Open-TEE - An Open Virtual Trusted Execution Environment

Brian McGillion*, Tanel Dettenborn†, Thomas Nyman‡
Intel Collaborative Research Institute for Secure Computing (ICRI-SC) at Aalto University, Finland.

N. Asokan§
Aalto University and University of Helsinki, Finland.

*brian.mcgillion@intel.com, †tanel.dettenborn@intel.com, ‡thomas.nyman@aalto.fi, §asokan@acm.org.

***Abstract*—Hardware-based Trusted Execution Environments (TEEs) are widely deployed in mobile devices. Yet their use has been limited primarily to applications developed by the device vendors. Recent standardization of TEE interfaces by GlobalPlatform (GP) promises to partially address this problem by enabling GP-compliant trusted applications to run on TEEs from different vendors. Nevertheless ordinary developers wishing to develop trusted applications face significant challenges. Access to hardware TEE interfaces are difficult to obtain without support from vendors. Tools and software needed to develop and debug trusted applications may be expensive or non-existent.**

In this paper, we describe Open-TEE, a virtual, hardware-independent TEE implemented in software. Open-TEE conforms to GP specifications. It allows developers to develop and debug trusted applications with the same tools they use for developing software in general. Once a trusted application is fully debugged, it can be compiled for any actual hardware TEE. Through performance measurements and a user study we demonstrate that Open-TEE is efficient and easy to use. We have made Open-TEE freely available as open source[1].

I. Introduction

Personal computing devices such as smartphones, tablets and laptops have become pervasive. They are used to store sensitive data and access critical services across a wide range of domains, such as banking, health care and safety, where privacy and security are paramount. On the other hand, traditional operating systems and the services that they provide are becoming so large and complex that the task of securing them is increasingly harder. Hardware-based trusted execution environments (TEEs) were developed to address this gap. A TEE on a device is isolated from its main operating environment by using hardware security features. It offers a smaller operating environment that provides just enough functionality so that sensitive data and operations can be offloaded to it.

Hardware-based TEEs have been widely deployed in mobile devices for over a decade [1]. TI M-Shield [2] and ARM TrustZone [3], [4] are early examples, followed by newer architectures like the Intel SEP security co-processor [5] and Apple's "Secure Enclave" co-processor [6]. Business requirements such as the need to enforce digital rights management and subsidy locks, as well as regulatory requirements like cloning- and theft protection have been the driving forces behind such large scale deployment [1]. Such requirements continue to appear: e.g., fingerprint scanners with hardware protection, hardware-backed keystores, and the recent "kill switch" [7] bill in California which mandates that a mobile device must be capable of being rendered inoperable if it is stolen.

Although the early hardware security modules (HSMs) like the IBM cryptocards[2] were programmable [8], the vast majority of HSMs used with personal computers and servers today are typically application-specific modules or fixed function co-processors like the Trusted Platform Modules (TPMs) [9]. In contrast, TEEs in mobile devices are programmable. However, despite widespread deployment of hardware-based TEEs in mobile devices, application developers have lacked the interfaces to use TEE functionality to protect their applications and services. Nor have they been researched extensively in the academic community. Recent efforts by GlobalPlatform [10] to specify standard interfaces for TEE functionality in mobile devices [11] will partially address this problem. However, there are a number of factors that stand in the way of widespread use of hardware-based TEEs in application development and research. Chief among them is the difficulty of developing applications for TEEs. Software development kits for TEE application development are often proprietary or expensive. Debugging low-level TEE applications either requires expensive hardware debugging tools, or leaves the developer with only primitive debugging techniques like "print tracing" (e.g., using printf statements in C to keep track of how values of variables change during program execution).

In this paper, we argue that a virtual standards-compliant TEE implemented entirely in software will allow developers to build TEE applications using tools and development environments that they are *already* familiar with. It will also allow applications to be tested and refined even when developers do not have access to devices where hardware TEE functionality has been made accessible to them. Such a facility will greatly ease TEE application development and can trigger new ways of using TEEs. We make the following contributions:

- We design and implement such a **virtual TEE called Open-TEE** which conforms to GlobalPlatform Specifications. We identify requirements that would make

[1]https://github.com/Open-TEE/project
[1]Name anonymized for submission

[2]http://www.ibm.com/security/cryptocards/

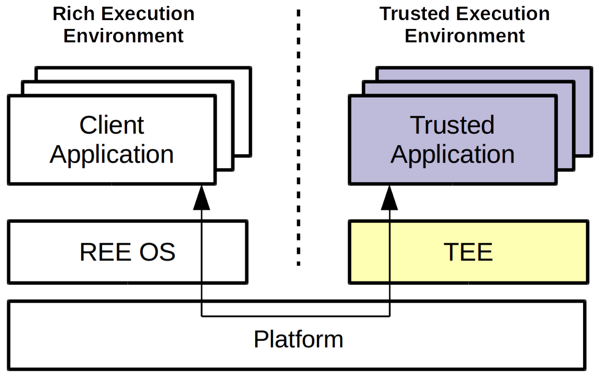

Fig. 1: A TEE in a Computing Device

Open-TEE acceptable to developers and make specific design choices informed by these requirements (Section III). Open-TEE is publicly available on GitHub.[3]

- We show that Open-TEE is **efficient, hardware-independent** and allows a developer to carry out much of the development life cycle of standard-compliant TEE applications using popular application development environments they already use. We demonstrate that Open-TEE **significantly improves the ease-of-use** of TEE application development by conducting a small-scale, yet rigorous, user study with experienced professional TEE developers (Section IV).

Given the demonstrable usability benefits, we recommend that organizations who develop applications for TEEs should consider incorporating Open-TEE into their development process. We also hope that this paper will enable more researchers to discover the power of TEEs and use Open-TEE to develop and experiment with new TEE applications.

II. Background

A. Structure of a TEE

A TEE is a secure, integrity-protected processing environment, consisting of processing, memory and storage capabilities. Figure 1 shows how a device can be visualized as a series of distinct environments with their own set of features and services. We adapt the terminology used by GlobalPlatform [12] to describe the concepts illustrated in Figure 1:

Rich Execution Environment (REE): The word "rich" here refers to an operating environment that is feature rich such as one would expect from a modern platforms such as Android, iOS, Windows, Linux or OS X.

