Strengthening embedded system security with PUF enhanced cryptographic engines

Matthew Areno

Matthew Areno

Candidate

Electrical and Computer Engineering

Department

This thesis is approved, and it is acceptable in quality
and form for publication:

Approved by the Thesis Committee:

Dr. James F. Plusquellic

Dr. Wei W. Shu

Dr. Jedediah R. Crandall

Dr. Brandon K. Eames

Strengthening Embedded System Security with PUF Enhanced Cryptographic Engines

by

Matthew Cody Areno

B.S., Utah State University, 2007
M.S., Utah State University, 2007

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Engineering

The University of New Mexico

Albuquerque, New Mexico

May, 2013

Dedication

To Slater, for always questioning me and forcing me to continue my never-ending quest for knowledge. To Sara, for always telling me how much you love me and how grateful you are to be in our family. To Freddie, for teaching me that it is always important to take a break and have some fun. To Henry, for always having a hug and kiss ready for me when I felt like I just could not keep going. To Nora, for being just so darn cute that I could never resist putting down the laptop to carry you around.

And to my beloved Cylinda. You are not only a wonderful wife, but a prized friend. You inspire others to believe in what they can accomplish, and then help them each day in their pursuit for success. You love unconditionally and support unwaveringly. I love you for everything you are, and everything you are not. You are my greatest friend, and my true companion. Te amo!

Strengthening Embedded System Security with PUF Enhanced Cryptographic Engines

by

Matthew Cody Areno

B.S., Utah State University, 2007

M.S., Utah State University, 2007

Ph.D., Engineering, University of New Mexico, 2013

Abstract

Mobile security is a vast research world with an ever-changing landscape. Researchers develop security solution one day just to see them overcome a week later. Protocols and specifications are generated that attempt to create order out of the chaos and provide a clear roadmap to the future of mobile security. Unfortunately, that road is often filled with switchbacks and U-turns as security measures are defeated and researchers return to the drawing board.

Research conducted over the past four years has provided significant information that can help to drastically alter the direction of mobile security and help it leapfrog efforts made by hackers around the world. During this time, an analysis of modern mobile security research was conducted. The research presented also includes research conducted on Physical Unclonable Functions and their impact to the world of mobile security. Finally, a novel mobile security architecture has been developed that can provide stronger authentication, unique encryption of security-critical data, and protection of sensitive information during transit between devices.

Contents

List of Figures

List of Tables

Glossary

ASIC	Application Specific Integrated Circuit
CA	Client Application
DRM	Digital Rights Management
EE	Execution Environment
EFT-POS	Electronic Funds Transfer - Point of Sale
FPGA	Field Programmable Gate Array
FSB	Flexible Secure Boot
GP	Global Platform
GPD/STIP	GlobalPlatform Device / Small Terminal Interoperable Platform
GUI	Graphical User Interface
HUK	Hardware Unique Key
IMEI	International Mobile Equipment Identity
IMSI	International Mobile Subscriber Identity
ME	Mobile Equipment

MLTM	Mobile Local-Owner Trusted Module
MPWF	Mobile Phone Working Group
MRTM	Mobile Remote-Owner Trusted Module
mTPM	mobile Trusted Platform Module
NFC	Near Field Communication
OS	Operating System
PRNG	Pseudo-Random Number Generator
REE	Rich-OS Execution Environment
RIC	Runtime Integrity Checking
RT	Root-of-Trust
RTM	Root-of-Trust-for-Measurement
RTR	Root-of-Trust-for-Reporting
RTS	Root-of-Trust-for-Storage
RTV	Root-of-Trust-for-Verification
SCMS	Smart Card Management System
SEE	Secure Execution Environment
SIM	Subscriber Identify Module
SoC	System-on-a-Chip
STIP	Small Terminal Interoperable Platform

TA	Trusted Application
TGG	Trusted Computing Group
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
TZ	TrustZone
UICC	Universal Integrated Circuit Card
WAC	Wholesale Applications Community

Chapter 1

Introduction

In an article presenting results of a recent report from Cisco's Global Mobile Data Traffic Forecast, it was stated, "By the end of 2012, the number of mobile-connected devices will exceed the number of people on Earth, and by 2016 there will be 1.4 mobile devices per capita." [5] Further, in 2011, mobile data traffic was 8 times the size of the entire global Internet in 2000. To say that mobile device usage is on the rise today may be one of the greatest understatements of our age.

While the number of devices being used, combined with their respective data usage, is in itself remarkable, the extent to which the devices are used is equally astounding. Current predictions show anywhere from 44 to 183 billion mobile application downloads by 2015 [6] [7]. Many of these applications, also known as "apps", are providing account access for membership organizations, financial institutions, email services, and much more. Unfortunately, the question of security during these interactions is often being answered long after these services have been deployed.

Users typically have no understanding of how their data and personal information is transmitted to the necessary entities, or how it is handled on the mobile device. A prime example of this is the recent disclosure that the iPhone mobile app for Southwest Airlines submitted a user's username and password in plain text over the Internet to their remote server [8]! This discovery was not found by the application developers, but by a hacker performing random attacks on mobile applications. While the issue was eventually addressed and an update was provided, the incident illustrates the fact that such issues are often discovered by normal users or by members of the security community, and not by application developers.

Issues such as these lead one to question the very nature and extent of the security provided by these mobile devices. However, such occurrences should not lead people to believe that security is not a factor in the design of mobile devices. Security is, in fact, one of the prime considerations by most device and software manufacturers around the world.

A question that often comes up is, "why can't we simply implement the same security methods on mobile devices that already exist on our desktop systems?" In order to answer this valid question, it is important to understand the vast differences between these two architectures.

Security for desktop systems is designed around the fact that information transfers happen via a limited number of controllable interfaces, i.e. networks and removable drives. While each of these provide the possibility for serious security issues, the User has greater control over the functionality of these interfaces. For instance, if users are concerned about network-based attacks, they can easily turn off their wireless card or unplug their network cable. If they are concerned about the contents of a specific CD or DVD, they can simply not insert the disk into their machine.

Users of mobile devices rarely have the same fine-grained control of their devices as desktop users. The Short Message Service (SMS) used by mobile communication networks to transmit small text messages between devices is not controllable by the User. These messages are not only queued by the network in order to help ensure delivery, but are received whether the User requests them or not. Because of this, SMS messages were used extensively in mobile attacks during the early 2000s, and are still used today.

Another major difference between desktop and mobile systems is the number of entities that have a vested interest in the operating state of the device. For desktop systems, the User is the owner of the entire system and has full control over what software is loaded onto the system, as well as when and how it operates. If the device is connected to a network and is found to be operating in a malicious manner, network service providers may disable services, but otherwise have no control over the device.

These controls, like the ones presented previously, are not the same for mobile devices. Operating System (OS) providers, third-party applications, service providers, and device manufacturers all have a justifiable need to control one or more elements on the device. A prime example is the baseband processor on a cellular device. This processor is responsible for controlling the modem in order to ensure the device is operating within the correct frequency spectrum to facilitate available communications mechanisms. A device manufacturer will typically control this in accordance with requirements dictated by the service provider. Control of this process must be available to the manufacturer, even after the user purchases the device.

1.1 Security Concerns in Embedded Systems

Due to the complexity of a system with multiple stakeholders, the idea of developing a "one-size-fits-all" solution is at best problematic. Further, mobile devices, or cellular devices as they are often called, are not the only hardware platforms that are in need of protection. Embedded systems that rely upon the same type of processing elements include healthcare systems, automotive computers, hand-held electronics, femtocell devices, and radio-frequency identification (RFID) equipment. Such devices often utilize ARM core processors, similar to the ones used in the majority of smart phone cellular devices. That being the case, solutions developed in the mobile world can potentially have implications in various other arenas as well.

1.1.1 Automotive Security

In 2010 and 2011, Checkoway et.al [9] [10] demonstrated the ability to hijack the computer system inside a number of automobiles. Most modern vehicles contain a number of different computers, each with a variety of capabilities and responsibilities in the overall operation of the automobile. An internal network is typically used to connect these computers, allowing them to share information and work together to control the vehicle. However, as demonstrated by the authors, these networks rarely have any form of authentication built in, but rather assume that commands and information received from any other computer is valid. The authors discovered that on the vehicles tested, any computer on the network was able to re-write the firmware of any other computer system. By gaining execution through the radio, On-STAR system, or even just through the standard On-Board Diagnostic (OBD-II) terminal, the attackers were able to unlock the car, disable the security system, turn the car on/off, disable the brakes, and perform a variety of other safety critical actions.

1.1.2 Medical Security

Security concerns with medical devices are also starting to crop up. The United States Government Accountability Office recently issued a report advocating the need for greater security examination of medical equipment [11], likely prompted by reports of successful hacks, such as that performed by Jerome Radcliffe [12]. In order to reduce costs and increase patient comfort, hospitals have begun networking medical equipment in order to allow doctors and nurses to make changes and monitor conditions without having to enter a patient's room. As with vehicular networks, there is often no authentication mechanism in place to prohibit an attacker from connecting to the network and sending bogus commands to connected devices. Further, there are serious privacy concerns in regards to an attacker being able to ascertain patient information and potential health issues by simply snooping a network connection.

1.1.3 Mobile Infrastructure Security

Femtocells are a new device being pushed by mobile service providers as a method for off-loading data usage loads on to landline based network connections. This has been done in large part due to the acceleration in the use of smart-phone devices that often consume what cellular provides consider "large" amounts of data. One of the primary concerns with this approach is that an attacker is provided with access to a resource that would normal be physically restricted, such as a cell tower. Researchers demonstrated an exploit based upon this very concept in 2011 in which they were able to monitor communications between the femtocell and the cellular network by tapping the hard-wire network connection [13]. The researchers were also able to rewrite the firmware of the femtocell and gain access on a device that can inject data directly into the cellular network, allowing them to send malicious and malformed data packets to the cellular network backbone. This revealed a vast amount of user information, including the locations of all femtocell devices in the entire country of France!

1.1.4 Mobile Device Theft

Theft of mobile devices also continues to be a problem today. Smart phones are significantly more advanced than previous mobile devices and as such command a much higher price on the black market. Users also tend to store much more personally identifiable information on these devices, such as contacts, email, credit card information, membership club details, and many other such items. Depending on the country, thieves are often able to simply remove the Subscriber Identification Module (SIM) card from the phone and replace it with a new one. Once done, the phone connects to the corresponding cellular network and continues to be usable. Despite the use deterrents, such as black lists and International Mobile Equipment Identity (IMEI) numbers, attackers have found methods of circumventing these roadblocks and regaining access to the underlying device. Without a non-circumventable method for associating a device with a specific user, this will continue to be a problem for the foreseeable future.

1.2 Security Approaches

Problems such as these are currently being researched by an assortment of different institutes delivering a potpourri of solutions. During this process, an analogy can be made to a lake into which a variety of different rocks are thrown. The result is chaos: waves of different sizes colliding and disrupting the beauty of the water. But as time goes on, the waters calm and the waves dissipate, leaving the viewer with a clear view of the lake. The conglomeration of these ideas into a simple, elegant solution that can calm the waters is often where the real science is born.

Each solution that is developed or presented is typically tailored towards the purposes of the originator and is not necessarily meant to provide an all-encompassing solution. This is not unexpected, nor is it a bad thing per say. However, it is left of others to determine the proper methodology for merging these technologies into a solution capable of addressing each of the concerns presented in Sec. 1.1. The three primary questions that must be resolved by any proposed solution are: 1) How can a family of devices that are all based off the same design be protect in a unique method that does not require differences in design or maintaining massive data about each device? 2) How can the mobile device be authenticated in a way that correlates a device to a user on a specific network without being subject to alteration by malicious software? and 3) How can the mechanisms developed to address these two question be protected from both external and internal attacks on the device?

1.2.1 Device Originality

Device originality may not initially appear to address any of the issues presented previously, but it is critical to authentication and true data protection. Changes in the hardware design of chips result in significant financial costs and are therefore not feasible as a method for device originality. This led to the use of the IMEI number as a method for uniquely identifying each device. However, this number is not a cryptographic key and has never been used for data protection. Further, this value is generally read from software and then returned. Any malicious code running on the device would have the potential to intercept such calls and manipulate this value. Further, the IMEI number is often not protected in any way and is easily read even by external software.

Most modern security solutions for desktop computing use public/private cryptographic key-pairs for uniquely identifying a device and ensuring that communications received originated from the expected party. These key-pairs are created through a special mathematical relationship that allows a message to be encrypted using one key and then decrypted only through the use of the other key. In this situation, a client would encrypt a message using a server's public key. The server, using its private key, would be the only entity capable of decrypting the message. If the client then receives a valid response from the server, it knows that it received the original message and was able to decrypt it, thereby validating that the server is who it claims to be.

Researchers over the past few years have identified the capability of generating unique values that can be used for identification or key generation through the use of Physical Unclonable Functions (PUFs). A PUF generates unique-per-device values by leveraging manufacturing differences between chips. Even though chips may contain the exact same hardware designs, the fabric upon with the logic is constructed has variations and defects that don't directly alter the functionality of the underlying logic, but can be utilized to generate random values that are unique for each device. This happens on each instance of the same chip, even for chips that are manufacturer at the same time and same location under all the same conditions. As a result, researchers have proposed their use in the generation of secret keys of use in Advanced Encryption Standard (AES) operations, as well as for random input into public/private key pair generators [14] [15].

Implementing a PUF circuit does not require differences in the hardware design, yet still provides variable, non-deterministic values for each device, thereby fulfilling the necessary requirements. By creating a unique-per-device AES key, a PUF enhanced design provides the capability to uniquely protect onboard data even on devices with identical hardware architectures and data. Additionally, no information about the key is required to be maintained off-chip.

A primary example of this is protection of device firmware. Transmission of the firmware to the device utilizes a public/private key-pair, but in a somewhat untraditional way. The server that maintains the firmware will encrypt the firmware and then sign it using its own private key. This encrypted blob of data is then sent to the mobile device along with the server's public key. In this case, the server is not concerned with someone being able to capture this transmission and decrypt the firmware, but is more concerned about proving to the devices that the packet is valid.

To support this approach, a mathematical operation known as a "hash" is performed on the public key, which results in a non-reproducible, and often smaller value. This hash can be stored in non-volatile storage on the mobile device. When the device receives the firmware update package from the server, it performs the same hash on the received key. If the result of the hash matches what is in memory, the device knows that the key is valid and has not been altered. It can then use the key to verify the signature of the firmware package and decrypt the remaining data. Once the data has been decrypted, it can then be re-encrypted using the device-unique secret key and stored on the device. This provides a unique instance of identical firmware on every device.

1.2.2 Device Authentication

The use of PUFs to generate a secret key and a public/private key pair also allows designers to address the issue of device authentication. A normal cellular device uses a SIM card to store user authentication information that is used to identify a user to the network. This exchange has worked well for years, but does not provide any association of the user to the actual device. In order to protect cellular networks, as well as automotive and medical networks, a method must be used that can associate the device itself to the network without being affected by native code executing on the device.

The use of a PUF generated value alleviates the ability of native code to simply read the identification value or authentication (public/private) keys. When authentication is requested, the processor sends a command to a PUF enhanced cryptographic unit instructing it to provide an authentication value. In conjunction with the SIM card, the cryptographic unit receives the authentication request and returns the necessary response. All that software running on the device can do is ignore the request, as it has no access to the authentication values. Doing so simply prohibits the device from connecting to the network and provides no information to the native code, malicious or otherwise.

The identity of the device can be done using a uniquely generated value, similar to the IMEI, or it could simply make use of the public/private key-pair for identification. For instance, consider an automotive network. The various computers on these networks need the ability to communicate with one another. However, a mechanism must exist for them to identify each other and know that received communications are legitimate and are not be spoofed by an unknown entity on the network. This problem can be solved using the public/private key-pairs.

When the automobile is first produced, the computer systems can be set in a “discovery” mode. Using an external computer, an identification packet can be sent to each automotive computer. This packet would contain the public key of the sending computer. The automotive computers then maintain this key by encrypting it with their own secret key and then storing it in non-volatile memory. Each computer can then also exchange public keys with all other computers in the automobile. Once this process has completed, a command from the external computer can be sent to move the computers from “discovery” mode to “operational” mode. At this point, all communications between computers can be encrypted and identities can be validated based upon results stored during the “discovery” phase. If at any time a failure occurs or another computer needs to be added/replaced, the dealer can implement such a change by authenticating itself with the public key of the external computer and place the automotive computers back into “discovery” mode.

While it would be possible to provide authentication using only the secret AES key, public/private keys are frequently used in RSA cryptographic operations in order to facilitate the exchange of secret keys. The reason for this is that public/private keys provide strong identification properties but are computationally expensive, especially in terms of time. For this reason, it is customary to use public/private keys only to establish a secure communications channel and to then exchange a symmetric secret key that can be used for all further communications.

1.2.3 Device Defense

In order for PUF generated values to be protected from attacks, the first step is access control. Values generated by the PUF are not required to be stored in non-volatile memory. A PUF is capable of not only producing unique-per-device values, but also doing so consistently. Each time the device boots, the PUF regenerates the exact same values, which can then be latched, i.e. stored, into volatile storage. By only maintaining these values in volatile storage, they will be erased each time the unit reboots or otherwise loses power.

The PUF values can also be used indirectly rather than allowing direct access to their contents. For instance, modern cryptographic units receive commands through memory-mapped registers that can be access from both the cryptographic unit and the main processor. The processor writes values to these memory locations that tell the crypto unit what operation to perform, what key to use, and where to get and store the necessary data. Then, through the use of a mux (a selectable-output device), the PUF value can be selected as the key value for the corresponding algorithm accelerator inside the crypto unit. This prevents the PUF generated values from being directly accessible by the cryptographic unit or by any other device in the system.

By using a PUF generated AES key, boot-loaders, i.e. firmware, used to load the operating system can also be protected in a unique manner. As mentioned previously, these boot-loaders can be sent to the device encrypted, then decrypted and re-encrypted with the device unique key. This results in identical firmwares being stored uniquely on each device, thereby preventing an attacker from creating a single, modified firmware and loading it on any device. The attacker would have to gain access to the AES key for each device before they were able to directly overwrite the firmware file. This helps to reduce the potential for attacks across an entire family of devices and instead makes attacks much more local in scope.

While access to the PUF enhance cryptographic unit is required, it must be restricted in order to prohibit arbitrary usage by malicious code. Some encryption algorithms are susceptible to what are called plain-text/cypher-text attacks wherein information about the value of the key can be inferred by analysis of plain-text and the resulting cypher-text. While there are no known plain-text/cypher-text attacks on AES, it may still be possible for an attacker to spoof identification if it is provided unrestricted access to the crypto unit. The use of Trusted Execution Environments (TEEs) provides a mechanism for creating isolated execution environments for trusted applications. Restricting access to the crypto unit only to applications executing within the TEE will help absolve potential issues of identity spoofing. Further, any application executing with the TEE is assumed to be trusted and should never act in a malicious manner that would re-open this possibility.

Additionally, a PUF enhanced cryptographic unit is capable of not only producing a single AES secret key, but can generate hundred or thousands. In a traditional approach where a secret key (or its hash) is stored in non-volatile storage, no known method exist for modifying this value in the instance that the key is compromised. Such values are often referred to as being "burned-in" meaning that once they are set, they can only be partially changed and only with physical access to the device. By contrast, if the secret key of a PUF enhanced architecture is ever compromised, the cryptographic unit can easily change to a new key. This can be done by decrypting any secret key protected elements using the original key and then re-encrypting them using a newly generated key. The crypto unit only needs to maintain information about which PUF circuitry is used for key generation at the time. Because of the nature of PUF circuits, such information provides no leakage of key values.

1.3 Research Objectives

The purpose of this research is to prove the viability of merging the various mobile security solutions into a singular solution that is capable of addressing each of the issues presented thus far. The novelty of this work comes not in the development of a new architecture or tool, but in how existing capabilities can be conjoined to provide enhanced mobile security. Specifically, this paper will explore the use of PUF technology to support the assurance of provenance of device boot software in mobile systems. Additionally, it will explore how PUF technology can support the concept of unique device authentication, together with user authentication, on mobile networks. Finally, it will detail how PUF technology can provide authenticated, private, non-reputable communication and protection of onboard data.

During this research, all of the modern mobile specifications and protocols, hardware and software security elements, as well as academic and professionally produced solutions were considered. Through scientific analysis of the benefits and drawbacks of each approach, a justification for the use of PUF technology has been founded which addresses the dire security concerns of our modern world. Rather than simply presenting yet another “security feature”, the goal of this work is to discover if a method exist for properly conjoining a number of these approaches into a single, PUF-based implementable solution that can be easily adopted by manufacturers and utilized by developers.

The details of how this analysis was performed are contained in the remaining chapters. In Chapter 2, a history of mobile security, as well as a discussion of prior and current work done both in the academic and corporate realms will be presented. Chapter 3 will present a detailed description ARM TrustZone technology and how it can support the creation of an isolated software execution environment, known as a Trusted Execution Environment (TEE). Chapter 4 will build upon this concept of a TEE and detail proposed modifications to standard mobile cryptographic units that will provide enhanced security features for mobile platforms. The results of this merger will be presented in Chapter 5, and the subsequent conclusions will be presented in Chapter 6.

Chapter 2

Background

The area of mobile security is vast and complex, consisting of a variety of standards, protocols, and specifications. Further, this topic has been and continues to be addressed by academia, private industry, manufacturers, independent organizations, and many other parties with a vested interest in mobile security. Unfortunately there is currently no “one-size-fits-all” solution for addressing all mobile security issues, with the best software approaches being as mutable as the underlying hardware architectures on which it runs.

In order to present information on the current secure platform ideologies in the most concise and straightforward way, a history of mobile security will first be presented. This history provides insight on how mobile security has evolved over the past twelve plus years and who the major players are that are impacting its direction. This also includes brief descriptions of many of the standards and specifications that have been developed to address issues relating to mobile security. Research performed by academia is presented to show how these standards and specifications may be used on various architectures, as well as modern uses of PUF technology in chip security applications. This chapter concludes with a few examples of commercial applications and how they are attempting to utilize these tools to provide platforms with secure execution environments.

2.1 A History of Mobile Security

As with most major technology families currently in use, the area of mobile security has multiple functional specifications that describe the various levels of operation that are necessary to support a so-called "secure" system. There are currently two primary organizations that are providing and maintaining these specifications: GlobalPlatform (GP) and the Trusted Computing Group (TCG). The Mobile Phone Working Group (MPWG) is a part of the TCG and focuses specifically on the development of specifications related to mobile security.

Before delving into the details of these specifications, it is important to understand the history of these two organizations and how they are being supported. In this section, information on the most commonly used specifications in mobile security will be presented. This is intended to be an informative overview of these documents, rather than an in-depth analysis. Future sections of the paper will dig deeper into the details of several of these documents, but at this point a general concept of each will be presented instead.

2.1.1 GPD/STIP

GlobalPlatform was formed in 1999 and was the primary entity overseeing the Visa Open Platform specification [16]. Visa Open Platform was originally created to support the development of Java-based applets that could be run on Smart Card devices. Since that time, GP has continued to focus on smart card development, with additional work in the areas of device and system specifications. While their work in card and system specifications is significant, their development of devices specifications is of the highest relevance to the research presented in subsequent chapters. The most influential of these is the Trusted Execution Environment (TEE), but everything started with their development of the GlobalPlatform Device/Small Terminal Interoperable Platform (GPD/STIP) specification.

In the year 2000, the Small Terminal Interoperable Platform (STIP) effort was started with the incorporation of the STIP Consortium. Most initial work performed by this group related to electronic-funds-transfer/point-of-sale (EFT-POS) terminals. Despite this primary emphasis, work with mobile phones was also included. After the release of the first comprehensive and flexible open specification for EFT-POS in 2002 [17], several new member companies made the decision to join. The EFT-POS and a mobile profile were both proposed the following year based upon input and support from these new members. The success of these proposals, as well as the fact that many members already had strong relationships with GP, lead to the transfer of this intellectual property from the STIP Consortium to GP. This was done with the intention of providing a more stable and long-term environment for supporting these standards, since the STIP Consortium was never intended to be a large standardization body. The STIP specification then became known as the GlobalPlatform Device / STIP, or GPD/STIP for short.

