Journal of Computing Research and Innovation (JCRINN) Vol. 8 No.2 (2023)
https://jcrinn.com : eISSN: 2600-8793 / https://dx.doi.org/10.24191/jcrinn.v8i2.371

Contribution of TEE and Parallel Computing to Performance and Security of Biometric Authentication Improvement

Maxim EDOH [1]*, Tahirou DJARA[2], Aziz SOBABE[3], Antoine VIANOU[4]
[1,2,3,4] Doctoral School of Engineering Sciences, University of Abomey-Calavi, Abomey-Calavi, Benin

*Corresponding author: * maxime.edoh@gmail.com*

Received Date : 6 June 2023
Accepted Date : 5 July 2023
Revised Date : 10 August 2023
Published Date : 1 September 2023

HIGHLIGHTS

- Fingerprints have not to be exported outside the secure runtime environment.
- If a TEE is compromised, it can attack more TAs and leak secrets such as fingerprint data or keys.
- Isolating tasks avoids potential interference between different biometric operations.
- Computation can be accelerated by using multiple threads.
- Intel SGX is a hardware-assisted EE isolation technology to secure computation.

Abstract

In this work, we present a new idea for improving biometric authentication of personal identities on mobile devices with significant accuracy and convenience because the problems that arise here are the vulnerability and confidentiality of biometric data before, during and after a biometric recognition operation, and the low processing speed during the same operation. The aim of this work is to propose a combination of the concepts of parallel computing and Trusted Execution Environments (TEE) as a solution to the problems raised. We thus address how parallel computing can be useful in accelerating the recognition of individuals, by reducing the interaction time with the database thus preventing some vulnerabilities. Finally, a hardware-assisted technology namely Intel SGX (Software Guard Extensions) is proposed for practical implementations.

Keywords *: parallel computing, trusted execution environment, biometric data, SGX.*

INTRODUCTION

Today, biometric recognition represents a major step forward in guaranteeing the authenticity of personal identity. The retrieval, storage and use of biometric data are required by many hardware and software platforms in a variety of applications. However, these data are subject to security and confidentiality threats. What's more, the operations involved in processing them are becoming increasingly resource-intensive.

Mobile devices are the main communication and entertainment device for many people, and frequently run trusted programs to handle sensitive and confidential data. Unfortunately, these mobile devices can also run a wide variety of potentially malicious programs. As such, they require isolation mechanisms that prevent untrusted programs from tampering with the code or data of trusted applications.

However, to provide a higher level of security as required by these applications, a number of proposals have been put forward, including the Trusted Execution Environment (TEE). In this article, we examine how TEE fits into the overall picture of services on a smartphone. We also analyze the current state of the art of the TEE proposition and the potential obstacles it may face due to the nature of current trends. Finally, we provide a potential pathway to overcome these issues in order to achieve large-scale deployment, enabling secure services to individual users.

Biometric authentication on mobile platforms

Some general information

To unlock their mobile devices more easily, users now prefer biometric authentication, such as fingerprint sensors, which also reduce the cognitive load associated with memorizing several long passwords.

Appropriate use of biometrics also increases security. Passwords are easy to steal; forging biometrics is much more difficult. The technology is ideal for providing role-based access controls and a high level of trust for business users.

How do biometrics work on mobile platforms?

Unlike passwords or PINs, biometric data is not stored on the network or transmitted between devices and servers. Instead, biometrics protect other authentication information (usually a digital certificate or private key), and it is this protected information that is actually used to verify the user.

The first step in understanding and implementing biometrics is to have a standardized service that can run on various mobile platform applications, such as a secure runtime environment (TEE[1]). In the Android v6.0 compatibility definition, for example, there are two notable requirements for the integration of this hardware-based measure:

- Smartphone vendors must use a TEE to match fingerprints.

- Fingerprints must be encrypted, signed and stored in such a way that they cannot be exported (or read) outside the TEE.

On platforms such as the Samsung Pass SDK, encryption stores data in the TEE and never leaves the device. A public/private key pair is "locked" with biometric data, and the fingerprint is used to unlock the keys implemented by the smartphone and Samsung Pass, to authenticate to and access the remote application.

TEE: Trusted execution environment

What is it?