Trusted Execution Environment (TEE): The TEE is a combination of features, both software and hardware, that isolate the execution of tasks from the REE. These environments have a limited set of features and services as they are intended to only address the security critical subset of an application's functionality such as offloading some cryptographic operations or key management.

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

Trusted Application (TA): An application encapsulating the security-critical functionality to be run within the TEE. This may be a service style application that provides a general feature, such as a generic cryptographic keystore, or it could be designed to offload a very specific part of an application that is running in the REE, such as a portion of the client state machine in a security protocol like TLS.

Client Application (CA): CAs are ordinary applications (e.g., browser or e-mail client) running in the REE. CAs are responsible for providing the majority of an application's functionality but can invoke TAs to offload sensitive operations.

As an example, consider a common use case for TEEs: the offloading of Digital Rights Management (DRM) protected content. The CA would be responsible for the majority of the tasks associated with viewing the content i.e. opening the media file, providing a region in the display into which it can be rendered (the window) and providing a mechanism to start, stop, rewind the media. A TA would be used to decrypt the protected media stream and make the decrypted content available directly to the graphics hardware that is responsible for rendering and displaying the stream.

B. TEE architectural options

A TEE can be realized in different ways, but the overall concept stays the same. Figure 2 shows a number of ways in which these TEEs can be realized:

Co-Processor: A separate core, generally with its own peripherals, is used to offload the security critical tasks from the main operating environment. The benefits of such a configuration are that the operation can generally be completely isolated and it can run simultaneously with the main core. The drawback is that there is an overhead associated with transferring the data to and from the core. Also, the co-processor is generally less powerful than the main core. The co-processor design can be further separated into two alternatives:

External Security co-processor: is a discrete hardware module outside the physical chip (commonly referred to as "System on Chip" or SoC) containing the main core, and is thus completely isolated from it, not sharing any resources with it.

Embedded Security co-processor: is embedded into the main SoC and thus has the capability to share some of the resources of the main system. It is still isolated from the main processor.

Processor Secure Environment: Many popular mobile TEE architectures follow a configuration where a single core supports multiple virtual cores that are mutually exclusive of one another i.e. when one is running the other is suspended. Generally there is some form of trigger to allow the core to switch from one state to the other. This configuration is sometimes referred to as the "processor secure environment" [13].

ARM TrustZone is an example of this configuration. In TrustZone, the processor core can be in one of two "worlds": a "secure world" (for the TEE) and a "normal world" (for the REE). A special instruction called Secure Monitor Call (SMC) can be executed to trigger the processor running in normal world to enter "monitor mode" that marshals the transition to secure world [3]. The advantage of this configuration is that there is no need to offload the data to and from the secure

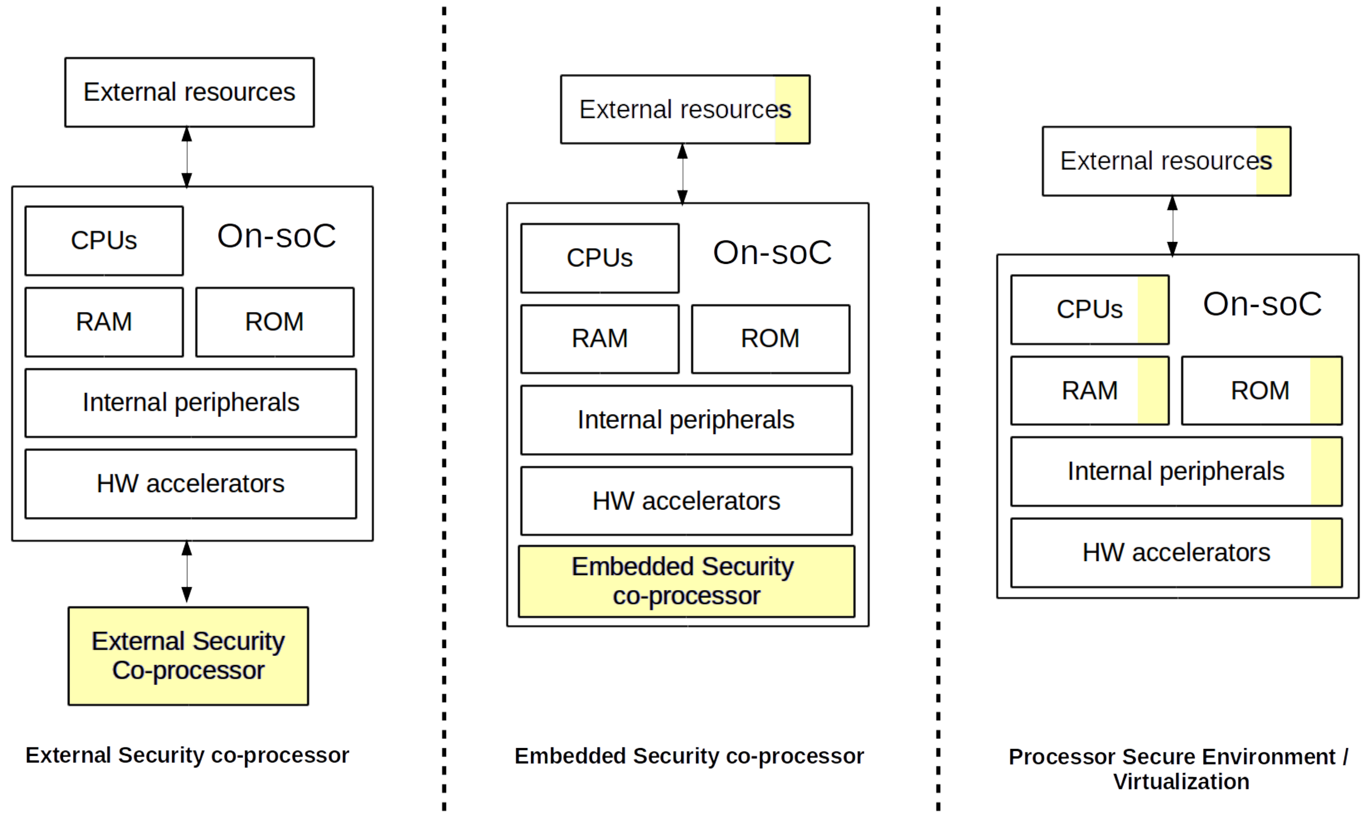

Fig. 2: Three potential architectural options for realizing a TEE architecture (adapted from [12])

world. However, there is a cost associated with having to store and restore the device state on entry and exit from a given mode. On single core devices there is also an added security benefit from having only one world running at a given time in that it ensures that the normal world OS cannot interfere with the secure world directly or indirectly (e.g., software side-channel attacks). However, this also has the disadvantage that when one world is active the other world must be completely halted, thus complicating interrupt handling.