GlobalPlatform continued to develop and support various specifications over the next few years. In 2007, GP published a white paper that became an announcement of their intention to take on a much stronger role in the development of standards in mobile security [18]. This paper, titled "Why The Mobile Industry is Evolving Towards Security", advocated the need for an isolated execution environment, dubbed the Secure Execution Environment (SEE), to provide proper security for mobile elements. At this time, GP pushed the use of their GPD/STIP technology to address four focal areas:

1. Interoperability

2. Security

3. Flexibility

4. Reactivity

To address these four paradoxes, the GPD/STIP card specifications were designed to utilize a portable base language, as well as a service control interface and service control manager. A portable base language refers to the use of a language that is agnostic to the underlying platform on which is it used. Supported languages must also be object-oriented and strongly typed, such as Java. This provides developers with assurances that operations performed on one architecture will be handled exactly the same on all other architectures. Additionally, data types and objects are maintained and stored in an identical manner on all platforms. The theology behind this approach was to allow developers to create single solutions that would work on any platform. Members of GP knew that developers would be unwilling to create unique implementations for every supporting device, nor did they wish to deal with fragmentation issues that can result for some requirements.

Objects known as service control elements are meant to address the last two requirements. Rather than requiring support for all possible resources that may exist on any platform, each resource is instead treated as a service and is represented by a software library. This library, and its associated functionality, is referred to as a service control element. Using this methodology allows manufacturers to implement a generic GPD/STIP compatible architecture, while including necessary libraries in order to support devices with different resources and capabilities.

2.1.2 Trusted Execution Environment

GlobalPlatform's work in mobile security continued with the development of the Trusted Execution Environment (TEE) specifications. This began in 2009 with the adaptation of the ARM TrustZone API specification [19], which will be discussed in more detail in section 2.2.1. After adopting this specification, GP released the TEE Client API [20] in July of 2010. This was followed by the TEE Internal API [21] and the TEE System Architecture [1]. (each of these will be discussed in greater detail in subsequent paragraphs) Additional TEE specifications are scheduled for release in late 2012 and early 2013.

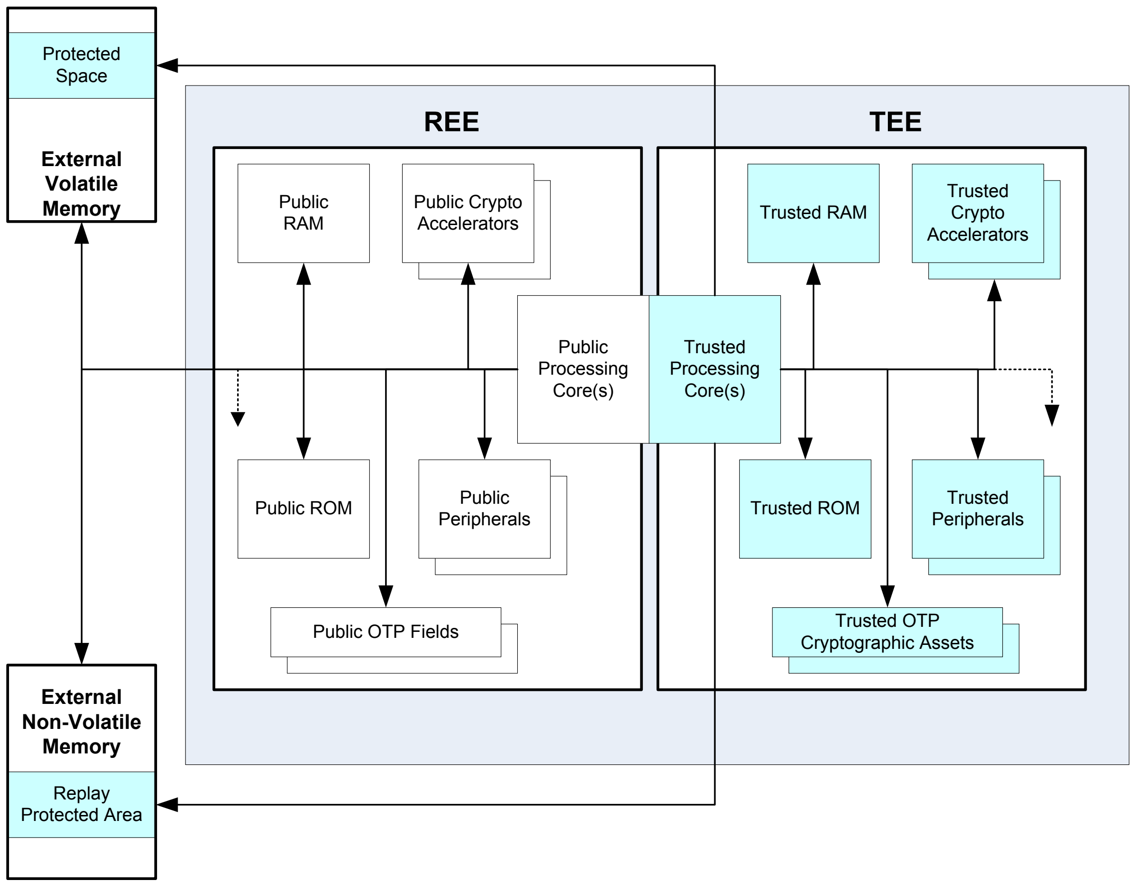

Protected Space
External Volatile Memory
REE
TEE
Public RAM
Public Crypto Accelerators
Trusted RAM
Trusted Crypto Accelerators
Public Processing Core(s)
Trusted Processing Core(s)
Public ROM
Public Peripherals
Trusted ROM
Trusted Peripherals
Public OTP Fields
Trusted OTP Cryptographic Assets
External Non-Volatile Memory
Replay Protected Area

Figure 2.1: TEE hardware design [1].

The motivating factor behind the development of the TEE specifications is the need to ensure isolated execution of critical code. The term “isolated” is meant to imply that the code being executed cannot be affected by the state or intent of any other code running on the system. This philosophy is meant to address the possibility of malicious code altering or intercepting information used to perform transactions of a sensitive nature, such as banking transactions. If the code used to perform these transactions can be “isolated” for this malicious code, a higher level of assurance can be provided for the legitimacy of such transactions.

In order for a TEE to provides this level of isolation, hardware support is necessary. Hardware elements must provide a mechanism for switching between the TEE and the standard execution environment, known as the Rich OS Execution Environment (REE). It must also provides security for data and instructions store in internal memory as well as external memory. Mechanisms must also exist for specifying which execution environments can access which peripherals. In essence, the entire hardware architecture must support partitioning of access based upon these two execution environments.

Further, in order for software developers to utilize these hardware features, a software Application Programmer Interface (API) is needed. Not every manufacturer will want the exact same hardware partitioning setup. For instance, one manufacturer may want the cryptographic unit to be accessible by both the TEE and the REE, while another may want it to only be accessible from the TEE. As a result, programmers need an API that allows them to customize the access levels of the hardware components. Further, the API must also provide methods for controlling the transition between the two execution environments, determining interrupt destinations, message passing, and a variety of other functionality.

In order to create an isolated environment, the hardware is partitioned into two execution levels: Trusted and Non-trusted. (These terms are used interchangeably

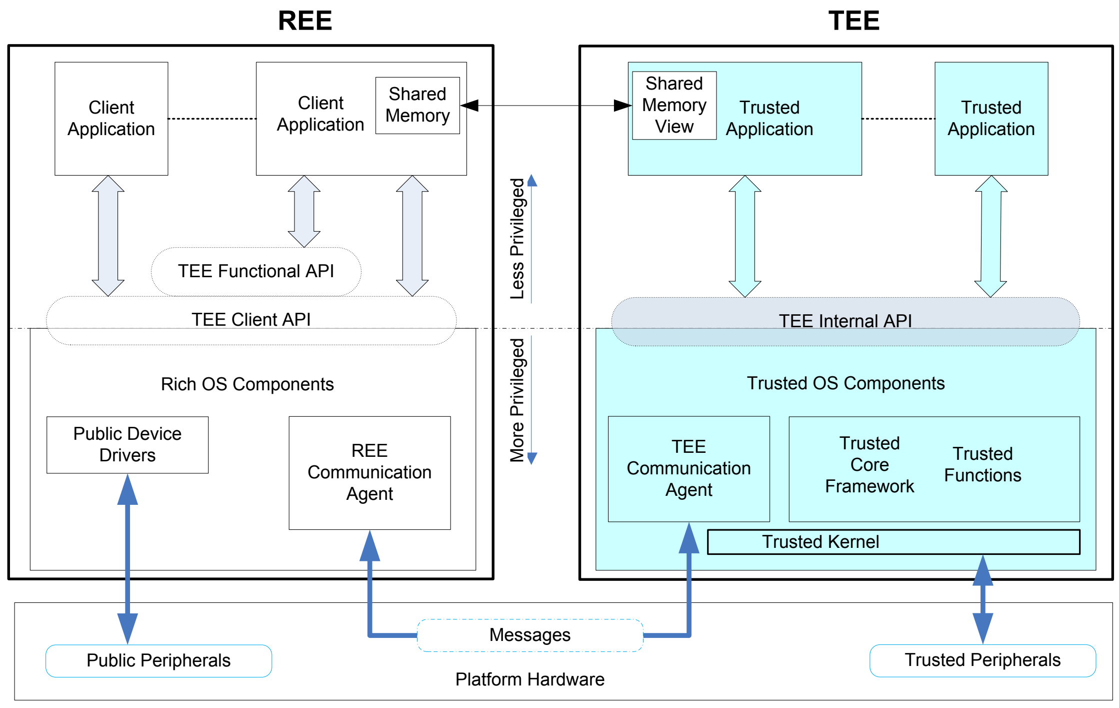


Figure 2.2: TEE Software Architecture [1].

with Secure/Non-secure, Trusted/Public, and Private/Public) This is illustrated in Fig.2.1. Each element may be represented by a single instance capable of switching between these two modes of operation, or by multiple independent instances. For instance, a TEE compliant System-on-Chip (SoC) may contain two separate ROMs, one that is used by the TEE and a separate one that is used by the REE. In contrast to this, there would likely be only one instance of a keyboard on the device. The keyboard would therefore need to be able to function in a manner that allowed it to be partitioned between the TEE and REE, thereby prohibiting access of TEE memory locations from within the REE.

From a software point of view, the execution environment is also partitioned between two states: a trusted and non-trusted, as depicted in Fig.2.2. In this figure, there are references to two specific software APIs: the TEE Client API and TEE

Internal API. The TEE Client API is used by Client Applications (CAs) to make calls into Trusted Applications (TAs) in order to access trusted operations. The TEE Internal API is provided to developers of Trusted Applications in order to access the underlying hardware elements included on the platform. Further, a third mechanism is also provided to facilitate efficient communication between the REE and TEE. This mechanism is called the REE Communications Agent and manages shared memory locations that are readable and writable by both worlds.

Inside of the TEE are two distinct classes of software: the Trusted OS Components and the Trust Applications. The purpose of the Trusted OS Components is to provide OS style functionality to the entire TEE. This includes scheduling and context switching, and should behave exactly like a standard kernel, only it exists inside the TEE.

The Trusted Applications then run on top of this trusted kernel and provide the actual functionality to be used by Client Applications. Each time a CA requests access to a TA, a new instance of the TA is created and stored with its own state and address space, thus isolating multiple instances of a TA from one another. These TAs then interface with the Trusted OS Components via the TEE Internal API, as defined in [21].

It is important to note that the TEE documentation does not define a specific architectural implementation, but rather dictates what an architecture must include in order to be compliant. The implementation details are then left to the manufacturer of the SoC. A more detailed analysis of TEEs will be provided in Section 3.

2.1.3 Open Mobile Terminal Platform

The concept of a Trusted Execution Environment was not introduced via the TEE specification. In 2009, two years prior to the first TEE specification, the Open Mobile

Terminal Platform (OMTP) released the latest versions of their Trusted Environment and Advanced Trusted Environment specifications [22] [23]. These two specifications helped to form the basis of many aspects of the TEE specifications.

The OMTP was originally founded in 2004. The purpose of the OMTP was for discussion of standards with various parties in the mobile community and the creation of standards that could be used across all areas and countries. While the majority of their work did focus on mobile systems, they also produced standards for a variety of wireless, embedded system type applications.

In 2010, the OMTP transitioned into a new entity called the Wholesale Applications Community [24]. At that time, the OMTP has nine full members and two sponsors, including AT&T, Ericsson, and Nokia. Although the Wholesale Applications Community, or WAC, does still maintain the original specifications developed by the OMTP, the primary focus has shifted to application development support across various mobile platforms. Despite this fact, the original OMTP specifications are still highly regarded and often sited in recently released specifications, such as the TEE.

2.1.4 Trusted Environment: OMTP TR0

In 2009, the OMTP group released the latest version of their initial Trusted Environment document, known as OMTP TR0 [22]. This document was released prior to GP's release of their TEE specifications. The TEE specifications are said to be in compliance with this document, not vice versa, though they can be considered complimentary. The main purpose of this document was to identify the necessary features and functions that must exist in various architectures that attempt to implement a Trusted Execution framework. This included the following applications:

- Debug port protection
- Mobile device ID
- Subscriber Identity Module (SIM) lock
- Mobile Equipment (ME) personalization
- Digital Rights Management (DRM)
- Secure Boot (SB)
- Secure binding
- Secure flash update.

Although this document covers an array of mobile security areas, the primary element of interest is the Secure Boot (SB) process. The SB process, as defined in this document, stipulates the exact requirements of each stage of the boot process. Since SB is a requirement of most other specifications, it is important to understand how it works.

SB starts by executing code from a hardware controlled, integrity-protected memory location from within the System-on-a-Chip (SoC) device. Prior to executing any additional code, such as multi-level bootloaders or the kernel, the integrity and authenticity of the code must be verified. Further, no commands received via external interfaces are processed as validation of user input can lead to potential security risks. This includes interfaces such as Bluetooth, Near Field Communication (NFC), and 802.11. And finally, in a multi-processor architecture, there are two options. Either all processors can implement all of the secure boot requirements, which will typically be wasteful and could lead to deadlock issues if not coded properly, or one processor can perform all secure boot requirements while all other processors only execute code that has already been properly authenticated.

In addition to the SB process, the OMTP TR0 document also presents requirements for a Hardware Unique Key (HUK). The document requires that a HUK must be generated using a Pseudo-Random Number Generator (PRNG) either on or off the device. Once loaded, this key can never be changed and must be stored in some form of secure hardware or hardware-controlled secure memory. Further, access to a HUK must be done only from secured/protected hardware or memory. External reading of the HUK should never be possible and the HUK must be a minimum of 128 bits. The use of any non-established algorithms or techniques is also prohibited. The full list of requirements is provided in HU 1 - HU 11 of Section 6.2 of [22].

2.1.5 Advanced Trusted Environment: OMTP TR1

After completing the TR0 document, the OMTP began working on TR1, known as the Advanced Trusted Environment [23]. This document followed the same methodology used in the development of TR0, but attempted to provide a more "comprehensive security roadmap." Topics covered in this document include:

- Trusted Execution Environment
- Secure Storage
- Flexible Secure Boot
- Generic Bootstrapping Architecture
- Run-Time Integrity Checking
- Secure Access to User Input/Output Facility
- Secure Interaction of UICC with Mobile Equipment

The TR1 document, which also had its latest release in 2009, was the first to formally specify requirements for a Trusted Execution Environment. The GP TEE specifications discussed previously are based off these requirements presented in TR1 for a Trusted Execution Environment, and any GP TEE compliant architecture would also be fully compliant with the TR1 document. In the TEE environment portion of the document, several specific requirements are listed that pertain to items such as a HUK, provisioning and storage of private, secret, and/or root public keys, application support requirements, and mechanisms for interactions between the TEE and any other execution environment (EE) on the device.

Aside for the TEE requirements, the flexible secure boot (FSB) and run-time integrity checking (RIC) sections are also of keen interest. The TR0 document first presented the concept of SB, but focused mostly on the integrity and verification metrics for the initial code. In TR1, the FSB is defined as a set of requirements regarding the update, or modification, process for the initial code. Because it is never expected that perfect code will (if ever) be developed the first time, a mechanism must exist for updating code that is part of the SB process and which will not adversely affect the security integrity of the SB process.

The RIC section explicitly details the requirements for analyzing software currently running on a platform in order to assure that it has not been modified in any way. This technique is meant to mitigate attack vectors that a hacker might use to modify the expected behavior of a specific element of code.

For instance, consider an Apple iPhone. Due to the restrictions that Apple enforces on application purchasing through their App Store, hackers have developed a method known as "jailbreaking" that provides the ability to run unsigned and unauthorized code on the device. This allows the device owner to install applications from any application store he is willing to use. In order to do this, modifications to the Kernel are required. This is typically done by exploiting some application on the device, gaining supervisor privileges, and then overwriting the kernel check in memory. The actual steps required are significantly more involved and have been published in detail [25].

The purpose of RIC is to specifically address issues that occur as a result of this type of attack. If Apple used a RIC method, it would be possible to detect alterations made to the Kernel after it had been loaded into memory. RIC is required to run either in the OS, EE, or run-time modifiable software, or to run in collaboration with software that as already been check for integrity. Ideal locations would then be either a separate device, such as a SIM card, or inside of a TEE that is loaded as part of the FSB process.

2.1.6 Mobile Trusted Platform Module

The Mobile Trusted Platform Module, or mTPM, is a standard developed by the Trusted Computing Group (TCG) to bring Trusted Platform Module (TPM) functionality to mobile devices [2]. TPMs are frequently found on standard desktop motherboards, as well as in laptop devices. Their primary function thus far has been to provide a measurement/verification mechanism for implementing the SB process on computers.

TPMs function by performing a hash of a given set of code. This value is then hashed with an internal register to create a new value for that register. This is called an "extend" operation. That value is then compared with a known, good value. If they are the same, the boot continues; otherwise, the boot fails. This continues throughout the boot process with each measurement being "extended" onto an internal register, called a Platform Configuration Register (PCR). A prime example of the use of TPMs is with Microsoft's Bitlocker application [26]. Microsoft has also mandated the use of TPMs starting with Windows 8 [27].

While the concept of TPMs certainly has application in the mobile arena, the implementation requirements are significantly different. The primary reason for this is because mobile equipment (ME), specifically a cellular device, has multiple stakeholders. A stakeholder, as defined by the TCG, is the owner of a trusted engine "... who has exclusive control over the data protection mechanisms in their own engine, and can permit data owners to use those data protection mechanisms." [28]

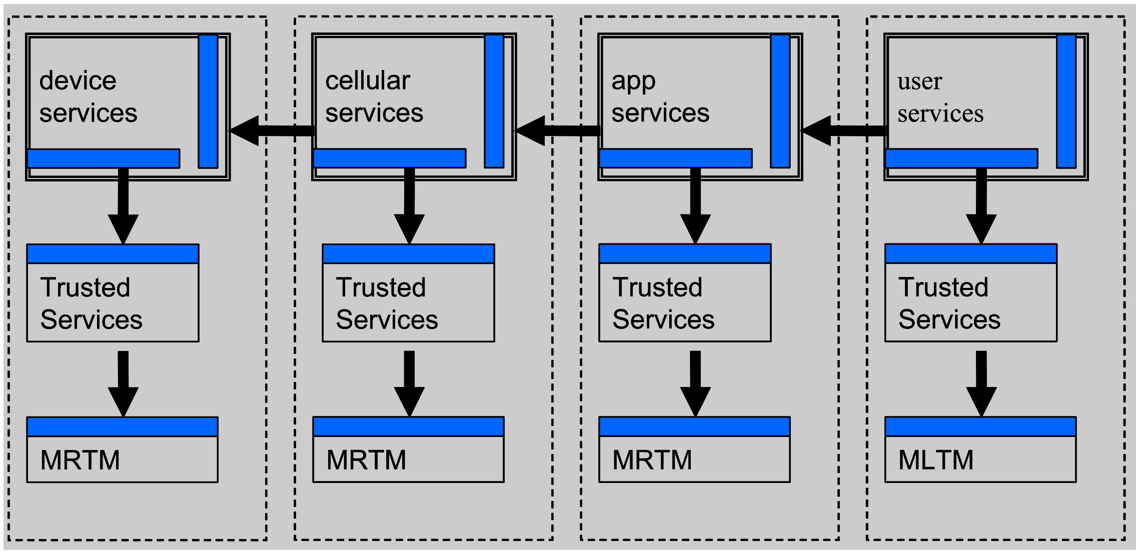


Figure 2.3: Example of a Generalized Mobile Platform [2].

The methodology behind multiple stakeholders is the fact that on ME the owner is not the only entity with execution requirements on the device. This is in stark contrast to standard desktop equipment, such as desktop computers and laptops. Such entities include the device manufacturer, communications carriers, service providers, and users/owners. The role played by each of these entities differs, as do their execution requirements. This is illustrated best in Figure2.3.

In this depiction of a generalized mobile platform, there are four different mTPM modules, one for each of the previously mentioned stakeholders. The stakeholders are divided into two types: Mobile Remote-Owner Trusted Modules (MRTMs) and

Mobile Local-Owner Trusted Modules (MLTMs). The designation between Remote and Local is based upon how execution is initiated on the ME. The User is considered the only Local agent as their execution is established locally through direct interaction with the ME, such as through a provided graphical user interface (GUI). The other three stakeholders must interact with the ME via an externally provided interface, such as cellular or Wi-Fi.

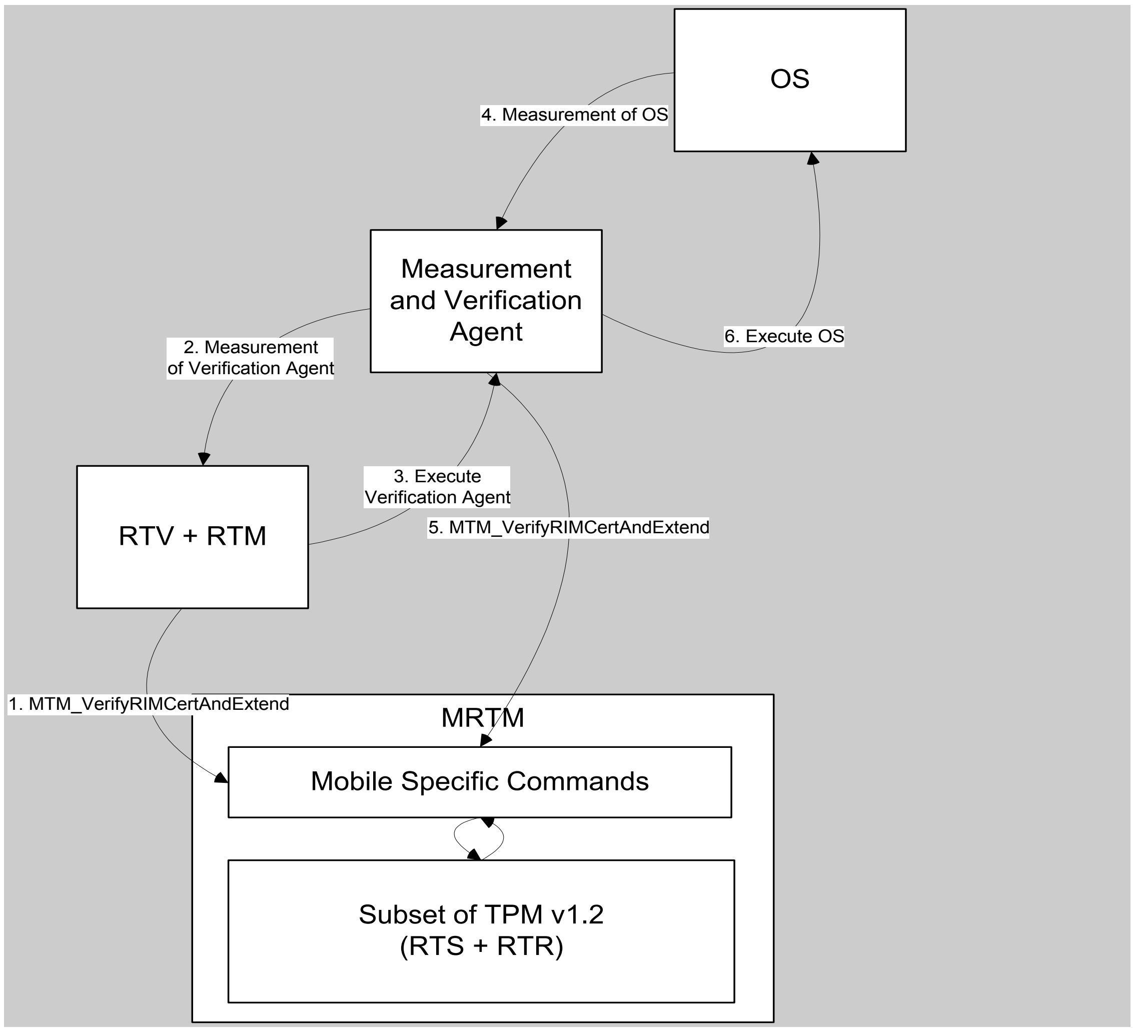

OS
4. Measurement of OS
Measurement and Verification Agent
2. Measurement of Verification Agent
6. Execute OS
RTV + RTM
3. Execute Verification Agent
5. MTM_VerifyRIMCertAndExtend
1. MTM_VerifyRIMCertAndExtend
MRTM
Mobile Specific Commands
Subset of TPM v1.2 (RTS + RTR)

Figure 2.4: Example execution flow of MRTM [2].

