Secure Kernel Execution with Intel SGX

Bruno Meneguele*, Keiko Fonseca†, Marcelo Rosa‡
Graduate Program in Electrical and Computer Engineering (CPGEI)
Federal University of Technology – Paraná (UTFPR)
Curitiba, Brazil
*bmeneg@redhat.com, †keiko@utfpr.edu.br, ‡mrosa@utfpr.edu.br

***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 allow internal kernel data to be stored or processed within SGX private enclaves.**

In this paper we present 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.

***Index Terms*—Intel SGX, Operating System, Linux Kernel, Isolation.**

I. Introduction

Security of Operating Systems (OS) is a broad subject mostly if the size and complexity of each of its subsystems [1] are considered. Recently, OS security has been a trending topic between researchers and conferences throughout the world due to its importance on current business models adoption leaning towards cloud computing solutions [2].

Cloud computing has its bases on virtualization technology, which in turn has its foundation built upon features provided by the host hardware and the underneath kernel abstraction layer of operating systems, like hypervisors and containers [3]. Although they have different architectures, both are susceptible to underneath OS kernel vulnerabilities: once OS components get compromised, internal functionalities are also exposed to the attacker, giving it partial or full control over system resources, including the virtualization layer [4].

Considering the correlation between system components, possibly sharing multiple points of exposure, a concept named Trusted Execution Environment (TEE) was defined. It was built upon other basic and older concepts of resource isolation, e.g. virtual barrier between the user applications and the rest of the system, including both, superuser applications and kernel [5].

Intel *Software Guard eXtension* (SGX) technology, is a materialization of the TEE concept released for the x86 architecture as a hardware-based implementation. By design, SGX defines a secure area (named *enclave*) that can be accessed only by the least privileged process (user applications running on processor ring 3), preventing access of any code with higher privilege, such as the operating system [6].

Although this limitation ensures that kernel code will not hijack userspace data processed within an enclave, kernel subsystems are unable to make any direct use of such hardware feature. Therefore, there is no way a kernel thread can process or store sensitive data inside enclaves to avoid kernel parts being exposed if any other component is compromised.

This paper will describe 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 builtin 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 to be transmitted to userspace and then 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.

This paper is organized as follows: Section II briefly presents a project with similar concepts proposed in this project; Section III presents few important topics for understanding the project; Section IV proposes some use cases for the proposed architecture; Section V describes the threat model and the assumptions made before conceiving this project; Section VI depicts the system architecture; Section VII presents the test case developed; Section VIII walks through future ideas to enhance the current state of this project, and Section IX concludes with all achievements and considerations brought by this project.

II. Similar projects

SKEEN was inspired on the TresorSGX [7] 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 interface usage, architecture design, and other internal details.

Some other comparisons can be done in relation to performance, security level (e.g. attack surface length), data

flow transparency, and interface extensibility. Nonetheless, a comparison of these dynamic factors (that may vary depending on the case or workload) is out of the scope for this paper.

III. Background

A. 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 [2]. 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 main memories 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. 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 [8].

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 [6]. 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 [9].

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.

B. 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) [10], 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 compiling 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 VI details about the UMH interface implementation, which was enhanced with shared memory handling code in order to improve overall performance.

IV. 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.

A. 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 [11]:

1) Dedicated: a real and small dedicated hardware implementation, delivered as a single microchip usually soldered to computer motherboard.
2) Integrated: part of a different component on the motherboard, sharing the same silicon dice of a certain microchip, for instance, inside PCH (also known as *chipset*).
3) Firmware: software implementation into platform TEE.

Although a TPM is usually meant to be used as a source of trust for the entire system - from its bootstrap to its runtime - 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 [12], that can be used for internal kernel subsystems like the *Integrity Measurement Architecture* (IMA) and *Extended Verification Module* (EVM) [13], through the concept of *encrypted* and *trusted* keys [14] from the *Key Retention Subsystem* (KRS) for ensuring some aspects of the system were not changed during normal operation.

B. 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.

V. Threat Model and Assumptions

The primary software assumption is that the overall system runtime 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) [15], are not checked for their presence, but we assume their respective mitigation are applied if necessary. Other side-channel attacks, like power analysis or any other with hardware access level are also out of scope of this project.

VI. Architecture

Figure 1 presents an overview of the SKEEN architecture proposed in this paper and used as the basis for further discussion.