Intel Software Guard Extensions (SGX) [14], [15] is another example of such a variant, the core does not perform a full transition to and from a secure world. Instead parts of a standard application, both code and data, are protected by mechanisms in the core. Parts of the application, called an Enclave, are encrypted by a key that is only accessible to the CPU. When an “enter enclave (EENTER)” [16] instruction is received the code and data are decrypted and operated upon in the core. They never leave the CPU package unencrypted, thus protecting them against external access. The benefits are that there is no need to transfer data back and forth between cores or to setup complicated transitions to and from a secure world, and there is no additional need for a separate operating environment as is required in other styles of TEE configuration.

Virtualization based on hardware features such as AMD-V and Intel VT-(x,d), have existed for many years and are used extensively to provide separation of resources between different operating environments especially in high density server configurations. They rely on processor support to allow virtualization of instructions and access to resources e.g. through the use of an IOMMU[4], access to and from peripheral devices can be restricted. Though in and of themselves they are not designed solely to provide a TEE, there is recent research [17], [18] to see how these can be used as an alternative to dedicated hardware based TEEs. When deployed as TEE environments they generally rely on a Virtual Machine Manager (VMM) to provide the marshaling of access to the resources. This poses a security concern as the VMM is considered to be part of the Trusted Computing Base (TCB) of such a system, thus increasing the attack surface.

C. Standardizing TEE Functionality

The landscape for TEEs has been very diverse, with a variety of different architectural options from multiple manufacturers. Even platforms using the same type of TEE are often not inter-operable. For example, an application written for one TrustZone-based platform will generally not run on a different TrustZone-based platform. They may be using a different TEE OSs or different REE OS drivers. On the other hand, developers and others who are higher up in the software ecosystem are less concerned with intricacies of low-level software or hardware but more concerned with their ability to use the capabilities of TEEs easily and across different platforms. This calls for standardization.

One initiative in TEE standardization has been undertaken by GlobalPlatform (GP) [10], which “is a cross industry, non-profit association which identifies, develops and publishes

[4]Input/Output Memory Management Unit

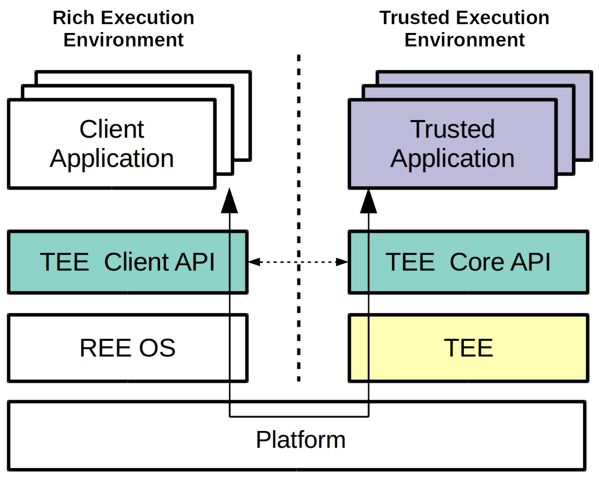

Fig. 3: The main APIs specified by GlobalPlatform [12]

specifications that promote the secure and inter-operable deployment and management of multiple applications on secure chip technology" [19]. GP offers specifications in three areas: smartcards, back-end support systems and devices. In this paper, we are concerned with specifications from the device working group related to the APIs for TAs.

Figure 3 shows the primary interfaces standardized by GP. The Core API provides an extensive set of features such as a crypto API and secure storage that can be used to implement a TA, for example a DRM decoder. The Client API is a very generic and thin layer consisting of a small number of functions and definitions that allow the transfer of data back and forth from the REE to a TA. A CA, for example a DRM player, will implement all complex but non-critical functionality by itself, but use the Client API to invoke the corresponding TA, such as the DRM decoder. Between the "TEE Client API" running on the REE and "TEE core API" running on the TEE we have an effective Remote Procedure Call (RPC) mechanism where a process running in the REE can invoke tasks in the TEE.

These standardization efforts in GlobalPlatform could resolve the issue of inter-operable TEEs. However, they do not remove the obstacle in gaining access to the requisite hardware nor does simplify the task of developing and testing TAs.

III. Open-TEE

In order to pave the way for the widespread use of TEE functionality by developers and researchers we propose an architecture and a software development kit (SDK) that implements this as a framework atop a set of tools that are familiar to the developer, thus removing the need for specialized hardware and the overheads that it incurs.

A. Motivation

To explain our motivations, we now revisit the difficulties in developing TEE applications that we alluded to in Section I:

a) Enable developer access to TEE functionality: For a variety of reasons, access to TEEs is generally restricted to developers working for chip manufacturers and to the original equipment manufacturers (OEMs) that make devices based on these chips. Usually, the technology is proprietary and easily deployable SDKs are not available. Furthermore, TEEs may not have a security architecture within them to safely allow complete outsider developers access to TEEs. However, there have been attempts to address this problem [20].

b) Provide a fast and efficient prototyping environment: The most common methods of debugging TAs are to either use expensive JTAG[5]debugging or resort to primitive "print tracing" by inserting diagnostic output in the source code. The former generally allows for detailed instruction level debugging. However, the costs associated with these debuggers can be prohibitively expensive, and the setup complex. Print tracing as a debugging technique is cumbersome and clutters up the source code even to locate the source of a problem. Another concern encountered by TEE developers is that if a TA running on actual device hardware crashes, a hard reset of the device maybe required to recover, thereby significantly increasing the time and effort of debugging.

c) Promote research into TEE Services: Ways to isolate TEEs from REEs are reasonably well understood as we saw in Section II. What is less well understood are the types of services that could benefit from using TEEs. As the app store model has proven, given an opportunity, the developer community at large is capable of pushing the boundaries and exploring new and novel ways to use technology. Making it possible for researchers to easily develop TAs could trigger the development of novel and innovative applications.

d) Promote community involvement: The pre-requisite for involving the developer community and researchers at large is to allow them access to a freely and easily available development environment, SDK and a platform with which to experiment. The financial and technical aspects of making hardware TEEs available for development on a large scale motivates the need for a software framework for TA development which is not bound to any particular hardware or vendor.