The primary purpose of creating four separate mTPM instances is to provide an isolated EE that is capable of performing required operations outside of the influence of other stakeholders. In addition to executing in an isolated environment, the mTPM also provides the ability to report the current state of each mTPM module. Stakeholders are therefore able to determine the current state of their respective modules on the ME and decide if their modules are in a state conducive to the operations they need to perform. Such operations might include firmware/software upgrades, network authentication, and e-Wallet transactions.

To better understand how an mTPM module would function, consider the example MRTM execution flow shown in Figure 2.4 [2]. The MRTM is based upon a subset of the TPM v1.2 specification, which includes a Root-of-Trust-for-Storage (RTS) and Root-of-Trust-for-Reporting (RTR) element. A Root-of-Trust (RT) basically consists of two things: a subset of trusted code along with an initialization value for a corresponding PCR. Each RT may be established with the same code, such as the code contained in the SecureROM, but each should have a unique initialization value for their respective PCR. Once the trusted code initializes the PCRs, it should then verify the integrity of any subsequent code that will run as part of the corresponding RT. Once this is done, the RT can validate its stated based upon the current value of the PCR, and can also perform any supported operations.

Added to these two RTs are the Root-of-Trust-for-Verification (RTV) and Root-of-Trust-for-Measurement (RTM). The RTV and RTM are the first two elements to execute on the system and represent the first stage trusted code described in the SB protocol. One of the requirements of this code is to perform a measurement of its execution code, which is then used to extend the PCRs into an initial state. Once this is done, each subsequent EE is measured and verified by the RTV+RTM and corresponding PCRs are extended with each operation. The last step of this process is loading the full operating system (OS), which can then query the MRTM to determine if it booted into a secure state. This is determined by examining the values in one or more PCRs. Because the PCRs are contained within a secure element and can only be indirectly modified via measured and verified code, the boot process is declared secure if the resulting PCR values match previously defined good values.

Using this basic methodology for both MRTM and MLTM modules, a mTPM compatible system would provide a means for multiple stakeholders to each have an isolated measurement and reporting system for their respective collection of code on the ME. To date, there have been no verified instantiations of a mTPM in a consumer ME device. The TCG is currently working to produce the second version of the mTPM specification and has already released a Use Cases document based upon this latest version [29].

2.1.7 Other Organizations

As stated previously, this section was not intended to cover every organization currently engaged in various aspects of mobile security. However, it is also important to mention two other organizations that are highly engaged in specification and requirements development of higher-level security functionality. These two organizations are the Open Mobile Alliance (OMA) [30] and the Third Generation Partnership Project (3GPP) [31].

The OMA is a group composed of various companies across the globe, typically focused in one of four categories: wireless vendors, information technology companies, mobile operators, and application and content providers. When OMA was originally form in June of 2002, there were nearly 200 companies associated with the organization. The primary initiative of this group has been to bring together various “mobile centered” organizations around the world to create unified specifications and services that could be used globally. OMA also creates open standards related to these four focal areas, such as the Multimedia Messaging Service (MMS) and Instant

Messaging (IM), both of which are heavily used on mobile devices today.

3GPP was created in December of 1998 and originally worked with specification development for 3G mobile systems based on the Global System for Mobile communications (GSM) standard. 3GPP has subsequently broadened its target technologies to include Edge, W-CDMA, HSPA, LTE, and LTE-Advanced. Standards developed by 3GPP typically focus around radio, core network, or service architecture. As such, their results are focused more on the use of ME on provider networks, as opposed to the development of the ME. Significant contributions of 3GPP include the Universal Mobile Telecommunications System (UMTS) and the Long Term Evolution (LTE) protocols.

2.2 Hardware Support

Thus far, this paper has present information centered on formal specifications and software support methods for mobile security. While these are certainly critical to the goal of creating a secure mobile platform, the inclusion of hardware support is equally critical. Further, some of the specifications developed, such as the TEE, require hardware features in order to ensure specified capabilities. It is important to note that all information presented on this topic was obtained through publicly available documentation and not through non-disclosure agreements.

To better understand how these hardware features work to enable these capabilities, information will be presented on several of the major security suites provided by chip manufacturers. This information is again provided as an overview; more extensive details will be presented in Chapter 3. It is important to note that implementation details are often confidential and require a non-disclosure agreement (NDA) with the corresponding company before any further information is released.

2.2.1 TrustZone

TrustZone was first released by ARM, Limited, in 2003 and is currently on its 3rd version [19]. TrustZone encompasses a collection of modifications and feature support that provide hardware enforced, software execution isolation on ARM core processors. In order to fully support an isolated EE, ARM updated its instruction set to include methods of switching between environments, as well as made major modifications to their hardware architecture in order to identify data and instructions as secure versus non-secure.

The primary hardware change was the addition of a 33rd bit that was added to the internal AXI bus, as well as all cache and internal memory elements. The value of this bit is used to determine whether the transaction or memory is secure (0) or non-secure (1). This bit is also present in a Current Status Register (CSR) and is used to indicate the current operating state of the system. There are significantly more details about how this functionality works, but this will be presented later in Chapter 3.1.

2.2.2 M-Shield

Texas Instruments (TI) is one of the primary licensees of ARM technology, using ARM based processors as the core of their OMAP processor line. M-Shield is the name of the TI provided security suite that interfaces with ARM TrustZone technology to provide enhanced security functionality. As stated in the TI M-Shield white paper, "...M-Shield mobile security technology is a system-level approach that intimately interleaves optimized hardware and software.." [32]

M-Shield consists primarily of hardware components that are added to the main SoC design, including SecureROM, SecureRAM, AES and public-key accelerators, SecureDMA, and a Secure State Machine. By utilizing TrustZone isolation mechanisms, M-Shield is able to provide these elements to users in both a secure and non-secure manner.

2.2.3 SecureMSM

SecureMSM is a collection of security features offered by Qualcomm that are very similar to those provided by TI's M-Shield technology. Unfortunately, little information is publicly available in with regards to implementation or functional details. Most of the information obtained is high-level and provided through 3^{rd} party developers.

SecureMSM is the name of Qualcomm's security suite for their Snapdragon processors. This suite encompasses in-house hardware security elements, such as secure boot, cryptographic accelerators, secure storage, etc., as well as a third-party TEE kernel called MobiCore [33]. MobiCore was developed by Giesecke & Devrient and was included to provided secure software functionality for financial transactions, like mobile payment and secure PIN entry, as well as key management and user authentication [34] [35]. The MobiCore kernel serves as the TEE and TAs for the Snapdragon platform (and possibly others).

2.3 Academic Research

Because the work being presented here focuses on secure execution on mobile devices through the use of a PUF generated secret key, academic research will be divided into two specific areas. First, research on the development of secure mobile devices will be presented. Second, papers specifically addressing PUF implementations and usage cases will be discussed. Finally, a few of the commercially developed security suites currently available will be discussed.

2.3.1 Mobile Security Research - IAIK

There are currently several groups around the world focused on the development of trusted execution environments for mobile devices. One of the most prominent groups is out of the Institute for Applied Information Processing and Communications (IAIK) at the Graz University of Technology in Austria. The primary authors in this group are Johannes Winter and Kurt Dietrich. Together, these two authors have written several papers since 2007, each describing various aspects of a secure mobile architecture for modern smartphone applications. The proposed architectural elements are based on the incorporation of Mobile Trust Module (MTM) functionality.

As noted by the author, there were three primary implementation possibilities for MTMs at that time. The first was completely in software, but was not feasible because no solid isolation mechanism existed at the time for code execution. (Although TrustZone had been developed by this time, it was not widely used or adopted) The second possibility was a separate IC on the device, which is a highly unlikely option due to the real estate restrictions on modern devices. The final option was to embed the functionality into an existing device or IC on the phone, such as the SIM card. It was also noted by the author that such an implementation would need to be done on an onboard card that is physically connected to the board, thereby reducing the chance of an attacker or thief simply removing or replacing the SIM card.

Moving forward with the use of SIM cards with JavaCard runtime environment support, the author presents a model architecture for a mobile device that could be used to provide MTM functionality in support of Secure Boot (SB) and attestation capabilities. While the idea is solid, the primary concern with this approach is the fact that trusted computing is being performed off chip, thereby opening the possibility of Man-in-the-Middle (MITM) attacks. The optimal architecture would have this functionality on-chip, whether through dedicated silicon or a software-only solution.

Johannes Winter presented a paper the following year that first introduced the concept of using ARM TrustZone to create a trusted execution environment [36]. Winter proposed a secure mobile environment that utilizes TrustZone to support MTM implementations in both a secure and non-secure environment. The architecture consisted of two separate operating systems, one executing inside the secure world and one executing in the non-secure world.

The architecture further supported the use of virtual machines for each world, allowing isolation between multiple processes and environments within each world. Such isolation is required in order to comply with engine execution requirements, as specified in the MTM reference architecture document [28]. The paper also addressed issues concerning virtualization on ARM platforms, interrupt handling between multiple guest systems, and VM creation and deletion.

This paper presented a solid architecture for secure processing, but leaves serious questions about performance hits and/or processing requirements. While most modern processors, including mobile processors, can support running two separate kernels simultaneously, devices that require high performance or deterministic behavior would possibly fail under such an architecture. The author is unable to address this concern due to lack of an actual hardware platform on which to test the proposed architecture. This has become a recurring issue faced by many researchers over the last few years, whether due to lack of "official" support for TrustZone, or inability to modify boot code due to implementations of the secure boot protocol.

Building upon these two papers, Dietrich and Winter released another paper in

2008 [37]. In their paper, the authors presented a software-only MTM architecture enforced via ARM TrustZone technology. The methodology presented makes use of SB to measure and verify the standard kernel image using MTM functionality. Embedded in the kernel, as well as other applications, is a Reference Integrity Metric (RIM) that can be used to validate the respective code. In order to provide this functionality with as little alteration as possible, the authors proposed incorporating RIM certificates inside the “notes” section of ELF binaries.

To prove that code has not been tampered with, the MTM performs a standard set of measure->verify->extend operations on each code element in the SB process. The result of each code segment measurement is compared with the value in the RIM certificate. If that value matches, the associated Platform Configuration Register (PCR) is extended and the boot sequence continues. Once the kernel is loaded and begins execution, the final value in the PCR can be checked to determine if the system booted into a trusted state, based on the final value of the PCRs.

The process described by the authors is effective and is actually quite similar to the process adopted by Apple for distribution of firmware files. Apple uses a proprietary format known as Img3 that uses tags to identify different sections of a firmware file. Each file contains at least these three sections: TYPE, DATA, and CERT. The TYPE section provides information on the type of firmware for which this file is used. The DATA section contains an encrypted copy of the compressed firmware and is the actual code that is run on the device. The CERT section contains an X.507 certificate that can be used to verify the firmware file and ensure it has not been tampered with. If at any stage in the boot process a firmware file does not pass the test, bootup is aborted and the system enters a recovery state where it stays until a restore is performed and valid code is again loaded onto the system. As noted previously, this is very similar to the approach presented by Dietrich and Winter.

These two authors continued their research in 2009 with another paper presenting two design approaches for providing trusted computing building blocks [38]. The first approach was based on a software emulated mobile TPM which used processor extensions to achieve isolation, while the second makes use of onboard smart cards as secure processing elements. These approaches were further discussed in a paper presented the following year [39]. Such approaches were most likely pursued at this point due to the lack of a customizable hardware platform that could be used to test the previously discussed architectures. This difficulty was addressed by Winter and presented in a paper in 2012 [40]. Having determined a hardware platform on which to proceed, it is assumed that follow-on papers will provide implementation results for each of these various architectures.

2.3.2 Mobile Security Research - Others

Dietrich and Winter were not the only two researchers to explore implementations for supporting TEEs or MTM functionality. In a paper by Grossschadl et al. [41], the authors discussed three concerns with the MTM specification that they felt needed to be addressed.

The first concern dealt with the isolation requirements required for proper MTM functionality. The authors made a valid point in stating that a separate chip with dedicated MTM functionality is often not feasible due to cost and/or board space. Their recommendation for attending to these concerns is the use of protected execution domains, such as the one provided via the use of ARM TrustZone. The second and third concerns deal with the minimal cryptographic primitives required by the MTM specification. The specification requires at least RSA with 2048-bit keys and SHA-1 hashes, both of which are significantly outdated.

Further, as noted by the authors, there is no defined mechanism for updating exploited or outdated algorithms. However, MTM implementations typically make use of the native functionality provided by the chip manufacturer in order to operate properly, which is part of the reason the MTM specification does not provide a specific design implementation. As such, it should be the responsibility of the chip manufacturer, or the TEE developer, to provide a mechanism for adding or removing cryptographic capabilities. Once that is done, the MTM specification does provide a method for updating its own code, which could then make use of the newly provided features.

Another group from Samsung also presented a collection of papers on the development of a new trusted mobile security architecture [42] [43]. This architecture is also based around the implementation of the MTM specification and details the use of Linux variants that support mandatory access control (MAC) mechanisms, such as Security Enhanced Linux (SELinux). The concept here is the exact same as what was presented by Dietrich and Winter in that such security mechanisms could be used to ensure isolation between multiple processes in the same operating system, thereby ensuring each stakeholder's MTMs are appropriately isolated from one another.

Although the authors present a novel approach, which was likely used by Dietrich and Winter in their work, it does not provide any information on how the "secure" kernel is protected other than to state that it must be loaded as part of a secure boot process. While the approach conceivably works for providing isolation between MTM engines, it still resolves down to a software-protecting-software approach if it is not accompanied by any hardware support mechanisms. If the "secure" kernel is not isolated from all other applications on the devices, it is still subject to attack by any installed application. However, considering the time at which the papers were written (2007 and 2008), such mechanisms were not widely available or even publicized. As such, the papers should be considered a novel and meaningful first step towards a secure mobile environment.

A paper presented by SuGil et al. also provides a MTM architecture for mobile devices [44]. The approach taken by the authors was very similar to the approaches already discussed. The primary features supported by this architecture were the measure->verify->extend and remote attestation. The authors also propose the use of RIM certificates for each application and element run on the device, allowing for the analysis of each code element prior to execution.

However, as the authors noted, this does not prevent code injection methods once the code has been stored in memory. Further, a white-list/black-list methodology was used that lacks re-validation capabilities. Once an application is verified, it is added to the white-list; any subsequent loads are not verified because the application would have been added to the white-list. This implies that the application must only pass verification once in order to be considered “good”. The application could be modified in any number of ways after that point, but the architecture would continue to load it based upon the whitepaper. Further, no information is provided on how these lists are maintained or protected.

Researchers have also explored higher-level isolation techniques, such as those presented by Muthukumaran et al. [45]. While their approach mentions dependence upon MTM functionality to ensure secure boot, their primary focus is on the development of secure measurement capabilities inside the kernel. The group discusses several integrity methodologies, such as Biba, LOMAC, Clark-Wilson, Clark-Wilson Lite, Linux Integrity Architecture, and PRIMA. Their final architecture utilized SELinux for MAC enforcement, together with PRIMA support, in order to protect the software installation process. This process is considered to be the primary concern due to its need for interaction with untrusted programs.

Software MTM solutions are not the only implementation being considered. In

2010, a paper by Kim et al. presented a complete MTM hardware design with strong performance results [46]. The solution presented provided the first known implementation of a dedicated chip that was compliant with the MTM specification. Comparisons were made between this device and a USIM chip, largely based on the functional similarities between the two elements.

However, as has been stated previously, performing such operations on a separate chip always broadens the attack space of the overall device. If the interface bus, which in this case is Inter-Integrated Circuit (I2C), can be monitored, input and output values are capable of being spoofed. Because the authors were primarily focused with minimizing power consumption requirements, it is unlikely that communications into and out of the chip are encrypted or otherwise obfuscated.

Despite addressing a serious security issue, the design does so by adding another security concern. A complete security suite must perform all security related operations on-chip, rather than relying on off-chip to provide critical information and results. Most modern SoC devices provide an internal cryptographic acceleration unit, making it the ideal element for performing such operations.

As claimed previously in Section 1.3, the best method for maintaining critical data elements, such as keys, known-good-results, and certificates, is through encryption with a unique-per-device key that is not stored in any form of non-volatile storage. Further, the most secure method for implementing secure boot, secure storage, MTMs, or any other secure service is via trusted software interfacing with internal cryptographic units. The best-known method for accomplishing this is through the use of a PUF generated secret key, as well as a Public/Private key pair. To better understand how this can be accomplished, information of PUF research will now be presented.

2.3.3 Physically Unclonable Function Research

The concept of a Physically Unclonable Functions (PUFs) where originally proposed in 1983 by D.W. Bauder at Sandia National Laboratory as a method for identifying counterfeiting, or alteration, of a chip design [47]. PUFs continued to be explored as a solution to such issues but soon found an additional use: random number generation. PUFs provide a path through a collection of decision nodes that result in a 0 or 1 value based upon the manufacturing characteristics of the device. A PUF is provided with a challenge, which represents a collection of decision values for each of the decision nodes, and produces a set of responses. Because of variations in the manufacturing process, each device will produce different responses to the same challenge.

As a result, many researchers have claimed that the "device unique" values generated by a PUF could be used as keys for cryptographic operations. This was first proposed in 2004 by Lee et al. [14], and then in 2007 by Suh and Devadas [15]. Their goal was to generate a key that could be used to provide device authentication. Their research provided several methods for attempting to stabilize the results of a PUF in order to provide consistent and repeatable results that could be used to generate a cryptographic key. Once stabilized, the authors proposed the ability to use this key in device authentication mechanism, such as IC identification. They also proposed the use of the data as a seed for a random number generator, as well as using the data as a key that could be tightly coupled with the processor in order to enable a physically secure processor. The inherent weakness of this approach is that if the key is successfully tied to the processor, an attacker may gain the ability to execute on the processor and then read back the resulting key.

Further, the authors describe using a generated key in collaboration with a set of challenges to provide an authentication mechanism. The process would require a trusted party to record responses produced by the IC to the provided challenges. These responses would be maintained in a database, allowing the trusted party to later send a challenge and compare the provided response with the one on record, thereby proving the identify of the device. However, as stated by the author, challenges are never reused, an idea meant to alleviate concerns with MITM attacks. Such a constraint though would either place an unnecessary constraint on the number of authentications possible, require a significant amount of storage for challenge/response pair per device, or necessitate a update policy in which the device much be taken back by the trusted party in order to create new sets of challenge/response pairs. This also begs the question of what would prohibit an attacker from stealing the device and running these tests on their own? While the attacker would not likely be able to generate or store all possible challenge/response pairs, they could simply re-invoke the authentication mechanism until they were provided with a challenge that they had a valid response for, thereby allowing them to spoof the actual device.

Finally, the authors presented information on using PUFs to further generate run-time cryptographic keys that could be used for standard cryptographic operations. The results produced by a PUF can at times change, based upon current physiological conditions, such as voltage and temperature. This characteristic, as mentioned by the authors, allows a PUF to be used as a random number generator (RNG) that could produce unique keys for cryptographic operations at any time. Some cryptographic operations might require a stable key, therefore obliging the developer to include error correction code (ECC), while others may only be able to use keys with specific characteristics, such as RSA. The authors do present satisfactory research into both of these areas, but do not provide further information about how these keys are accessed or used on the platform. This is a topic that will be addressed in the next two chapters.

The concept of using PUFs to authenticate an entire system, rather than just a single device, was presented by Ibrahim and Nair [48]. The authors proposed the use of a trusted party to maintain challenge/response pairs, similar to that of Suh and Devadas. Various PUF capable elements on the device are provided with a challenge, and then provide their responses to a reader in the system. This reader then aggregates all the responses received in order to generate a single system response. This response can then be used to verify the entire system, as well as isolate which element generated an invalid response if the overall system response was incorrect.

While this idea is certainly feasible and provides a strong status mechanism for the system, it is still limited by the same problems mentioned previously for Suh and Devadas. Further, the architecture presented by the authors lists wireless transmission mechanisms for transmission of PUF responses, and touts the fact that all processing is performed off-chip. As a result, there is no mechanism for proving that the IC inside the device actually generated the response. The original chip could be removed, replaced by another, and then powered up at a nearby location. Because all challenge/responses are conducted wirelessly, the chip can still provide the necessary responses without even being on the device, thereby presenting a false-positive response for a valid operating state. Again, the concept of what was present is solid, but the implementation mechanism is subject to a variety of attack methods.

Ibrahim and Nair also discussed the ability to generate a key and use it to aid in the development of a Cyber Physical System, or CPS [48]. To this end, the authors discussed the ability to incorporate multiple PUFs into a system in order to create a paradigm called security fusion. Although details of how this new paradigm would permeate into software were not included, the authors presented a system architecture wherein multiple elements, equipped with their own PUFs, would communicate with a reader to provide results for generation of a system response. The collated results from each of the elements would be compared with a known result to ensure each element is functioning properly. A match would indicate that all systems are verified, while an incorrect value would cause the system to track down and identify the malfunctioning element. This approach does seem feasible for helping to ensure overall hardware security, but as it currently stands does nothing to ensure secure execution of software.

The usage of PUF results in cryptographic applications has also been discussed. To date, most of the applied uses involve using PUF generated results with RFID elements [49] [50]. While this is a valid and useful application for PUFs, our primary concern is the security of the TEE executing on the processor, an approach that to date has not been formally defined by any research groups.

A critical aspect of using PUFs to provide cryptographic information of identification is dependent upon the ability to provide stable, repeatable results. As such, a variety of groups began looking into methods of stabilizing the output generated by a PUF. Such research included the ECC scheme presented previously by Suh and Devadas [15], as well as the use of a SRAM-PUF by Bohm et al. [51], 2D Hamming codes [52], XOR masking [53], longest increasing subsequence algorithm (LISA) [54], majority voting [55], and soft decision decoders [56].

The primary security concern generated by these proposed solutions is the requirement to store the syndrome, or helper data, about the PUF publicly. This allows an attacker to ascertain a certain amount of information about the PUF simply by observing these data. This problem was specifically addressed by [57]. A second method was also proposed by Paral and Davadas that use pattern matching to regenerate keys, but uses indices into sets of PUF challenges as secret key bits, rather than the typical approach which simply uses the PUF response bits as the actual secret key [58].

Other security concerns also cropped up in some of these approaches. The SRAM PUF [51] is a prime example. As the authors noted, the ability of a software application to read the contents of the SRAM would prove critical to the security of the generated key. While the ability to use the SRAM to create a stabilized key may be a viable method, from a security standpoint it would be difficult to ensure that no attacker was ever able to gain execution privileges or code access to the SecureROM. A prominent example of such an attack is the A4 bootROM exploit developed by GeoHot [59].