In the field of data security and confidentiality, Trusted Execution Environments (TEEs) are emerging as a bastion of protection within IT environments. A TEE is a secure enclave where code and data can be executed with the highest degree of confidence, safe from outside interference and malicious attack

[1] In this article, by TEE we mean software running in the secure world of ARM TrustZone, with the exception of trusted applications (TAs). TEE provides an execution environment for TAs.

(Sabt et al., 2015). It forms a hardware fortress that protects sensitive operations, cryptographic keys and authentication processes from unauthorized access and compromise. TEEs enable applications to operate in a confidential environment, even in the presence of untrusted or compromised components, underpinning secure operations in areas such as mobile devices, cloud computing and Internet of Things (IoT) ecosystems. This trusted enclave is the cornerstone for securing critical data and computational integrity in an increasingly interconnected, data-centric world.

Interest

Trusted Execution Environments (TEEs) have a number of advantages, including :

- **Secure isolation:** TEEs provide a secure enclave in which code can be safely executed. This enclave is isolated from the rest of the operating system and applications, reducing the risk of malicious interference.
- **Key and data protection:** encryption keys, biometric data and processing operations are stored and executed within the TEE's secure environment. This reduces the risk of exposure of sensitive keys and data.
- **Authenticity and integrity:** TEEs are designed to guarantee the authenticity and integrity of the code executed inside the enclave. This ensures that the code has not been modified or altered, which is crucial for the security of biometric operations.
- **Protection against physical and logical attacks:** TEEs offer protection against a range of attacks, including auxiliary channel attacks, reverse engineering attacks and brute force attacks. Even if an attacker manages to compromise the operating system, the integrity of the TEE generally remains intact.
- **ID and session management:** TEEs can be used to manage user IDs and biometric sessions securely, ensuring that authentication information is protected against interception or manipulation.
- **Extended security functions:** some TEEs offer extended security functions, such as secure key generation, secure encryption/decryption operations and secure time-stamping functions.

TEE deployment on mobile devices

Trusted execution environments (TEEs) are widely deployed, particularly on smartphones. A recent trend in TEE development is the transition from vendor-controlled, single-use TEEs to open TEEs that host trusted applications (TAs) from multiple sources with independent tasks. This transition should create the TA ecosystem needed to provide enhanced, customized security for the applications and operating system running in the rich runtime environment (REE). However, the transition also poses two security problems: an enlarged attack surface resulting from the increased complexity of TA and TEE; and the lack of trust (or isolation) between TA and TEE.

TEE (Trusted Execution Environment) is becoming increasingly popular, especially on mobile devices. Since its inception, TEE development has gone through the following stages.

In the first stage, TEE is mainly used for a secure boot that checks the validity of the operating system loaded in the rich runtime environment (REE), for example, in Motorola X/G/E cell phones.

In the second stage, TEE begins to support more features, including encryption, fingerprint authentication, mobile payment, Trusted User Interface (TUI), Digital Rights Management (DRM) and more. Some manufacturers are also exploiting TEE's high privilege to provide runtime protection for REE, for example, TrustZone Integrity Measurement Architecture (TIMA) in Samsung's KNOX