B. Requirements

Motivated by the above discussion, our aim is to develop an SDK and framework that allows for the development and testing of standard-compliant TEE applications. The framework should allow development of GP-compliant CA and TA functionality without having to rely on any particular hardware support. We intend this to be a fast prototyping and development environment that also provides a platform from which to conduct further research into TEE functionality. Our fundamental design principle is that it should require as little configuration and maintenance as possible, allowing the developer to focus on the task at hand.

We identify the following criteria by which we can measure our TEE framework's usefulness and hence its potential success in addressing the issues that motivated it:

Compliance: Having chosen GP as the standard, our framework should comply with its main interfaces, the Client and Core APIs.

Hardware-independence: As a software based solution our framework should not be dependent on a particular TEE

[5]Joint Test Action Group standard addresses debugging of integrated circuits

hardware environment. Furthermore, it should also not be dependent on any particular hardware for the development system itself.

Reasonable Performance: To be readily deployed, our framework must not suffer from code bloat that adds to the on-disk footprint nor to the memory consumption required to run the environment. In addition the start-up and restart times of the environment, especially that of the CAs and TAs should not be excessive.

Ease-of-use: The solution should be easily deployed and configured. It should use tools that are widely available making it more attractive (e.g., there should be no need for extra package/tool configuration on the development system).

We now describe our design and implementation of such a software framework which we call Open-TEE.

C. Architecture

We begin with an overview of the structure of the Open-TEE environment. Figure 4 identifies the main components and their relationships. The color code used in Figure 4 is the same as that used for Figure 3 to make the correspondence between the Open-TEE implementation architecture and the GP conceptual architecture is clear. We describe in each component in detail below.

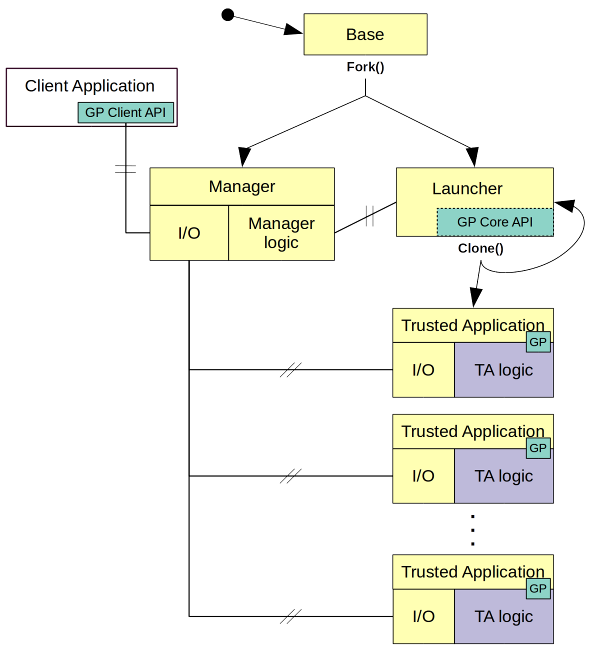

Fig. 4: Open-TEE architecture

a) Base: Open-TEE is designed to function as a daemon process in user space. It starts executing Base, a process that encapsulates the TEE functionality as a whole. Base is responsible for loading the configuration and preparing the common parts of the system. Once initialized the Base will fork to create two independent but related processes. One process becomes Manager and the other, Launcher which serves as a prototype for TAs.

b) Manager: Manager can be visualized as Open-TEE's "operating system." Its main responsibilities are: managing connections between applications, monitoring TA state, providing secure storage for a TA and controlling shared memory regions for the connected applications. Centralizing this functionality into a control process can also be seen as a wrapper abstracting the running environment (e.g. GNU/Linux) and reconciling it with the requirements imposed by the GP TEE standards. GP requirements and the host environment's functionality are not always aligned. For example, GP requirements stipulate that if a TA/CA process crashes unexpectedly, all shared resources of the connected processes must be released. To achieve this functionality in a typical running environment, additional steps are needed beyond just terminating the process. For example all shared memory must be unregistered – this needs to be a distinct action from normal process termination.

c) Launcher: The sole purpose of Launcher is to create new TA processes efficiently. When it is first created, Launcher will load a shared library implementing the GP Core API and will wait for further commands from Manager. Manager will signal Launcher when there is a need to launch a new TA (for example, when there is a request from a CA). Upon receiving the signal, Launcher will clone itself. The clone will then load the shared library corresponding to the requested TA. The design of Launcher follows the "zygote" design pattern (such as that used in Android [21]) of pre-loading common components. This is intended to improve the perceived performance of starting a new TA in Open-TEE: because shared libraries and configurations common to all TAs are pre-loaded into Launcher, the time required to start and configure the new process is minimal. A newly created TA process is then re-parented onto Manager so that it is possible for it to control the TA (so that, for example, it can enforce the type of GP requirements discussed in the paragraph above).

d) Trusted Application Processes: The architecture of the TA processes is inspired by the multi-process architecture utilized in the Chromium Project [22]. Each process has been divided into two threads[6]. The first handles Inter-Process Communication (IPC) and the second is the working thread, referred to respectively as the IO and TA Logic threads. This architectural model enables the process to be interrupted without halting it, as occurs when changing status flags and adding new tasks to the task queue. Additional benefits of this model are that it allows greater separation and abstraction of the TA functionality from the Open-TEE framework.

e) GP TEE APIs: The GP TEE Client API and GP TEE Core API are implemented as shared libraries in order to reduce code and memory consumption.

f) IPC: Open-TEE implements a communication protocol on top of Unix domain sockets and inter-process signals as the means to both control the system and transfer the messages between the CA and TA.

D. Implementation and Tooling

a) Utilizing existing functionality: To meet the hardware-independence requirement, we do not emulate specific TEE hardware with software based emulators, such as

[6]The architecture of Manager follows the same division

QEMU [23]. Instead we rely on existing technologies and the services offered by the mainstream OS of Open-TEE's running environment rather than developing a new TEE OS to deploy the GP APIs in. In addition we reuse software from existing open source projects, such as the OpenSSL crypto library and the GNU tool suite, thereby reducing the amount of time required to develop and test the Open-TEE framework.