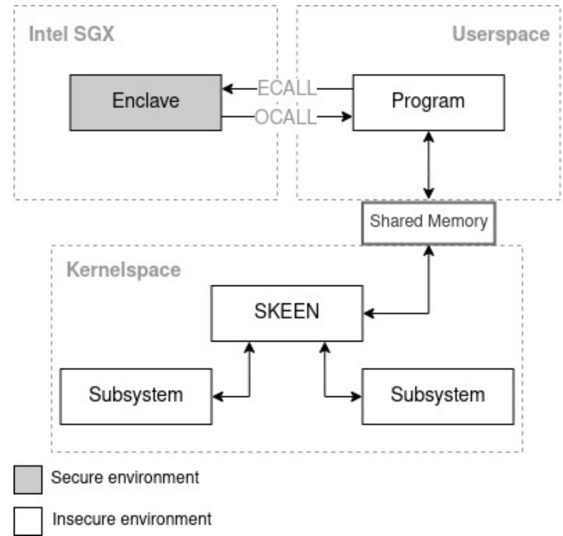

Fig. 1. In the insecure side of the platform the SKEEN kernel module is used as the interface for the underneath kernel subsystems to interact with the userspace program, that is launched to every new subsystem request, through a shared memory IPC scheme. The data is then insecurely forwarded to secure SGX enclaves where the data is finally protected against eavesdropping and malicious modification.

Nonetheless, Figure 1 can be exploded in order to observe the layered design of the architecture as shown in Figure 2. This design was used to accommodate differences between each customer 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.

A. Userspace Program

To allow better isolation between subsystems operating with SKEEN, each initialization request made to its core launches a new process application in a unique process memory and also a unique shared memory region (not shared to any other subsystem). Once all transactions are finished, the customer can request to SKEEN to terminate - by freeing and zeroing any memory region allocated - the userspace program and

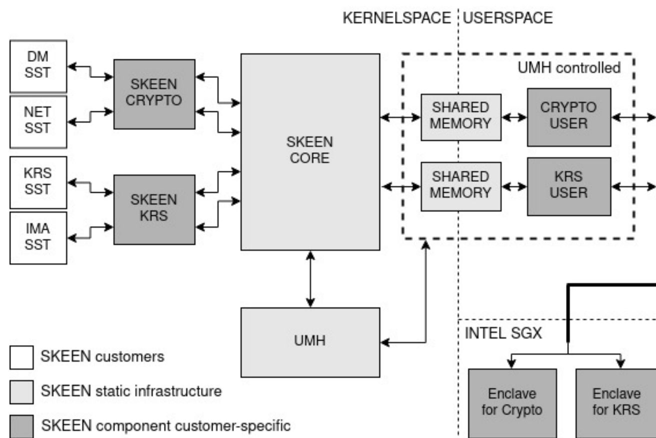

Fig. 2. Layered view from customers in the kernelspace to the Intel SGX enclaves, where the core acts as the main component and communication arbitrator between kernel and userspace. Each customer-specific component has its relative counter part in both userspace and within Intel SGX, creating the code and data isolation between customers.

other structures held within the architecture used to manage each operation context.

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

B. Interprocess Communication

Shared memory was the chosen approach to be used on SKEEN IPC mechanism, employing near-zero overhead in its raw format. However, two this approach has two issues: data synchronization and access control.

1) Data synchronization: Since there is no data synchronization in shared memory, the concurrency control in the data exchange among different processes needed to be implemented.

Our strategy consisted on not sharing the same memory region among different subsystems, which allow us to manage a small data rate flowing through the shared memory. Also, each half of this memory is used strictly by each flow direction: upwards for data flowing from kernel to userspace, denoted by *request*, and downwards for data flowing in the other direction, denoted by *response*. These data objects have fixed size and are composed of different fields.

This memory split, plus the fact that the userspace program operates reactively, helps to mitigate data concurrence issues. Thus, each side of the channel needs to be careful only to not exceed the buffer boundary and to signalize which message was already handled. Also, requests and responses are handled sequentially, thus data ordering is kept and parallelism is not supported.

2) Access control: The shared memory mechanism does not imply any access control to the memory, allowing any concurrent process to map the same memory the other process is using. However, any shared memory created to be used as the IPC is tied to a single user process in the SKEEN architecture, the one launched by the SKEEN core code. Any mapping operation by another process to the same memory region is denied.

