Survey on Trusted Execution Environments

Nicolas Buchner, Holger Kinkelin, Filip Rezabek*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: nicolas.buchner@tum.de, kinkelin@net.in.tum.de, frezabek@net.in.tum.de

***Abstract*—Confidentiality of code and data is an essential part of modern computing. As cloud services become more important as an easy way to use computational power, the need for keeping the exact code of the running applications from the companies that offer these services. A Trusted Execution Environment (TEE) is an option to remove the need for trusting the device the code is executed on and hide data from other processes. In this paper, the concept of a TEE is shown as well, as two implementations of TEE are being analyzed. First, a general overview of TEEs is given. Next, the functionality of Intel SGX and ARM TrustZone is being explained. Afterwards, the features of both implementations are shown, and the problems they have are being analyzed. Next, there is a comparison between the two implementations. Finally, other options to achieve trusted execution are being shown.**

***Index Terms*—trusted execution, privacy, security**

1. Motivation

In today's world, the need for security of the data used every day has become a significant factor in determining how to transport and process this data. Whilst looking at security during communication between different devices is common, the need to process data and execute proprietary code on client devices brought a new problem. The need to keep program code away from others has been the main reason these problems have become of more interest to companies. This is because the creation of programs has become more expensive and time-intensive, especially when Machine Learning models are a part of the application. This leads to the following questions. Does one trust the client's device? How does one execute code on data that the client's device should not have access to? Furthermore, is there a way to keep the client device's OS from accessing or influencing the execution of the application's code?

In recent years, research on ways to provide secure execution of code on untrusted devices has progressed. There are multiple ways to archieve the desired goal of code and data confidentiality on remote devices. Trusted Execution Environments (TEE) are one such option to provide the secure execution of code without interference from any other processes on the device. A TEE restricts access to the code and data of an application inside of it. Furthermore, it allows for verification of the TEE's content. A TEE requires some additional hardware on the device as well as software to manage the interactions. There are different implementations of TEE, and each comes with different strengths and weaknesses, some of which will be discussed later in this paper.

In this paper, the main components of a Trusted Executions Environment (TEE) are shown. Furthermore, two implementations of a TEE are being analyzed in terms of their functionality. These implementations are Intel SGX and ARM Trust Zone. Next, in Section 3 the features of both Intel SGX and ARM TrustZone are being shown. Afterwards, the problems of the respective implementation are analyzed. Finally, Section 4 shows different frameworks to help create applications intended for use with TEE.

2. Trusted Execution Environments

The TEE is a Secure Operating System separated from the original device's Operating System (OS). Additionally, it is supported by hardware components to provide the functions required by the applications, which are executed inside the TEE.

The features a TEE provides, depend on the individual implementations. Most commonly, a TEE provides some isolation of the processes from any process running in the normal OS of the device and from other processes inside the TEE. Furthermore, they allow verification of the executed code and data.

As shown by Arfaoui et al. [1], there are different ways to implement the hardware components of a TEE. The first option for hardware components of a TEE includes a trusted ROM, RAM, and a trusted processing environment. Furthermore, the TEE has its own crypto accelerators and can support trusted peripherals. The second option for the hardware components is to share the hardware with the regular OS and have a state that specifies if the currently executed process is trusted or untrusted.

The software part of a TEE is the TEE kernel, which is an OS, that is different from the host OS [1]. It is authenticated and validated during the start of the device and takes control upon the execution of the TEE application. Another software part is the TEE APIs. These APIs can be differentiated into private and public APIs. The private APIs provide a way for the Trusted Applications to use the functions provided by the TEE. The public APIs offer an interaction between applications running in the devices OS and the TEE applications.

Two such implementations are being shown in the following, and their functionality is being explained. Furthermore, the requirements of each implementation are being looked at.