This also contributes towards meeting the ease of use requirement in that developers can easily set up Open-TEE and start developing TAs using a set of familiar tools, editors, IDEs, compilers and debuggers. For example, a developer utilizing Open-TEE can connect to a TA process with a cheap reliable software debugger such as GDB [24] for detailed debugging tasks like stepping through the code, inspecting variables and registers etc.

b) Development process: The intended user base for Open-TEE consists of seasoned developers. To ensure viability in such a demanding user base, we adopted a rigorous development process for Open-TEE so that the end result will be perceived as robust and usable. Open-TEE is developed as an open source project and as such there are a number of powerful tools that are freely available for this type of project. As previously mentioned GitHub is used for hosting the code. GerritHub[7] is used for performing peer-review of all code before it is submitted to the code base. In addition to the manual review process we leverage the power of Coverity[8] to perform in depth static analysis scans. This enforces secure coding practices and helps to find potential functional bugs that may have been missed during the manual code review. In addition, we have deployed a continuous integration (CI) server running Jenkins[9], which we have connected to GerritHub. Its main task is to perform a number of "smoke tests"[10] on the new patches. These tests ensure that the patches conform to the coding guidelines, build successfully and that the basic system is usable after the patches are applied.

c) Open-TEE in Use: Being designed as an open source framework upon which to build and test features that will utilize a TEE, Open-TEE has been implemented to be as inconspicuous as possible. The complexity of the system is hidden from the users of Open-TEE. They are presented with an SDK that exposes the Client and Core APIs without being required to have a deep understanding of how the overall framework works, thereby allowing them to focus on the development of their own TAs. However, Open-TEE is already being extended by the community. The ongoing implementation of the GP Trusted UI specification is an example.

IV. Evaluation

We now return to the requirements from Section III and evaluate how well Open-TEE meets those requirements.

A. Compliance

Every effort has been made to comply with the GP standard. Whenever this has not been feasible, due to time constraints or in the interest of providing a platform upon which to build, the deviation has been documented and a debug message is logged to inform the user of the non-compliance. The Client API is fully implemented. The Core API implementation has 100% function coverage, however, the algorithm coverage is currently 80% due to the use of existing libraries that do not support the remaining algorithms. A compliance test suite is commercially available from GP. There is no information about how well existing TEE hardware conform to the GP specifications. Based on our experience in implementing version 1.0.26 of the GP Core API, we provided detailed feedback, including on errors and ambiguities in the specification, to GlobalPlatform in response to their solicitation of public comments. Several items in our feedback have been addressed in the released version 1.1.

B. Hardware-independence

By following the GP standard and not emulating any specific TEE hardware, Open-TEE is independent of TEE hardware. TAs developed with Open-TEE can be compiled to any target TEE hardware architecture. The implication of this is seen in the performance measurements of the TAs and CAs that are being developed. Open-TEE can provide coverage reports to help highlight hot-spots in the code, generate call graphs etc. However, as the actual TEE is potentially running a different environment than that offered by Open-TEE– possibly utilizing hardware based cryptographic accelerators, potentially having a different CPU, with different clock speed and throughput characteristics – it will result in different timing characteristics. In this sense, as with all virtual environments, Open-TEE can not fully replace the actual hardware environment for the final stages of the development cycle. However, with judicious use of coverage reports and other generic analysis techniques, one should be fairly confident that the hardware-independent parts of the code have been optimally implemented.

Open-TEE has been deployed and used on various development environments ranging from servers to desktops and laptops. We have tested Open-TEE on both ARM and x86 architectures. Open-TEE requires Linux but has been run successfully on virtual machines hosted on other OSs.

C. Footprints and Performance

To evaluate our performance we deployed Open-TEE on a desktop machine (Intel i7-2600 CPU with 8GB RAM) running 64-bit Ubuntu 14.04. All performance tests were run 40 times while the machine was under normal load e.g. having editors and browsers open.

1) Disk and Memory consumption: Open-TEE is written in ANSI C with a total of 12423 lines[11] spread over 78 source and header files. Table I shows the total size of the framework and highlights two libraries from the framework that are of most interest to developers, being "libInternalApi.so" against which the TAs are linked and "libtee.so" against which CAs are linked. As is standard on operating systems that support shared libraries the "Text" section, containing the programs code, can be shared among the different processes that link against it. The "Data" and "BSS" respectively refer to the initialized and uninitialized data parts of the library that can be shared in a

[7]http://gerrithub.io/

[8]https://scan.coverity.com/projects/3441

[9]http://jenkins-ci.org/

[10]A suite of tests intended to ensure that the basic functionality of a system are intact.

[11]gathered using sloccount: http://www.dwheeler.com/sloccount/

	Text	Data	BSS	overall
libInternalApi.so	117448	2248	160	119856
libtee.so	18617	880	152	19649
Total Framework	224948	7760	1664	234372

TABLE I: Binary sizes (bytes)

	RSS	Shared	Private	PSS
Manager	1024	764	260	305
Launcher	1624	1232	392	558
Manager	1112	832	280	316
Launcher	1648	1548	100	397
Test TA1[12]	1072	932	140	308
Manager	1116	832	284	319
Launcher	1648	1548	100	337
Test TA1	1072	944	128	245
Test TA2[13]	1236	1068	168	299

TABLE II: Memory usage (KB)

Copy-On-Write (COW) basis. As the table highlights the vast majority of the libraries' size can be shared, thus reducing the required footprint.

Table II shows the memory consumption of the running process under three different scenarios. The first shows the memory consumption of Manager and Launcher immediately after they have been launched, i.e. before any TAs have been launched. The next section shows how the memory consumption increases when one TA is launched while the last sections shows the situation when two TAs are running simultaneously.

RSS (Resident Set Size) shows how much memory has been allocated for a process, this includes all memory that a process shares with other processes. As such it is very naive measurement of a processes memory impact.

Shared is the memory that a process shares with other processes, i.e. through the use of shared libraries.

Private is the memory that is private to a process and will be returned to the system when the process terminates, however, Copy-On-Write semantics after a process fork may complicate this calculation. The Private pages may actually be shared until one or the other of the processes tries to write to the page, at which time it will be given its own copy of the Private page.

PSS (Proportional Set Size) is a realistic indicator of the actual memory footprint of a process. It is calculated as the sum of the Private memory and the (Shared memory / number of processes sharing it) e.g. if there is 100KB of Private memory and 1000KB of memory shared with 10 processes, there is an impact of 200KB[14] on the systems memory. Taking the example of Launcher between runs 2 and 3 it can be seen that while the RSS, Shared and Private memory usage stays constant the PSS decreases as more pages are shared with the new TA.

