On Making Emerging Trusted Execution Environments Accessible to Developers

Thomas Nyman[1,*], Brian McGillion[1,†], and N. Asokan[2]

[1] Intel Collaborative Research Institute for Secure Computing (ICRI-SC)
at Aalto University, Finland. *thomas.nyman@aalto.fi
†brian.mcgillion@aalto.fi
[2] Aalto University and University of Helsinki, Finland. asokan@acm.org

Abstract. New types of Trusted Execution Environment (TEE) architectures like TrustLite and Intel Software Guard Extensions (SGX) are emerging. They bring new features that can lead to innovative security and privacy solutions. But each new TEE environment comes with its own set of interfaces and programming paradigms, thus raising the barrier for entry for developers who want to make use of these TEEs. In this paper, we motivate the need for realizing standard TEE interfaces on such emerging TEE architectures and show that this exercise is not straight forward. We report on our on-going work in mapping GlobalPlatform standard interfaces on TrustLite and SGX.

1 Introduction

For more than a decade the vast majority of smartphones and tablets have been equipped with hardware security functionality, usually referred to as *Trusted Execution Environments* (TEEs) [3]. A TEE is an isolated and integrity-protected processing environment where sensitive computations, such as cryptographic operations, can be safely carried out. Until recently, application developers have not had the means to make use of TEEs to enhance the security and privacy of their applications. New standardization efforts, such as GlobalPlatform (GP) [5], and open source implementation initiatives, such as OP-TEE [8] and Trusted Little Kernel [11] have the potential for ushering in widespread use of TEEs by application developers.

Although the deployed base of mobile devices with TEEs is very large, numbering in hundreds of millions, they predominantly follow the same architectural pattern: a computing device containing a physical or logical TEE where small amounts of sensitive computation can be carried out in conjunction with a larger software components operating outside the TEE. The chief example of such a TEE architecture is ARM Trust-Zone [1] which is widely deployed in smartphones.

Recently new types of TEE architectures have been proposed. They range from TEEs like TrustLite [7] and SMART [4] designed for tiny resource constrained devices to Intel Software Guard Extensions (SGX) [10] intended primarily for servers and desktops. They provide novel functionality but come with their own *Application Program Interfaces* (APIs) and development environment. This constitutes a high barrier for entry for developers depriving them of the chance to benefit from such TEE functionality.

A natural solution approach to this problem would be to support a set of standard TEE interfaces, such as those specified by GP, on these emerging TEE architectures thereby allowing developers familiar with the standard interfaces to readily make use of the new TEEs. However, subtle differences in architectural assumptions pose some challenges in mapping GP interfaces to these TEEs. In this work-in-progress paper, we examine these challenges.

Our contribution two-fold. First, we briefly juxtapose the features of the emerging TEE architectures with the assumptions behind TEE standards (Section 2). We then show the subtleties and challenges in realizing the GlobalPlatform model on two such example TEE architectures: TrustLite and SGX (Section 3. We briefly describe our experience so far in resolving these challenges.

2 Background

The isolation and integrity-protection of the processing environment in a TEE can be achieved in different ways. Contemporary TEE architectures commonly utilize either a dedicated security co-processor or CPU extensions that allow one physical processor to operate efficiently in two distinct isolated modes in a so called *split-world* configuration. In either case, the device may viewed as having two separate environments, each with its own set of features, as shown in Fig. 1. The *Rich Execution Environment* (REE) refers to the operating environment that houses the conventional OS and applications. In contrast, the TEE typically only has a limited set of features that are only intended to address security critical functionality offloaded onto the TEE by *Client Applications* (CAs) in the REE. The isolation of the TEE itself and discrete pieces of software inside the TEE, referred to as *Trusted Applications* (TAs) is realized by means of hardware support. Services and APIs available to TAs, such as access to trusted storage, are provided by a TEE OS, part of the TEE *Trusted Computing Base* (TCB).
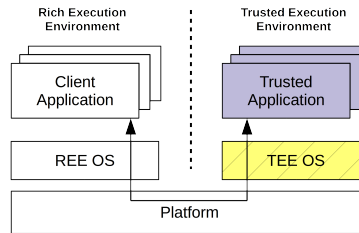
Fig. 1. Abstract view of computing device equipped with a TEE [5]

