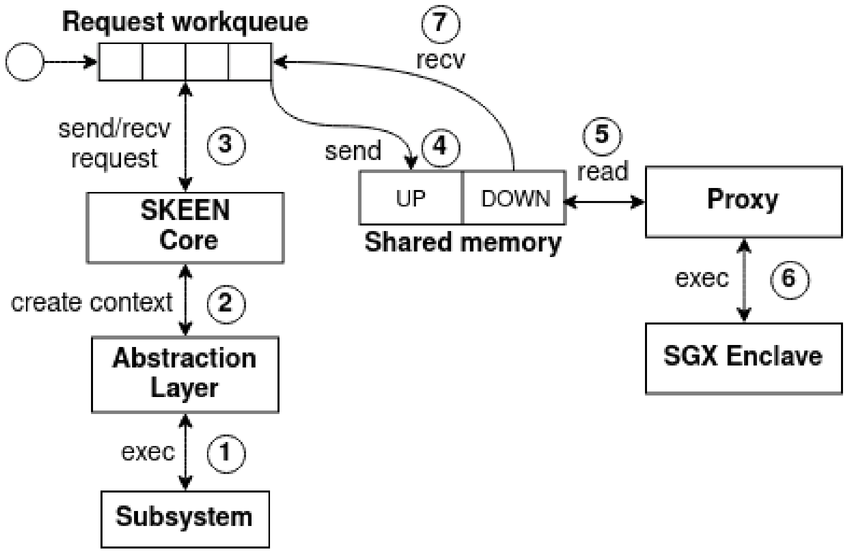


Source: Own authorship (2022).

The *context* is the core data transferred between kernel components, since it holds both the userspace program information, *UMH info*, and both the request data to be processed in userspace and its response. SKEEN core wraps the *context* in another *request* abstraction to maintain and trace the state of that specific flow, thus it has the ability to destroy it when requested.

Figure 6 depicts the data flow through the architecture using a generic abstraction layer (GAL) for a subsystem as an example.

Figure 6 – Data flow throughout the SKEEN layered architecture considering a test case with a cryptographic abstraction layer.



Source: Own authorship (2022).

A subsystem first calls a common operation, *exec* ①, as it would normally do if the actual interface is used, and not an abstraction layer to SKEEN; then a *context* is created ② by the *core* and kept within GAL for possible further operations and for matching the response code coming from userspace. The *core* then wraps the *context* content, as shown in Figure 5, with enough data to trace it in a work queue, and launches the *send* (or *receive* depending on what the *exec* call actually does) operation ③ that moves the most basic object (*kernel request*) from within the *context* through the IPC channel ④. The userspace program, polling the shared memory, notices a non-processed request is waiting in the shared memory, and then reads ⑤ the request by checking its internal content in order to select the correct ECALL that matches the *exec* operation to be performed within the SGX enclave ⑥. Once the process is completed an OCALL, from the enclave to the userspace program, is performed, returning the response value. The returned value is then wrapped in the object *user response* and send back to kernelspace through the IPC channel. In the kernel, once the *user response* data is available in memory, an event triggers the *receive* operation ⑦, waking up the *dispatcher* component in the core, as seen in Figure 4. The *dispatcher* evaluates the overall state of that request and execute the callback placed in the GAL code. A final check or processing might take place before returning the response value to the subsystem.

5 RESULTS AND DISCUSSION

We used a cryptographic algorithm driver as the test case for validating the entire architecture behavior. A cryptographic driver exposes an interface of allowed operations to be performed with specific algorithms. Our test driver wraps the AES algorithm, which uses an implementation from within the SGX enclave instead of using the existent kernel implementation. Therefore, the cryptographic operations are guaranteed to be isolated and confidential.

The abstraction layer follows the specification for registering a driver in the kernel and, at the same time, implements the wrapper functions that will send data to be encrypted or decrypted through the SKEEN infrastructure. In this way, any other code directly requesting AES encryption/decryption operation can normally use the generic in-kernel cryptographic interface, while our driver handles the translation between crypto data structures and SKEEN structures. We present in Appendix A.3 the details used in our cryptographic abstraction layer.

Another important aspect to be noted is that both the abstraction component and the userspace program must be aware of specific data requirements that an AES driver demands, such as: data size bigger than the algorithm block size are split in different chunks of fixed size. With that, both sides of the channel must have a common way of handling it. Situations like this may force the subsystem-specific components to handle fragmentation in somewhat non-trivial ways, possibly creating different internal fields for both request and response, for instance, on header, payload or other data field.

5.1 Security Implications

The userspace program's code can be tampered only in a short time window from within the kernel execution thread, meaning that the kernel as a whole must be already compromised by a malicious attacker. However, it is important to note that once the program is fully loaded into userspace's memory and is in normal operation, standard vulnerabilities can still be exploited by a malicious user. With that, keeping the userspace program as simple as possible for improving its auditability is important.

With the SKEEN userspace program being built into the kernel image, integrity checks can be added at early boot stages of the system, possibly opening a new set of features related to the system's chain of trust when leveraging SKEEN with other mechanism such as *Secure Boot*, *Integrity Measurement Architecture* or any other Linux Security Modules handling the system's chain of trust.

Another aspect to be considered in the current state of the project is the fact that the data being transmitted between kernelspace and userspace is not encrypted, meaning that after the data is copied from SKEEN's shared memory into userspace's program memory its confidentiality can be compromised upon runtime vulnerabilities.

5.2 Performance Comparison

One of the goals of this work, presented in the Section 1.2, is to implement a interprocess communication mechanism different from the one used in the project TresorSGX. On their work (RICHTER, 2016), the suggestion was to use a standard shared memory to store the data from kernel and userspace to be used with the proposed architecture instead of the well-known kernel-userspace Netlink IPC protocol; the major assumption was that the overhead required by the protocol was considerably delaying the data transference, and thus, limiting the architecture usage.

Since the SGX SDK used in both SKEEN and TresorSGX projects are the same, the data transfer rate and protocol between userspace and SGX are also the same and, thus, was not considered or measured. However, we were able to measure a considerable difference with relation to the interprocess communication channel.

Considering we are not measuring SGX data exchange, the comparison could be done in a system without Intel processor: the machine used was an AMD Ryzen 5 3600X 6-Core Processor with 16GB of DDR4 3200MHz RAM memory running Fedora 36 Linux distribution.

5.2.1 Architecture Bootstrap

Generally speaking, the bootstrap performance is not important to this work's scope. However, it is important to mention the findings during the comparison.

In both SKEEN and TresorSGX the in-kernel core component is first loaded and initialized, but the userspace application has a bootstrap phase where the memories and structures for establishing the communication channel with the Kernel is required. On TresorSGX the required bootstrap includes a structure allocation and declaration and a standard socket creation for the `AF_NETLINK` address family, being all cheap operations to the system; since TresorSGX's architecture was strictly built to work as a loadable kernel module, the userspace application has all system's features at its disposal at initialization time.

On SKEEN, on the other hand, the userspace application is required to wait until `/dev/usermode` is available to be used, which also depends on the kernel device initialization process, which takes, on average $120ms$ (milliseconds). Once the character device is ready, SKEEN userspace program needs to open and map the shared memory into its own memory space, but both operations are also cheap to the system.