(Samsung Inc., 2018). Up to this point, the functionality of a TEE is generally fixed (i.e. it cannot change after manufacture). For example, if a third-party company needs to deploy a trusted application (TA) for mobile payment in TEE, it must pre-install the TA before the devices leave the factories. Today, the third stage of TEE development is underway. The new TEEs now support dynamic (post-manufacturing) installation of TAs. There are already a few TA app stores from which phone users can easily download and install TAs, just like installing an ordinary app, for example Samsung's trustlets (Samsung Inc., 2018) and TrustKernel's TEEReady (TrustKernel, 2018). GlobalPlatform (GlobalPlatform, 2018) has proposed a set of APIs for communication between TEEs and REEs, which most commercial TEEs now follow as a de facto standard. ARM is also leading a group of TEE vendors to create the Open Trust Protocol (OTrP) (ARM, 2016), which combines a secure architecture with TA management. The aim of these efforts is to improve the compatibility and deployability of TAs with different TEEs. However, this increasing openness and flexibility makes TEE more complex and widens the attack surface. To support various TAs, TEEs are being developed with more features, leading to a significant increase in the size of the Trusted Computing Base (TCB). Meanwhile, as TEEs have to support the dynamic installation of new TAs, it is no longer possible for manufacturers to carry out comprehensive security tests at the factory. At present, there are more than ten TEE suppliers on the market. TEE's larger TCB and more dynamic TEE ecosystem pose two challenges: weakening security and increasing distrust. Firstly, TEE generally has the highest privilege in the system, so if TEE is compromised, the security of the whole system can be compromised. For example, an attacker can take advantage of a TEE bug to write arbitrary REE memory, known as the Boomerang attack (CVE-2016-8764). Meanwhile, if a TEE is compromised, it can attack more TAs and leak secrets such as fingerprint data (CVE-2015-4422) or keys (CVE-2015-6639). It is expected that the number of TEE vulnerabilities will continue to increase in the near future due to the expanded attack surface. Secondly, since there is currently only one TEE in each device, all TAs must trust the TEE unconditionally. However, this trust is becoming increasingly difficult to establish as more and more TAs from different sources enter the TEE. TA suppliers may demand a higher security standard or more security features than those provided by some TEEs. It is also possible that a TA supplier has a conflict of interest with a TEE supplier and prefers to trust/execute in another TEE.

Table 1: Real-world mobile commercial TEE suppliers and products

	Sales	**TEE name**	**Architecture used**
Flea marketer	Qualcomm	QSEE	ARM32, ARM64
	Spectrum	Spectrum TEE	ARM32
	HiSilicon	TrustedCore	ARM32
TEE seller	Apple	Enclave	ARM32
	TrustKernel	T6	ARM32
	Trustonic	Kinibi	ARM32
	Google	Trustee	ARM32
	Linaro	OP-TEE	ARM32
	SierraWare	Sierra TEE	ARM32
	Proven&Run	ProvenCore	N.A.

TEE vulnerabilities

Many vulnerabilities are critical for a large number of devices. An article from Google's Project Zero (Google Project Zero, 2017) shows how to exploit two major TEEs on real mobile devices. It concludes that "... despite their highly sensitive point of view, these operating systems currently lag behind modern operating systems in terms of mitigation and security practices." Many CVEs are not well documented and are slow to be published. For example, CVE-2016-10238 was first discovered in 2016 but only published in March 2017, and the description is brief: "... Technical details are unknown and an exploit is not publicly available." Yet we try to analyze each vulnerability with the best effort. We find that there are three basic categories of TEE-related vulnerabilities: first, TEE vulnerabilities can lead to leakage of secret data or execution of arbitrary code, e.g., CVE-2017-0518/0519, CVE-2016-2431/2432, etc. Secondly, TEE vulnerabilities allow one TA to affect the security of other TAs, for example, CVE-2016-0825 and CVE-2015-6639/6647. Thirdly, TEE vulnerabilities can be exploited to obtain an elevation of privileges in the REE, for example, CVE-2016-8762/8763/8764. The causes of these attacks include the lack of isolation and the semantic gap between different execution environments. One of the main reasons behind the above vulnerabilities is that there are many interactions between applications running in REE and TEE, which usually take place via shared memory. As stated by Machiry et al (Machiry et al., 2017): "TEE has very limited visibility into the security mechanisms of the unapproved environment", which is referred to as the "semantic gap" between TEE and REE. Thus, a malicious CA (client application running in REE) can ask the TEE to overwrite data in the REE kernel. Although the CA itself cannot do this because it has no privilege, the TEE has a higher privilege and can violate the REE's security mechanism through logical errors.

Need for several isolated TEEs

Currently, all TAs installed on a device must trust the only TEE kernel available on that device. This forced trust is increasingly called into question as the TA ecosystem becomes more diverse and open. Let's take the example of a mobile payment TA running in a TEE offered by the phone manufacturer. If an attacker exploits a TEE bug to steal the TA payment's private key, he can in fact steal the user's money directly. In practice, the payment company ends up compensating the user for the TEE fault, like AliPay (Alipay Member Protection, 2018). The point here is that currently, a TA must trust the only TEE available on a device, even though the TEE may not meet the TA's security requirements or is not trustworthy to the TA. On the other hand, if the system supports multiple TEEs, the phone manufacturer may offer a default TEE (system TEE in this article) to run the manufacturer's TAs. At the same time, TAs with different security requirements can install TEE instances they trust. Protecting the TEE from attack by hardening the TEE itself could be a research direction for the security problem we are addressing. However, there are various TEE OS products from different vendors widely deployed in billions of devices, each with divergent design and implementation. It is difficult to protect them one by one in practice.