2.1 TrustLite and TyTAN

TrustLite [7] is a generic hardware security architecture intended for low-cost and exceedingly resource constrained embedded systems, such as automotive electronics, industrial control systems, medical implants or wearables. Such classes of devices have

particularly strong economic incentives to minimize development and production costs, and hence typically lack hardware support for isolated execution or even support for paging and virtual memory. Nevertheless, in many cases such devices are employed in security-sensive and safety-critical applications, which not only require real-time guarantees, but can also benefit from certain security features, such as strong isolation and the possibility of attestating local state to a (remote) verifier.

Lacking a conventional *Memory Management Unit* (MMU), memory access in these constrained environments is typically mediated by a *Memory Protection Unit* (MPU), which can be programmed in supervisor mode by the device OS with memory access rules for the next task scheduled to run. In TrustLite the basis for strong isolation between tasks is an *Execution Aware Memory Protection Unit* (EA-MPU), which not only enforces access control on all memory accesses, but does so considering the current program counter value when validating a particular access to memory. Hence, a platform equipped with an EA-MPU can enforce fine-grained access control based on the individual code regions executed independently of the OS. TrustLite also introduces a secure exception engine which maintains the memory isolation of secure tasks protected by the EA-MPU even in the case of hardware and software interrupts.

Later work by Brasser et. al. leverage the TrustLite architecture to realize Ty-TAN [2], a security architecture for embedded systems that provides strong isolation of dynamically configurable tasks assisted by hardware features introduced by TrusLite, as well real-time scheduling guarantees. TyTAN utilizes the secure exception engine of TrustLite to provide a secure IPC mechanism, where both the sender and receiver can be authenticated using the digest of respective tasks measured upon loading. TyTAN does not provide any built-in access control for IPC. Instead, e.g., the task on the receiving end of an IPC call can make access control decisions based on the verified identiy of the sender. TyTAN differs from the traditional TEE model in Fig. 1 in that normal and secure tasks co-exist in a single environment[3], and rely on scheduling provided by a untrusted (real time) operating system. The OS is assumed schedule tasks fairly without starvation. Normal tasks are isolated from other tasks but are fully controlled and accessible by the OS. In contrast, secure tasks are isolated by the EA-MPU from the rest of the system, including the OS. Whereas CAs in Fig. 1 are thought to include the advanced domain logic of an application, and TAs only very limited security sensitive functionality, all tasks in TyTAN are relatively simple. The primarily concern in TyTAN is the correct operation of secure tasks, even in the face of an adversary who has full control of the untrusted OS and normal tasks running on the platform.

2.2 Intel SGX

Intel Software Guard Extensions [10] is another hardware-based approach to realize an isolated environment for preserving the confidentiality and integrity of sensitive code and data. SGX consists of a set of new CPU instructions and memory access changes to the Intel CPU architecture, which allows parts of the application code and data residing in main memory to be encrypted using a key accessible only in the CPU core.

---

[3] Instead, the class of devices TrustLite represents may potentially be used as part of a programmable secure co-processor.

The protected portions of the application's virtual address space together with the corresponding SGX control data structures are referred to as an *enclave*. Enclave creation and initialization is handled via two dedicated instructions, *Enclave CREATE* (`ECREATE`) and *Enclave INIT* (`EINIT`). When an enclave is operated on, a dedicated hardware unit on the CPU *System on Chip* (SoC) decrypts incoming, and encrypts outgoing traffic between the main memory and CPU package, so that sensitive code and data never leave the CPU package unencrypted. Transfer of execution into and out of an enclave is strictly controlled.