Adding up all the requirements for the bootstrap phase, we come up with the Table 1 conclusion. The average time was taken from 20 separated runs.

Table 1 – Bootstrap time for both TresorSGX and SKEEN projects, summing all requirements for getting the userspace application running.

Process	Average Time
TresorSGX	32.090μs
Structure allocation and initialization	32.090μs
SKEEN	120.032ms
Time waiting on `/dev/usermode`	120.016ms
Shared memory open and mapping	15.990μs

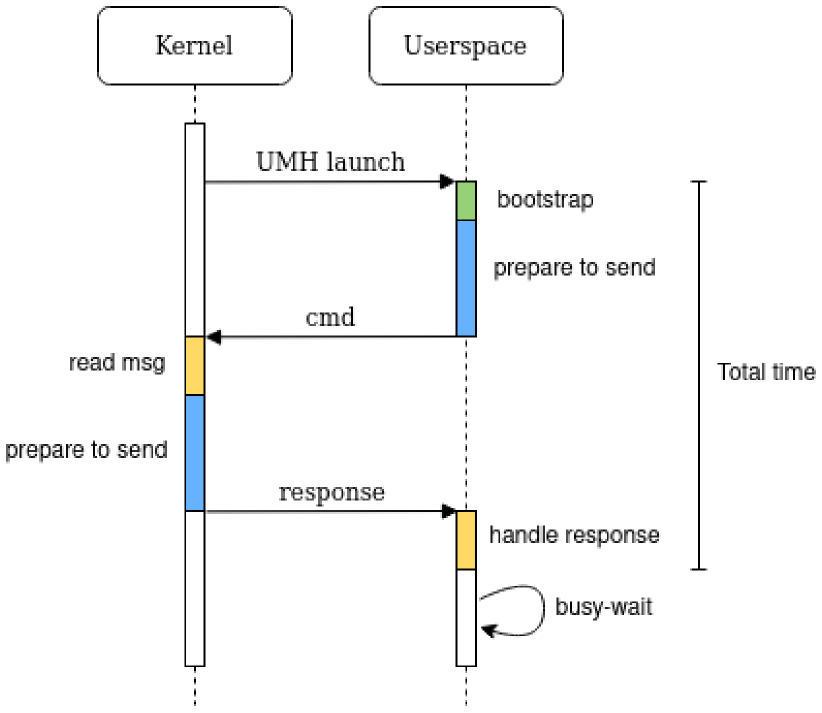
Source: Own authorship (2022).

5.2.2 Interprocess Communication

The idea of using a simple shared memory was to create a specific protocol to suffice SKEEN's requirement, leaving the generic aspect of Netlink API aside, improving the overall performance by decreasing the overhead needed to maintain protocol's flexibility.

To measure both SKEEN and TresorSGX IPC mechanisms timing we need to consider their particularities regarding requests and responses, since they do not follow the same protocol or message ordering and, for that, we present in Figures 7 and 8 their respective sequence diagram.

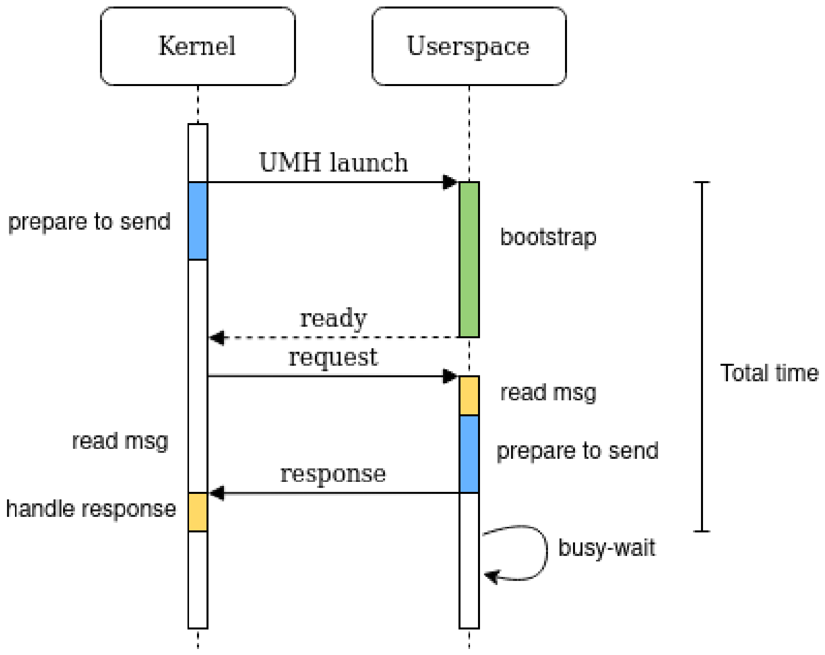
Figure 7 – TresorSGX data flow used to time the Netlink IPC mechanism.



Source: Own authorship (2022).

To measure the time spent in each function or code portion in each userspace program we used the `clock_gettime()` function from the standard C timer library with the `CLOCK_MONOTONIC` clock type to avoid disconnected jumps in the system time and also to do not depend on specific CPU timers. For the kernel portion, the *dynamic ftrace* tracing

Figure 8 – SKEEN data flow used to time the Shared Memory IPC mechanism.



Source: Own authorship (2022).

method (Steven Rostedt, 2017) was used; *ftrace* allows the user to trace internal kernel function behaviors and timing, from which CPU is running a certain function to when a certain event was generated. To setup the trace points to TresorSGX the user can make use of *ftrace* interface at *sysfs* at system's runtime, since the kernel and userspace application can be executed anytime after the system is up. However, for SKEEN, early boot tracing must be enabled, which can be done by setting specific kernel command line at boot time. The results for the average time are shown in the Table 2.

Table 2 – Interprocess communication time for both TresorSGX and SKEEN projects. The clock being used by *ftrace*, *CLOCK_MONOTONIC*, could not measure SKEEN kernel *send* operation; a higher resolution clock is required. At the same time, the number might be lower then 1 microsecond, not adding much to the final time.

Process	**TresorSGX**	**SKEEN**
Userspace send message	56.286μs	1.122μs
Userspace receive message	1.848ms	1.876μs
Kernel send message	1.243μs	N/A
Kernel receive message	2,624μs	1.425ms
Total	**1.911**ms	**1.428**ms

Source: Own authorship (2022).

One important note to the results presented in Table 2 is the fact that SKEEN's kernel *receive* operation operates in a polling approach, waiting on the userspace program to fill the shared memory space with a response, increasing the number of wasted CPU cycles and increasing the time presented in the table. And on TresorSGX userspace read operation the assumption for the time spent is related to the number of message data lookups in different header layers.

In average, the SKEEN's shared memory IPC mechanism shows 25.27% better performance than Netlink IPC mechanism. Although it does not imply an overall architecture better performance, due to other design choices, it indicates that in the exact same scenarios SKEEN can perform better up to this percentage, encouraging further studies and investigations.

5.3 Difficulties found

During the development of this work important obstacles had to be handled, otherwise the conclusion of it would not be possible. In the following subsections these difficulties are going to be detailed.