Biometric data protection during read/write operations

Isolating biometric tasks is of crucial importance to guarantee the security, confidentiality and integrity of biometric data, as well as to avoid potential interference between different biometric operations. As shown by Ayaz Akram (Akram, 2021), Trusted Execution Environments (TEEs) are ideally suited to this task.

We propose the use of TEE in a parallel processing environment to isolate biometric tasks and protect them from each other.

In a parallel processing environment, several biometric tasks may be running simultaneously on different cores or processors. TEE will enable each biometric task to be isolated in its own secure space. So, even if one task is compromised, the other tasks remain protected and secure.

Furthermore, thanks to this process, biometric data is not exposed to unauthorized processes, minimizing the risk of leakage or unauthorized access.

When it comes to encrypting biometric data, TEEs will provide a secure environment for the use of keys, preventing their exposure to unauthorized parties.

Similarly, TEEs are designed to resist both physical and lateral attacks. This means they are less susceptible to attacks based on power consumption measurement, timing analysis or other methods used to compromise system security.

Parallel calculation

Definition

In contrast to sequential computing, parallel computing involves the simultaneous execution of a single task, partitioned and adapted so that it can be distributed between several processors to process larger problems more quickly:

$$sequential\_time \rightarrow parallel\_time = sequential\_time/number\_of\_resources$$

Interest

Parallel computing has a wide range of applications, as specified in (Abdellatif, 2016) and (Wang et al., 2018). Among the many advantages it offers, we would highlight the following:

- performance enhancement
- better use of resources
- a trade-off between performance and price
- edge computing
- resource sharing
- scalability

Some examples of works using parallel computing to enhance biometric authentication

In (Mangata et al., 2022), Mangata et al. evaluated the runtime performance of a single-mode biometric recognition system for fingerprint-based access control to secure premises. To accelerate computation time in this system, they resorted to parallel programming, targeting more loops in the verification module. Their approach was to parallelize all computationally-intensive loops when verifying fingerprints in the database by leveraging Microsoft's parallel task library, specifically exploiting the for and for each loops.

Supatmi et al. in (Supatmi et al., 2020), studied fingerprint matching using the Bozorth3 algorithm to match fingerprints and parallel computing using NVIDIA Compute Unified Device Architecture (NVIDIA CUDA). In this study, fingerprint matching is performed with parallel computing applied to the Graphics Processing Unit (GPU). The GPU device used in this study is the CUDA (Compute Unified Device Architecture), which is an application programming interface (API) developed by NVIDIA. Their results show that the CUDA runtime process is better than the CPU runtime process.

Our technology proposal

Intel SGX (Intel Software Guard Extensions (SGX)) (McKeen et al., 2013) is a processor instruction set extension that enables the creation of a 32-bit trusted execution environment (TEE), a secure enclave.

By incorporating reliable runtime technologies such as the widely available Intel Software Guard (SGX) extensions into mobile devices, applications can be made secure. However, software engineers need to align the development process with the capabilities and properties of such technology, in order to properly secure applications while achieving good performance.

METHODOLOGY

Performance Overhead of Native SGX

Switching Delay

The switching delay comes from the switching from normal execution environment to trusted execution environment, or vice versa. Many operations can introduce the switching delay, e.g, an ECALL or OCALL, or even a system call. An ECALL or OCALL will obviously slow down the execution because of switching between trusted environment and untrusted environment. As for a system call, it cannot be executed inside a SGX environment directly, and needs to be executed outside. Therefore, a system call is comprised in an OCALL for native SGX applications. Same as an OCALL, switching to the normal execution environment to perform a system call will inevitably introduce a delay. Switching delay is the major source of overhead for some I/O intensive applications, to reduce the overhead, developers need to invoke ECALLs, OCALLs wisely.