Overall, we can conclude that (a) the memory footprint of Open-TEE is low and (b) the extensive use of shared libraries implies that the marginal memory cost of launching a new TA is small, as shown by the PSS figures.

[12]ta_conn_test_app

[13]example_digest_ta

[14]100KB + (1000KB / 10) = 200KB

	Time
Build	147 ms $\pm$ 10.95
Execute	430.5 μs $\pm$ 32.6

TABLE III: Average build and execute times of a TA, including standard deviations

2) Build and Run performance: One of the driving requirements of Open-TEE is the need to have short build and deploy cycles to help reduce the overall development effort. Table III highlights that Open-TEE does not pose a significant overhead to the developer, taking an average of just 147 ms to perform an incremental build of a TA. The time required for an incremental build was comparable to that of a clean build, falling within the standard deviation of the former, this can be attributed to the source code being confined to a single C file. Comparative results are not available for deployed hardware-based development environments. However, considering that a full reset of the target device and the subsequent boot of its OS may be required before the CA can be launched, Open-TEE's performance is likely to be perceived as being superior.

D. Ease of use

Determining whether Open-TEE eases the burden of TA development is particularly challenging because, until now, TA development has been limited to a very small set of developers. Fortunately, we were able to recruit several experienced TA developers from multiple organizations to participate in a user study. Our user study was conducted as follows.

Participants: Fourteen people participated in the study. All had prior software experience (between 3 and 33 years, $M = 13$, $SD = 8.2$). Eleven had prior experience developing/debugging TAs (between $\frac{1}{2}$ and 15 years, $M = 5.1$, $SD = 4.2$).

Materials: The standard System Usability Scale (SUS) [25], [26] questionnaire was used to elicit the participants' estimates of the ease of use in developing TAs. We used a pre-study and a post-study questionnaire. In addition to demographic information, the pre-study questionnaire included free-form questions about the current software development environment (if any) they use for TA development. It also contained a SUS questionnaire which the participants were asked to complete with their current TA development environment in mind. This was completed only by those participants who had prior TA development experience.

The material for the user study[15]task was a sample CA/TA pair, provided as part of the Open-TEE source tree. A software flaw had been introduced to the TA, which, when executed, would result in a segmentation fault and subsequent premature termination of the TA. The CA was free of error and was only used to interact with the TA running in Open-TEE.

The post-study questionnaire consisted of a SUS form which the participants were asked to complete with Open-TEE in mind. The questionnaire also had open ended questions about specific difficulties they face in TA development.

Procedure: The user study was conducted in three steps. In the first step, participants were first asked to complete the pre-

[15] The user study materials can be found at http://open-tee.github.io/userstudy/

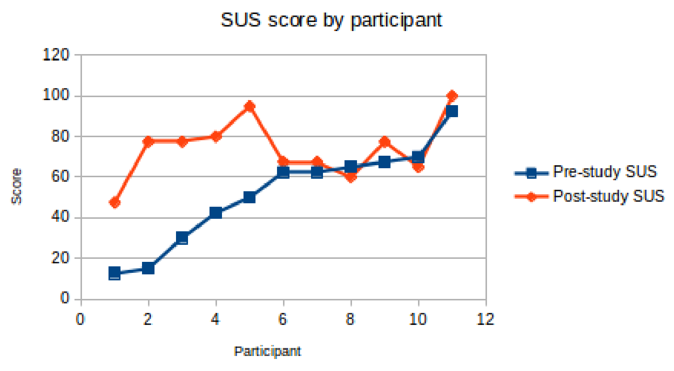

Fig. 5: Pre- and post-study SUS score (for participants with prior TA development experience)

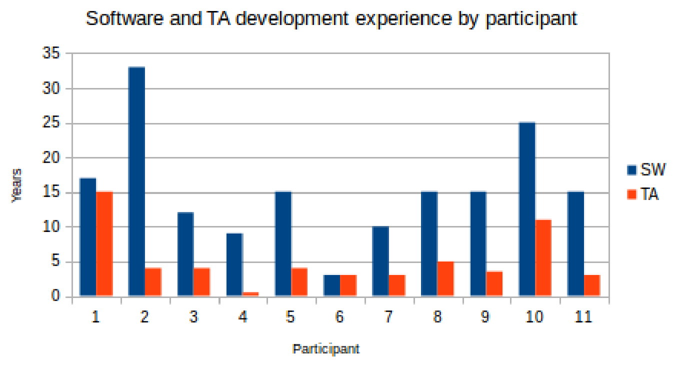

Fig. 6: General software and TA development experience by participant (for participants with prior TA development experience)

study questionnaire. After this they were pointed to a web page containing brief instructions on how to install and use Open-TEE. In the second step, once the participants completed the tutorial they were told about the flawed TA. They were tasked to identify the reason for the TA malfunction using Open-TEE and correct the flaw in the TA. Finally, in the third step, after the participants had completed the debugging exercise, they were asked to complete the post-study questionnaire.

Results: The mean, standard deviation and median of the SUS scores for all participants, including those without prior TA development experience are shown in Table IV. With both sets of participants, the post-study questionnaire yields a mean score above 68, which is considered the threshold value for an above average SUS score, indicating an acceptable level of usability in Open-TEE.

Figure 5 shows the scores reported both before and after the use of Open-TEE by participants with prior TA development experience. Nine out of the eleven participants (82%) rated Open-TEE higher than the development environment they are

	Mean	Std.dev.	Median
Pre-study SUS	51.82	24.70	62.50
Post-study SUS	74.09	15.01	77.50
Post-study SUS (all participants)	69.92	18.09	68.75

TABLE IV: Mean, standard deviation and median for the pre- and post-study SUS scores

using currently. This suggests that the perceived usability of Open-TEE is higher than that of the current tools used by the experienced TA developers. In five cases (46 %), the difference in SUS scores was 35 or more. In the remaining six cases, the difference in SUS scores was 10 or less. A Wilcoxon signed-rank test showed that the difference in SUS scores is statistically significant ($z = -2.50$, $p < .05$, $r = -0.53$).