5.3.1 Static library

The choice of enabling SKEEN as soon as possible in the Kernel loading/booting cycle forced the entire architecture to be implemented as a built-in feature, instead of a Loadable Kernel Module, as it was done in TresorSGX (RICHTER; GöTZFRIED; MüLLER, 2016). Because of that, the userspace program is loaded before the *rootfs* is fully loaded, requiring it to be statically linked against every necessary library, including those shipped with the SGX SDK being used, which are only shipped as shared objects.

Mangling kernel build system with userspace options showed to be really difficult, to the point the project could not be tested to its full extent: a full encryption and decryption cycle, being applied to a real scenario, like done in TresorSGX with block volume encryption, was not tested. However, considering the good results regarding IPC performance 5.2.2, further investigation on different SGX SDKs and kernel build system improvements shows to be worthy.

5.4 Future work

Shared memory is the mechanism chosen as the IPC mechanism between kernel and proxy programs in the current state of this project, however new IPC mechanisms are proposed to upstream Linux kernel community regularly for many different use cases (BROWN, 2011). With that in mind, deeper research on new IPC mechanisms or improvements on well-known ones (NetOS Group, 2019) is a tackle point for future enhancement. Also, a research on how to control memory access from kernel components to the SKEEN spawned shared memory is being performed.

Integrating SKEEN to early boot security mechanisms could also improve the overall security of the system and enhance the SKEEN security scope: leveraging Intel TXT (Intel Corporation, 2019c) technology in order to gather pre-boot platforms measurement values (BIOS/UEFI, Chipset, and others) enabling a more robust chain of trust for the entire system. Also, enabling an early handshake between SKEEN and SGX to establish the key to encrypt the data being transmitted through the shared memory with the proxy program greatly enhances overall architecture security.

Following the same thought of booting a system with possible malicious components, the proxy could have its hash measured and verified before effectively executing it with the help of the IMA (Integrity Measurement Architecture) Linux subsystem; the enclaves could also be attested before being actively executed. Also, the proxy program can apply the usage of a sandbox mechanism like *seccomp* (KERNEL.ORG, 2019), preventing any not allowed system calls to be performed in case it gets compromised.

As previously mentioned in Section 3, using SKEEN for implementing a software *Trusted Platform Module* which can not be tampered with when the kernel is compromised, can mitigate security concerns related to default software TPM implementations/emulators.

Another point to be considered is the number of different SGX SDKs created in the past years, during the time this work was being written. Intel presents a list of different available SDKs in their own SDK page (Intel Corporation, 2022).

A final suggestion is to slightly tweak some implementation decisions in order to get better overall performance, for instance, (i) bound the request list to a specific CPU to take advantage of cache locality or use *semaphores* instead of *polling* for waiting requests and responses in the shared memory in both Kernel and proxy sides.

5.4.1 Development Environment

Since the SKEEN project is directly tied to the Kernel version and also the SGX SDK being used. With that, for future work it is important to meet the follwing requirements:

- **Kernel version**: SKEEN's kernel code was built using the available interfaces exposed by the Kernel version 5.19.0, released on July 31 2022. The UMH interface is the most susceptible to change due to its relation with other important subsystems that are continuously being modified, such as *btfilter*.
- **Intel SGX SDK**: The SGX SDK used in this project was the official Intel SDK (Intel Corporation, 2022), consequently, the userspace's program Makefile is dependent on Intel's SDK particularities, from runtime libraries to compilation and linking options.
- **Compiler toolset**: Since the userspace program is being compiled with kernel's build system, the toolset for compiling the SKEEN-enabled kernel must use the SGX-aware tools, such as *as*, *ld*, *ld.gold* and *objdump*. These are all delivered as part of the SDK.

A set of scripts were created to help on the compilation and testing phases, these were called *SKEEN-OSTest* and they can be found on their own Git repository[1]. The most important scripts are:

- `run.pl`: Perl script to launch a *qemu* instance running a custom kernel with SKEEN enabled and allow remote *GDB* debugging session.
- `kernel-compile.sh`: Bash script for compiling a minimal kernel image with the SKEEN configuration options enabled and other different configuration options to improve testing, such as dynamic tracing enabled, address space layout randomization and stack frame pointer optimization disabled.
- `osimage-gen.sh`: Bash script to generate a distribution image with required packages defined by the user using the *mkosi* tool. This allows distribution level testing instead of only basic system tests in a basic *ramdisk* image console.

Usage for each of these scripts can be found on their own help message by executing them with the flag `--help`.

1 https://gitlab.com/radlab-utfpr/skeen-ostest - Scripts to help compiling and testing SKEEN features

6 CONCLUSION

The architecture proposed in this paper employs the Intel SGX technology for data and process isolation of internal kernel components, which, at first glance, are not allowed to have access to such technology. It is accomplished by moving data from kernel to userspace and then wrapping it into SGX enclaves. A test case creating a crypto algorithm driver was created, giving in-kernel code the ability to perform encryption and decryption of data directly from within SGX enclaves using the standard Linux Kernel Crypto API.

The code for both the architecture implementation and test cases are being kept with open source license[1].

[1] https://gitlab.com/radlab-utfpr/skeen-linux-kernel - Linux Kernel code with SKEEN patches applied on top.

BIBLIOGRAPHY

Android Open Source Project. **Trusty TEE**. 2022. Disponível em: https://source.android.com/docs/security/features/trusty.

Apple Inc. **Apple Platform Security**. *[S.l.]*, 2022. Disponível em: https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf.

ARM. **Learn the architecture - TrustZone for AArch64**. *[S.l.]*, 2021. Disponível em: file:///C:/Users/Bruno/Downloads/learn_the_architecture_-_trustzone_for_aarch64_102418_0101_01_en.pdf.

ARNAUTOV, S. *et al.* SCONE: Secure linux containers with intel SGX. *In*: **12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)**. Savannah, GA: USENIX Association, 2016. p. 689–703. ISBN 978-1-931971-33-1. Disponível em: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

BROWN, N. **Fast interprocess communication revisited**. 2011. Disponível em: https://lwn.net/Articles/466304/.

European Union Agency for Network and Information Security. **Security aspects of virtualization**. *[S.l.]*, 2017.

IMA Project. **Integrity Measurement Architecture Wiki**. 2020. Disponível em: https://sourceforge.net/p/linux-ima/wiki/Home/.

Intel Corporation. **L1 Terminal Fault**. 2018. Disponível em: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

Intel Corporation. **Intel® 64 and IA-32 Architecutres Software Developer's Manual, Volume 3C: System Programming Guide, Part 3**. *[S.l.]*, 2019. v. 3C, n. 326019-071US. Disponível em: https://software.intel.com/sites/default/files/managed/7c/f1/326019-sdm-vol-3c.pdf.

Intel Corporation. **Intel® 64 and IA-32 Architecutres Software Developer's Manual, Volume 3D: System Programming Guide, Part 4**. *[S.l.]*, 2019. v. 3D, n. 332831-071US. Disponível em: https://software.intel.com/sites/default/files/managed/7c/f1/332831-sdm-vol-3d.pdf.

Intel Corporation. **Intel® Trusted Execution Technology (Intel TXT), Software Developement Guide, Measured Launched Environment**. *[S.l.]*, 2019. Disponível em: http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html.