Analysis of Switching Delay

The switching delay of the Intel SGX happens every time when a SGX program tries to cross the boundary between trusted and untrusted environment, i.e., whenever an ECALL or OCALL happens. In this section, we will have discussion on how it can affect the performance of SGX applications and evaluate it.

To evaluate the switching delay, we use two groups of functions. In general, functions in group 1 do not have switching delay and functions in group 2 contain switching delay. For group 1 which listed in

Listing 3.1, it contains three functions which execute in the normal execution environment. They can be executed in the normal environment directly, which means invoking them does not require switching the environment, and therefore will not introduce any switching delay. For group 2 which listed in Listing 3.2 and Listing 3.3, it contains one OCALL and four ECALLs. To execute these functions, we must start from untrusted part of the application and then switch to the SGX environment, therefore the switching delay will be introduced. And an nested ecall_ocall function will even experience such delay twice.
We write some simple and representative functions for normal and SGX environment. The functions *normal_empty* and *ecall_empty* are the simplest functions that can reflect the influence of the switching delay; the function *ecall_ocall* is simplest ECALL-OCALL pair; the functions *normal* and *ecall_print* can be used to show how print statements can slowdown the execution; and the functions *normal_malloc* and *ecall_malloc* can be used to show whether a simple memory allocation can affect the performance.

Listing 3.1: Normal functions

```
void normal_empty()
{
    ; // do nothing
}

void normal_print()
{
    printf("print something.\n");
    // print directly
}

void normal_malloc()
{
    void *ptr =
        malloc((size_t)1024);
    free(ptr);
}
```

Listing 3.2: OCALL function

```
void ocall_empty()
{
    ; // do nothing
}
```

Listing 3.3: ECALL functions

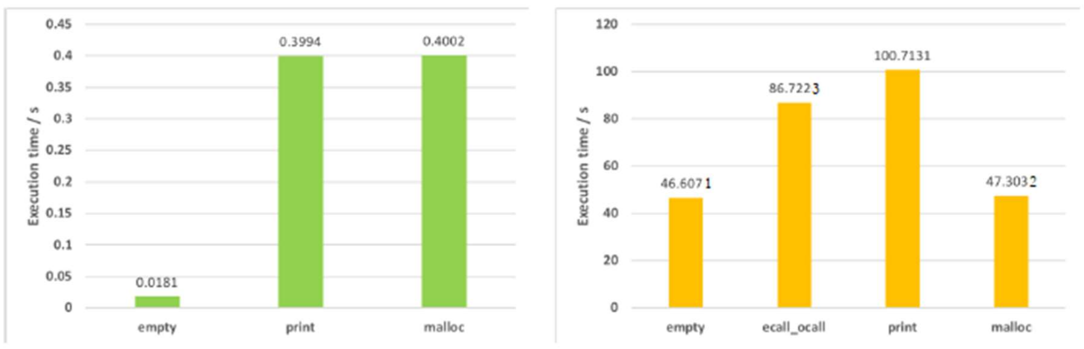
```
void ecall_empty()
{
    ; // do nothing
}

void ecall_ocall()
{
    ocall_empty(); // nested call
}

void ecall_print()
{
    printf("print something.\n");
    // do an ocall and print
}

void ecall_malloc()
{
    void *ptr =
        malloc((size_t)1024);
    free(ptr);
}
```

All experiments use Intel NUC7i5BNH with an i5-7260U processor with 4 cores at 2.20 GHz with 4 MB cache and 12 GB main memory.

For each experiment, we execute each function for one million times, and 50 runs for each experiment are reported so that we can calculate the average. Figure 2 shows the results. From the figure, we can see that the switching delay is absolutely significant, e.g., an empty ECALL *ecall_empty* is more than 2500x times slower than the normal_empty function in the normal environment. The ecall_ocall function takes approximately double time of *ecall_empty* and this is consistent with the fact that an ECALL which contains an OCALL would experience the switching delay twice.

For the performance of *normal_print* and *ecall_print* functions, we can see printing statements will heavily affected the execution performance. In the normal environment, because invoking a *printf* function involves system calls which can take a relatively longer time, the performance is affected. And in the SGX environment, a print statement will introduce an ECALL delay and an OCALL switching delay plus preparing time for the string, as well as the real printing time in the normal environment. In general, using a print statement in the SGX environment is more than 250x times slower than using a printf directly in the outside.