2.1. Intel SGX

The first implementation of TEE looked at is created by Intel, called Software Guard Extensions (SGX). According to Intel [2], SGX provides isolation of program code and data in memory through hardware-based encryption of the memory. They claim this prevents more privileged processes, like the OS, from accessing this information. To use Intel SGX, the device needs to have an Intel CPU that supports SGX and have SGX enabled in the BIOS. The Intel CPUs that support SGX are all 6th to 10th generation processors as well as the server processors of the 11th and 12th generation. SGX is depricated in non server versions of th 11th and 12th generations due to lack of use cases for private users.

The hardware requirements of SGX are similar to the second option shown in Section 2. As such, the hardware used for SGX is primarily the original hardware, with some minor changes required. A Memory Encryption Engine is needed to input and output data from the TEE safely. The Memory Encryption Engine also stores the keys used for encryption of each secure memory area. Furthermore, SGX requires some additional microcode.

In SGX, the encrypted memory section and the respective key that belongs to it are called enclaves. Any user process that wants to use the SGX functions can create an enclave, as shown by Gu et a. [3]. On startup, the enclave is verified, usually through a hash, either on the local machine or remotely. The hash is 256 bits long and includes the code and the initial data of the enclave, as well as security flags and page locations within the enclave. This hash is then signed by either Intel or the program developer. If the developer signed the hash himself, the public key that is needed for verification of the signature has to be signed by Intel and added by the executing device to SGX [4]. The untrusted process, which the enclave belongs to, and the enclave share the same memory address space. The difference being that the enclave's memory is encrypted, as described in Jauernig et al. [5]. Any OS functionality, like memory management, interrupt handling, and I/O is still done by the device's OS. Still, neither the OS nor any other process can access or change any data or code inside the enclave's memory. When an enclave function is called, the encrypted memory section that belongs to this function is loaded into the CPU and then decrypted on the CPU for execution.

Any developer who wants to use SGX for their application has to create two parts. As shown in Figure 1, there is an untrusted part of the application and a trusted part, which is the enclave. The developer decides which functions of the application and which data require the enclave's security and then creates the enclave part out of it. The enclave needs to be verifiable, so a hash of the enclave's contents needs to be made. This hash is used to verify the enclave after the creation on the client device, most likely remotely by a server belonging to the program's creator. The untrusted part initializes the enclave on the client device, which is then verified. Once the application requires the contents of the enclave, the untrusted part calls the respective function of the enclave. The enclave then takes over and executes the called function and returns the output of this function. Then, the program's normal execution continues until an enclave function is required again. For any communication with the enclave that contains data, a secure channel is created by the enclave, and the enclave includes its verification for the other process to verify the data is coming from or going to the correct enclave [4].

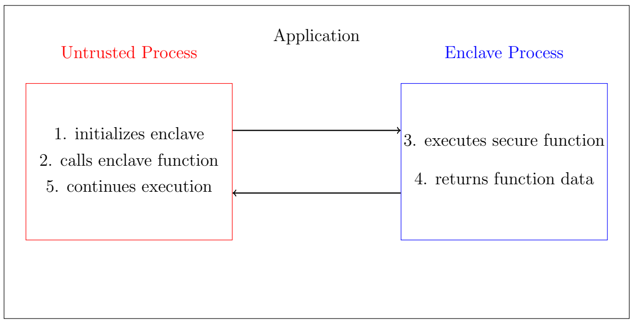

Figure 1: Application using SGX enclave

2.2. ARM TrustZone

The other implementation being analyzed is called TrustZone. It is created by ARM [6]. Similar to SGX, TrustZone's hardware components are the ones described as the second option in Section 2. In TrustZone, the execution environment is split into two parts: the secure world and normal world. These two worlds are not just different terms used by ARM to describe the same thing as untrusted and enclave parts in SGX. While the untrusted part is about the same as the normal world, an enclave does not represent the same as the secure world in TrustZone. To use TrustZone, an ARM processor with TrustZone support is required. The ARM processors that the Cortex-A and Cortex-M classes of their processors.

The hardware changes required for TrustZone are very minimal. The memory controller needs to be able to differentiate between secure and normal world processes. The device's OS is considered a normal world process in this context. The CPU needs to be able to do the same differentiation between secure and normal world. This ability to differentiate keeps anything running in the normal world from interfering with secure world applications. The context switch between these two worlds is done by trusted firmware on the device [5].