Intel Corporation. **Software Guard Extensions - Get Started**. 2022. Disponível em: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/get-started.html.

Kernel.org. **Generic Netlink HowTo**. 2017. Disponível em: https://wiki.linuxfoundation.org/networking/generic_netlink_howto.

Kernel.org. **Generic TEE Subsystem**. 2017. Disponível em: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a2d9214c730f54ff72c2940bcd7f22d1fccb26ec.

KERNEL.ORG. **Linux Userspace SECCOMP manpage**. 2019. Disponível em: http://man7.org/linux/man-pages/man2/seccomp.2.html.

Kernel.org. **Trusted and Encrypted Keys**. 2020. Disponível em: https://www.kernel.org/doc/html/latest/security/keys/trusted-encrypted.html.

M. Jones. **Invoking user-space applications from the kernel**. 2010. Disponível em: https://developer.ibm.com/articles/l-user-space-apps/.

MCKEEN, F. *et al.* Innovative instructions and software model for isolated execution. *In*: **Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy**. New York, NY, USA: ACM, 2013. (HASP '13), p. 10:1–10:1. ISBN 978-1-4503-2118-1. Disponível em: http://doi.acm.org/10.1145/2487726.2488368.

MCKUSICK, M. K.; NEVILLE-NEIL, G.; WATSON, R. N. **The Design and Implementation of the FreeBSD Operating System**. 2nd. ed. *[S.l.]*: Addison-Wesley Professional, 2014. ISBN 0321968972, 9780321968975.

MENEGUELE, B.; FONSECA, K.; ROSA, M. Secure kernel execution with intel sgx. *In*: **X Simpósio Brasileiro de Engenharia de Sistemas Computacionais**. Porto Alegre, RS, Brasil: SBC, 2020. p. 168–173. ISSN 2763-9002. Disponível em: https://sol.sbc.org.br/index.php/sbesc_estendido/article/view/13108.

Netlink Community. **Netlink Protocol Library Suite (libnl)**. 2022. Disponível em: https://www.infradead.org/~tgr/libnl/.

NetOS Group. **ipc-bench: A UNIX inter-process communication benchmark**. 2019. Disponível em: https://www.cl.cam.ac.uk/research/srg/netos/projects/ipc-bench/.

RAJ, H. *et al.* ftpm: A software-only implementation of a tpm chip. *In*: **USENIX Security**. *[s.n.]*, 2016. Disponível em: https://www.microsoft.com/en-us/research/publication/ftpm-software-implementation-tpm-chip/.

Red Hat Inc. **What is Virtualization**. 2019. Disponível em: https://www.redhat.com/en/topics/virtualization/what-is-virtualization.

RICHTER, L. **Isolation of Operating System Components with Intel SGX**. May 2016. Dissertação (Mestrado) — Friedrich-Alexander-Universitä, May 2016.

RICHTER, L.; GöTZFRIED, J.; MüLLER, T. Isolating operating system components with intel sgx. *In*: **Proceedings of the 1st Workshop on System Software for Trusted Execution**. New York, NY, USA: ACM, 2016. (SysTEX '16), p. 8:1–8:6. ISBN 978-1-4503-4670-2. Disponível em: http://doi.acm.org/10.1145/3007788.3007796.

Sabt, M.; Achemlal, M.; Bouabdallah, A. Trusted execution environment: What it is, and what it is not. *In*: **2015 IEEE Trustcom/BigDataSE/ISPA**. *[S.l.: s.n.]*, 2015. v. 1, p. 57–64. ISSN null.

SPREITZER, R. *et al.* Systematic classification of side-channel attacks: A case study for mobile devices. **IEEE Communications Surveys & Tutorials**, v. 20, n. 1, p. 465–488, 2018.

Steven Rostedt. **Kernel ftrace - Kernel Function Tracer Documentation**. 2017. Disponível em: https://www.kernel.org/doc/html/latest/trace/ftrace.html.

TAN, B. *et al.* Towards lightweight serverless computing via unikernel as a function. *In*: **2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)**. *[S.l.: s.n.]*, 2020. p. 1–10.

Thomas Graf. **Netlink Library Documentation**. 2011. Disponível em: https://www.infradead.org/~tgr/libnl/doc/core.html.

Trusted Computing Group. **TPM 2.0 - A Brief Introduction**. 2019. Disponível em: https://trustedcomputinggroup.org/wp-content/uploads/2019_TCG_TPM2_BriefOverview_DR02web.pdf.

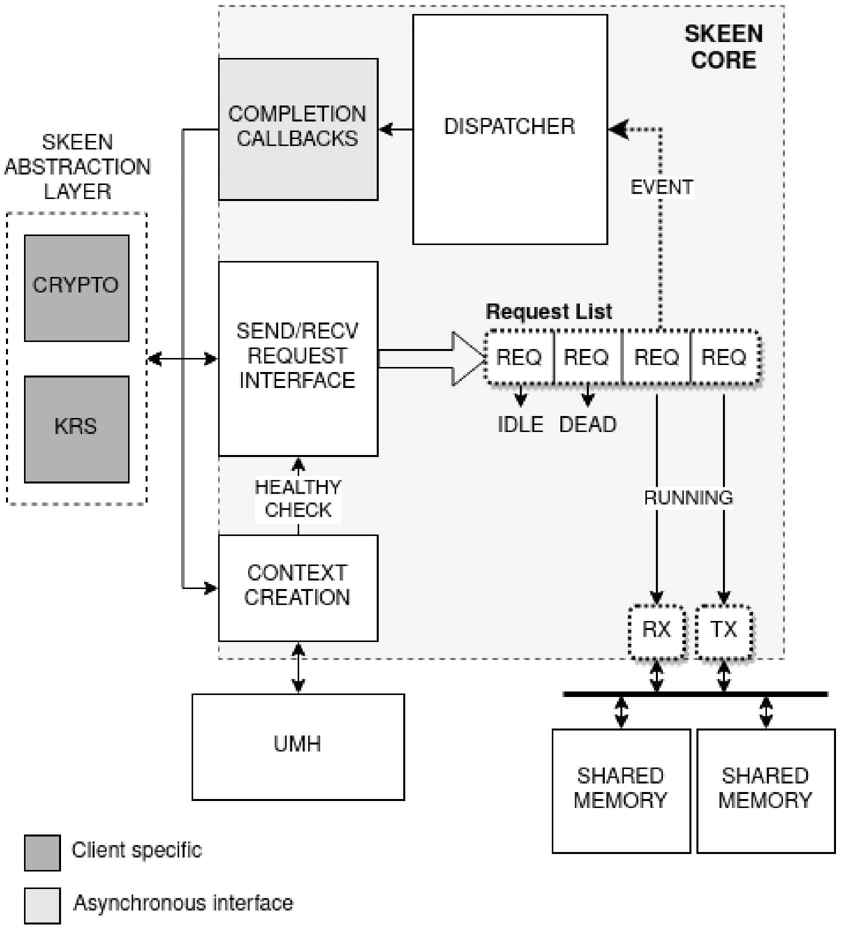
APPENDIX

APPENDIX A – SKEEN Architecture Internals

A.1 Core

SKEEN's core is responsible for creating and maintaining all the infrastructure needed to keep the communication from the external module, user of SKEEN's abstraction layers, and the userspace program interacting with the SGX enclaves. The way the core components are tied together was previously presented in Figure 4, but to improve readability the same image is shown as Figure 9 below.