As for the *ecall_malloc*, its execution time is only slightly higher than the simplest ECALL *ecall_empty*. This means the switching delay is the major source of performance overhead compared to libc functions in the enclave and system calls outside.
In general, the surprisingly high performance overhead introduced by the switching delay of Intel SGX needs to be paid attention with. Developers must build the SGX applications wisely and try not to abuse the ECALL, OCALL or any other functions that will introduce the significant switching delay.



(a) Group 1: normal functions (b) Group 2: SGX functions

Figure 2: Performance of normal and SGX functions

DISCUSSION

Practically it is observed from (Ngoc et al., 2019) that "SGX imposes a heavy performance penalty upon switching between the application and the enclave, ranging from 10,000 to 18,000 cycles per call depending on the call mechanism used". It is well known that energy saving is the ultimate aim of any mobile device as they are battery operated. If such heavy mechanism is employed then battery of mobile may drain fast.

Optimization

Tian et al. in (Tian et al., 2018) pointed out that SGX will cause CPU performance loss. The reason is the frequent enclave switch. SGX provides standard functions OCall and ECall for users to get in and out of the enclave. They performed more than 8000 CPU cycles. So researchers used “Switchless Calls” to improve SGX performance by minimizing the number of OCall and ECall calls. They present another shared memory based switchless enclave function call schema for Intel SGX. The schema has been included in recent versions of the Intel SGX SDK as an official feature. They argue that it is not always worth dedicating an entire logical core to an enclave worker thread in exchange for faster enclave transitions. The novelty of this implementation is that it makes it possible to decide at runtime whether to use switchless enclave function calls or normal ECalls. This technique aims to utilize the available CPU resources more efficiently. The general idea is that at points in time when the frequency of enclave function calls is low, then an ECall is affordable. However, switchless enclave function calls should be used at points in time where enclave functions are called in high frequency.

In effect, a recent patch in Linux SGX SDK contains switchless calls, which reduces an enclave mode switch overhead during the enclave transition. The goal of switchless calls is to eliminate enclave switches from SGX applications by making ECALL and OCALL themselves switchless, which are functions used for entering/leaving SGX enclaves. For this, an SGX run-time library executes two worker threads, one in the application (untrusted) memory region and the other in the enclave (trusted) memory region. The application worker thread handles ECALL, while the enclave worker thread

handles OCALL, respectively. There are two thread pools to handle switchless ECALL and OCALL, and worker threads are executed asynchronously. For asynchronous execution, switchless calls utilize two shared queues: a request queue and a response queue.
Figure 1 illustrates the workflow of a switchless OCALL. The implementation of switchless calls adopts a sleep-wake approach for efficiency. When a caller thread inside an enclave invokes an OCALL, it first updates the request queue. Then, one of the worker threads from the thread pool in the untrusted region is assigned and handles the OCALL.
Finally, the worker thread updates the response queue. Note that the current version of Linux SDK reflects switchless SGX implementation for common operations inside an enclave, such as threading, file I/O, and system clock, to eliminate OCALLs.

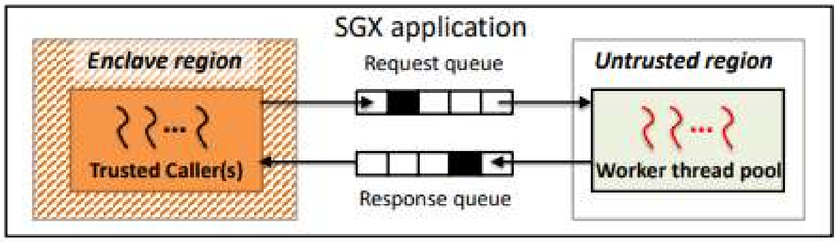


Figure 1 : Workflow of OCALL

CONCLUSION

The issue of security and confidentiality of biometric information is one that our solution addresses. Firstly, the trusted execution environment approach is used to guarantee the confidentiality of said data during processing. Finally, we reinforce the implementation of these different operations by the principle of parallel computing. SGX is a credible enforcement technology that is not too trendy, but the research of it is still on the road. As one of the new solutions to cloud security issues, SGX provides a new perspective and strategy for rethinking our approach to cloud security. However, the experimental results of the performance analysis of SGX have shown that the overhead of SGX runtime is enough to affect the performance of cloud applications. Given the considerable performance costs and the complexity of breaking down the application structure into multiple parts, SGX seems ill-suited for today's cloud environments. To summarize, one of SGX's open challenges for large-scale, complex cloud applications is how to map an application to enclaves to provide the best balance of TCB size, performance, and data security. Future work can focus on finding an effective solution to the challenges posed by SGX performance issues in the cloud.