2.4 Commercial Implementations

Now beginning to realize the vast market currently available in the mobile world, companies have also begun creating "secure" execution environments for consumer devices. These companies, like the academic researchers, are attempting to address the need for security during critical mobile transactions, such as banking and NFC. Although most major banking companies will likely wait for a definitive standard before committing to full entry into the mobile world, that is not stopping security companies from touting the high security services provided by their solutions. Further, if their solutions can align themselves with the current trajectory of many of the specifications presented previously, adaptation once the specification is approved should be achieved relatively quickly.

OKLabs out of Australia was one of the first mobile security companies to present a security solution. Their approach involves the use of virtualization to provide isolation between various processes running on the system [60] [61] [62]. OKLabs also heavily advertises the use of a microkernel in conjunction with virtualization as a method for minimizing the overhead of multiple simultaneous OS instances [63] [64] [65]. To prove the feasibility of their approach, OK Labs presented a case study of their technology on the Motorola Evoke QA4 device [66]. While their work is impressive, it is yet again another example of software protecting software.

Further, the attack space on a software application that attempts to virtualize all the underlying hardware is enormous in comparison to two execution environments with hardware-enforced control of access privileges.

Another company that has been diligently working in the mobile security area is Trusted Logic in France. Trusted Logic boasts a security suite called Trusted Foundations [67] that provides a software API that interfaces with on-chip security peripherals, such as cryptographic engines. This software suite has been incorporated into variants of the TI OMAP 4 processor family [68]. Further, the financial firm Gemalto has even worked with TI to create a secure banking product that can be run on OMAP processors equipped with the Trusted Foundations software [69]. The fact that Trusted Logic has been able to garner such support for its product just further illustrates that companies are very interested in utilizing this type of technology.

The last company that will be discussed is Giesecke & Devrient out of Germany. This company developed the MobiCore security suite that was mentioned as part of the Qualcomm SecureMSM solution in Section 2.2.3. MobiCore is a secure OS that makes use of ARM TrustZone technology to create a TEE [34]. G&D have lauded this new technology for providing DRM protection [70] [71], DRM for mobile [72], virtual private networking (VPN) [73], and even secure smartphone applications [35]. MobiCore got a big lift when it was announced that it would be incorporated in the eagerly anticipated Samsung Galaxy S III devices [74]. The incorporation of this technology allowed Samsung to begin using the NFC communications provided with the Google Android OS.

As can be seen from the vast amount of research and development that is currently being conducted, mobile security is an extremely popular and prized area of interest. While all of these solutions are cropping up for securing execution on mobile devices, the primary question of interest should be, to paraphrase Lois Lane, "who's got you?" It can be seen how all of these standards and technologies provide protection for applications and providing secure execution, but how are they being protected when they are not running? Is the secure boot process sufficient to protect these executing environments and prove that everything is running properly? In order to address these questions, it is necessary to better understand what ARM TrustZone is and exactly how it provides the means for creating a trusted execution environment. Further, how can ARM TrustZone work with a TEE to control access to critical data and properly authenticate a device?

Chapter 3

TrustZone and Trusted Execution Environments

In 2003, ARM released the first version of a new technology suite called TrustZone [3]. In addition to the hardware features provided, ARM developed a TZ Application Programmer Interface (API) that could be used by software developers to interact with the underlying hardware features. This API, currently at version 3.0, became the basis for the Trusted Execution Environment (TEE) specifications [19]. It is important to note that TZ is not a software security implementation; it is a collection of hardware extensions to the ARM core architecture that support software isolation. The provided API is a suggested framework and can be altered or replaced by any software developer.

Using TZ to provide software isolation has allowed for the creation of a variety of mobile security solutions and specifications, such as TEEs. The question becomes whether or not this functionality is capable of addressing the issues of security for system critical resources and authentication. This question is addressed in this chapter, as greater details about TrustZone and Trusted Execution Environments

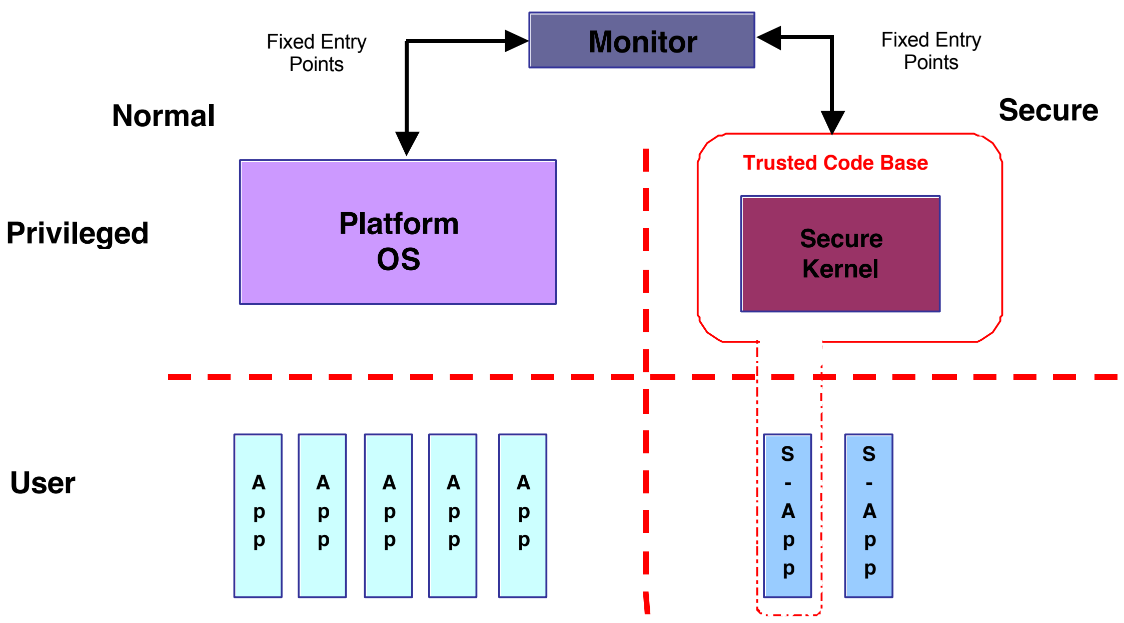


Figure 3.1: Security structure of the original TrustZone technology [3].

are presented.

3.1 TrustZone Architecture

When ARM initially presented TrustZone to the electronics community at large [3] [75], the primary new architectural feature they touted was the inclusion of a 33rd address/data bit known as the “security” or “S-bit”. This bit serves a single purpose: identify the address, data value, or system state as either secure or non-secure.

The value of this bit is controlled via a special software element known as the “Monitor”. The monitor runs in a special operation mode (Monitor mode) and is in charge of handling the transition between the secure and non-secure worlds, as well as intercepting interrupts and forwarding them to their corresponding world. The Monitor can only be entered by way of a fixed set of entry points, thereby reducing the attack surface through which attackers can inject data. This approach allows testing of the interface to become a somewhat practical idea, which in turn allows the idea of a secure system to reemerge. The layout of this design is shown in Fig. 3.1.

In addition to being used within the ARM core itself, the S-bit has also been incorporated into the Advanced Microcontroller Bus Architecture (AMBA) bus. This bus served as the primary bus at the time TZ was originally developed and was used to connect peripherals to the microprocessor. The inclusion of the S-bit allows designers to establish peripheral devices and their coinciding transactions as secure or non-secure. Developers of external peripherals only need to add a small wrapper to their design that is capable of reading this bit and handling it accordingly. In most cases this allows the majority of hardware designs to remain unaltered.

The S-bit has also been merged into the cache and Memory Management Unit (MMU). Doing so allows data and instructions fetched from memory to remain in the cache regardless of the current state of the system. One of the biggest performance hits of this style of segregation is the flushing of memory locations when the processor moves between states. Ordinarily failure to perform such a flush would pose a potential security risk as the normal world might be able to analyze what the secure world was doing and possibly even retrieve secure data still located in the cache. By identifying each memory address as containing a secure or non-secure value, flushes are no longer needed because the processor can enforce access controls on the data based upon the current state.

Since TZ was initially created in 2003, the S-bit has continued to be a critical element of the overall design. It is difficult to determine exactly what changes have occurred since the original TZ architecture was released, as documentation of earlier versions is no longer available. However, recent documentation emphasizes several features not discussed in the original white paper provided by ARM.

As depicted in a paper in 2009 [4], such features include:

1. AMBA3 Advanced eXtensible Interface (AXI) bus support
2. The Secure Configuration Register (SCR) in the system control CoProcessor 15 (CP15)
3. The Secure Monitor Call (SMC) instruction

While there are also other important and noteworthy components of TZ, such as secure debugging capabilities, only the portions that relate directly to the research being presented will be covered. The next three sections will discuss each of these in greater detail.

3.1.1 TrustZone AXI Support

In order to best understand the changes made to the TZ-compatible devices, consider the generic mobile architecture presented in Fig. 3.2. The ARM processor core is only a small piece of the overall device architecture, and as such must maintain a mechanism for identifying peripherals and transactions as secure or non-secure. By incorporating support for TZ into the AXI bus via the inclusion of the S-bit, TZ aware peripherals can be properly configured and act accordingly based upon the state of each transaction.

It is important to note that not every element in this architecture must be TZ-aware. For instance, by making only the MMU TZ aware, other memory peripherals, such as the Boot ROM, SRAM, DRAM, and Flash, can be automatically protected by requiring all memory accesses to be controlled by the MMU. Further, some elements may not even need to be included. Which elements are and are not included is dictated by the system designer and is often based upon what is needed by the

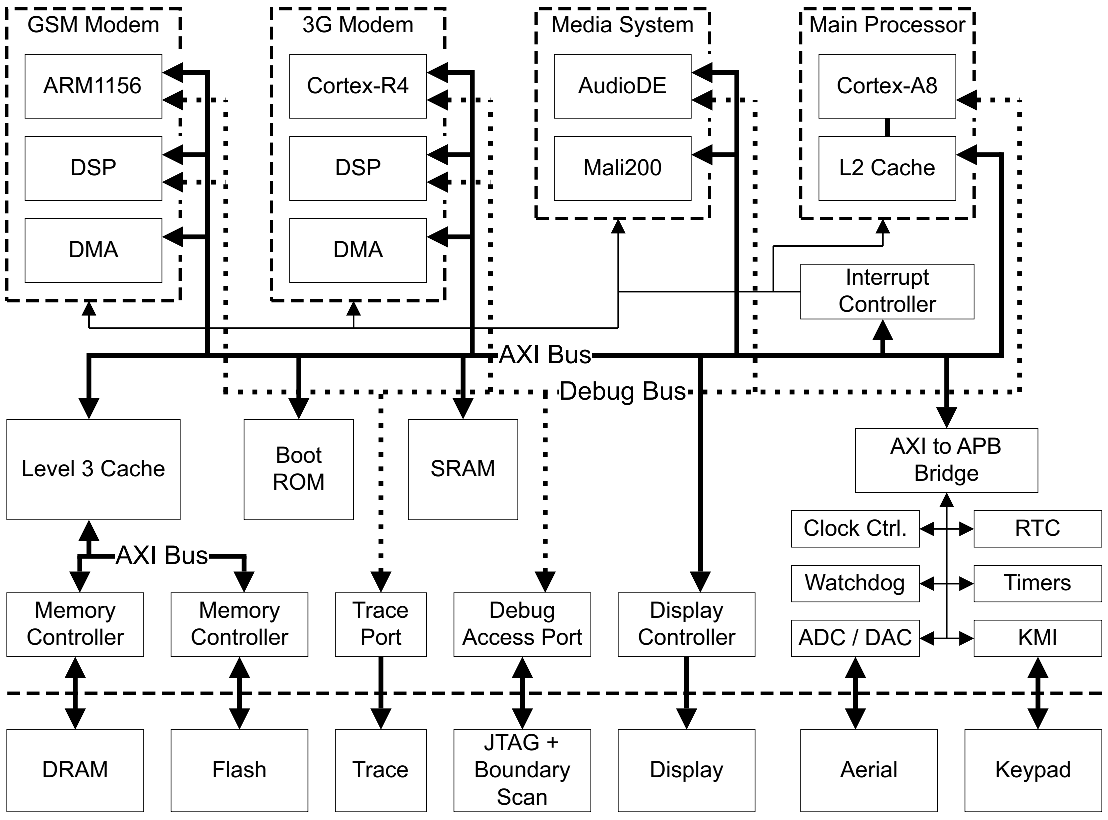


Figure 3.2: Generic mobile device architecture with ARM processor core [4].

architecture, rather than simply including everything that is available. A designer may want one timer to be TZ-aware, but may leave another as generally accessible. This is illustrated in Fig 3.3, an example architecture showing how TZ support may be partitioned throughout a mobile architecture design.

3.1.2 TrustZone Secure Configuration Register

The Secure Configuration Register, or SCR, is used to maintain information about the TrustZone security settings currently being used. While the specific bits and their associated purposes may vary from one architecture to another, the least significant

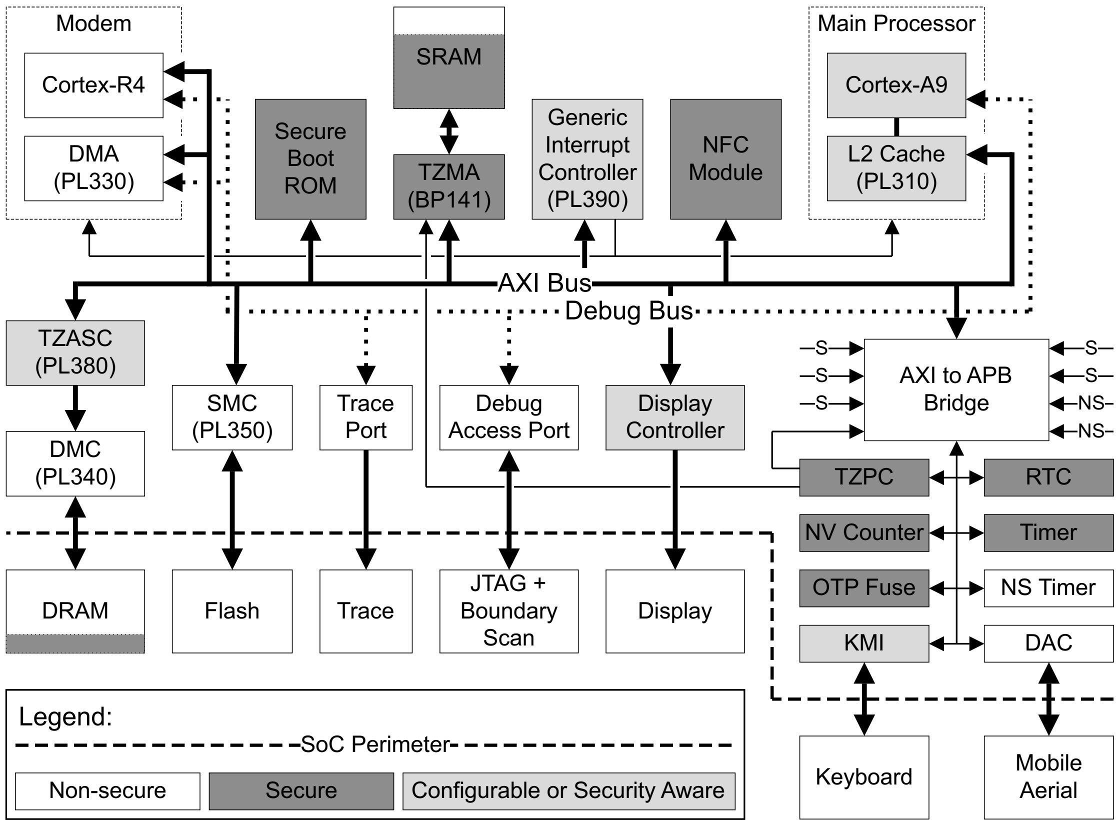


Figure 3.3: Example mobile architecture supporting ARM TrustZone [4].

bit is always used to identify the current TZ state of the system with "0" representing secure and "1" representing non-secure. Although the SCR is a 32-bit register, only 8-10 bits are typically ever used. These bits are usually used to dictate how the system responds to specific actions, such as an IRQ or FIQ interrupt, and what functionality is present while in the secure and non-secure worlds.

For example, the ARMv7 architecture uses 10 bits of the SCR [76]. One of these bits, called the F-bit Writable (FW) bit, is used to control access to the F-bit of the Current Program Status Register (CPSR). The F-bit of the CPSR is used to enable masking of the Fast Interrupt (FIQ). By disabling the ability of the non-secure world to mask this interrupt, the secure world can establish a timer that triggers on a regular interval and generates an FIQ exception, thereby assuring that the secure world executes on a deterministic period. This is the most common approach used and is recommend by ARM as the defacto method for preemptive execution of the TEE.

Access to this register is also restricted to software running in the “supervisor” mode within the secure world. For systems that implement a full-fledged kernel within the secure world, this provides access restrictions that the kernel can utilize to ensure individual applications running within the TEE cannot change these values. While it would be assumed that any application executing inside the secure world has been proven “trusted”, placing all control into the hands of the trusted kernel provides a small amount of added security.

3.1.3 TrustZone Secure Monitor Call

The Secure Monitor Call (SMC) instruction, formerly known as the Secure Monitor Interrupt (SMI), is a part of the security extensions added to the ARM core. The intended use of this function it to provide a mechanism through which the non-secure world can call into the Monitor, thereby invoking a switch to the secure world. This capability is controlled through the SCR discussed previously. The execution of the SMC instruction will either result in the normal execution of the SMC or will be treated as an “UNDEFINED” instruction and generate the corresponding exception, depending upon how its functionality is defined in the by the secure world.

This instruction is the primary mechanism used by drivers developed within the REE, or more specifically the REE communications agent. Rather than forcing the system to wait until the TEE next executes, the SMC provides a mechanism by which the REE can call into the TEE for access to trusted peripherals or for execution of trusted applications. Once the secure world completes the requested operation, the

Monitor will restore execution to the REE, one instruction after the SMC instruction.

3.1.4 TrustZone Memory Support

Memory support in a TZ architecture is significantly different than what is found in most modern CPU architectures. This is primarily due to the support requirements inherent to implementing two isolated execution environments. As discussed previously, flushing of the internal cache, instruction pipelines, and Translation Lookaside Buffers (TLBs) can result in significant performance hits. To address these concerns, ARM decided to integrate the S-bit directly into the internal level one cache of its processor cores.

The internal memory structure in an ARM core consists of two primary elements, a TLB for maintaining virtual-to-physical mappings, and the level one (L1) cache area. Using the standard cache methodology, instructions and data are prefetched into the L1 for access by the processor. When the processor requests this information, it includes a marker called the Non-Secure (NS) bit. The NS-bit is another name for the S-bit and represents the value of the NS-bit in the CPSR at the time of the access. Each instruction and data value fetched is correspondingly labeled when it arrives in the cache.

The virtual address that is used by the processor to request a specific piece of data also includes an additional bit, called the Non-Secure Table Identifier (NSTID). This bit does not indicate the state of the system when the request is made, but rather identifies the state to which the memory belongs. It is important to note that in TZ architectures, it is possible for the secure world and the non-secure world to share memory, as well as for the secure world to provide introspection of non-secure memory. Therefore, even if the processor is currently executing in the secure world, it may wish to retrieve data values from the non-secure world memory. In this case, the NSTID bit would be set to "1", indicating that the memory address is to be retrieved from the non-secure world memory map even though the current value of the NS-bit in the SCR is "0". The TLB would then contain a mapping to the appropriate physical address and compare the current state with the NS bit for that entry. As long as the values match, or the current state is secure, access should be granted. Otherwise, an exception is raised.

Other memory devices on the system typically act in a similar fashion, but usually require some type of TZ interface logic to maintain the NS bits for each address or block of addresses, rather than requiring incorporation of this bit into the chip fabric.

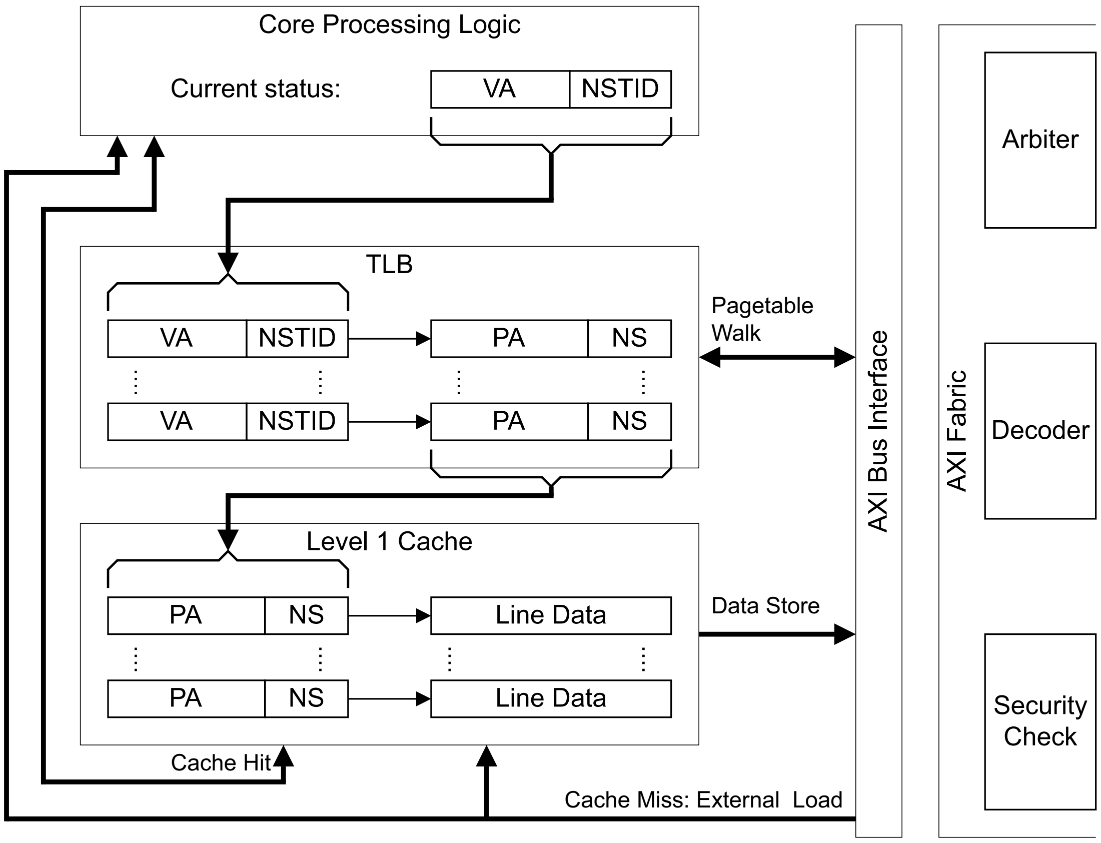


Figure 3.4: Address translation in TrustZone aware level one cache [4].

An example of this was shown in Fig. 3.3. This figure contains a module called the TrustZone Address Space Controller, or TZASC. Such a module can be placed in-between the main processor and the memory element, thereby allowing the memory element to remain unchanged. The purpose of the TZASC is to monitor transactions to the memory and determine access permissions based upon the value of the NS bit. The TZASC must maintain partition information in the event physical memory is partitioned between secure and non-secure memory, again allowing for the co-existence of mixed memory elements. To reduce storage requirements for security settings, memory is usually identified as secure or non-secure in variably sized pages, rather than individual byte addresses.

3.1.5 TrustZone Interrupt Handling

Interrupt handling is the last major change required by the TZ architecture. Interrupts are used to indicate to the processor that some external peripheral needs to communicate with the processor. Based upon how this peripheral has been set up, such a request could indicate the need to transfer non-secure or secure information. It is therefore necessary for the processor to determine which type of interrupt has been generated and whether a change in the execution state is necessary in order to service the interrupt.