Additionally, from kernel side, it is still unknown how to control the memory access by different kernel threads. This topic is revisited in Section VIII.

C. Kernelspace Module

The kernel module contains many 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 generating requests and the userspace program responding back to subsystems data that was processed inside SGX enclaves. This in-between component, structured as presented in Figure 3, was designed to be as transparent as possible, hiding the entire bookkeeping, data tracking, and IPC mechanism that guarantees that data reaches its respective destination. Also, considering that each customer 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 customer specificity be handled and maintained in a separate codebase.

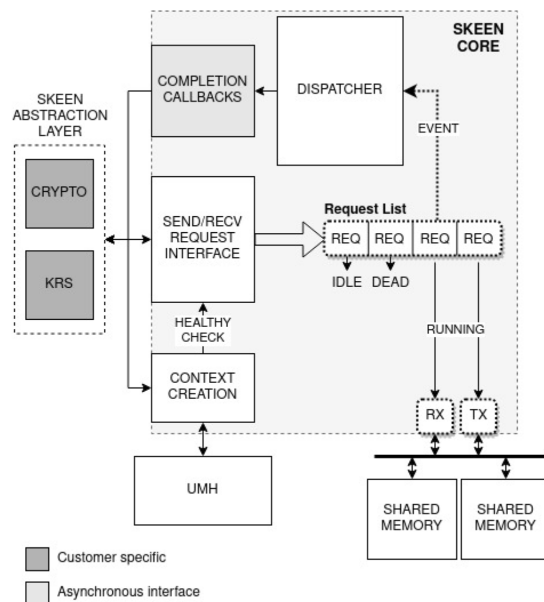

Fig. 3. The *core* component coordinates different instances of customer-specifics smaller components and also manages the data flow from both up and downwards directions with a work queue holding operation requests in different states. Customers wait on completion callbacks that are triggered by the event dispatcher.

The only object that is shared among core and customer-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 customer explicitly destroys it when the answers to the requests are sent to userspace. A more detailed explanation of the data flow throughout the architecture will be given in Section VI-E

D. SGX Enclave

The trusted program, running within the SGX enclave, is built alongside the userspace program since it is also customer-specific and also because both must be aware of each other existence with the notion of what interface is being exposed. The user (untrusted) program makes *ECALLs* to functions from the program running on the trusted side. While *OCALLs* are performed for the other way.

E. 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 4 depicts the core data components from the subsystem operation request to the highest level of abstraction.

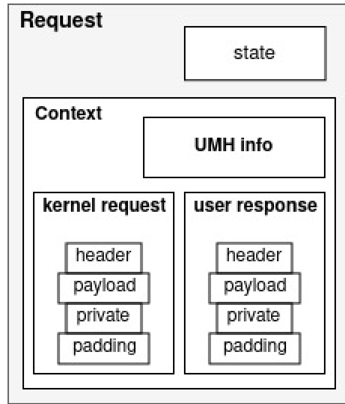

Fig. 4. Kernel module layered data view. The deeper inside the data is, the closer to the subsystem requesting data process on Intel SGX.

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

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 it will be sent from userspace.

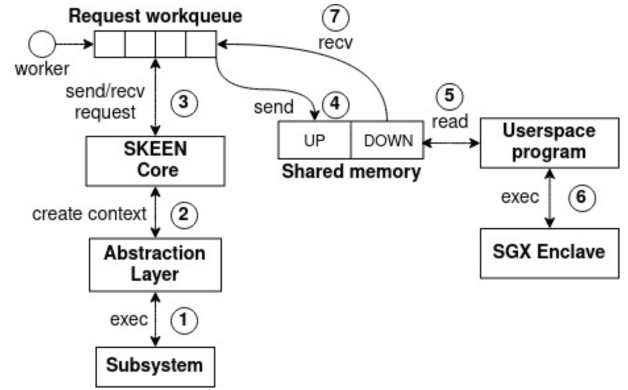

Fig. 5. Data flow throughout the SKEEN layered architecture considering a test case with a cryptographic abstraction layer.

The *core* then wraps the *context* content, as shown in Figure 4, 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 notices a non-processed request is waiting in the shared memory, and then reads ⑤ the request by checks 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 ⑧, waking up the *dispatcher* component in the core, as seen in Figure 3. 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.