Figure 9 – The *core* coordinates different instances of customer-specific components and also manages the data flow from both up and downwards directions with a work queue holding operation requests in different states. Customers wait on completion callbacks that are triggered by the dispatcher upon request state change events.



Source: Own Authorship (2022).

Each core component has its particularity and importance to keep the parts working together. In the following sections these components are detailed with code snippets taken from the SKEEN project source code repository[1].

[1] https://gitlab.com/radlab-utfpr/skeen-linux-kernel/-/tree/rebase-5.19/security/skeen

A.1.1 Contexts

Within SKEEN, a context is a structure used for holding information that connects SKEEN's user request to the userspace program response and also information regarding the userspace program itself[2] handling that request. With that, it can be considered that the context is the most important data structure within SKEEN core and is used by all its components on their operations, being passed directly as function parameter or being retrieved from a SKEEN base request. The Listing A.1.1 shows the context structure declaration and its members.

Listing A.1 – Context structure at *include/linux/skeen.h* used for store information regarding the entire data flow of a specific request.

```c
/*
 * Context related structure and functions
 */
struct skeen_context {
    struct umd_info umd_info; /* general information about umh */

    struct skeen_kreq *kreq; /* request from kernel to us */
    struct skeen_ures *ures; /* response from us to kernel */
    unsigned int timeout; /* how long should polling lasts */

    struct skeen_completion_operations *cop;
};

/*
 * Internal kernel interface (callbacks) structure
 */
struct skeen_completion_operations {
    void (*recvd)(struct skeen_context *ctx);
    void (*sent)(struct skeen_context *ctx);
};
```

Source: Own Authorship (2022).

Contexts are created from abstraction layer level, as detailed in the Appendix A.3, by calling `skeen_context_create()`. While some structure members are simple to initialize,

[2] A more detailed information on how the userspace program is launched and its execution is presented in the Appendix A.2.

the `umh_info` requires specific information and attention, since it is directly related to the userspace program memory loading and execution.

A.1.2 Requests

Requests are the base communication unit used between in the SKEEN core code. They basically hold a context (Section A.1.1) reference and the data structures needed to maintain a *request state* in a *request linked list*. The Listing A.1.2 shows the inner details of a request structure.

Listing A.2 – Base request structure at *security/skeen/core.c* with the data to place it into a linked list, work queue and to maintain the life cycle of a SKEEN context.

```c
struct skeen_request {
    struct list_head list;
    struct work_struct work;

    struct skeen_context *ctx;
    req_state_t state;
};
```

Source: Own Authorship (2022).

To help the reader, this base request is not the same as a SKEEN's user request, that is transferred to the userspace program as part of the communication protocol. The base request is used as a helper structure within SKEEN core to aggregate both SKEEN's user request and userspace program response in a single chunk of data, easing their life cycle management. Both user's request and userspace responses are going to be better detailed in the Section A.1.3.

Base requests are created at the same time as the contexts, as previously explained at Section A.1.1, but different from a context, they are used as both tasks in an internal work queue and items in a *request list*. The work queue allows parallel processing of different requests, while the request list is used for maintaining the request life cycle within SKEEN's core even after the task related to that request were already completed.

Requests' life cycle were presented at Section 4.3.3.

A.1.3 Communication Protocol

As presented in the Section 4.3.2, every request and its respective response is passed through a memory that is shared between the kernel and the userspace program and is used for a single SKEEN's costumer at a time, preventing concurrent subsystems access vulnerabilities.

Before diving into the actual request and response protocol, it is fundamental to present how the shared memory between the kernel and the user process is managed.

Shared Memory Management

The shared memory used for SKEEN's communication protocol was implemented as a new option for the IPC mode with UMH processes, whose had only the option of communication through *pipes*. With that, the solution created also benefits any other module in the kernel willing to use UMH for launching userspace programs from within the kernelspace.

For that, a new character driver was created exposing an interface file under `/dev/usermode`. The code at `drivers/char/usermode_shmem.c` implements handlers for the `open()`, `close()` and `mmap()` system calls with additional attention to prevent memory swap and dump on the memory being allocated.

Since the userspace program execution and memory allocation happens in different kernel threads, SKEEN's context creation must be aware when both components are ready to be used, thus begin any request/response processing. To keep these three entities in sync, a wait queue is used to put the SKEEN context launch on hold. The wake up event is only sent when the userspace program finally `mmap()`s the shared memory interface file. The Listing A.1.3 presents the `mmap()` handler where the wake up event is sent to all threads waiting on the wait queue (in this case, SKEEN context launch thread).

Listing A.3 – MMAP handler in the UMH shared memory IPC driver at *drivers/char/usermode_-shmem.c*. The $wake\_up()$ call at the end of the handler is going to wake up SKEEN's context launch thread.

```
static int shmem_mmap(struct file *file, struct vm_area_struct *vma)
{
    int err;
    /* ipc.shm_info was already assigned on userprocess creation */
    struct umd_shmem *shm_info = &(current->umd_info->ipc.shm_info);
    void *addr = kzalloc(UMD_SHMEM_LEN, GFP_KERNEL);
    phys_addr_t paddr = virt_to_phys(addr);

    /* make page non-swappable, always on memory after first access
   */
    SetPageReserved(virt_to_page(addr));

    vma->vm_ops = &vma_remap_ops;
    /* protect memory from some runtime operations */
    vma->vm_flags = VM_SHARED|VM_LOCKED|VM_DONTEXPAND|VM_DONTDUMP;
    err = remap_pfn_range(vma, vma->vm_start, paddr >> PAGE_SHIFT,
                  vma->vm_end - vma->vm_start,
                  vma->vm_page_prot);
    if (err)
        return -EAGAIN;

    vma->vm_private_data = addr;
    /* prepare to launch kthreads waiting on shm */
    shm_info->mm_map = addr;
    shm_info->wait_event = 1;
    wake_up(&shmem_wait_queue);
    return 0;
}
```

Source: Own Authorship (2022).

And the code being waked up by the `mmap()` is presented in the Listing A.1.3.

Listing A.4 – Code to launch SKEEN's context processing at *security/skeen/core.c*, from receiving the kernelspace module request, to launching the userspace program. This code waits until the userspace program *mmaps* the UMH shared memory.

```
static void skeen_context_launch(struct work_struct *work)
        // ...

    err = umd_load_blob(&ctx->umd_info, &skeen_umh_start,
                 &skeen_umh_end - &skeen_umh_start);
    if (err)
        return;

    fork_usermode_driver(&ctx->umd_info, UMD_IPC_SHMEM);
    pr_info("skeen: userspace program called with pid: %d\n",
        pid_nr(ctx->umd_info.tgid));
    shm_info = &ctx->umd_info.ipc.shm_info;

    /* Wait until userspace process mmap's the usermode shm */
    wait_event(shmem_wait_queue, shm_info->wait_event != 0);
    kreq = kzalloc(sizeof(struct skeen_kreq), GFP_KERNEL);
```

```c
    if (!kreq)
        return;
    ctx->kreq = kreq;

        // ...
}
```

Source: Own Authorship (2022).