Entry to an enclave occurs via a dedicated *Enclave ENTER* (`EENTER`) CPU instruction, which causes any cached page table translations overlapping with the protected address region of an enclave to be invalidated and transfer control to code inside the enclave. While the CPU is executing in *enclave mode*, it has access to the protected pages belonging to the currently executing enclave, as well as any unprotected pages in the current processes's virtual address space. Accesses to protected pages belonging to other enclaves is prevented. Outside of enclave mode, access attempts by the CPU to enclave pages are treated as references to nonexistent memory, as are physical memory accesses by other agents, such as DMA access by capable disk drive controllers, graphics cards etc.

Exit from an enclave may occur either as a result of an *Enclave EXIT* (`EEXIT`) instruction, or asynchronously, such as as when exceptions or interrupts occur during enclave execution. In the latter case, an *Asynchronous Enclave eXit* (`AEX`) instruction is used to securely save the processor state inside protected enclave memory and scrub the CPU registers to avoid leakage of sensitive data. Finally, the CPU leaves enclave mode. The enclave may subsequently be re-entered with a *Enclave RESUME* (`ERESUME`) instruction, which restores the execution state of the enclave. In both cases, any cached page translations referring to the enclave's protected address region are cleared.

In contrast to conventional split-world-based isolated execution approaches, such as ARM TrustZone [1], the SGX hardware architecture results in much simpler transitions to and from the secure CPU mode. There is no need to manually transfer data back and forth between a secure and normal world, as the isolated enclaves execute within the address space of the host process. In addition there is no need for a separate operating environment to provide further isolation between TAs/enclaves, as in other TEE configurations.

2.3 Standardization

To date there have been many proprietary APIs developed for TEEs. Almost every vendor of TEE technology has supplied their own implementations, with varying levels of complexity and functionality. Many of these solutions are aimed at the same market segments thus impacting the consumers, OEMs and operators in this case, in similar ways. This fragmentation has been partly to blame for the slow uptake in the use of TEE technology, OEMs are finding that they need to provide more engineering effort to support essentially the same functionality on different hardware platforms. With ever increasing demands for more services and growing awareness of end users' need for privacy protection, OS vendors such as Google are mandating that more of the security

critical components of the operating system are protected by hardware security mechanisms.

This fragmentation and the need for OEMs to pass *Compliance Test Suites* (CTS), that are designed to test that a platform is protecting the security critical components as mandated by the OS vendor, have given rise to the need to address this through some form of standardization. There have been a number of efforts to address this by providing a standard operating system to run within the TEE, such as Nvidia's *Trusted Little Kernel* [11] (TLK) and Linaro's *OP-TEE* [8]. Though a common operating system is a good start to help adoption as it removes a lot of engineering effort it does not go far enough to address the needs of application developers on platforms such as Android or iOS. To benefit these application developers there must be a consistent API by which the TEE functionality can be accessed and invoked.

To this end *GlobalPlatform* (GP) [5], a non-profit association focused on promoting the industry wide adoption of security standards, has been formulating and driving standards for TEE APIs. Within the scope of the TEEs that we describe in this paper, i.e. extensible TEEs that allow for the deployment of trusted third-party code, GP has defined two main standards. The TEE Client API, running in the host operating system, and the TEE Core API, running inside the TEE. This provides standardization of both the TAs (so they can be deployed in any GP compliant TEE irrespective of the HW vendor) and the CAs that run in the host operating system. GP's architecture is built around the model defined in Fig. 1 where a CA running in the REE creates one or more sessions to TAs in the TEE. This session establishes an effective Remote Procedure Call (RPC) mechanism through which it is possible for the CA to invoke commands in the TA. Via a well defined API a TA is able to provide its services while also protecting itself from misuse. The first column in Table 1 summarizes the GP Core API internal TA interface.

To date GP is the forerunner in TEE standardization – numerous GP implementations are starting to emerge. Both TLK and OP-TEE provide some form of GP compliance, thought he exact extent of the compliance is unknown as the GP TEE compliance test suite is not readily available.