The difference in SUS scores divides the participants into two distinct groups. The five participants for whom the difference was 35 or more had SUS scores below 60 in their pre-study questionnaire. The remaining six for whom the difference was 10 or less had pre-study SUS scores over 60. A natural question is whether we can discern any other difference between the two groups that might explain the difference in SUS scores. One possible explanation was that experienced software developers were comfortable with their current tools and hence did not perceive Open-TEE as being easier to use. If this explanation is correct then one can hypothesize that developers with many years of general software or TA development experience will rate their current development tools higher than their counterparts with fewer years of experience would. However, a Spearman's rho correlation test indicated no significant correlation between the years of general software development experience and the SUS score in the pre-study questionnaire ($r_s = -.042$, $p > .05$), nor between the years of TA development experience, and the SUS score in the pre-study questionnaire ($r_s = -.204$, $p > .05$). Figure 6 shows the software development experience (both general and TA) reported by each participant whose SUS scores are shown in Figure 5.

A majority of the experienced TA developers (7 out of 11, 64 %), reported using hardware tools for debugging TAs under development. Four (36 %) used Lauterbach[16] hardware assisted debug tools. Three (27 %) used other development boards such as Arndale[17], Fido[18] or DS-5[19] or actual mobile devices. Participant responses highlighted different types of difficulties in debugging TAs using only hardware:

- workflow slowdown due to the need to (cross) compile, load and execute TAs on separate hardware (*"slow execution (flash, download, reboot, run)"*, *"debugging TA is slow, you need to cross compile and push binary into target hardware"*),
- problems due to the hardware itself being under development and hence exhibiting flaws, (*"TEE itself might not work without problems, because some change have been made"*), and
- inconvenience caused by the restricted access to prototype hardware *"Main difficulty is that you need development hardware, which is problematic when working outside the office."*).

Six participants (55 %) reported that their current development environment does not support interactive debugging. But even the rest, who used tools like Lauterbach tracing, reported that they found it easier to resort to print tracing, whenever they needed to examine values of TA variables.

[16]http://www.lauterbach.com/
[17]www.arndaleboard.org/
[18]http://www.liewenthal.ee/projects/fido/
[19]http://ds.arm.com/

After having used Open-TEE, several participants commented *"debugging is easy"* or *"debugging is fast"* in the post-study questionnaire. One participant characterized how Open-TEE could be integrated into his existing workflow before cross compiling to target hardware: *"[Open-TEE] complements nicely my previous SDE - first preliminary testing with Open-TEE & `gdb` & `OT_LOG(..)`, and only after that ARM cross compiler & FVP emulation"*. The dominant suggestion for improvement was a desire to see more extensive documentation for Open-TEE.

Given the sample size, the results should be taken as indicative rather than definitive. However, it is reasonable to conclude that Open-TEE has the potential to improve the ease of use of developing TEE applications.

V. Related Work

Ekberg et al. [1] describe a number of reasons for the under utilization of TEEs in devices. Ranging from lack of standard APIs, availability of SDKs; lack of trust between the different stakeholders, with OEMs being unwilling to open up their security environments to third parties. In this section we will review a number of initiatives that have been undertaken to address some of these issues and compare these efforts to Open-TEE.

ObC [20] was one of the first attempts to address the problem of opening the TEE to third party developers by challenging the prevailing opinion that a credential system must be centralized and closed. ObC predates many standardization efforts and as such defines a proprietary mechanism by which to enable the CA/TA communication and synchronization while leveraging the TrustZone architecture to enforce the security. On the other hand our work aims to promote standards adoption in order to proliferate TEE research and deployment.

Muthu [27] analyzes extending QEMU to support TrustZone, the feasibility of such a solution, and tries to determine if it would be beneficial to the developer community. Winter et al. [28] go one step further and implement a TrustZone emulator as an open source project. However, we were not able to find the source code. Open-TEE addresses the issue of virtualizing the TEE, however, in contrast we are not tied to the emulation of a specific TEE implement. One issue with developing an emulator for the TEE is that it still lacks an operating system to run. In section II-C we highlight the lack of a standardized OS even among the different TrustZone implementations.

To this end there have been a number of efforts to create an OS that is suitable to be deployed in TrustZone [29] [30] [31]. All of these are open source solutions which are released under various licenses (Table V). In addition to providing an operating system for the TEE both OP-TEE [30] and T6 [31] choose to rely on GP as their RPC mechanism between the REE and TEE. TLK [29] on the other hand chooses to provide a proprietary communication mechanism.

Sierraware's Open Virtualization [32] provides a dual-licensed OS implementation[20] that also supports the GP standards. The commercial products (sierraVisor, SierraTEE) provide extended functionality and not being GPL there is no requirement for any changes to be made publicly available by the license holders as is required with their open source offering. Open-TEE is licensed under Apache-V2 giving users the flexibility of an open source license without the strict copyleft requirements.

[20]proprietary, GPL

	Compliance	HW-independence	License
Open-TEE	yes	yes	Apache-V2
Open Virtualization [32]	yes	no	proprietary, GPL
OP-TEE [30]	yes	no	BSD-2,BSD-3
T6 [31]	?	no	?
TLK [29]	no	no	MIT,FreeBSD
TrustZone Emulator [28]	?	no	?

TABLE V: Comparison of available alternatives to Open-TEE

Trustonic's <t-dev developers program [33] was created to support Trustonic partners who have deployed the <t-base TEE implementation. This program provides an SDK, tools and consulting with the aim of easing the development and testing of TEE applications in deployed hardware solutions.

All of the OS based solutions have to be ported to support the various HW environments, increasing the effort of maintaining the OS and reducing the users available options. Many of the OS based solutions also require that the HW be configured in a developer mode, without this setting it is generally not possible to deploy custom SW to the TEE, for obvious reasons, further restricting the developers options. Open-TEE in contrast does not have this HW dependency, thus enabling the users to start developing with the framework once they have cloned the repositories. Based on the references listed above, we conclude that no other project fills the niche of a fast prototyping SDK framework that we have described in this paper.

VI. Discussion and Conclusion

We have demonstrated that Open-TEE meets our objective of an easy-to-use, hardware-independent software framework that allows developers to write and debug GP-compliant TEE applications. We made a deliberate decision to open source Open-TEE under Apache-V2 license [34]. The Apache license was selected because it is a recognized open source license and it provides additional flexibility for those wishing to use the framework. All third party components have been carefully selected and we have used only components that have been properly licensed and do not set any restrictions for future use. This has made it possible for people from outside the research group to contribute to Open-TEE. Currently a number of extensions are being worked on including support for other GP APIs and supporting Client API bindings in Java (for Android applications).