To handle this type of situation, ARM has advocated the use of multiple Exception Vector Tables (EVTs), used to maintain the address of interrupt service routines. Such an implementation is shown in Fig. 3.5. As illustrated, the ARM architecture supports seven possible interrupts. In order to firewall the transfer of information between the non-secure and secure world, it is necessary to keep separate EVTs for each world, plus one additional EVT for the Monitor. Once an interrupt occurs, the current state of the processor is compared with the state of the interrupt. If they are the same, the interrupt is simply forwarded on to the appropriate execution environ-

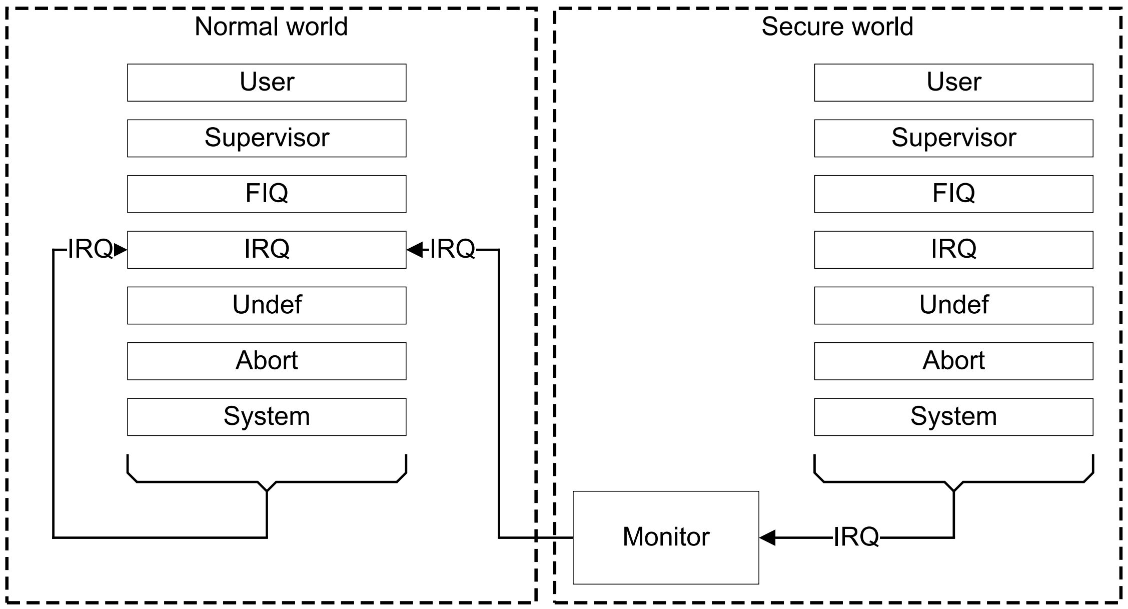


Figure 3.5: Example interrupt table structure for TrustZone architectures [4].

ment. If not, execution is transferred to the Monitor, which determines the correct state and performs the required transition. Isolation of the EVTs can be done in two ways: separate interrupt controllers, or the use of a single TZ-aware interrupt controller, such as the PL390 shown in Fig. 3.3.

Trapping interrupts in the Monitor is not possible for all interrupts. Such trapping is only supported for IRQ and FIQ interrupts, configurable in the SCR. Because IRQs are the most common interrupt, it is suggested that this interrupt maintain its normal functionality whenever possible. As mentioned in Sec. 3.1.2, the secure world can setup the FIQ to trap into the Monitor and then disable masking of this interrupt by the non-secure world. This ensures the ability of the secure world to run and prohibits malicious code running in the non-secure world from preventing execution of the TEE.

3.2 Trusted Execution Environments

The Trusted Execution Environment (TEE) specification was originally started in 2009 after the TZ API was donated to the GlobalPlatform group. The TEE is based on security requirements outlines in the OMTP TR1 document [1] [23], but resembles the TZ examples from an architectural standpoint. The collection of TEE documents released thus far are explicit in affirming that the specifications do not advocate or require any specific hardware implementation, but rather state what functionality is necessary in a supporting architecture.

An initial overview of the TEE specification was presented previously in Sec. 2.1.2. Because the TEE specification is mostly focused on the software implementation, this section will focus predominately on software only. It will be assumed that whatever architecture is used to execute the TEE is fully compatible with hardware requirements, or has full ARM TZ support, which is TEE compliant.

3.2.1 TEE Software Architecture

As stated by the TEE documentation, "the goal of the TEE Software Architecture is to enable Trusted Applications to provide isolated and trustworthy capabilities for service providers, which can then be used through intermediary Client Applications" [1]. This is illustrated in the previously presented TEE reference software architecture, shown again Fig. 3.6.

Based upon this design concept, the entire execution environment and device peripherals can be divided into two groups: public (non-secure) and trusted (secure). The Rich-OS Execution Environment (REE) consists of the Rich-OS, as well as all untrusted applications and public peripherals. The Trusted Execution Environment consists of some primary program, perhaps a full-fledged kernel, with trusted appli-

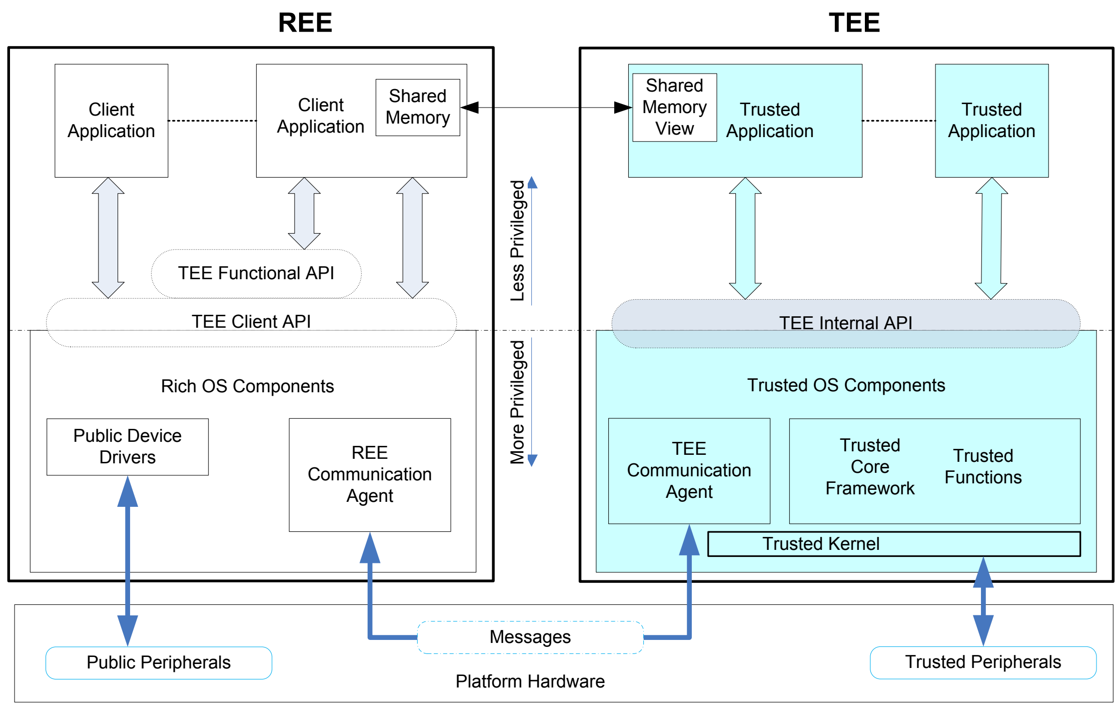


Figure 3.6: TEE reference software architecture [1].

cations and trusted peripherals. Any peripheral available to both environments is considered public. Additionally, in order to help facilitate communications between the two worlds, shared memory locations can be established for the transfer of data.

The REE consists of three primary elements: TEE Functional Application Programming Interface (API), TEE Client API, and the REE Communications Agent. In most cases the communications agent will consist of any required kernel drivers that will populate communication structures for message and data passing between the REE and TEE, as well as the invocation of SMC instructions where possible. The TEE Client API is used to facilitate communications between the REE and the TEE by making appropriate calls to the communications agent. The TEE Functional API sits at the highest level and is meant to provide a set of Rich-OS friendly APIs that can utilize the Client API and communications agent to access TEE applications.

A communications agent also exists on the TEE side. The communications agent is responsible for interpreting requests from the REE and forwarding the request to the corresponding Trusted Application (TA) or Trusted Kernel. The TEE Internal API "defines a set of C APIs for the development of Trusted Applications running inside a Trusted Execution Environment." [21]. In addition, the API provides access to the trusted peripherals via interactions with the Trusted OS Components, also knows as the "hosting code".

When a Client Application (CA) needs to communicate with a TA, a session is opened. The session begins by first invoking an instance of the TA. Each session involves a unique instance of the TA, each with its own independent physical memory space. Once the instance is created, commands and data can be exchanged between the CA and TA, either through the communications agents, or via a shared memory location. Once the transaction completes, the TA instance is terminated and execution resumes.

3.3 Security and Customization of Trusted Execution Environments

At the time this paper was written, GP was still working on the development of additional TEE documents, including information on the lifecycle of TAs. Some of the primary issues in this regard are: "What services are available within the TEE?", "How long do TAs stay on the device?", and "How are new services uploaded to the device?" Additionally, "How are updates of the TEE handled?"

TrustZone provides a mechanism for implementing and support TEEs, but it provides no inherent mechanism for upgrading or customizing such an implementation. The Secure Boot (SB) specification discussed previously requires measurement and verification of every software element used in the boot process prior to its execution. Storage of "known-good" values used for comparison is often located on the device, and a TZ-TEE combination could be used to protect these values.

But what happens if a device is compromised? Firmware files are almost always unique per platform, not per device. Leakage of a key, certificate, hash, or other value, may result in the compromise of a single system initially, but because such information is not unique, a leakage on one device results in a leakage to all such devices. Firmware files could be decrypted, modified, encrypted, and then written to any other device with no way of knowing that such files are not legitimate.

Addressing such concerns requires the ability to secure this type of information in a unique manner on each device, yet still provide a mechanism for SB. Not many companies are going to want to independently encrypt the firmware for each device. Such an approach would not only be time consuming, but would also require the company to know the secure key for each device in order to encrypt it prior to uploading or burning the information into the device. This would necessitate storage and resource consumption that not only cost time, but also would require a significant financial investment. The maintenance of this information would also provide a huge financial liability because loss of this information would prohibit a user from updating his or her device, thereby exposing the company to potential lawsuits. Further, this method would still require a key to be burned into the device, or stored in some form of non-volatile, secure memory. Thus, there is still no mechanism for updating the key in the case of compromise.

In order to develop a mobile security solution that is capable of addressing all of the issues presented thus far, research needed to be conducted into each of the following areas:

1. Generation of unique-per-device secret key

2. Storage of unique-per-device secret key
3. Protection of TEE in storage
4. Protection of TEE measurement/verification values
5. Update/modification mechanism for TEE

In Sec. 1.1, a collection of concerns were presented that are all a direct result of these types of security holes. Properly merging and applying a set of solutions from various mobile platform can generate a new solution that is capable of alleviating such concerns by providing a mechanism to handle each of these areas. Additionally, this solution is also capable of addressing the need for a device-unique authentication method, as discussed in Sec. 1.3.

The solution presented in this research is based upon the use of a Physically Unclonable Function (PUF) to generate a "unique-per-device" secret key, as well as a random number to be used as a seed for generation of a RSA public/private key pair. These three keys can be reproduced at boot time, thereby eliminating the need to store them in any form of nonvolatile storage. Once generated, they can also be used to protect system critical information in a unique manner without altering or diminishing any of the original protection mechanisms. A detailed explanation of this approach, as well as a comprehensive view of its internal structure, will be presented in Chapter 4.

Chapter 4

ECE Architecture

As stated previously, the purpose of this research is an analysis of current mobile security mechanisms and a determination of how PUF technology can be leveraged to provide enhanced data security and device authentication on mobile platforms. A variety of implementations and methodologies have been presented that have attempted to address many of the security issues that the mobile world faces today. But in order for those mechanisms to keep a system safe, they must themselves be kept safe from attacks that would seek to bypass or disable their functionality.

Currently, most systems rely upon cryptography to protect sensitive information stored on a device, such as trusted code used during the secure boot process. Symmetric encryption is typically used for protecting data, and SoC manufacturers have even begun providing locations for "secure" storage of cryptographic keys with the device [32] [33]. Such locations include secure ROMs and electronically programmable eFuse arrays, as well as off chip devices, like smart cards and SIM cards. Based upon the increased use of these elements in a growing number of applications, it would be difficult to make a claim that mobile security has not been enhanced by their incorporation into designs. However, that does not imply that the security of these architectures cannot be improved upon, or that their security is infallible.

The primary security risk inherent to storing cryptographic information on these devices is the potential for such value to be leaked or otherwise exposed. Since these keys are often used to protect security critical elements, access to the key or its value would allow for the unauthorized modification of such elements on any affected device. The risk to devices is further compounded by the potential for key values to be shared across multiple devices.

Manufacturers occasionally send firmware updates to devices in order address software bugs and security issues. These files are usually encrypted and/or digitally signed. Rather than creating a unique version of the firmware file for every device manufactured, it is more logical that a manufacturer would use a single secret key or public/private key pair to perform cryptographic operations on these files. In order for each device to able to verify the authenticity of the firmware files, the corresponding key(s) must be stored in some form of non-volatile storage on the device. Additionally, this key would be the same on all supporting devices.

In cases such as this, leakage of the key or information about the key can have a much broader impact. Once an attacker has been able to obtain the necessary information from one device, that information can be used in attacks against all other devices that utilize the same key(s). Because there have been no recorded incidents of leakage of such keys, it is logical to assume that the odds of such a leak or exposure are small. It is, nonetheless, an unnecessary risk.

In addition to concerns about the security of the system if the key is every exposed, theft of mobile devices is also a concern that needs to be addressed. To protect against theft, there must exist a mechanism for accurately identifying mobile equipment and associating this equipment with a mobile user. The reason for this is that it is often possible for a thief to steal a device and then active it for use on another cellular account. The International Mobile Equipment Identifier, or IMEI, was developed specifically for the purposes of uniquely identifying mobile devices.

Because IMEI numbers are supposed to be unique per device, most manufacturers store the IMEI in a one-time programmable, non-volatile memory location, such as an eFuse array. The use of an eFuse array provides manufacturers with an easy method for setting this information on a per device basis without having to modify the design of every device produced. What this means though is that the value can be changed by anyone with the necessary equipment to blow the fuses, though changes can only be made to previously un-blown fuses. Since the eFuse array value is frequently read and returned via software, an exploited device can be programmed to return any arbitrary value, rather than the valid IMEI number programmed by the manufacturer.

Considering the security issues presented thus far, it is easy to see that a better system is needed if any progress is to be made in the field of mobile security. In this chapter, an architecture for mobile devices is presented that addresses many of the security issues discussed here and in previous chapters. Specifically, this architecture utilizes a PUF to generate unique, per device values that may be used for cryptographic operations and device identification. It is proposed that by integrating PUF technology into a cryptographic processor in an isolated environment within the SoC, secure cryptographic key generation is possible and that the results will be unique for every supporting device. Further, it is proposed that a software Application Programming Interface (API) operating inside of a Trusted Execution Environment can be used to provide the required functionality in the most secure manner possible. By combining these two architectural features, a new mobile security framework can be developed that will greatly increase the overall security of the device, as well as providing stronger mechanisms for device authentication and protection of security critical code and data.

4.1 PUF Enhanced Cryptographic Engine

Of all the issues that have been discussed thus far, some of the primary questions that could be asked are:

- "Why is so much of the security analysis done on the CPU, where every attacker yearns to live?",
- "Why are these security critical elements accessible by the processor, and thus by the associated OS?",
- "How can the processor be trusted to accurately identify itself if unaltered execution of the corresponding code cannot be guaranteed?",
- "Would moving this functionality to a more secure location to which attackers don't have access fix the problem?", and
- "If so, is creation of such an environment possible?"

Section 2.3.3 discussed the capability of generating secret keys, as well as public/private key pairs, through the use of PUFs. Because most modern SoC designers have begun adding cryptographic accelerators to their chips [32] [33], it would be fairly straightforward to simply tie in the response of a PUF challenge directly to the cryptographic unit. Since cryptographic accelerators are considered separate elements in a SoC and are not a part of the processor, access to the results of the PUF challenge can be isolated from the main processor. This method would restrict access to the PUF results to within the cryptographic engine, while still providing the process with the capability to interface with the crypto engine and perform operations that utilize the provided results.

To better illustrate the proposed architecture, consider the generic SoC design depicted in Figure 4.1. This figure shows a very simple SoC architecture consisting

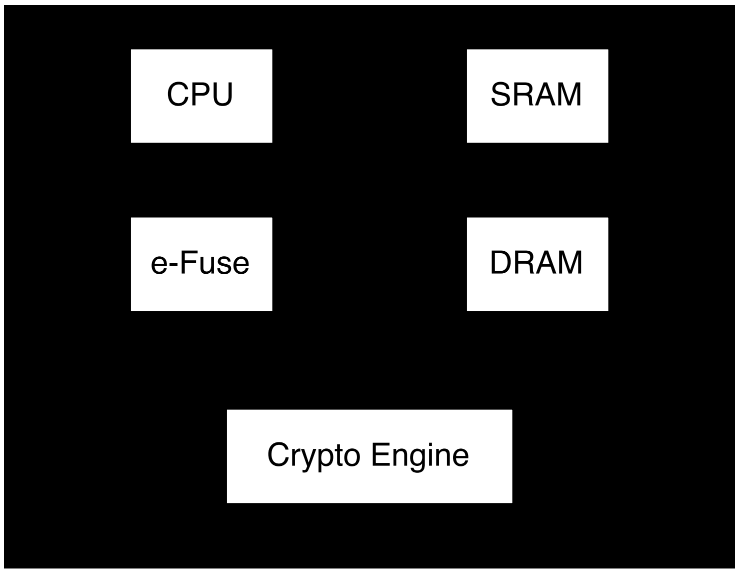


Figure 4.1: Simplified, generic SoC architecture

of a CPU with connections to DRAM, SRAM, eFuse array, and the crypto engine. Figure 4.2 shows what might be contained with the crypto engine. The engine will typically have some form of FIFO/DMA input/output features for moving data through the engine, as well as a small micro-controller and various cryptographic accelerator units.

In order to perform a cryptographic operation, the processor will write information into the control registers for the crypto engine, which are usually memory-mapped into the address space of the processor. These registers will provide information that tells the crypto engine what addresses to access, what type of cryptographic algorithm to use, whether to encrypt or decrypt the data, and a variety of other platform specific information. One of the critical pieces of information is which key to use, and if it is a custom key, where it is located.

In order to make a crypto engine as flexible as possible, designers have typically provided a set of registers that can contain the key to be used, or instead may contain

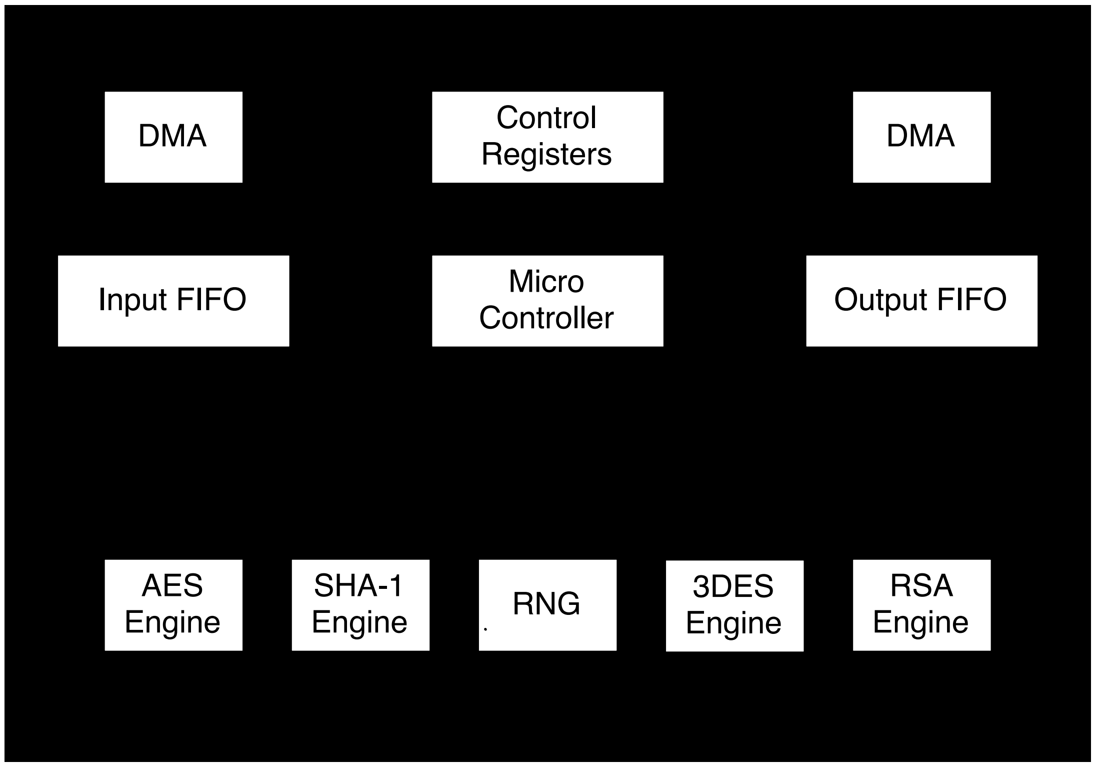


Figure 4.2: Generic cryptographic engine with various cryptographic accelerators

the address of where to find the key. This may not always be the case, and if the same manufacturer that produced the SoC designs the crypto engine, some key addresses could be hard-coded into the engine. How each chip handles this is almost always confidential and can only be obtained through a non-disclosure agreement (NDA), if at all.

It is assumed that most modern SoC architectures store cryptographic keys in the eFuse array, or some other memory-mapped non-volatile memory location. eFuses where initially designed by IBM as a mechanism for enabling or disable functionality within a SoC [77]. Use cases for eFuse arrays continued to grow and quickly found their way into mobile security applications. Motorola is rumored to have used them to block downgrading of the Operating System on Droid devices [78]. Subsequently, chip manufacturers have begun offering eFuse arrays as a location for the secure storage of cryptographic keys [32] [33]. While specifics of which manufacturers are providing this capability and which OS vendors are utilizing it is difficult to determine, it can be determined that this is the approach being pushed by a number of manufacturers.

Unfortunately, by storing keys in non-volatile storage, the value of the key becomes susceptible to a variety of attacks, many of which were discussed by Chakraborty [79]. Most of the reasons cited by the author for such attacks are centered more on the acquisition of intellectual property (IP) rather than keys. However, as stated by Simons et al. [80], "Tools for attacking hardware have become very advanced, which has decreased the protection provided by storing a key in memory to a minimum." Therefore, these keys must be protected from discovery through attacks like those presented by these two authors.

In order to protect against hardware attacks aimed at key extraction, such as those mentioned by these two authors, it is necessary to remove the keys from any form of non-volatile storage. This can be accomplished with PUFs, as shown in Figure 4.3. The first step is simply adding a PUF to the overall SoC architecture and connecting the results directly into the cryptographic engine.