VII. Results

We used a crypto algorithm driver as the test case for validating the entire architecture behavior. A crypto driver exposes an interface of allowed operations to be performed with specific algorithms. The test driver wraps the AES algorithm, which uses an implementation from within the SGX enclave instead of using the existent kernel implementation. Therefore, although the data arriving to the enclave might have been tampered, the cryptographic operations are guaranteed to be correct.

The abstraction layer follow the specification for registering the driver in the kernel and, at the same time, implement the wrapper functions that will send all data to be encrypted or decrypted through our SKEEN infrastructure. In this way, any other code directly request AES encryption/decryption operation can normally use the generic crypto interface, while our driver handles all translation between crypto data structures to SKEEN structures.

Another important aspect to be noted is that both the abstraction component and the userspace program must be aware of specific cases an AES driver may face: data 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 usages for the request and response internal fields - header, payload and generic data field.

VIII. Future work

Shared memory is the mechanism chosen to represent the IPC among kernel and userspace 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 [16]. With that in mind, deeper research on new IPC mechanisms or improvements on well-known ones [17] 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 [18] technology in order to gather pre-boot platforms measurement values (BIOS, Chipset, and others) enabling a more robust chain of trust for the entire system.

Following the same thought of booting a system with possible malicious components, SKEEN userspace process could have its hash measured and verified before it is effectively running with the help of the IMA (Integrity Measurement Architecture) Linux subsytem. Also, the userspace program can apply the usage of a sandbox mechanism like *seccomp* [19], preventing any not allowed system calls to be performed in case it gets compromised.

A final suggestion is to compare the overall architecture performance and security against TresorSGX project, mostly due to the mechanisms and technologies differences applied in each project.

IX. 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.

References

[1] M. K. McKusick, G. Neville-Neil, and R. N. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Addison-Wesley Professional, 2014.

[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure linux containers with intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, November 2016, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[3] Red Hat Inc, "What is virtualization," 2019. [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-virtualization

[4] European Union Agency for Network and Information Security, "Security aspects of virtualization," ENISA, Tech. Rep., February 2017.

[5] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, Aug 2015, pp. 57–64.

[6] Intel Corporation, *Intel® 64 and IA-32 Architecutres Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*, October 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/7c/f1/332831-sdm-vol-3d.pdf

[7] L. Richter, J. Götzfried, and T. Müller, "Isolating operating system components with intel sgx," in *Proceedings of the 1st Workshop on System Software for Trusted Execution*, ser. SysTEX '16. New York, NY, USA: ACM, 2016, pp. 8:1–8:6. [Online]. Available: http://doi.acm.org/10.1145/3007788.3007796

[8] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:1. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

[9] Intel Corporation, *Intel® 64 and IA-32 Architecutres Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*, October 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/7c/f1/326019-sdm-vol-3c.pdf

[10] M. Jones, "Invoking user-space applications from the kernel," February 2010. [Online]. Available: https://developer.ibm.com/articles/l-user-space-apps/

[11] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, "ftpm: A software-only implementation of a tpm chip," in *USENIX Security*, August 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/ftpm-software-implementation-tpm-chip/

[12] Trusted Computing Group, "Tpm 2.0 - a brief introduction," 2019. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/2019_TCG_TPM2_BriefOverview_DR02web.pdf

[13] IMA Project, "Integrity measurement architecture wiki," August 2020. [Online]. Available: https://sourceforge.net/p/linux-ima/wiki/Home/

[14] Kernel.org, "Trusted and encrypted keys," 2020. [Online]. Available: https://www.kernel.org/doc/html/latest/security/keys/trusted-encrypted.html

[15] Intel Corporation, "L1 Terminal Fault," October 2018. [Online]. Available: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault

[16] N. Brown, "Fast interprocess communication revisited," November 2011. [Online]. Available: https://lwn.net/Articles/466304/

[17] NetOS Group, "ipc-bench: A unix inter-process communication benchmark," November 2019. [Online]. Available: https://www.cl.cam.ac.uk/research/srg/netos/projects/ipc-bench/

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

[19] Kernel.org, "Linux userspace seccomp manpage," 2019. [Online]. Available: http://man7.org/linux/man-pages/man2/seccomp.2.html