In-depth details regarding the userspace program loading and execution is presented in the Appendix A.2.

A.1.4 Requests and Responses

Both requests and responses flowing through SKEEN's communication channel have a specific data structure to allow continuous mapping of their data to which kernelspace subsystem that data request or response belongs to.

The data structure for both request and response are similar, with only one different field, and could have been merged into a single structure, however, to increase flexibility for future features, two different structures were created. In the Listing A.1.4 both request and response structures are presented.

Listing A.5 – Request and response data structure declaration at *include/uapi/linux/skeen/protocol.h*. They are quite similar and the only field that differentiates is the first field into *fields* structure, where for the userspace response *retval* is used while for the kernelspace request it is an *opcode* field.

```c
/* SKEEN underneath protocol structure.
 * Every other abstraction should fit it */
struct skeen_ures {
    struct {
        uint32_t id;
    } header;
    union {
        uint8_t data[SKEEN_PROTO_MAX_PAYLOAD_LEN];
        struct {
            skeen_proto_retval_t retval;
            uint32_t data_len;
            uint8_t data[SKEEN_PROTO_MAX_DATALEN];
        } fields;
    } payload;
    uint8_t private[8];
    /* align structure to 64 bits */
    uint8_t reserved[12];
} __attribute__((packed));

struct skeen_kreq {
    struct {
        uint32_t id;
    } header;
    union {
        uint8_t data[SKEEN_PROTO_MAX_PAYLOAD_LEN];
        struct {
            skeen_proto_opcode_t opcode;
            uint32_t data_len;
            uint8_t data[SKEEN_PROTO_MAX_DATALEN];
        } fields;
    } payload;
    uint8_t private[8];
    /* align structure to 64 bits */
    uint8_t reserved[12];
} __attribute__((packed));
```

Source: Own Authorship (2022).

The declaration was placed in a specific SKEEN UAPI header, which is installed into system's userspace at `/usr/include/skeen/` directory, making it available to userspace programs and, consequently, preventing structure declaration issues.

The *private* field was added to for internal SKEEN use, with no direct relation to the data being transmitted. However, in the current state of the project, this field is not used.

A.2 Userspace Program

In order to narrow the attacker's surface to SKEEN's userspace program the mechanism *Usermode Helper* was used to load and execute a binary blob from within the kernel image as a userspace application. The same mechanism is used by the kernel initialization code to instantiate the *init* process.

A.2.1 Binary Placement

At first glance, the userspace application can be compiled as usual: using the standard kernel **Kbuild** infrastructure with specific userspace Makefile targets, however, the binary will be executed by the UMH mechanism later, which explicitly requires a reference to the binary's start point (memory address) and also its total size, forcing an explicit amendment to the kernel image, as shown in Listing A.2.1.

Listing A.6 – Assembly code at *security/skeen/umh_blob.S* to explicitly include the binary blob into kernel's image as a read-only data.

```
    .section .rodata, "a"
    .global skeen_umh_start
skeen_umh_start:
    .incbin "security/skeen/skeen_us"
    .global skeen_umh_end
skeen_umh_end:
```

Source: Own Authorship (2022).

Compiling the above code appends the userspace binary blob to the kernel's image. Later, both `skeen_umh_start` and `skeen_umh_end` are used to calculate the binary size to be used at binary execution.

A.2.2 Application Compilation

The userspace program must be compiled alongside the kernel image, but many different cares must be taken with relation to linkage options to allow it to be compiled with SGX support. The Listing A.2.2 presents part of the linking flags used to link the trusted portions of SKEEN's userspace program. Most of the options were taken from the Intel SGX SDK[3] code and are required to make sure the default options do not interfere with the SGX usage: no system standard library or header should be used, but only those exposed by the SDK, also, the trusted portion is loaded into an SGX enclave at runtime as a shared library, meaning that the standard execution stack should not be handled by regular rules, and so forth.

Listing A.7 – Compilation and linking options for the trusted portion of userspace's program at *security/skeen/Makefile*.

```
T_CFLAGS := $(T_INCLUDE_PATH) -nostdinc -fvisibility=hidden -fpie -
   ffunction-sections -fdata-sections
T_LDFLAGS_SEC := -Wl,-z,relro,-z,now,-z,noexecstack -Wl,-fuse-ld=gold
    -Wl,--rosegment
```

[3] https://github.com/intel/linux-sgx

```

T_LDFLAGS := $(T_LDFLAGS_SEC) \
    -nostdlib -nodefaultlibs -nostartfiles -L$(SGX_LIBRARY_PATH) \
    -Wl,--no-undefined -Wl,--whole-archive -l$(TRTS_LIB) -Wl,--no-
   whole-archive \
    -Wl,--start-group -lsgx_tstdc -lsgx_tcrypto -l$(TSERVICE_LIB) -Wl
   ,--end-group \
    -Wl,-Bstatic -Wl,-Bsymbolic -Wl,--no-undefined \
    -Wl,-pie,-eenclave_entry -Wl,--export-dynamic \
    -Wl,--defsym,__ImageBase=0 -Wl,--gc-sections \
    -Wl,--version-script=$(T_LDS)
```

Source: Own Authorship (2022).

Any library needed by the userspace application must be statically linked in this step, otherwise the application will not be able to find the shared library in case it is launched before the *rootfs* is fully mounted and ready to be used. This relates directly to the issues mentioned at Section 5.3.1.

A.2.3 Application Execution

Executing the userspace program demands three different stages, binary blob memory loading, launching and healthy check, the following sections show each of these stages.

Binary Loading

When a new request is created by a SKEEN's abstraction layer user a new context is created and the userspace application to handle the SGX enclaves starts to be prepared. The first step is to load the binary blob into userspace memory and save a reference to it into a `umd_driver` structure, as shown in the Listing A.2.3 and Listing A.2.3.

Listing A.8 – UMH call at *security/skeen/core.c* to explicitly load the binary into userspace's memory.

```c
static void skeen_context_launch(struct work_struct *work)
        // ...
    err = umd_load_blob(&ctx->umd_info, &skeen_umh_start,
                &skeen_umh_end - &skeen_umh_start);
    if (err)
        return;
    // ...
}
```

Source: Own Authorship (2022).

Listing A.9 – UMH driver declaration at *include/linux/usermode_driver.h* with the reference to the userspace memory.

```
struct umd_shmem {
    void *mm_map;
    int wait_event;
};

struct umd_ipc {
    int mode;
    struct umd_shmem shm_info;
    struct file *pipe_to_umh;
    struct file *pipe_from_umh;
};

struct umd_info {
    const char *driver_name;
    struct umd_ipc ipc;
    struct path wd;
    struct pid *tgid;
};
```

Source: Own Authorship (2022).

In the Listing A.2.3, both *umd_ipc* and *umd_shmem* structures were added to the UMH mechanism on SKEEN development as part of the Shared Memory IPC mode support implementation, in order to allow better separation and selection of IPC modes.