Although the sample in our user study is small, participants were drawn from several different organizations with track records of TA development. This makes us confident that our user study results are valid. It is very difficult at this time to conduct larger-scale user studies of TA development because the community of TA developers is tiny. Recall that expanding the size of the TA developer base is the very motivation for Open-TEE in the first place.

We initially intended Open-TEE as a developer tool. However, an alternative use has become evident in our discussions with service providers. Although use of TEEs can improve the security and usability of their service, not all their clients may have TEE-equipped devices. Yet the service provider would like to present a consistent user experience for their entire client base. A possible approach for them is to ship their application (CA and TA) with Open-TEE and arrange for the CA to use Open-TEE if it cannot detect a real hardware TEE on the device. This would allow the service provider to have a common provisioning mechanism and offer a consistent user experience for all their clients. However, once we cast Open-TEE as a potential fall-back TEE in this manner, we need to address the question of how best to isolate it from the REE in the absence of any hardware support. This is part of our current work.

Our hope in writing this paper is to make the research community aware of Open-TEE and encourage researchers to use it and contribute to its development. We also believe that organizations and developers who already develop TA applications will benefit from incorporating Open-TEE into their development process.

Acknowledgments: We thank the participants of the user study for their time and their useful feedback. Thanks to Kumaripaba Athukorala for advice on conducting the user study. This work was supported by the Intel Collaborative Institute for Secure Computing at Aalto University (http://www.icri-sc.org/), which is jointly funded by Intel Corporation and Aalto University.

References

[1] J. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," *IEEE Security & Privacy*, vol. 12, no. 4, pp. 29–37, 2014. [Online]. Available: http://dx.doi.org/10.1109/MSP.2014.38

[2] J. Azema and G. Fayad, "M-Shield mobile security technology," 2008, TI White paper. http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.

[3] ARM, "ARM security technology — Building a secure system using TrustZone technology," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html, April 2009.

[4] ——, "Technical reference manual: ARM 1176jzf-s (trustzone-enabled processor)," http://www.arm.com/pdfs/DDI0301D_arm1176jzfs_r0p2_trm.pdf.

[5] Intel, "SEP driver," https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/drivers/staging/sep?id=refs/tags/v3.14.32.

[6] Apple, "iOS security," https://www.apple.com/ca/iphone/business/docs/iOS_Security_Feb14.pdf.

[7] M. Leno, "Senate bill 962, leno. smartphones." http://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201320140SB962.

[8] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart, "Building the IBM 4758 secure coprocessor," *IEEE Computer*, vol. 34, no. 10, pp. 57–66, 2001. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/2.955100

[9] "Trusted Platform Module (TPM) Specifications," https://www.trustedcomputinggroup.org/specs/TPM/.

[10] GlobalPlatform, "Home page." http://www.globalplatform.org.

[11] ——, "Device specifications for trusted execution environment." http://www.globalplatform.org/specificationsdevice.asp.

[12] ——, "TEE System Architecture," http://www.globalplatform.org/specificationsdevice.asp.

[13] J.-E. Ekberg, "Securing software architectures for trusted processor environments," Doctoral dissertation, Aalto University, May 2013, http://urn.fi/URN:ISBN:978-952-60-3632-8.

[14] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:1. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

[15] Intel, "Intel software guard extensions (intel sgx)," https://software.intel.com/en-us/intel-isa-extensions#pid-19539-1495.

[16] ——, "Software guard extensions programming reference," https://software.intel.com/sites/default/files/329298-001.pdf.

[17] Y. Cheng, X. Ding, and R. Deng, "Appshield: Protecting applications against untrusted operating system," *Singaport Management University Technical Report, SMU-SIS-13*, vol. 101, 2013.

[18] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for TCB minimization," in *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, 2008, pp. 315–328. [Online]. Available: http://doi.acm.org/10.1145/1352592.1352625

[19] GlobalPlatform, "About." http://www.globalplatform.org/aboutus.

[20] K. Kostiainen, J. Ekberg, N. Asokan, and A. Rantala, "On-board credentials with open provisioning," in *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia, March 10-12, 2009*, 2009, pp. 104–115. [Online]. Available: http://doi.acm.org/10.1145/1533057.1533074

[21] Android Open Source Project, "Managing your app's memory," https://developer.android.com/training/articles/memory.html.

[22] The Chromium Projects, "Multi-process architecture," http://www.chromium.org/developers/design-documents/multi-process-architecture.

[23] QEMU, "Open source processor emulator," http://wiki.qemu.org/Main_Page.

[24] GNU, "GDB: The GNU project debugger," http://www.gnu.org/software/gdb/.

[25] J. Brooke, *Usability evaluation in industry*. Taylor & Francis, London, 1996, ch. SUS: A "quick and dirty" usability scale, pp. 189–194.

[26] A. Bangor, P. T. Kortum, and J. T. Miller, "An empirical evaluation of the system usability," *International Journal of Human-Computer Interaction*, pp. 574–594, 2008, http://dx.doi.org/10.1080%2F10447310802205776.

[27] A. Muthu, "Emulating trust zone feature in android emulator by extending qemu," Master's thesis, KTH Royal Institute of Technology, 2013.

[28] J. Winter, P. Wiegele, M. Pirker, and R. Tögl, "A flexible software development and emulation framework for ARM TrustZone," in *Trusted Systems - Third International Conference, INTRUST 2011, Beijing, China, November 27-29, 2011, Revised Selected Papers*, 2011, pp. 1–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32298-3_1

[29] NVIDIA, "Trusted little kernel (tlk)," http://nv-tegra.nvidia.com/gitweb/?p=3rdparty/ote_partner/tlk.git;a=summary.

[30] Linaro, "OP-TEE," https://wiki.linaro.org/WorkingGroups/Security/OP-TEE.

[31] TrustKernel, "T6," http://trustkernel.org/.

[32] Sierraware, "Open virtualization's sierravisor and SierraTEE," http://www.openvirtualization.org/.

[33] Trustonic, "‹t-dev developer program," https://www.trustonic.com/products-services/developer-program/.

[34] The Apache Software Foundation, "Apache license, version 2.0," http://www.apache.org/licenses/LICENSE-2.0.