When presented with a set of challenges, the PUF generates a set of responses which are feed into a key generation unit, shown in Figure 4.4. As proven by previously discussed research, these responses should contain random data that is unique for each device. The random data is used for two purposes: first, as an actual secret key for symmetric key operations, and second, as random input into an RSA key pair generation unit. As there are no mathematical constraints on the value of an AES symmetric key (aside from size), the random data can simply be fed straight through. On the other hand, RSA key pairs must be related mathematically and therefore cannot be random data values. Rather, the random data is presented as a seed for generation of random RSA keys. Since RSA key values are generated based

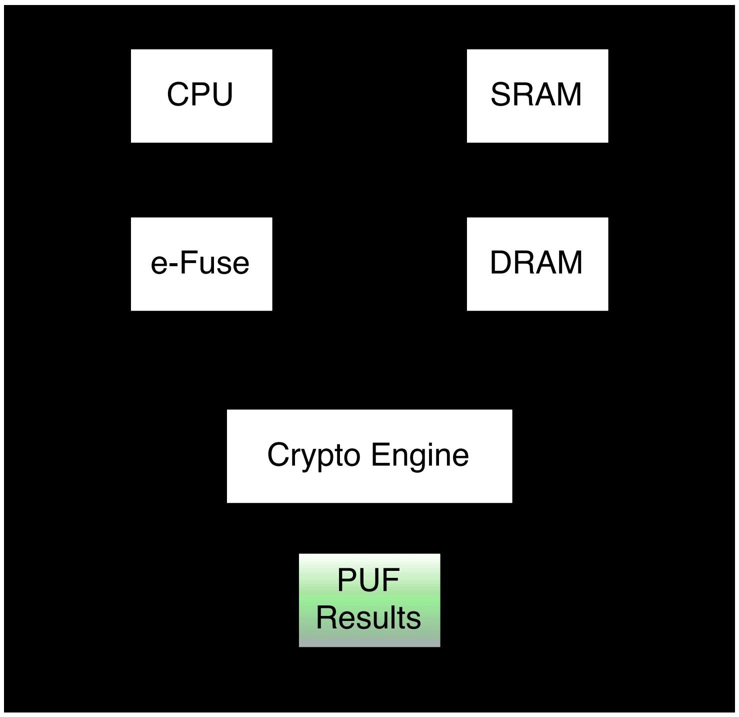


Figure 4.3: PUF Enhanced SoC Architecture

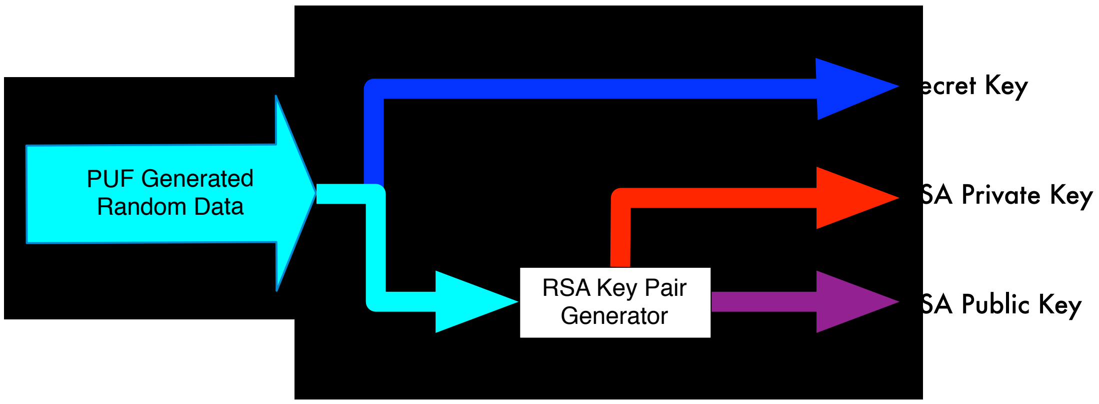


Figure 4.4: PUF Result Key Generator (PRBKG)

upon the value of the seed, the resulting keys will be unique per device.

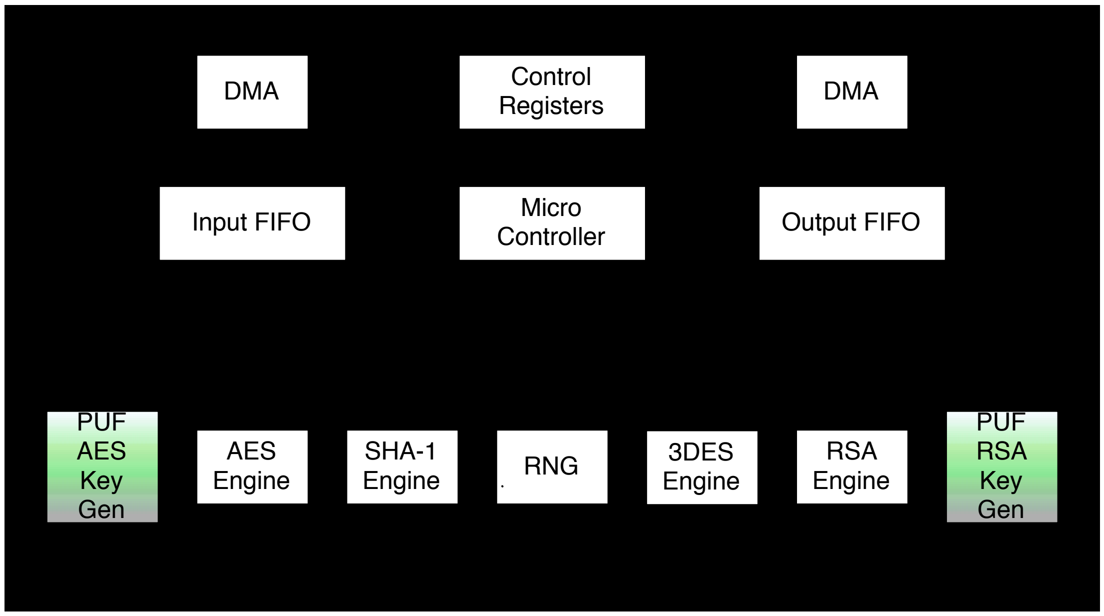


Figure 4.5: PUF Enhanced Cryptographic Engine

Once the PUF successfully generates the necessary keys and values, they can be fed into the crypto engine for access during cryptographic operations. Figure 4.5 shows the resulting architecture from adding the key generation modules into the generic architecture presented previously. The keys generated for use in this case must be static, meaning the PUF must always generate the exact same keys every time it is run. It will likely also be of interest to include key generation functionality that is based upon random PUF results and could be used sporadically by the device. The purposes of these random keys will be discussed in greater detail later in this chapter.

4.2 Crypto Engine Communication Architecture

A PUF-enhanced cryptographic engine can be integrated into a mobile system to support secure operations. However, several key considerations must be weighed in order to achieve the goals of improved system level security. First, any access methods that might exist that would allow code running on the processor to access the key must be eliminated. While it would be unusual for a designer to establish a direct communication mechanism between the processor and the micro-controller running inside the crypto engine, which does not mean that such an architecture will never exist. As such, a truly secure cryptographic engine should never make assumptions about which element it is communicating with and should therefore restrict all access to critical components.

Therefore, it is proposed that the key value be stored in some form of latched, volatile memory and fed directly into the corresponding cryptographic accelerators or muxed with any other key input lines going into the accelerator. The crypto micro-controller would then only control the mux that feeds the accelerators, rather than directly supplying the requisite values. In this way, no device or logic unit has direct access to the generated keys and they are only available when the device is running. The processor can then utilize these keys in the same method described previously: through the use of control registers.

As for when the keys are used, it is necessary to first understand their purpose. The purpose of the secret key is encryption of sensitive information, such as the TEE. It could also be used to encrypt known good values for MTM functionality, public/private key pairs, DRM authorization information, authentication, and various other applications which will be discussed in greater detail in Section 4.3. However, the use of the device secret keys should be as limited as possible. Instead, additional keys can be generated and encrypted with the device secret key. Upon boot up, these keys can be decrypted and used to protect the majority of sensitive information, while encryption with the device secret key is disabled. This provides a dramatically smaller window in which an attacker might attempt to gain access to the device secret keys.

The public/private pair would be used in conjunction with the secret key for authentication. Authentication can be used to verify the identity of the mobile device based upon the processor it's using, as this is unlikely to change. Instead of burning a key into the eFuse array, a hash of the network's public key can be stored instead. This will allow the device to verify any authentication requests. When the device is first associated with the network, a simple value exchange can occur, using the following process:

$$Alice : C = E_{A_{priv}}(T_{nonce} + T_{challenge})) + A_{pub} \tag{4.2.1}$$

$$Bob : T_{nonce} + T_{challenge} = D_{A_{pub}}(C) \tag{4.2.2}$$

$$Bob : C = E_{A_{pub}}(B_{pub} + E_{B_{priv}}(T_{nonce} + E_{B_{secret}}(T_{challenge}))) \tag{4.2.3}$$

$$Alice : B_{pub} + E_{B_{priv}}(T_{nonce} + E_{B_{secret}}(T_{data})) = D_{A_{priv}}(C) \tag{4.2.4}$$

$$Alice : T_{nonce} + E_{B_{secret}}(T_{challenge}) = D_{B_{pub}}(E_{B_{priv}}(T_{nonce} + E_{B_{secret}}(T_{challenge}))) \tag{4.2.5}$$

$$Alice : T_{response} = E_{B_{secret}}(T_{challenge}) \tag{4.2.6}$$

$$\tag{4.2.7}$$

For this example, the two users "Alice" and "Bob" will be used to represent the network and device respectively. In step 4.2.1, Alice will generate two items, a nonce and a challenge, and encrypt them using her private key. This information, along with Alice's public key, is then sent to Bob. Assuming Bob has a hash of Alice's public key stored in some form of non-volatile memory, he can verify the provided key and then use it to decrypt the nonce and challenge (step 4.2.2). Bob then encrypts the challenge using his device secret key, combines it with the nonce, and encrypts them together using his device private key. Bob combines that with his device public key, encrypts it all with Alice's public key, and returns the result to Alice (step 4.2.3). Finally, Alice can decrypt all of this information, thereby providing her with the Bob's public key (step 4.2.4), which can then be used to decrypt the origin nonce and the result of the challenge issued to Bob (step 4.2.5).

In order for this approach to work, there are a few conditions required in order to ensure security. First, this transaction must take place over a closed-network environment. With mobile devices, this could be done by a carrier prior to selling the device, or be done over hard-wire with the customer. Second, it should be understood that the nonce and data value being sent over in step 4.2.1 could be seen by anyone on the network. This value is not meant to be a secret, but rather a signature generated by Alice. Bob can store the challenge and use it to verify that any further authentication request do in fact originate from Alice. By adding the nonce, we eliminate the ability of an attacker to replay this communication or spoof these values in the future. This exchange of the nonce and data value is consistent with the challenge/response methodology discussed in Chapter 2.

In addition, the nonce is always encrypted prior to the challenge. The reason for this is that the nonce should always change, or repeat so infrequently as to make storage of all values in between infeasible. If the key were placed in front, an attacker could ascertain when a challenge is being issued, even if they cannot determine the value of the challenge. This could be somewhat mitigated by making the challenge longer than the block size when using a block cypher, or by changing the padding scheme used by a streaming cypher. However, it is simply a better idea to place the nonce first followed by the challenge, thereby making comparisons of subsequent cypher-text basically pointless. Further, this transaction should only need to occur once, as Alice should be able to use Bob's public key for any follow-on transactions. Should the need ever arise, Alice can issue a new challenge to Bob at any time over a completely encrypted channel.

The primary purpose of this approach is to provide a method of authentication for the device. Rather than using an approach similar to the ones presented in previous research efforts where a database of challenge/response pairs is maintained, this methodology requires only a single challenge/response pair to be maintained per device. In addition, it also provides a simple method for updating this information should any security compromise occur with regards to the stored data or keys.

From the users' point of view, it would be impractical to expect them to bring their device in to an authorized repair facility or to the Information Technologies (IT) department at work every time a potential security incident occurs. Not only would such requirements result in personal inconvenience, they could also create significant support requirements for carriers and/or manufacturers. Therefore, a policy must exist that allows for the update, modification, or removal of these challenge/response pair that can be done without direct impact on the user. This approach directly addresses this need.

Once this initial transaction occurs, subsequent exchanges can be optimized by removing the need to exchange Alice and Bob's public key. Instead, Bob can maintain an encrypted copy of Alice's public key in some form of non-volatile storage. Each time a transaction occurs, Bob simply decrypts Alice's public key using his own secret key, verifies the hash, and then uses the key as though it were passed in directly from Alice. This then reduces the exchange requirements, as shown below.

$$Alice : C = (E_{B_{pub}}(T_{nonce} + T_{challenge}) \tag{4.2.8}$$

$$Bob : T_{nonce} + T_{challenge} = D_{B_{priv}}(C) \tag{4.2.9}$$

$$Bob : C = E_{A_{pub}}(T_{nonce} + E_{B_{secret}}(T_{challenge})) \tag{4.2.10}$$

$$Alice : T_{nonce} + E_{B_{secret}}(T_{challenge}) = D_{A_{priv}}(C) \tag{4.2.11}$$

Although Alice's public key is public and disclosure of it should not be considered a significant security risk, it can provide information to an attacker who might have a presence on the device. If the attacker is able to simply read a section of memory and pull out all public keys that are stored by the device, (s)he could ascertain certain information about whom the device communicates with. Further, an attacker could modify the key with the public key of another device, thereby allowing the attacker to spoof the identity of the original device. For these reasons, it is highly recommended that all association and authentication keys, challenges, and supporting information be maintained completely encrypted while not in use and removed from memory as soon as possible after use.

4.3 PUF Generated Key Usage

The previous sections described how it would be possible to leverage a PUF-enhanced cryptographic engine to produce keys that can be used to perform secure authentication, as well as protecting security sensitive information on mobile devices. The capabilities provided by such an architecture are critical to the development of a secure system. Although the user is the primary owner of a mobile device after it has been purchased, the incorporation of these features can greatly reduce the adverse effect of malicious software on network providers and OS vendors. Additionally, each feature presented has applicability in all three arenas: network provider, OS vendor, and user.

To illustrate exactly how effective this architecture can be at addressing the secure concerned presented thus far, a number of use cases will be presented. First, it will be shown how the use of device unique secret key can be used to protect the TEE during the secure-boot process. Second, an example of how public/private key pairs can be used to protect and enforce DRM protections on streaming media will be presented. Finally, a device authentication mechanism aimed at eliminating theft that utilizes the authentication scheme detailed previously will be discussed.

4.3.1 Use Case: “Secure” Secure-Boot

Throughout this paper the concept of securing the TEE has been mentioned. It is proposed that the best method for doing this is by incorporating the use of the PUF enhanced crypto engine with its unique-per-device secret key. The generated secret key can be used to encrypt the entire TEE as it resides on disk, or to just encrypt another key that is used to encrypt the TEE. For this reason, it is critical that the PUF be able to generate a consistent, repeatable value for the device secret key. A number of approaches have been taken to address this concern, as discussed in Section 2.3.3. Multiple secret keys can be generated if needed, allowing for redundant copies in the event that one of the PUFs fails. For the purposes of this research, it is assumed that the PUF is stable, though contingencies are discussed should they ever be necessary.

When a device boots for the first time, it is expected that the non-volatile storage will be empty and no software will be loaded, other than initial boot code located in the secure-ROM. From here, the first step should be loading a TEE onto the device. In order to support encryption and protection of the TEE, it is necessary to store a header at the beginning of the disk, similar to the Master Boot Record (MBR)

Section	**Size**
TEE marker	4 Bytes
TEE version	4 Bytes
TEE SHA-256 hash encrypted	20 Bytes
TEE encryption routine	4 Bytes
TEE size	8 Bytes
Offset to boot-loader	8 Bytes
TEE Manufacturer	80 Bytes
Padding	384 Bytes

Table 4.1: Example TEE header

used on many desktop systems. This header contains all the information necessary to identify the location of, and then decrypt, the TEE. An example of such a header is shown in Table 4.1.

The example header shown provides a number of fields that can be used to store information about the TEE. For the decryption process, the critical pieces of information are the encryption routine used and the SHA-256 hash of the decrypted TEE. The identification of the algorithm allows for future changes, though it is expected that this will always be some form of AES. The hash value is used to verify that the TEE was not maliciously altered at any time and represents a hash of the decrypted TEE, not the encrypted version.

The header also contains a number of fields that can be used to provide information about the TEE and who manufactured it. The offset field provides information on where the TEE resides on disk, and the size field contains the size of the encrypted TEE. All other fields are for manufacturer identification and backward compatibility support, assuming manufacturers will produce multiple versions of their TEE over the years.

Once the TEE has been located, decrypted, and verified, execution can be passed to the TEE and the boot process can continue. The inclusion of this mechanism

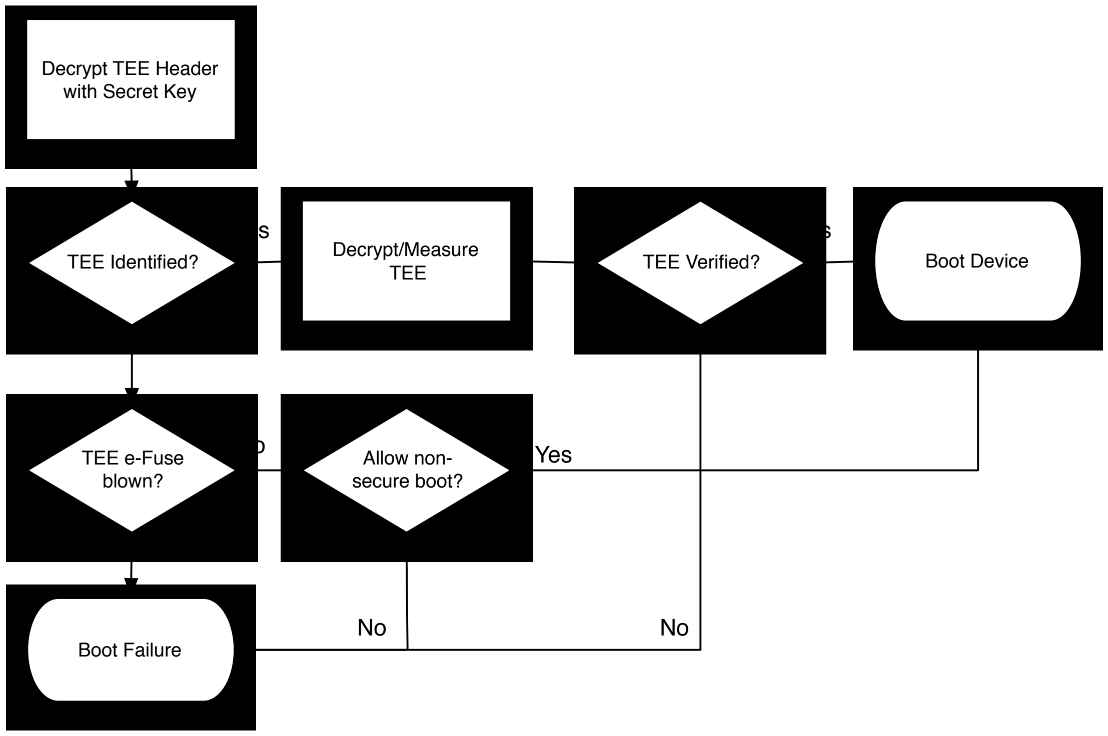


Figure 4.6: TEE boot process options

results in the boot process shown in Figure 4.6. As illustrated in this figure, the first step in the process is decrypting the TEE header. This should be done using the device secret key. If a TEE is identified, it is then decrypted, measured, and verified, and then executed. If no TEE is found on disk, a check should be performed to determine if a TEE was ever loaded onto the device. This can be done using a single eFuses bit. The eFuse bit should be enabled initially until a TEE is loaded for the first time. Once the TEE is loaded, the fuse can be blown. If the fuse is blown, if not TEE is found or if the TEE verification process fails, the device will not boot. Manufacturers may also allow for non-secure execution, though this will be implementation specific.

To protect against "bricking" the device in the event that a flash disk fails or the

TEE is inadvertently modified, manufacturers can upload a new TEE to the device. The process should be similar to what is done when the TEE is initially loaded. For optimal security, this process should require authentication of any device attempting to upload a new TEE. The process described in Section 4.2 would work perfectly for this issue.

It is important to note that this approach does not prevent the alteration of the TEE; instead, it intentionally allows the TEE to be updated. This is possible because the header that identifies the TEE only contains a SHA-256 hash of the decrypted TEE. In order to update the TEE, the new TEE sent to the device, encrypted using the device secret key, written to disk, and a new hash is generated and stored at the proper location in the TEE header. The ability to perform an update must be controlled via the TEE or the secure-ROM, which must both include functionality for verifying the authenticity of any TEE updates received. This architecture merely allows such an update to occur, while prohibiting an attacker from removing the TEE and booting in a non-secure fashion, nor modifying the TEE and overwriting the hash.

For this to work, both the TEE and the crypto engine will require enhanced support and greater API functionality. From the TEE perspective, it must include mechanisms for communicating with the crypto engine, performing updates, and restricting access to critical information in memory. The crypto engine must not ever assume that the entity using it is functioning properly and therefore must not allow arbitrary usage of either the secret key or the private key. How this can be handled will be addressed in Section 4.4.

4.3.2 Use Case: DRM Protection

Digital Rights Management, or DRM, is a term for the collection of protection mechanisms used by various digital media companies to protect media from copyright infringement and unauthorized usage. DRM incorporates a variety of different protection mechanisms and is used to protect a variety of electronic media from video games to streaming video. Rather than attempting to address how this technology could help each DRM technology available, the use case discussed here is a single specific technology. In this section, it will be shown how a protocol called FairPlay such could be enhanced with the PUF enhanced crypto engine. The FairPlay protocol is used extensively by Apple, Inc. [81].

FairPlay is a DRM protection mechanism used and developed by Apple to protect audio and video media. The protocol calls for the encryption of an audio/video stream of each file with a "master" key that is stored encrypted within the metadata for the file. Each time a user downloads a file, a random "user" key is generated and used to encrypt the master key. FairPlay servers maintain a list of user keys assigned to each user and download this information to client applications for each authorized machine. The client application then maintains this information in its own internal key repository. Whenever a user wants to play a file, the application pulls the user key from its key repository, decrypts the master key, and then uses it to decrypt the audio/video stream.

Although Apple stopped using FairPlay in 2009 for audio tracks, it continues to use this technology for videos purchased or rented on various devices. However, in the words of Steve Jobs,

> "The problem, of course, is that there are many smart people in the world, some with a lot of time on their hands, who love to discover such secrets and publish a way for everyone to get free (and stolen) music.

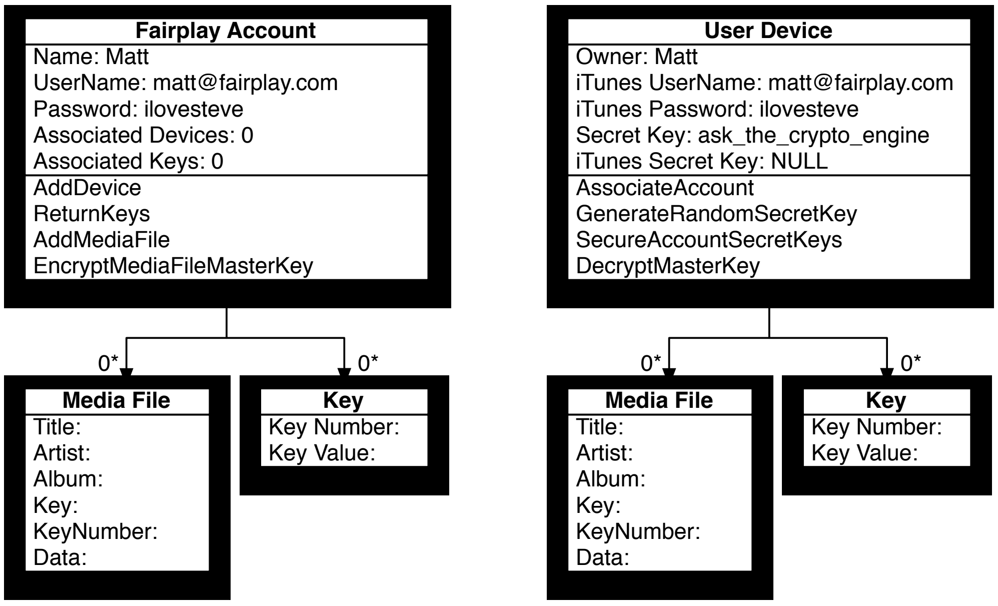
They are often successful in doing just that, so any company trying to protect content using a DRM must frequently update it with new and harder to discover secrets. It is a cat-and-mouse game.' ' [82]

Steve was correct about hackers breaking DRM, and FairPlay was no exception. Various groups and individuals were able to reverse engineer iTunes and discover methods for obtaining the master keys used to protect files, as well as intercepting the unencrypted traffic delivered to sound cards [81].

In addition to security issues related to the DRM protection scheme, FairPlay and Apple have both received a lot of backlash from consumers about the process of "authorizing" their devices. For instance, Apple restricts the number of authorize devices per account, currently set at five. Additionally, problems have arisen from the need to have an Internet connection available in order to authorize a device to play protected media. Even if a device is authorized with iTunes and almost has music purchased with the correct account, if the key for each file is not in the iTunes key repository, the device will not play the respective file. So the question is, how can DRM be used in a way that is not inhibited by network connectivity and that is not so restrictive on device usage?