Binary Launching

Once the binary is loaded, it can be executed by directly calling kernel's *exec* system call through a wrapper on UMH module. The binary memory is allocated to a new userspace process, which is asynchronously placed in a work queue for the `execve()` call. The Listing A.2.3 shows the wrapper being called at SKEEN's core code.

Listing A.10 – UMH execution call at *security/skeen/core.c*. *fork_usermode_driver()* is a wrapper around the *execve()* operation with additional code to handle the IPC setup process. The previously loaded memory is then executed as a userspace process.

```c
static void skeen_context_launch(struct work_struct *work)
        // ...
    fork_usermode_driver(&ctx->umd_info, UMD_IPC_SHMEM);
    pr_info("skeen: userspace program called with pid: %d\n",
        pid_nr(ctx->umd_info.tgid));
    // ...
}
```

Source: Own Authorship (2022).

Healthy Check

Once the binary is executed, the first thing that should happen is a healthy check from SKEEN's core to userspace's program, ensuring the program started fine and that the communication channel is working correctly. However, as mentioned in Section A.1.3, the core code will wait until userspace program `mmap()`s the UMH shared memory created by SKEEN to then perform the healthy check, preventing data loss due to launching delays.

In Listing A.2.3 it is possible to see different error cases where the healthy check might fail. In Listing A.2.3 is the actual check code to ensure a dummy request with a small data (e.g. *hello*) reaches userspace. In the same way, a response from userspace is expected to make kernel side under a certain timeout threshold.

Listing A.11 – Valid return codes for the healthy check code at *security/skeen/core.c*, triggered on context launch phase.

```c
static void skeen_context_launch(struct work_struct *work)
        // ...
    if ((err = skeen_umh_health_check(ctx)) != 0) {
        switch (err) {
        case -ENOMEM:
            pr_err("skeen: failed to allocate data\n");
            break;
        case -ETIME:
            pr_err("skeen: failed to receive data in time\n");
            break;
        case 1:
            pr_err("skeen: userspace program not healthy\n");
            break;
        default:
            pr_info("skeen: healthy check succeeded\n");
            break;
        }

        return;
    }
        // ...
}
```

Source: Own Authorship (2022).

Listing A.12 – Code to handle the healthy check logic at *security/skeen/core.c*. Since this code is ran strictly by SKEEN core before enabling any other data flow starts, the request list and work queue is bypassed.

```c
static int skeen_umh_health_check(struct skeen_context *ctx)
{
    struct skeen_request *req;
    struct skeen_context **pctx = &ctx;
    char *kreq_data = "hello";
    char *ures_data;

    req = container_of(pctx, struct skeen_request, ctx);

    /* send the healthy data to userspace process */
    skeen_proto_set_opcode(ctx->kreq, SKEEN_PROTO_OP_HEALTHY);
    skeen_proto_set_data(ctx->kreq, kreq_data, 6);

    /* health check is executed to make sure the userspace process is
     * actually running, before any other work can actually be
     * performed, because of that, call the __request_* functions
     * directly, bypassing workqueue */
    __request_send(&req->work);

    /* [debug-mode] poll the memory forever */
    ctx->timeout = 0;
    __request_recv_poll(&req->work);
    if (PTR_ERR(ctx->ures) == -ETIME)
        return -ETIME;

    ures_data = kmalloc(SKEEN_PROTO_MAX_DATALEN, GFP_KERNEL);
    if (!skeen_proto_check_crc32(ctx->ures) ||
        (skeen_proto_get_retval(ctx->ures) != SKEEN_PROTO_RET_OK) ||
        (skeen_proto_get_data(ctx->ures, ures_data,
                    SKEEN_PROTO_MAX_DATALEN) != 6)) {
        pr_info("skeen: health check failed: %d %s\n",
            ctx->ures->header.id, ures_data);
        return -1;
    }

    if (!memcmp(ures_data, kreq_data, 6))
        return -1;

    return 0;
}
```

Source: Own Authorship (2022).

On userspace side the code is similar, the only caution is to read the shared memory following the convention presented in Section 4.3.2, where the memory is divided into two parts, being that one is a circular buffer for requests and the second is a circular buffer for responses. With that, SKEEN core must write directly to the first address of the shared memory, while the userspace writes to the beginning of `UMH_SHM_LEN/2`, where `UMH_SHM_LEN` is the total size of the shared memory allocated by UMH.

Once the healthy check is complete, requests and responses start to be processed following the standard mechanisms implemented.

A.3 Abstraction Layer

When a new module is created to implement a new cryptographic algorithm or just to implement a known algorithm using different techniques within the kernel, this new module must be **registered** and **adhere** to the standard crypto subsystem interface, allowing other modules to use this new implementation just by changing the algorithm driver name, but keeping the function calls and callbacks the same.

In this appendix some snippets are shown to exemplify how an AES algorithm implemented within a SGX enclave can be called from another kernel module using the SKEEN abstraction layer for cryptography.

A.3.1 Driver Registration

The SKEEN abstraction layer for the AES cryptographic algorithm implemented within a SGX enclave is used as the *new module* that must be registered into the crypto subsystem as an **algorithm driver** and also adhere to the subsystem's interface. The Listing A.3.1 shows how the registration is done by defining the `crypto_alg` structure.

Listing A.13 – SKEEN crypto abstraction layer registration as a new and valid cryptographic module. It is valid to note that the headers needed to compile this code were not included just for the sake of brevity.

```
static struct crypto_alg skeen_aes_alg = {
    .cra_name = "skeen-aes",
    .cra_driver_name = "skeen-aes",
    .cra_priority = 100,
    .cra_flags = CRYPTO_ALG_TYPE_CIPHER,
    .cra_blocksize = AES_BLOCK_SIZE,
    .cra_ctxsize  = sizeof(struct crypto_aes_ctx),
    .cra_module = THIS_MODULE,
    .cra_u = {
        .cipher = {
            .cia_min_keysize = AES_MIN_KEY_SIZE,
            .cia_max_keysize = AES_MAX_KEY_SIZE,
            .cia_setkey = skeen_aes_setkey,
            .cia_encrypt = skeen_aes_encrypt,
            .cia_decrypt = skeen_aes_decrypt,
        }
    },
    .cra_init = skeen_aes_init,
    .cra_exit = skeen_aes_exit,
};

```

```c
static int __init skeen_aes_mod_init(void)
{
    pr_info("skeen-crypto: registering AES alg\n");
    return crypto_register_alg(&skeen_aes_alg);
}

static void __exit skeen_aes_mod_exit(void)
{
    crypto_unregister_alg(&skeen_aes_alg);
}

late_initcall(skeen_aes_mod_init);
module_exit(skeen_aes_mod_exit);

MODULE_DESCRIPTION("SKEEN AES Crypto Interface");
MODULE_ALIAS_CRYPTO("skeen-aes");
```

Source: Own Authorship (2022).

Since the AES algorithm is also present in the generic kernel code, the default values for some of the fields were used from the generic kernel header `crypto/aes.h`, but in case a different algorithm, not present in the current version of the kernel, is used, the values would respect the algorithm specification.