2.4 Open-TEE

In our previous work [9] we highlight and applaud the efforts that have been undertaken to address the issue of TEE compliance, however, we highlight other issues that hinder the widespread adoption of TEEs. Chief among them is the lack of access to debug-enabled versions of existing TEEs and the difficulty of developing applications for the TEE. This insight led to the development of Open-TEE, a virtual TEE that complies with the GP standard and provides a fast and efficient prototyping platform for TAs / CAs. Fig. 2 shows Open-TEE architecture as a series of processes running on a development machine, thus allowing developers to leverage the tools, e.g. editors and debuggers, that they are familar with in order to complete their projects. The *Manager* process provides the TEE runtime with services usually expected from the TEE OS. Unlike a proper TEE OS, Manager is not self-contained in the sense that it would provide various OS services itself, rather it runs as a regular host process and leverages to a large extent services provided by the host OS. The sole purpose of the *Launcher*
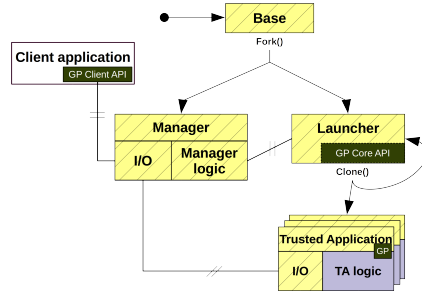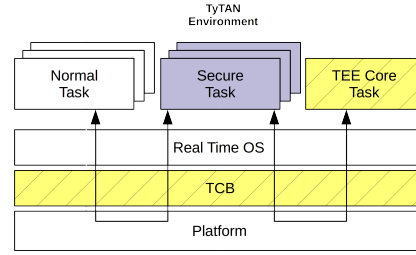
Fig. 2. Open-TEE architecture [9]

Fig. 3. GP architecture on TyTAN

process is to pre-load the shared library implementing the TEE Core API and serve as a base process used to clone the actual TA processes, which are subsequently reparented onto Manager. Each TA process is divided into two separate threads; the *I/O* thread and *TA logic* thread. The I/O thread facilitates communication with Manager, whereas the TA logic thread executes the TA domain logic. We called this work Open-TEE and have made it publically available [4] under the permissive Apache-V2 license.

3 Mapping GlobalPlatform Interfaces to New TEE Architectures

Although not conventional TEE environments with their own OSs, TrustLite and SGX provide the same security assurances i.e. integrity and confidentiality of both code and data. They do no enforce the split world view that has become the de facto standard for extensible TEEs. However, there are still a number of use cases where applying the GP concepts and having access to a GP compliant implementation can be beneficial. First and foremost is the question of portability – GP compliant applications should be readily deployable in any compliant TEE. Second is the notion of a services framework – there are many cases when a developer would like to use existing services provided by a TEE, e.g. a keystore with key management and cryptographic routines, without the need to reimplement these.

3.1 Realizing GP interfaces on TrustLite / TyTAN

In Section 2.1 we noted how the isolation model in TyTAN differs from the typical GP model. Consequently, mapping the GP APIs to platforms such as TyTAN presents some challenges:

Placement of TEE Core Typically, the TEE Core APIs and support for CA-to-TA communication would be provided as services by the TEE OS. As in TrustLite and Ty-TAN the OS itself is excluded from the platform TCB, the Core API implementation needs to be provided in some other manner. An obvious alternative would be to provide the functionality purely in a library linked to each trusted task designated as a GP-style TA. This has a number of drawbacks. First, due the the lack of paging memory management and caching, this leads to suboptimal memory utilization as identical functionality

---
[4] http://open-tee.github.io/

would be replicated for each TA. Second, the implementation of certain APIs, such as trusted storage access would be greatly simplified by platform support for centralized secure storage. Therefore TEE Core API functionality could either be provided directly by the TyTAN TCB, or in a less invasive approach, by a separate secure task, referred to as *TEE Core task*, separately protected by the EA-MPU. The Core API library linked to TAs can then invoke IPC calls to the TEE Core task, which services these appropriately. The overall architecture is shown in the Fig. 3. In this case, the challenge stems from the fact that the strong isolation provided by TrustLite is not based on a completely separate environment, but on per-task isolation. Therefore, there is no separate trusted OS part of the TCB as is the case in established TEEs.