In order to make the most of the capabilities provided by the ECE, an update is required to the mechanism for associating devices with FairPlay compatible accounts. Consider a simplified UML model of a FairPlay account stored on a FairPlay Server and a user device, as shown in Fig 4.7. These models show the necessary information and functions that are used by each entity to support this relationship. In addition, each has zero or more associated keys and media files.

Now, let us consider how a device would associate itself with a FairPlay server. The first step is to generate a random secret key that can be used to encrypt the "master" key for any media purchased via this device. This key, along with the



(a) FairPlay Account (b) User Device

Figure 4.7: Simplified UML models of a FairPlay account and user device.

account information and whatever FairPlay currently uses as the unique identifier, are sent to the FairPlay server. The FairPlay server then verifies the account information and determines if there are any device slots available for the account. If so, the secret key provided is added to the account. The FairPlay server then returns to the device all secret keys currently associated with the account. These keys are then stored on the device and encrypted using the device's secret key. This process is illustrated in Figure 4.8.

Most of the steps in this process should be fairly straightforward and easy to understand. However, the purpose behind sending each device all the secret keys associated with the account might appear somewhat suspicious. The reason for doing this is simple: by allowing all devices to know the secret key used by all other devices on the account, each device can play any supported media file, regardless of where it was purchased and how it was transferred to the device. As long as the

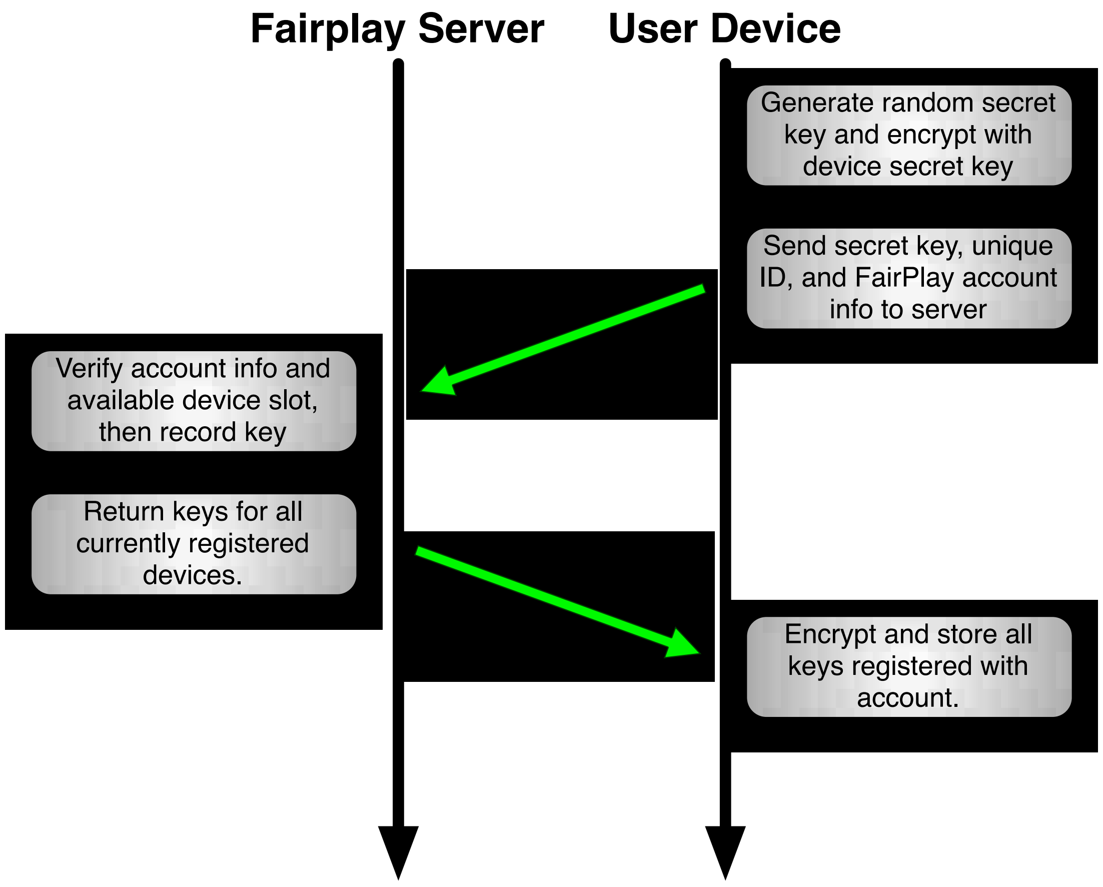

Fairplay Server
User Device
Generate random secret key and encrypt with device secret key
Send secret key, unique ID, and FairPlay account info to server
Verify account info and available device slot, then record key
Return keys for all currently registered devices.
Encrypt and store all keys registered with account.

Figure 4.8: Device association with FairPlay servers

device has been registered with the FairPlay account, it will have access to all keys used to encrypt any media files belonging to the account. Additionally, there is no inherent risk to the overall security of the device if these keys are exposed as they are only used to protect media files and nothing else.

If a device ever needs to be removed from the account, whether due to security issues or because it is no longer being used, the FairPlay servers simply need to go through the account and change the Key and KeyNumber attributes for each media file that was purchased by that devices. If the account maintains a "primary device", its key can be used to re-encrypt the master key and then the KeyNumber value can be set to that key. If there are no other devices on the account, the key can be reset to the master key and stay that way until a new device is associated with the account.

This technique can also be used support movie rentals where not only is there data encryption to consider, but also a timeframe. For instance, most movies rented through the Apple iTunes store or Amazon are only valid for 24 hours from the time it was rented. But what happens if during that time the user wants to watch the video on a different device, but doesn't have Internet access in order to authorize such a transaction? There is currently no mechanism for transferring a rental movie from one device to another. There is now.

By maintaining a list of keys for all devices on an account on each device, the capability exists to transfer files between devices and maintain the ability to play them. All information about the movie, including expiration time, can be secured on the initial device through the device's iTunes secret key. If a user wants to move the movie to another device, all it needs to do is transfer the file. Because each of the devices have all the keys, each will be able to decrypt a file meant for any other device on the account. Once on the new device, the new device can decrypt the information and play the movie using the appropriate account key.

The "key" here is that any device on the account can transfer a protected file to any other device without needing to determine whether or not it is associated with the same account. Because the information needed to play the file is encrypted, the only way another device could play the file is if it was already registered with the FairPlay account and had the full list of keys associated with the account. The only information necessary needed by an associated device in order to decrypt the file is which of the keys was used to encrypt the data. This could easily be stored in the file metadata, as shown in Fig 4.7, or it could use a simple trial-and-error approach wherein the device tries decrypting the file with each account key until it finds the one the correctly decrypts the data.

As can be seen from these DRM examples, this architecture provides the capability of providing significantly enhanced functionality and security. The primary advantage that this approach provides is in the ability to provide unique and protected keys that can be easily changed when needed without significant impact on a user's ability to access their files. Further, it provides a means of securely transferring files between multiple devices associated with the same FairPlay account that does not infringe on any copyright policies. DRM is a "cat-and-mouse" game and it is not expected that this will be a definitive security solution to the problems faced by DRM. However, this approach does provide significant benefits over any know approach used today.

4.3.3 Use Case: Deterring Theft

The final use case being presented details how this architecture can assist in deterring theft of mobile devices. The currently proposed approach is to use International Mobile Equipment Identity (IMEI) numbers to uniquely identify mobile devices. This should not be confused with the International Mobile Subscriber Identity (IMSI) number, which is usually stored within a Subscriber Identity Module (SIM) card. The IMSI is used to associate the user with the network, while the IMEI is used to associate the device with the network. However, there is currently no mechanism for associating the user with the device, and both together with the network.

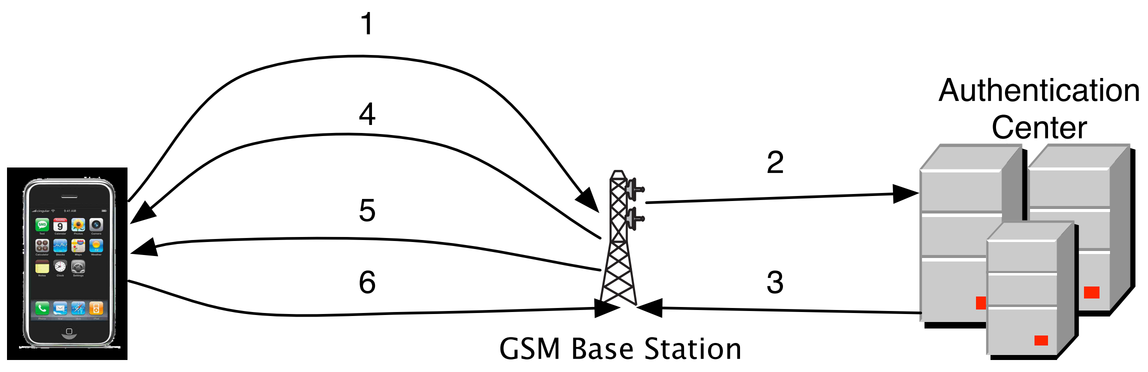
In addition to this lack of functionality, the usage of the IMEI number has not been as effective as was originally hoped. The first problem revolves around the fact that even if the device is reported as stolen and the IMEI is known, the device may only be placed on a blacklist for a specific carrier. This means a thief only needs to take the device to a different carrier in order to use it. While many devices may be "locked" to a specific carrier, or carrier frequency rather, there almost always exist tools for "unlocking" these devices in order to allow them to work on any carrier. Further, many countries don't even use black-lists thereby creating hot spots for thieves to buy and sell mobile devices.

Second, if the IMEI number is read and returned via software running on the device, the number can never be trusted to be accurate. If the OS has been hacked, the user can hard-code any value (s)he wants and thereby make the device appear to have a different number. Without any reliable verification mechanism, there is no method available to determine from the carrier end whether or not the real IMEI was returned or a spoofed version generated by some arbitrary code.

And finally, because this value is often stored in a programmable, non-volatile memory location, such as an eFuse array, they can be modified. If a device manufacturer has the capability of programming this area after manufacturing, a hacker will likely have the capability as well. This often requires nothing more than re-soldering a few connections and then having the necessary programming equipment. Such alterations are typically illegal, but then so is theft, so that's often of little consequence to a thief.

To best understand how an ECE can help address these problems, it is necessary to understand exactly how a mobile device user is authenticated with a cellular network. For this example, a Global System for Mobile computing, or GSM, network will be considered.

The first time a user attempts to connect to the network with their device, it sends the IMSI number to the network. The network verifies this IMSI number and a new number is generated, called the Temporary Mobile Subscriber Identity (TMSI). From this point on, the device uses this number to associate itself with the



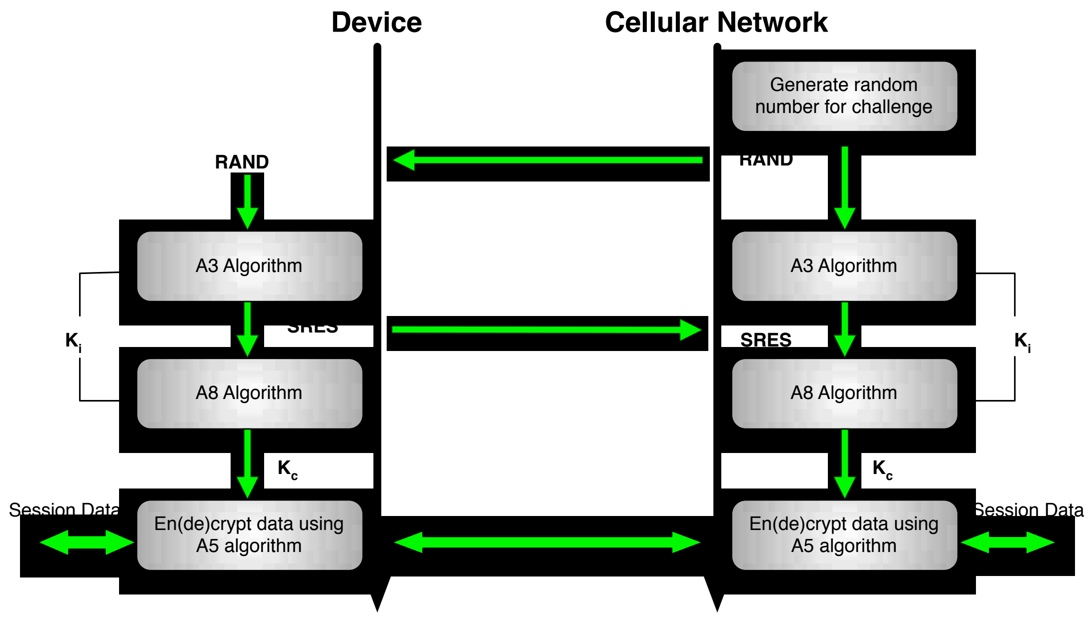
STEP	DESCRIPTION
1	Transmit IMSI to base station
2	Send IMSI number to network
3	Return TMSI, random number, and expected response to base station
4	Send TMSI to mobile device
5	Send random number to mobile device
6	Retrieve response from mobile device and compare to expected result

Figure 4.9: Device authentication on GSM networks

network rather than the IMEI. This is illustrated in Fig 4.9.

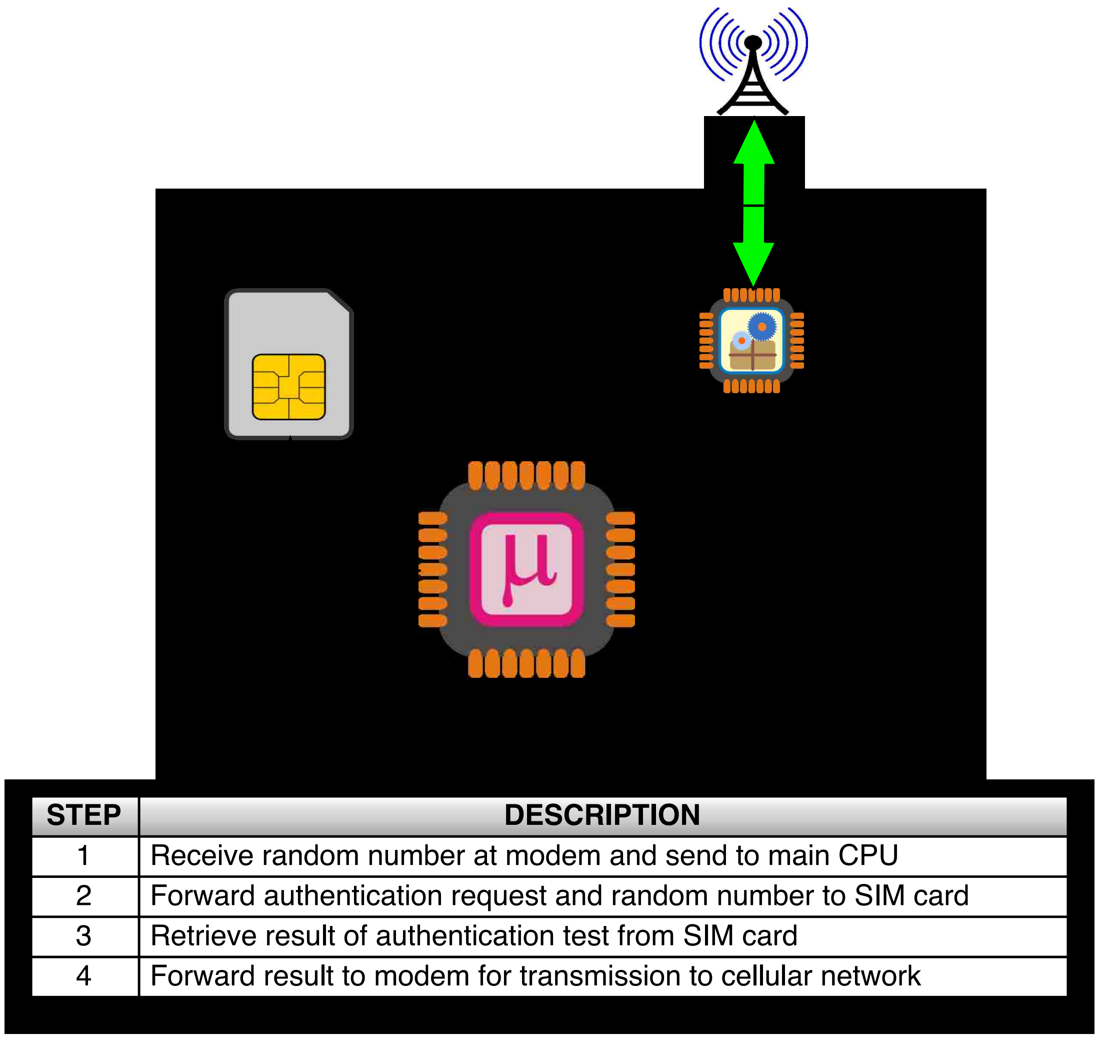
Each network also contains an Authentication Center (AC) that handles the association of devices with the network. Both the SIM card and the AC have access to a user authentication code and mathematical algorithm that are used in this process. To initiate the association, the AC sends a random number to the mobile device. This random number, along with the user code, is fed into the algorithm on the SIM card of the device. The AC also performs the exact same operation. Once the device completes the operation, it returns the response to the AC. The AC can then compare the device's result with its own and verify that they are the same. If so, the device is authenticated. Otherwise, the device is considered "unknown" and is denied access to the network.

While the ECE is not meant to provide any enhanced security to the communications channel between the device and the base station, it is important to see how this authentication works. Figure 4.10 shows more detail about how this is done. Using a shared key, K_i, which is programmed into the SIM card, the random number is run through the A3 algorithm on both sides. This results in a Signed RESponse (SRES) that is transmitted back to the base station to prove the authenticity of the device. The result is also feed into the A8 algorithm in order to create a session key, K_c. (The A3 and A8 algorithms are key-dependent, one-way hashes that today have been combined into what is known as the COMP128 algorithm) This key is then used to encrypt/decrypt all traffic between the device and the network. This approach is effective in that it allows encryption of traffic without ever having to transmit the key between element, but it should be noted that these algorithms and



Key	Purpose
K_i	Individual Subscriber Authentication Key, pre-programmed into SIM
K_c	Session key

Figure 4.10: Cryptographic operations during the authentication process



STEP	DESCRIPTION
1	Receive random number at modem and send to main CPU
2	Forward authentication request and random number to SIM card
3	Retrieve result of authentication test from SIM card
4	Forward result to modem for transmission to cellular network

Figure 4.11: Example internal communication flow during network authentication

the keys involved can typically be broken anywhere between a few minutes (A5) to a few hours (A3 and A8) [83].

It is also important to understand exactly how these operations are performed on the device, as well as how the data is handled. As shown in Fig 4.11, the random number generated by the cellular network is received at the modem and then transmitted to the main processor. Once the main CPU receives the request, it is forwarded on to the SIM card along with the random number that was received. The SIM card then sends the random number, plus the user authentication number, through the appropriate algorithm and generates the response. This response is then returned to the CPU, which then passes it along to the modem for broadcast to the cellular network.

Once the device is authenticated, the AC will periodically send out requests for the IMEI number of the device. The process for generating a request is similar to the cellular authentication process, except that the CPU, not the SIM card, provides the IMEI value. The reason for this is that the IMEI number is typically not stored inside the SIM card. Instead, it is likely stored in an eFuse array, which is internal to the processor and thus inaccessible to the SIM card.

The primary security concern in each of these communication processes is the fact that everything is channeled through the main CPU which means that whatever software is running on the CPU has the ability to manufacture whatever value it wants. What this means is that any form of malicious code could manufacture a bogus value for the IMEI number rather than the actual value stored on the chip. While it would be unusual for an attacker to interfere with the initial authentication mechanism (since they likely want the device connected to the cellular network), there is certainly reason to alter the results of an IMEI request. As mentioned previously, if a thief or attacker wanted to bypass blacklisting functionality, they need only intercept these requests and change the value returned for the IMEI number.

To counter the problem of IMEI spoofing or tampering, as well as to protect against theft, the ECE can be used to provide a simple solution. For authentication with the cellular network, a challenge/response test can be issued once a secure channel has been established between the device and the cellular network. The challenge/response test would utilize the device secret key to encrypt a challenge value and then return the response. This approach would require the least changes in existing cellular network protocols. However, a better method would be for the mobile device to provide the cellular network with its device public key. It could then sign the response provided to the CPU in Step 3 of Figure 4.11 and return it with the response. The cellular network could then verify the signature with the device's public key, thereby providing authentication information on the user and the device simultaneously.

In regards to IMEI spoofing, the ECE can also provide a random number that would take the place of the IMEI number and would also be inaccessible to the processor. The difficulty with this is having the processor acknowledge and response to the IMEI request, but not be able to manipulate the value. Unfortunately, there is no easy method for doing this. However, modern SoC designs provide a solution.

In addition to the main processor, most cellular devices have an additional processor called the baseband processor (BBP). This processor, also known as the modem, handles the cellular communications into and out of the mobile device. As such, it has the capability to identify authentication and IMEI request from the cellular network. Using its own PUF to generate a random, unique-per-device value, the BBP could provide an identification value for the mobile device that could be used in place of the IMEI number. By doing so, the main processor, or Application Processor as it is often called, has no control over the value returned during IMEI requests, nor would the cellular network have any idea the number was generated by the BBP rather than the application processor.

The solutions presented here provide two critical points. First, the ECE can be used by the application processor to counter theft in mobile devices. Second, the ECE has potential uses in more than just the application processor. While the identification of additional elements that can utilize an ECE is a subject for further research, it is important to note that even on a mobile processor, benefits can be gained by implementation on multiple elements.

4.4 Enhanced Crypto Engine API

In order for the Enhanced Cryptographic Engine (ECE) to be of use to the mobile device, it is necessary to provide an Application Programming Interface (API) that will allow the unit and its associated functionality to be accessible to developers. The API proposed in this section consists of a small number of functions providing the necessary interface, as well as *enums* and *#defines* to generate parameter values. The complete API header file is shown in Appendix A.

While the API does provide the functionality to support increased mobile security, it is the usage implementation, requirements, and restrictions that provide the enhanced security capabilities discussed thus far. In this section, the functions defined by the API will be presented. After discussing these functions, the next section will discuss the restrictions and requirements that must be met in order to ensure the highest level of security on the mobile device. In Chapter 5, a complete implementation will be presented that shows an example of a software API that implements all the specified functionality and does so in accordance with all listed requirements and restrictions.

4.4.1 ECE API Functions

Although the API functions may be accessed for a number of different purposes, it is easiest to consider their use in one of four different operational modes in which the processor may be executing. These modes are illustrated in Fig. 4.12. As shown, the processor can be in one of four different operating modes: Uninitialized mode, TEE Loading mode, Discovery mode, or Standard mode. During execution within each of these modes, the processor has specific responsibilities that require certain interactions with the ECE. To depict the functionality provided by the API, each mode will be considered independently.

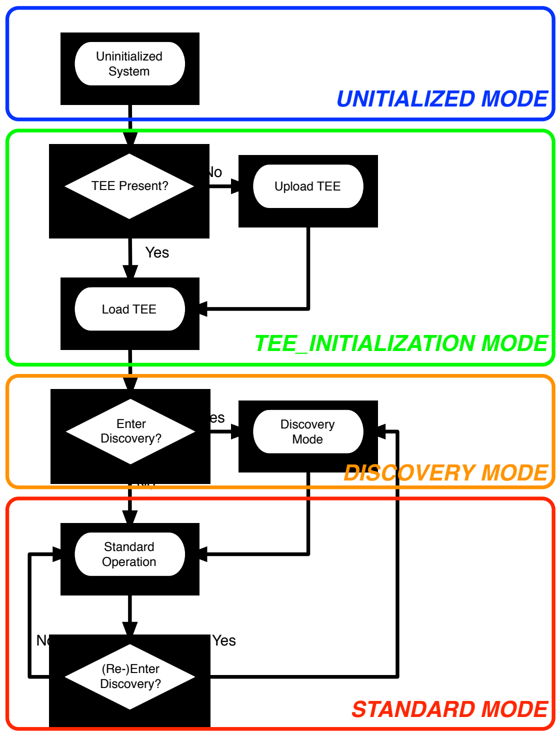


Figure 4.12: Operational modes of a mobile architecture using an enhanced cryptographic engine

Uninitialized Mode

A device enters the uninitialized mode when no TEE has been loaded into memory, or when a corrupted TEE has been detected. This will be the case when the device is turned on for the first time, or whenever a problem has occurred during the attempted boot of the TEE. Prior to loading a TEE onto the device, or executing an existing TEE, it is necessary to generate the AES secret key and the RSA key-pair from the PUF results inside the ECE. To support this need, two functions are necessary to instruct the ECE to generate the needed keys:

Once these two functions have been called from the SecureROM code on the device, the processor can move to the next mode. Because the TEE header is encrypted using the device secret key, it is necessary to enable access to this key prior to moving to the next mode.