On the software side, there is a separate OS that runs in the secure world, unlike in SGX. During the device's boot, the image of TrustZone is verified and authenticated. This is not done for individual application initializations as all the applications run in the same TEE instance for TrustZone, while in SGX, each enclave is a separate TEE instance. Thus any program in the secure world can influence other programs in it. In order to deal with this problem, an application can only be added to the TrustZone of a device if the device's creator allows this application's inclusion.

A developer who wants to use TrustZone can decide to create a secure world-only application. Developers can also choose to develop both an application that runs in the normal world and have an application in the secure world for the security-relevant features.

3. Features and Problems

In the following, different implementations of TEE are being analyzed with a focus on Intel SGX and ARM TrustZone. The features of each implementation are being shown, and finally, the problems of the implementation are being analyzed.

3.1. Intel SGX Analysis

As described in Section 2.1, SGX enclaves are encrypted while they are in the device's memory. Thus an attacker can not read any of the contents of an enclave. Furthermore, the attacker can not access the encryption key, as it is stored on the Memory Encryption Engine. The attacker could still change any part of the encrypted enclave, but the enclave is verified when it is used, which means the attack would be noticed. The enclave can be initialized again to match the actual content the developer intended. These features result in the fact that the program's developer no longer needs to trust the entire device the program runs on. The developer's trust must now only be in the CPU that the code is executed on. SGX provides separation between processes executed inside of enclaves as well, which means the execution of one enclave can not impact any other enclave. However, one feature is not provided by SGX, which is support for trusted peripherals. The fact that this feature is missing means that no device directly connected to the client can influence anything in the enclave or be used directly for I/O purposes.

As for the disadvantages of using SGX, the enclave needs to be decrypted each time its functions are executed. After execution of the enclave function, the entire enclave needs to be encrypted again before it can be stored in the device's memory. Both of these operations increase the application's execution time on the client device. Depending on the size of the enclave and the frequency of calls to enclave functions, this en-/decryption time can be a significant part of the overall execution time of the application. Furthermore, Section 2.1 shows the need for the developer to split the application into two parts and also create means to verify the content, i. e. creating a hash of it and signing the enclave. This increases the development time of such an application.

To summarize, SGX provides an environment to execute code without the interference of other processes on the device and guarantees the confidentiality and integrity of both code and data, as described by Narra et al. [7]. Furthermore, it allows for the remote verification of the created enclave. As for the drawbacks of SGX, the development effort is more significant than making a regular program. On the client-side, the processing overhead increases because the enclave needs to be decrypted for execution and the verification of the data entering the enclave. Dinh Ngoc et al. [8] describe the preformance overhead based on the different calls in SGX and the amount of CPU cycles each of them takes. Finally, SGX does not protect against side-channel attacks. Thus some information about the program can still be inferred.

3.2. ARM TrustZone Analysis

In Section 2.2, TrustZone was shown to have the secure world and the normal world. The secure world is also described to be the part of TrustZone, which holds the TEE part. In the secure world, the applications are shielded from any influences coming from the normal world. Thus no normal world process can read or change the content of the memory areas belonging to any secure world application. Neither can any normal world process interfere in the execution of code belonging to the secure world. Should the attacker have access to the hardware itself though, TrustZone does not offer any protection. This comes from the fact that the memory is not encrypted like it is in SGX, and the context switch is done by the trusted firmware. In TrustZone's case, the developers' trust needs to be put into the TrustZone firmware and the additional hardware parts of TrustZone. However, in contrast to SGX, TrustZone does offer support for trusted peripherals by including the drivers for the peripherals in the secure OS [5]. Unlike SGX, there is no en-/decryption needed for execution, only a context switch that takes very little time to do.

The disadvantages of using TrustZone are, as previously mentioned the fact that there is no separation between applications running in the secure world. Additionally, TrustZone only protects against software attackers and does not protect against hardware-level attackers, as there is no encryption of the secure memory parts. Furthermore, the developers of applications, which are running inside the TrustZone of a device, need to trust each other because of the missing separation. Finally, TrustZone is only verified on boot of the device and does not verify applications before they are executed.