Session access control The TEE Client API is used for communication between normal and secure tasks. The TEE Core API also provides an Internal Client API, which allows one TA to act as a client to another TA. With regards to TyTAN tasks, this is utilized for communication from one secure task to another. One challenge with this mapping is the need to enforce access restrictions on the TEE Core task so that only secure tasks are allowed to invoke Core API functionality. In addition, the TEE Client API provides session-based access to TAs. As noted in Section 2.1, TyTAN does not provide any IPC access control by itself, but leaves this up to individual secure tasks based on the hash digest of the sender as reported by the IPC primitives part of the TyTAN TCB. While it would be possible to implement session, Core API task multiplexing and access control in each secure task individually, providing this functionality as part of a proxy in the TyTAN TCB has the same advantages as the centralized TEE Core placement which is an important consideration in the highly resource constrained environments that TrustLite / TyTAN targets. The challenge here stems from a large disparity in levels of IPC abstraction between TyTAN, and the GP specifications.

Other issues Apart from the Internal Client API, the TEE Core API provides the TAs with internal programming interfaces for trusted storage, cryptographic operations, time API and an artihmetic API intended for as building blocks for developers to implement further asymmetric cryptographic algorithms. For performance reasons, platforms based on TrustLite may require additional cryptographic hardware accelerators to meet operational requirements, hence full coverage of the TEE Core API is not only impractical, but also unnecessary for many intended use cases. Due to its nature, it is likely that instantiations of TrustLite for different use cases will have varying degrees of hardware support for aforementioned features, and it is unlikely that a one-size-fits-all solution would be applicable for devices of this scale.

3.2   Realizing GP with SGX through Open-TEE

Being a set of instructions in the CPU and associated hardware memory protection SGX has the ability to run independently of the OS. This has a number of side effects that may not be immediately obvious. For example, the code in an enclave must be self-contained: it is not possible to make any system calls or invoke services provided by the OS directly from within an enclave. In order for these types of accesses to be possible an enclave must synchronize with the non-enclave part of the application to perform these tasks on its behalf.

Open-TEE was conceived as a GP compliant tool for fast prototyping. However, throughout its design, choices were made that would allow it to work as a fully functional TEE environment when combined with the right hardware security mechanisms, like the SGX enclaves.

GP	SGX
TA_CreateEntryPoint	ECREATE, EINIT, EENTER, EEXIT
TA_DestroyEntryPoint	EENTER, EEXIT
TA_OpenSessionEntryPoint	EENTER, AEX, ERESUME, EEXIT
TA_CloseSessionEntry	EENTER, AEX, ERESUME, EEXIT
TA_InvokeCommandEntryPoint	EENTER, AEX, ERESUME, EEXIT

Table 1. Mapping GP entry points to SGX instructions

If we take a common usage scenario of a CA wishing to interact with a TA that performs some operation involving access to secure storage, we can see how Open-TEE's architecture (Fig 2) can be mapped to SGX [6]. The choice to split the TA process into two distinct parts, handling I/O and logic functionality facilitates this mapping. The I/O thread is responsible for all the communications back and forth to Manager and the CA and is also responsible for all system interactions that may be required, this is Open-TEE specific. The TA logic thread is then solely responsible for providing the GP core API compliant functionality, this is the thread in which the developers TA code will be executed.