FUNDING

The authors received no financial support for the research, authorship and publication of this article.

ACKNOWLEDGMENTS

The authors appreciate the reviewers for their contributions towards improving the quality of this research.

CONFLICT OF INTEREST DISCLOSURE

All authors declare that they have no conflicts of interest to disclose.

REFERENCES

Sabt, M., Achemlal, M. & Bouabdallah, A. (2015). Trusted Execution Environment: What It is, and What It is Not, IEEE Trustcom/BigDataSE/ISPA, Helsinki, Finland, 2015, pp. 57-64, doi: 10.1109/Trustcom.2015.357. https://ieeexplore.ieee.org/abstract/document/7345265

Samsung Inc. (2018). « Samsung KNOX ». Retrieved December 01, 2018, 2018, https://www.samsungknox.com/en/knox-platform/knox- security

TrustKernel. (2018). « TrustKernel TEEReady ». Retrieved December 01, 2018, 2018, https://dev.trustkernel.com/ready

GlobalPlatform. (2018). Retrieved December 01, 2018, 2018, https://www.globalplatform.org/

ARM. (2016). « Connected devices need e-commerce standard security say cyber security experts. », Retrieved January 01, 2016, 2016, https://goo.gl/1ePiQC

Google Project Zero. (2017). Retrieved March 11, 2017, 2017, https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html

Machiry, A., Gustafson, E., Spensky, C., Salls, C., Stephens, N., Wang, R., Bianchi A., Choe, Y. R., Kruegel, C. & Vigna, G. (2017). BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments.

Alipay Member Protection. (2018). Retrieved December 01, 2018, 2018, https://intl.alipay.com/ihome/user/protect/memberProtect.htm

Akram, A. (2021), Trusted Execution for High-Performance Computing, http://www.ayazakram.com/papers/eurodw.pdf

Abdellatif, M., (2016). Accelerating mobile security processing with parallel computing. Electronic master's thesis, Montreal, High Technology School.

Wang, T. & Kemao, Q. (2018). Parallel computing in experimental mechanics and optical measurement: A review (II), https://www.sciencedirect.com/science/article/pii/S0143816617302154

Padua D., (2011). *Encyclopedia of parallel computing, volume 4*,

https://books.google.com/books?hl=fr&lr=&id=Hm6LaufVKFEC&oi=fnd&pg=PR1&dq=Padua+D.,+2011.+Encyclopedia+of+parallel+computing,+volume+4&ots=uFGOhSz9dU&sig=gyggeh8xovYXwcAXBoqa8Gf9LOA

Mangata, B. B., Muamba, K., Khalaba, F., Bukanga, C. P. & Kisiaka, M., (2022), Parallel and Distributed Computation of a Fingerprint Access Control System, *Journal of Computing Research and Innovation (JCRINN) Vol. 7 No. 2 (2022) (pp1-9)*

Supatmi, S., Sumitra, I. D., (2020), Fingerprint Matching Using Bozorth3 Algorithm and Parallel Computation on NVIDIA Compute Unified Device Architecture, *IOP Conference Series: Materials Science and Engineering*

McKeen, F. et al. (2013). Innovative Instructions and Software Model for Isolated Execution. In *HASP*.

Dinh Ngoc, T., Bui, B., Bitchebe, S., Tchana, A., Schiavoni, V., Felber, P., & Hagimont, D. (2019), Everything you should know about Intel SGX performance on virtualized systems, *Proceedings of the ACM on Measurement and Analysis of Computing Systems*.

Tian, H.; Zhang, Q.; Yan, S.; Rudnitsky, A.; Shacham, L.; Yariv, R.; Milshten, N. (2018), Switchless Calls Made Practical in Intel SGX. In *Proceedings of the 3rdWorkshop on System Software for Trusted Execution*.