In summary, TrustZone offers an execution environment for code without the influence of processes in the normal world. The integrity of the secure world is checked on boot, and the contents of the secure world are kept confidential from the normal world but not from the secure world. In terms of drawbacks, TrustZone only protects against software attackers. Furthermore, there is no separation between programs inside the secure world, forcing developers to additionally trust other applications in the device'sTrustZone. Similar to SGX, TrustZone does not protect against side-channel attacks.

TABLE 1: SGX and TrustZone features

	Intel SGX	**ARM TrustZone**
Trusted Part	CPU	Firmware and Hardware
Protection against Hardware Attacks	yes	no
Separation of TEE applications	yes	no
Verification of Trusted application	remotely or locally when called	only on system boot (entire TEE not individual applications)
Trusted Peripherals	no	yes

4. Applications

After understanding how Intel SGX and ARM TrustZone work and what they offer, we take a look at how to create applications that use either SGX or TrustZone.

For Intel SGX, there are multiple frameworks, which are meant to help develop an application for it. There are two kinds of frameworks for SGX. The first of them is intended for the creation of new applications which use SGX. The other type of framework can make an existing application run in an SGX environment.

Intel themselves created one framework that is of the first kind, and it is called SGX SDK [9]. SGX SDK is a framework that supports C and C++ programming languages, and its development tools can be used on both Windows and Linux operating systems. It does not only include the libraries for SGX but also contains tools for debugging and some examples to help understand how to use this framework. Intel actively updates the SGX SDK.

Another framework for creating new TEE applications is Open Enclave SDK [10]. Open Enclave is an open-source framework for TEE applications. Same as SGX SDK, it works with C and C++ languages and has versions for both Windows and Linux. Unlike SGX SDK, though, it does not only work for developing applications intended for Intel SGX. It also allows for creating programs designed for use on ARM TrustZone. Open Enclave is also regularly updated by multiple authors.

The other kind of framework is used to make an already existing application run in an SGX enclave. There are multiple commercial frameworks for this purpose, like Fortanix [11]. Furthermore, there are some open-source frameworks like Graphene [12]. They work based on LibOS. To use Graphene, the host it should run on needs to support SGX SDK. The application is signed together with the Graphene enclave part to form the enclave. Then the created enclave is sent to the host it should run on together with Graphene.

Both of the shown TEE implementations are also available for use on different devices. While ARM TrustZone is mainly used on mobile devices, Intel SGX is aimed for use on client pcs and servers. Some cloud service providers already include support for SGX on their platform. One such Cloud service is Microsoft Azure Confidential Computing [13]. Microsoft Azure supports applications using SGX SDK, Open Enclave and some other frameworks used to create SGX programs. Furthermore, it allows for remote attestation of the enclaves. On Azure, there is also support for another kind of TEE called AMD SEV, which was not presented in this paper.

5. Related work

There are other papers, which are looking into different implementations of TEE. One such paper is Jauernig et al. [5], describing five different TEE implementations there. The implementations described are Intel SGX, AMD SEV, ARM TrustZone, as well as the two academic TEE implementations, Sanctum and Sanctuary. They are described in terms of their functionality and the provided features.

Other concepts can provide security features to a device. One such concept is the Trusted Platform Module (TPM). A TPM is a hardware module that contains some data storage capacity, as well as the hardware required to generate both symmetric and asymmetric encryption keys and can create cryptographic hashes. Aaraj et al. [14] explain that a TPM offers cryptographic functions as well as protected storage to perform integrity checks on the platform or any application that is using it. The TPM itself is only encrypted storage for keys in a hierarchical system of keys, some of which are bound to data on the system's storage, and others are just used to keep the bound keys safe. The TPM is only a cryptographic co-processor and cannot be used for general computation.