The CA initiates an open session call towards the TA. Manager notices that the TA is not running so a request is made to Launcher to create a TA process. At this point the TA is a standard process and loads in accordance with the OS requirements. Once the TA is created we must initialize the TA logic code which implements the GP standard [5] by invoking `TA_CreateEntryPoint`. As all of the GP functionality is implemented within the enclave it is at this point that the enclave must be created. The TA hands over the enclave code along with any additional configuration data to the enclave creation service running in the supervisor mode. Once the enclave has been created and initialized control is handed back to the TA application in order that it might enter the enclave to perform the required tasks e.g. finalizing the GP initialization and establishing the session to the CA.

Now that the session is established between the CA and the TA, the CA can invoke TA commands using `TEEC_InvokeCommand`. Imagine that the CA wishes to provision some data to the secure storage. It invokes the corresponding TA command, when the TA logic thread wants to access the secure storage this requires a read or write operation to the storage media, which is a system call and is thus prohibited by SGX. In this case the Open-TEE implementation of the GP secure storage API initiates an interrupt towards the I/O thread causing an *Asynchronous Enclave eXit* event which stores the enclave state and control is passed back to the I/O thread. The I/O thread then invokes Manager's secure storage API to read/write encrypted data from/to persistent storage. When this action has been completed the TA can resume execution in the TA logic thread. When the invoked task is complete the enclave can be exited thereby returning

control to the I/O thread which can then respond to the CA on the status of the invoked command. Other system services needed in the TEE core API such time functionality can be handled in a similar fashion. Table 1 summarizes the mapping of GP entry points to SGX instructions.

4   Conclusion

The GP interfaces were designed to support split-world-based TEEs. We have shown that, even though beneficial for easing the adoption of new, emerging TEE environments, mapping existing GP interfaces to them has a number of challenges. In ongoing work we are implementing the GlobalPlatform TEE Client API, and a subset of the TEE Core APIs on TyTAN. We are also exploring practical subsets of the TEE Core API applicable to different use cases. In future work we plan to investigate possible extensions to the TEE Core APIs relevant for small scale devices. We also plan to continue our efforts in realizing Open-TEE on top of SGX.

References

1. ARM Security Technology - Building a Secure System using TrustZone Technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf` (2009)
2. Brasser, F., Koeberl, P., Mahjoub, B.E., Sadeghi, A.R., Wachsmann, C.: TyTAN: Tiny trust anchor for tiny devices. In: 52nd Design Automation Conference (DAC) 2015 (Jun 2015)
3. Ekberg, J., Kostiainen, K., Asokan, N.: The untapped potential of trusted execution environments on mobile devices. IEEE Security & Privacy 12(4), 29–37 (2014), `http://dx.doi.org/10.1109/MSP.2014.38`
4. Eldefrawy, K., Tsudik, G., Francillon, A., Perito, D.: SMART: secure and minimal architecture for (establishing dynamic) root of trust. In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012. The Internet Society (2012), `http://www.internetsociety.org/smart-secure-and-minimal-architecture-establishing-dynamic-root-trust`
5. GlobalPlatform: Device specifications for trusted execution environment. `http://www.globalplatform.org/specificationsdevice.asp`
6. Intel: Software Guard Extensions Programming Reference (2013), `https://software.intel.com/sites/default/files/329298-001.pdf`
7. Koeberl, P., Schulz, S., Sadeghi, A.R., Varadharajan, V.: TrustLite: A security architecture for tiny embedded devices. In: Proceedings of the Ninth European Conference on Computer Systems. pp. 10:1–10:14. EuroSys '14, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2592798.2592824`
8. Linaro: OP-TEE. `https://wiki.linaro.org/WorkingGroups/Security/OP-TEE`
9. McGillion, B., Dettenborn, T., Nyman, T., Asokan, N.: Open-TEE – an open virtual trusted execution environment. Tech. rep., Aalto University (2015), `http://open-tee.github.io/publications/`

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

11. NVIDIA: Trusted little kernel (tlk). `http://nv-tegra.nvidia.com/gitweb/?p=3rdparty/ote_partner/tlk.git;a=summary`