Once the abstraction layer is registered as a bew algorithm driver in the crypto subsystem, a module, user of the abstraction layer, can use it only by changing the driver name to `skeen-aes` in the `crypto_alloc_tfm()` call, or any other wrapper. The Listing A.3.1 shows how the request for SKEEN's abstraction layer can be done from an external module willing to use it.

Listing A.14 – Initial algorithm driver request using the standard *SKCipher* crypto subsystem interface, which wraps more basic and low-level functions, and the *set key* operation for later encryption and decryption operations.

```c
char key[16] = {0}; /* BAD key, that's just an _example_ */
struct crypto_skcipher *tfm = crypto_alloc_skcipher("skeen-aes", 0,
   0);
crypto_skcipher_setkey(tfm, key, sizeof(key));
```

Source: Own Authorship (2022).

A.3.2 Driver Usage

To help understanding the module usage it is important to note that the abstraction layer was divided in two different scoped modules:

1. **AES Module**: used to register the module in the crypto subsystem;
2. **SKEEN Core Interface**: used to communicate the crypto data to the SKEEN core. Therefore, hereafter called **inner scope** due to its proximity to the SKEEN core when compared to the other scope.

The code shown in the Listing A.3.1 comes from the **AES Module** scope and every function pointed in the `crypto_alg` structure definition is present in the same scope. These functions are simple wrappers calling code from the **SKEEN Core Interface** scope with arguments passed by the user of crypto abstraction layer, for instance, buffers containing data and keys and operation code for both encryption and decryption operations, as can be seen in the Listing A.3.2

Listing A.15 – Functions from the abstraction layer *AES Module* scope, calling a single function from *SKEEN Core Interface* scope to pass specific information from the abstraction layer user and the required code of the cryptographic operation.

```
static int skeen_aes_setkey(struct crypto_tfm *tfm, const u8 *key,
                unsigned int keylen)
{
    u8 aes_key = key;
    skeen_crypto_op_t opcode = SKEEN_CRYPTO_OP_SETKEY;
    return skeen_crypto_exec(tfm, opcode, aes_key, &keylen);
}

static void skeen_aes_encrypt(struct crypto_tfm *tfm, u8 *dst,
                  const u8 *src)
{
    u8 *input_buffer = src;
    skeen_crypto_op_t opcode = SKEEN_CRYPTO_OP_ENCRYPT;

    if (skeen_crypto_exec(tfm, opcode, dst, input_buffer))
        memset(dst, 0, 16);
}

```

```
static void skeen_aes_decrypt(struct crypto_tfm *tfm, u8 *dst,
                    const u8 *src)
{
    u8 *input_buffer = src;
    skeen_crypto_op_t opcode = SKEEN_CRYPTO_OP_DECRYPT;

    if (skeen_crypto_exec(tfm, opcode, dst, input_buffer))
        memset(dst, 0, 16);
}
```

Source: Own Authorship (2022).

The function `skeen_crypto_exec()` is the entry point for the inner scope that handles the actual SKEEN core interface, maintaining the data flow from the kernelspace subsystem SKEEN costumer and the userspace program through an *internal context* reference, which is used for waiting the whole data flow cycle completion.

The aforementioned *internal context* that is kept for future use is created when a module willing to use a cryptographic algorithm calls the function `crypto_alloc_tfm()` or other wrappers like `crypto_alloc_skcipher()`, leading directly to the execution of the function defined at the `crypto_alg.cra_init` structure field, and thus, to the creation of the context that is used throughout the abstraction layer as reference for the caller's request.

Each time a new module allocates SKEEN's abstraction layer as their crypto algorithm driver, a new context is created and kept in a dedicated linked list known only by the abstraction layer, meaning that this list is not maintained by SKEEN core. The reason for keeping the context control at the abstraction layer level is to avoid increasing core's complexity for each SKEEN costumer with different requirements.

The Listing A.3.2 shows the code that creates the *internal context* and later, in the Listing A.3.2, how it is used for setting the key for a specific request.

Listing A.16 – Internal context creation to maintain the data flow from kernelspace module to the userspace program interacting with SGX enclaves. This context is a way to keep track of the actual crypto subsystem context, the module that called for it (user) and SKEEN core context.

```
/*
 * AES Module scope at security/skeen/crypto_aes.c
 */

/* We need a way to link crypto_aes_ctx to our internal skeen_context
   . It's
 * done with an intermediate context object, skeen_crypto_context,
   which
 * basically contains both contexts into it and must be initialized
   before
 * the crypto algo actually is ready for performing any action */
static int skeen_aes_init(struct crypto_tfm *tfm)
{
    struct crypto_aes_ctx *tfm_ctx = crypto_tfm_ctx(tfm);
    int err;

    err = skeen_crypto_ctx_init(tfm_ctx);
    if (err)
        return err;

    return 0;
}

/*
 * Inner scope at security/skeen/crypto.c
 */

LIST_HEAD(ctx_list);

int skeen_crypto_ctx_init(void *tfm_ctx)
{
    struct skeen_crypto_context *ictx;

    ictx = kzalloc(sizeof(struct skeen_crypto_context), GFP_KERNEL);
    if (!ictx)
        return -ENOMEM;

    ictx->skeen_ctx = skeen_context_create("skeen_crypto");
    if (IS_ERR(ictx->skeen_ctx))
        return PTR_ERR(ictx->skeen_ctx);

    ictx->skeen_ctx->cop = &complete_ops;
    ictx->tfm_ctx = tfm_ctx;
    init_completion(&ictx->completion);
    list_add_tail(&ictx->list, &ctx_list);

    return 0;
}
```

Source: Own Authorship (2022).

Listing A.17 – Function for the *setkey* operation. The code initializes the request and response structures and populates the request with the key to be set and the operation code to be passed to the AES algorithm within the SGX enclave. The request is then sent and the function waits for the response.

```
static int __crypto_exec_setkey(struct skeen_crypto_context *ctx,
                const u8 *key, unsigned int keylen)
{
    struct skeen_kreq *kreq;
    struct skeen_ures *ures;
    u8 *data = (u8 *)key;
    u32 data_len = (u32)keylen;
    int err = 0;

    kreq = kzalloc(sizeof(struct skeen_kreq), GFP_KERNEL);
    ures = kzalloc(sizeof(struct skeen_ures), GFP_KERNEL);
    ctx->skeen_ctx->kreq = kreq;
    ctx->skeen_ctx->ures = ures;

    skeen_proto_set_opcode(kreq, SKEEN_PROTO_CRYPTO_OP_SETKEY);
    skeen_proto_set_data(kreq, data, data_len);

    err = skeen_request_send(ctx->skeen_ctx);
    if (err)
        return err;

    wait_for_completion(&ctx->completion);
    err = skeen_proto_get_retval(ures);
    return err;
}
```

Source: Own Authorship (2022).

The waiting operation uses the standard kernel's *completion* API, which is built on top of the *wait queue* and *wake up* infrastructure of the scheduler, meaning that it does not use conventional mechanisms as *mutex*, *semaphore* or *busy-wait*, but a lightweight and low-level solutions that allows other threads in the same CPU core to run until a specific event is yielded.

The code for the entire SKEEN project can be found on its own Git repository[4].

[4] https://gitlab.com/radlab-utfpr/skeen-linux-kernel/-/tree/rebase-5.19/security/skeen