Another option for trusted execution is smart cards. Smart cards are small chips protected from the physical environment and usually break if one tries to overcome the physical protection of the chip. A smart card has a small connection interface, and some smart cards can even be accessed remotely. A typical example of a smart card is a credit card. Naccache and M'Raihi [15] explains, that the connection interface of a smart card is standardized and smart cards do not possess a power source of their own. In essence, a smart card is a small computer that is powered as long as it is connected to another device and can be authenticated against the connected device or remote users and then do some computation on the smart card. As a smart card is equipped with cryptographic functions, all of the data entering and leaving the card can be secured as well. There are two types of smart cards, cryptographic smart cards, that only offer cryptographic functions like authentication or encryption. The other type of smart card is called the java smart card. This second type of smart card allows for more general computational use of the computer inside.

6. Conclusion

In conclusion, we looked at what a TEE is and what is required to implement a TEE and then analyzed two different implementations. The requirements for a TEE are split into hardware and software parts, and there exist multiple ways to implement a TEE in both of these parts. Of the implementations looked at, each has its respective advantages and disadvantages. SGX is built only to require minimal hardware changes and relies primarily on encrypting memory areas. These areas are decrypted only for the CPU upon execution of the contained code and can not be read or altered without detection. The enclaves in SGX are also kept separate to prevent one from influencing another.

In TrustZone, on the other hand, the separation in different worlds is mainly achieved through the trusted firmware performing a context switch. In Section 2.2, we saw that TrustZone allows for secure peripherals, in contrast to SGX, but it does not separate applications running in the secure world.

Finally, other ways to achieve trusted execution on untrusted devices are shown. Each of these options has its own individual features and problems, each with slightly different use cases. TPM offers cryptographic operations on a coprocessor; smart cards can either be used for cryptographic functions only or be a general-purpose execution environment on a small card that needs to be connected to another device for power.

A TEE does not guarantee that an attacker can not gain information about the code or data executed inside the TEE despite the features provided. The vulnerabilities of specific implementations could be an exciting topic for further research, as well as general problems of TEE.

Another subject for further research could be an analysis of the performance overhead of different implementations. Different implementations use different mechanisms to achieve their features and have very different execution times.

References

[1] G. Arfaoui, S. Gharout, and J. Traoré, "Trusted execution environments: A look under the hood," in *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, 2014, pp. 259–266.

[2] [Online]. Available: https://www.intel.de/content/www/de/de/architecture-and-technology/software-guard-extensions.html

[3] Z. Gu, H. Jamjoom, D. Su, H. Huang, J. Zhang, T. Ma, D. Pendarakis, and I. Molloy, "Reaching data confidentiality and model accountability on the caltrain," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 336–348.

[4] [Online]. Available: https://sgx101.gitbook.io/sgx101/sgx-bootstrap/attestation

[5] P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Trusted execution environments: Properties, applications, and challenges," *IEEE Security Privacy*, vol. 18, no. 2, pp. 56–60, 2020.

[6] [Online]. Available: https://developer.arm.com/ip-products/security-ip/trustzone

[7] K. G. Narra, Z. Lin, Y. Wang, K. Balasubramaniam, and M. Annavaram, "Privacy-preserving inference in machine learning services using trusted execution environments," *arXiv preprint arXiv:1912.03485*, 2019.

[8] T. Dinh Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont, "Everything you should know about intel sgx performance on virtualized systems," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, pp. 1–21, 2019.

[9] [Online]. Available: https://01.org/intel-softwareguard-extensions

[10] [Online]. Available: https://github.com/openenclave/openenclave/tree/master/docs/GettingStartedDocs

[11] [Online]. Available: https://fortanix.com

[12] [Online]. Available: https://graphene.readthedocs.io/en/latest/oldwiki/Introduction-to-Graphene-SGX.html

[13] [Online]. Available: https://docs.microsoft.com/de-de/azure/confidential-computing/enclave-development-oss

[14] N. Aaraj, A. Raghunathan, and N. K. Jha, "Analysis and design of a hardware/software trusted platform module for embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 8, no. 1, pp. 1–31, 2009.

[15] D. Naccache and D. M'Raihi, "Cryptographic smart cards," *IEEE micro*, vol. 16, no. 3, pp. 14–24, 1996.