Listing 4.1: Uninitialized Mode Functions

```
ERROR_CODE ECE_Generate_Secret_Key( void );
ERROR_CODE ECE_Generate_RSA_Key_Pair( void );
```

TEE Loading Mode

With the device-unique key loaded and accessible, the next step is discovery and verification of the TEE. Discovery of an existing TEE is handled via the process outlined in Figure 4.6. Using an approach similar to the Master Boot Record (MBR) used in desktop systems, the TEE header should be located at the beginning of the first sector of the disk. If no TEE is present, it will first be necessary to upload a TEE onto the system before continuing execution. The ability to scan for and load a TEE can be handled with one function, and the ability to load a new TEE and generate the corresponding header can be handled with another. These two functions are:

Listing 4.2: TEE Loading Mode Functions

```
ERROR_CODE ECE_TEE_Verification( uint32_t * load_address );
ERROR_CODE ECE_TEE_Initialization( void );
```

The first function, ECE_TEE_Verification, is used to scan for an existing TEE and then decrypt and load it at a specified address. Before the TEE header can be analyzed, it must first be decrypted using the device secret key. Once decrypted, the header provides all necessary information about the TEE in order to verify its integrity and to load the TEE into memory prior to execution.

The ECE_TEE_Initialization function is used for loading an initial TEE onto an internal storage device, as well as providing updates to an existing TEE. This function must therefore support the use of one or more communication protocols for uploading the TEE to the device. Which protocols are supported is platform dependent and is not dictated by this API. Once a TEE has been uploaded or modified, this function then invokes ECE_TEE_Verification to begin execution of the new TEE.

Additional functions are necessary to facilitate the cryptographic operations necessary to decrypt or encrypt the TEE and its header. The functions provided must also support access to the device secret key. To meet this need, two more functions are included that allow code to encrypt or decrypt data using a variety of modes and customized features. These functions are:

Listing 4.3: Crypto Loading Mode Functions

```
ERROR_CODE ECE_AES_Crypto( uint8_t *input, uint8_t *output, uint32_t length, uint32_t op_type, uint8_t *key,
     uint8_t *iv );
ERROR_CODE ECE_RSA_Crytpo( uint8_t *input, uint8_t *output, uint32_t length, uint32_t op_type, uint8_t *key );
```

Each function contains an *op_type* parameter that is used to define specific information about the respective operation, such as key size, encryption or decryption, and algorithm selection. A full listing of possible values is shown in Appendix A and is based off the open source OpenSSL library. This portion of the API is platform dependent and may include fewer, more, or the same number of options. The lack of a desired implementation in the provided API does not correlate to a lack of functionality in the design. Designers have full liberty to modify the provided API in any manner that fits their needs.

After the TEE has been loaded and begun execution, there are two possible modes into which the system may move. The Discovery' mode is used when it is necessary to associate one device with another device through the exchange of public keys. The other supported mode is called Standard mode and provides all other generic capabilities and interfaces to the ECE. The Discovery mode will be considered next.

Discovery Mode

Discovery mode is a special mode that allows for the exchange of public keys between two devices. Once in Discovery mode, a mobile device may issue a discovery request to another device, or it may received a discover request from another device. The required parameters for a discovery request consist of an address and an interface method, i.e. communication protocol. The device then issues a request using this information and exchanges keys using the algorithm defined in Section 4.2.

Discovery mode also supports the ability to authenticate a mobile device with a cellular network, Wi-Fi network, or any other necessary device or infrastructure. This authentication method includes the ability to associate a device with a user during authentication with a cellular network, as discussed in Section 4.3.3.

To support the ability to association and authenticate a device, five additional functions are needed. These functions are shown below.

Listing 4.4: Discovery Mode Functions

```
ERROR_CODE ECE_Device_Authentication( uint8_t *key, uint32_t method, uint8_t *address );
ERROR_CODE ECE_Device_Association( uint32_t method, uint8_t *address );
ERROR_CODE ECE_Device_Disassociation( uint32_t method, uint32_t *address );
uint8_t * ECE_AES_Keygen( void );
uint8_t * ECE_RSA_Keygen( void );
```

The first function, ECE_Device_Authentication, is used to verify the identity of a connecting device. Once in discovery mode, the function ECE_Device_Association is used to association with a given device. The function ECE_Device_Disassociation is used to remove a previously recorded association with another device. The last two functions, ECE_AES_Keygen and ECE_RSA-_Keygen, are used by both devices to generate a public/private key pair and a shared secret key that may be used to facilitate secure communications between the two devices. Once all discovery operations have been performed, the system moves into Standard mode.

Standard Mode

Standard mode is where the device will spend most of its time. This mode is predominately concerned with the protection of data and information controlled by both the REE and the TEE. The TEE should not rely upon TZ protections alone to protect access to sensitive information, but should instead generate keys for encryption of such information. The REE can also utilize key generation capabilities to protect its data and information, as well as to support standard cryptographic operations, such as Secure Sockets Layer (SSL) and Transport Layer Security (TLS).

Support of this functionality requires additional methods, though it may still utilize some of the functions previously listed, like ECE_AES_Keygen. The additional functions provided are shown below.

Listing 4.5: Standard Mode Functions

```
ERROR_CODE ECE_Change_Secret_Key( void );
ERROR_CODE ECE_Change_RSA_Key_Pair( void );
```

The last two functions provided are ECE_Change_Secret_Key and ECE_Change_RSA_Key_Pair. These functions are used to change the primary keys used by the device. This may be done at any time if it is believed that the device has potentially been compromised. Performing this operation is a fairly simple and straightforward procedure. The TEE is the only element that is encrypted with the device secret key. Therefore, it must be decrypted prior to changing the key and then re-encrypted after a new key has been generated.

To update the RSA key pair, a command must be sent to any authenticated network alerting the appropriate entity of the expected change. Once a new key pair is generated, a new challenge/response exchange can be initiated in order to re-authenticate the device with the network. This can be done by following the same procedure that was performed initially to authenticate the device. As long as the network is expecting this change, it is possible to initiate this update in a secure fashion.

4.4.2 ECE API Restrictions and Requirements

True security in almost any application does not come in the features that are provided, but in the restrictions and requirements for utilizing the provided functionality. The ECE is no exception. While the use of the ECE will likely vary from one implementation to another, there are a few critical rules that must be met completely in all applications. Other restrictions and requirements are flexible based upon the implementation and platform details for the design. This should not be mistaken as meaning that the restrictions and requirements could be ignored. Rather, this means that some platforms may wish to adhere to them in one way, while it would be more logical to do so in a different manner on another platform. Similar to the previous section, it will be easiest to consider these restrictions and requirements by looking at the four modes of operation.

The uninitialized mode is where the device keys are generated. The functionality provided by this mode must only be available during the first stage of boot. At no other time should any code be able to regenerate the device keys. At a later stage access to these keys must be disabled. Therefore it is absolutely critical that the functionality to regenerate these keys not be available at any other stage. Otherwise, an attacker could potentially gain access to the ECE and perform arbitrary encryption and decryption of device critical data with the device secret key, allowing them the ability to modify or remove the data.

The TEE Loading mode provides the mechanisms necessary to modify and upload a TEE to the device. Because the TEE will be responsible for controlling access to the ECE, it is critical that the TEE be properly verified and tested prior to use. All

TEEs must be signed before being uploaded to the device. A public key must be used to verify the signature for the TEE prior to loading it onto disk. To accommodate this need, a hash of the key must be stored on the device. The recommended location is the onboard eFuse array. In the case that an eFuse array is not available, the hash can also be stored in an unused section of the SecureROM. If neither of these two components is available, any other form of non-volatile, one-time programmable memory can be used. Developers may also choose to store the entire public key on the device rather than the hash. This is an acceptable alternative, but will likely restrict the number of suitable locations due to the size of a public key.

The TEE loading mode must also only be entered prior to loading of any existing TEE. A TEE may be partitioned between the trusted kernel and trusted applications. If such partitioning were done, it would be acceptable to allow for the uploading of new trusted applications or the updating of existing applications from within the TEE kernel while operating in Standard mode. However, the TEE kernel itself must never be updated at any time other than during the TEE Loading mode. This ensures that the proper verification of any provided update is performed prior to accepting the changes. While the TEE may initiate the update process while running in the Standard mode, the update must only be performed in TEE Loading mode prior to execution of the TEE. This ensures that even if an attacker were to gain execution within the TEE, (s)he cannot upload an arbitrary TEE update to the device. This is critical since the TEE is not re-verified after beginning executed. If an attacker is able to modify the TEE after it has been loaded into memory, there is not mechanism for detection. Therefore, even the TEE has an element of untrustworthiness of which must be accounted.

Discovery mode is the most highly implementation dependent mode. This is primarily due to the fact that the need to associate with other devices, as well as the method of association, is very implementation dependent. Regardless of the implementation, discovery mode must rely upon a trusted entity to approve its use. For a large number of devices, this can be achieved via user interactions. In other instances, user interaction is not possible and the trusted entity must be provided in another fashion. In either case, device association must be instituted via the TEE running in Standard mode. However, the association must only occur in Discovery mode. Therefore, if the device is running in Standard mode and an association is requested, the device must reboot into Discovery mode. A direct transition from Standard mode back to Discovery mode is not allowed. This requirement is in place for the same reason that was presented for TEE updating. An attacker must not be provided with the ability to arbitrarily associate or disassociate with other devices.

Another reason that discovery must only be performed while in Discovery mode is that the device secret key must be disabled prior to leaving Discovery mode and entering Standard mode. At this point, the TEE header and TEE have been decrypted and the TEE is running. The only remaining use for the device secret key is decryption of stored association information. A device will always enter discovery mode even if it is only to check for existing association information. After existing association information is decrypted, the device secret key must be un-latched. During Uninitialized mode, the ECE latches the PUF generated keys into registers so they can be accessed. These latches should be cleared once the association information is decrypted, thereby prohibiting access to the device secret keys. The device public/private key pair can still be enabled as they are used primarily for continual authentication purposes and do not provide protection of any device critical data.

In regards to how discovery it initiated, the simplest case is that association is only performed as needed by the user and with the user's permission. In other situations, such as automotive networks, discovery mode can be initiated via an external mechanism. In automotive systems, the On-Board Network (OBD) interface can be used to facilitate communications between computer systems. Networked devices can store the public key or key-hash internally for the manufacturer of the automobile. An external device can be pre-loaded with the manufacturer's private key and connected to the OBD network. This device can then send a command to each computer instructing it to enter discovery mode. This command must be signed using the private key, thereby allowing each networked computer to verify the authenticity of the command prior to transitioning to Standard mode. Once every internal computer has associated with all other computers, the external device then broadcasts an "end-association" command that tells each device to leave Discovery mode and enter Standard mode.

While such an approach might be appropriate for automotive networks, it is not recommended for standard user operated devices. User operated devices must always obtain user permission prior to entering Discovery mode. In the event a user is not available, some trust mechanism must be established prior to allowing any device to put another device into Discovery mode, such as the one presented previously. Although the exact approach presented here is not required, the secure establishment of trust between the two devices is, however that may be accomplished.

As stated before, once in Standard mode, access to the device secret key must be disabled. The TEE may continue to utilize the ECE to generate other secret keys in order to form a key-chain for encryption of other security sensitive data values. At this point, the majority of restrictions and requirements will be the responsibility of the TEE developer. Aside from access to the device secret key, all other ECE functions should be available. The functions for changing the device secret keys must also only initiate the action, rather than perform them. As with the transition to Discovery mode and the modification of the TEE, this must be performed at an earlier level. In this case, changing of the device secret keys must take place from with code executing out of the SecureROM. Any commands needed to support the changing of the private key can occur outside the SecureROM code since such modem and/or baseband processor support is typically not available here. However, the actual command to change the key must occur only in the SecureROM code.

The ECE contains a small amount of non-volatile memory that can maintain helper data for identifying viable PUF circuits for use in generating these keys. By changing which PUF circuits are used, the resulting keys can be changed. The new circuit information is stored as helper data within the non-volatile memory. The use of helper data does not provide any information about the key itself, as discussed in Section 2.3.3. After changing the helper data, the device will then have a new secret key that can be used to protect the TEE.

It is expected that the functionality to change the device secret key will be maintained within the TEE, though the request to do so will likely come from code executing within the REE. To provide the highest degree of assurance that the request is valid, the REE must be measured and verified prior to accepting any such requests. Further, the user must be prompted from the TEE, not the REE, for approval of such an action. All ECE requests that necessitate user interaction must be performed via the TEE, not the REE. This provides the highest level of confidence that such requests come from a trusted source. Additionally, all requests that require user approval must include a cool-down timer. If an attacker was able to gain access inside the TEE, (s)he must not be allowed to continually change the key as it might be possible for them to eventually guess the right value. It is recommended that a cool-down time of 24 hours be used, though shorter periods may be used if necessary. It is highly recommended that no time period shorter than one hour be used. Where possible, the timer should be placed inside the ECE and not the TEE. If an attacker has gained execution within the TEE, they would likely be able to bypass any internal cool-down timers.

Chapter 5

Implementation and Results

In Chapter 4, the ECE architecture was presented, along with a collection of use cases that can benefit from the functionality provided by the ECE. While a full hardware implementation of the ECE would be ideal, the purpose of this research is to prove the feasibility of this approach and determine any weaknesses that may exist or any changes that might strengthen its security. Further, complications with hardware platforms that would support such an architecture have made a hardware implementation impractical at this time.

In order to overcome the limitation presented by not having a viable hardware platform, while still providing solid results on the effectiveness of this architecture, a software emulator of the ECE has been developed. In this chapter, the structure of this emulator will be discussed and a number of test cases that were conducted with the emulator will be presented. The test cases include protection of a TEE and association between a number of devices. Details about the API presented in Section 4.4 and how the functions were implemented in this emulator will also be covered. In conclusion, this chapter will present the results discovered through the use of this software emulator and how they affect the proposed architecture and security approach.

5.1 ECE Software Emulator Architecture

The software emulator consists of two executables: ece_emulator and tee_client. As discussed in Section 4.4 and shown in Fig. 4.12, the ECE is accessed by a processor running in one of four execution modes. These four execution modes are split between these two applications. The ece_emulator handles the Uninitialized and TEE_Initialization modes, while the tee_client handles Discovery and Standard modes. This was done because logically the first two modes happen outside of any TEE on the device, while the last two modes are either handled or invoked by the TEE. While the tee_client is not meant to emulate a fully compliant GlobalPlatform TEE, it does contain all the necessary functionality to support the ECE API functions that must exist within a TEE. Both applications utilize the OpenSSL library [84] in order to implement cryptographic operations, such as key generation and encryption and decryption routines.

To better understand how these two applications work together to emulate the ECE and its corresponding software API, consider the execution flow of the ece_emulator shown in Fig. 5.1. When the ece_emulator application is started, it is provided with a command line argument that represents the name of an ECE disk file. An ECE disk file is a 128MB file that is used to emulate a simple flash chip on a mobile device. This file maintains several items that are necessary for the proper functionality of the ece_emulator application. The size of this file is completely arbitrary, with 128MB being chosen because it is large enough to store a significant amount of test data while not taking an excessive amount of disk space on any modern desktop computer. The ECE disk file is also mapped into memory when used by the ece_emulator, so that was a factor in its size as well.

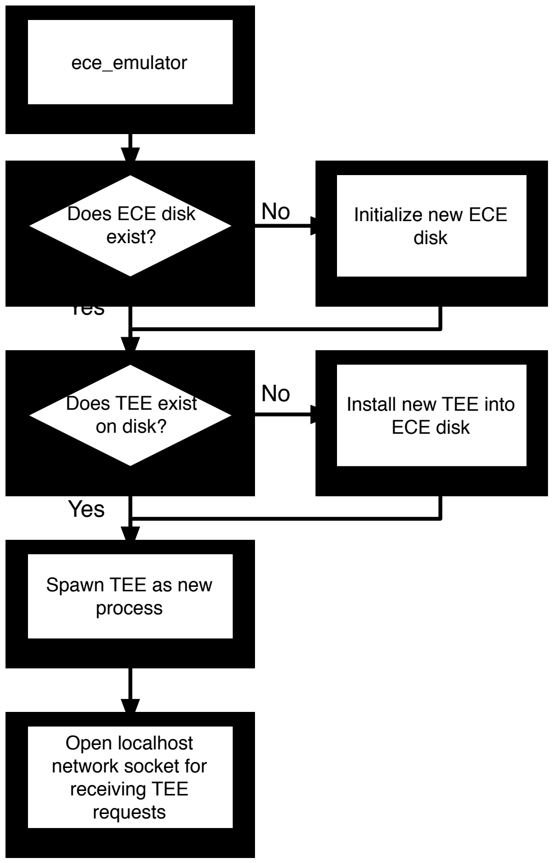


Figure 5.1: Operational flow of ece_emulator application.

The ECE disk file starts with a header that identifies the files as containing a valid ECE format. The header used is shown in Table 5.1. This header contains all information necessary to initialize the ECE environment. Because OpenSSL is not setup to generate the same key each time its key generation functions are called, it is necessary to store the secret keys inside the ECE disk file after they are generated. This allows the application to maintain the same secret keys across multiple runs and represents the only significant deviance from the actual ECE architecture.

This portion of execution represents the Unitialized mode. The first time the program is run, the ECE disk file will not exist. The program therefore initializes a new

Section	Size
ECE marker	4 Bytes
ECE header size	4 Bytes
ECE section size	4 Bytes
Offset to secret AES key	8 Bytes
Offset to secret RSA key pair	8 Bytes
Offset to TEE header	8 Bytes
Padding	476 Bytes

Table 5.1: ECE disk header

disk file and generates the necessary device secret keys using the ECE_Generate_Secret_Key and ECE_Generate_RSA_Key_Pair functions, as shown in Figure 5.2. Initialization consists of creating the file, expanding it, generating the keys, writing them to the file, and then writing the ECE header to file. Memory mapping is not absolutely necessary, but rather makes access to the header and corresponding data

```
ECE_Error_Code  ECE_Generate_Secret_Key()
  key <- RAND_bytes( 32 );
  iv <- RAND_bytes( 16 );

ECE_Error_Code  ECE_Generate_RSA_Key_Pair()
  keyPair <- RSA_generate_key( size, exponent, NULL, NULL );

ECE_Error_Code  ECE_Initialize_New_Disk( diskName )
  if( diskName does not exist )
    CreateFile( diskName );
    ExpandFileSize( diskName, 128 * 1024 * 1024 );
    data = MapFileToMemory( diskName );
    deviceAESKey = ECE_Generate_Secret_Key();
    deviceRSAKeyPair = ECE_Generate_RSA_Secret_Key_Pair();
    hdr = ECE_Generate_ECE_Header( deviceAESKey, deviceRSAKeyPair );
    ECE_Write_AES_Key_To_File( data, deviceAESKey );
    ECE_Write_RSA_Key_Pair_To_File( data, deviceRSAKeyPair );
    ECE_Write_ECE_Header_To_File( data, hdr );
  else
    data = MapFileToMemory( diskName );
    hdr = ECE_Read_ECE_Header_From_File( deviceAESKey, deviceRSAKeyPair );
    deviceAESKey = ECE_Read_Secret_Key_From_File();
    deviceRSAKeyPair = ECE_Read_RSA_Secret_Key_Pair_From_File();
```

Figure 5.2: ECE disk initialization

easier. If the disk file already exists, the header and keys need to be read from the file.

In addition to the secret key information, the ECE header also provides information about the location of the TEE header. In a standard ECE implementation, the ECE header will not exist and the TEE header will be located at the first sector of the disk. However, due to the need to maintain key information, the TEE is located at a different location in the ECE disk file.

Once the ece_emulator has verified and parsed the information contained in the ECE header, it moves on to the next operational mode. In the TEE Loading mode, the ece_emulator is responsible for implementing the ECE_TEE_Verification and ECE_TEE_Initialization functions by utilizing the ECE_AES_Crypto and ECE_RSA-_Crypto routines. If the disk file is brand new, a new TEE will need to be loaded. For

```
int32_t ECE_AES_Crypto( input, output, length, op_type, key, iv )
  EVP_CIPHER_CTX_init( cryptCTX );
  EVP_CIPHER_CTX_set_padding( cryptCTX, 0 );
  algorithm = ECE_Get_AES_Algorithm( op_type );
  if( op_type is encrypt )
    EVP_EncryptInit_ex( cryptCTX, algorithm, NULL, key, iv );
    EVP_EncryptUpdate( cryptCTX, output, bytesEncrypted, input, length );
    EVP_EncryptFinal_ex( cryptCTX, output+bytesEncrypted, bytesEncrypted );
    return bytesEncrypted;
  else
    EVP_DecryptInit_ex( cryptCTX, algorithm, NULL, key, iv );
    EVP_DecryptUpdate( cryptCTX, output, bytesDecrypted, input, length );
    EVP_DecryptFinal_ex( cryptCTX, output+bytesDecrypted, bytesDecrypted );
    return bytesDecrypted;

int32_t ECE_RSA_Crypto( input, output, length, op_type, key )
  padding = RSA_PKCS1_OAEP_PADDING;
  if( op_type is encrypt )
    bytesEncrypted = RSA_public_encrypt( length, input, output, keyPair,
        padding );
    return bytesEncrypted;
  else
    bytesDecrypted = RSA_private_decrypt( length, input, output, keyPair,
        padding );
    return bytesDecrypted;
```

Figure 5.3: ECE AES and RSA crypto operations

```
ECE_Error_Code ECE_TEE_Initialization( teeFileName )
  teeHdr = ECE_Generate_Blank_Header();
  ECE_Fill_TEE_Manufacturer_Info( teeHdr );
  teeData = ECE_Read_TEE_Executable( teeFileName );
  if( ECE_Verify_Signature( teeData ) < 0 )
    return ECE_ERROR;
  teeHdr->hash = ECE_SHA_256( teeData );
  teeHdr->routine = AES_256_CBC;
  teeHdr->size = ECE_AES_Crypto( teeData, encData, fileSize, encyrypt,
      secretKey, secretIV );
  memcpy( teeHdr + teeHdr->offset}, teeData, teeHdr->size );
  ECE_AES_Crypto( teeHdr, encHdr, teeHdrSize, encrypt, secretKey, secretIV );
  memcpy( teeHdr, encHdr, teeHdrSize );

ECE_Error_Code ECE_TEE_Verification( diskData )
  if( ECE_Does_TEE_Header_Exist( diskData ) == FALSE )
    if( ECE_TEE_Initialization( diskData ) < 0 )
      return ECE_ERROR;
  teeData = ECE_Read_TEE_Header( diskData );
  if( ECE_Verify_TEE_Header( teeData ) < 0 )
    return ECE_ERROR;
  ECE_Execute_TEE( teeData );
```

Figure 5.4: TEE initialization and verification

the emulator, this is the tee_client executable. Initialization consists of reading in the tee_client executable, encrypting it using the device secret key, and then writing it to disk. Additionally, a TEE header must be created based upon file attributes for the tee_client file, such as size. This header must also be encrypted prior to storing it in the ECE disk file. The complete pseudo code for the ECE_AES_Crypto and ECE_RSA_Crypto functions are shown in Figure 5.3 and the TEE functions are shown in Figure 5.4.

Although OpenSSL provides a significant number of functions and features that make this emulator possible, it is important to note that there is no RSA_private_-encrypt and RSA_public_decrypt functions. Or rather, there are none that function the same as the converse functions. The RSA_private_encrypt function is really a signature generation function as opposed to a function that will actually encrypt a specified string of data with a private key. The same is true for public decryption. Fortunately this was only slightly limiting and is not a problem that should occur

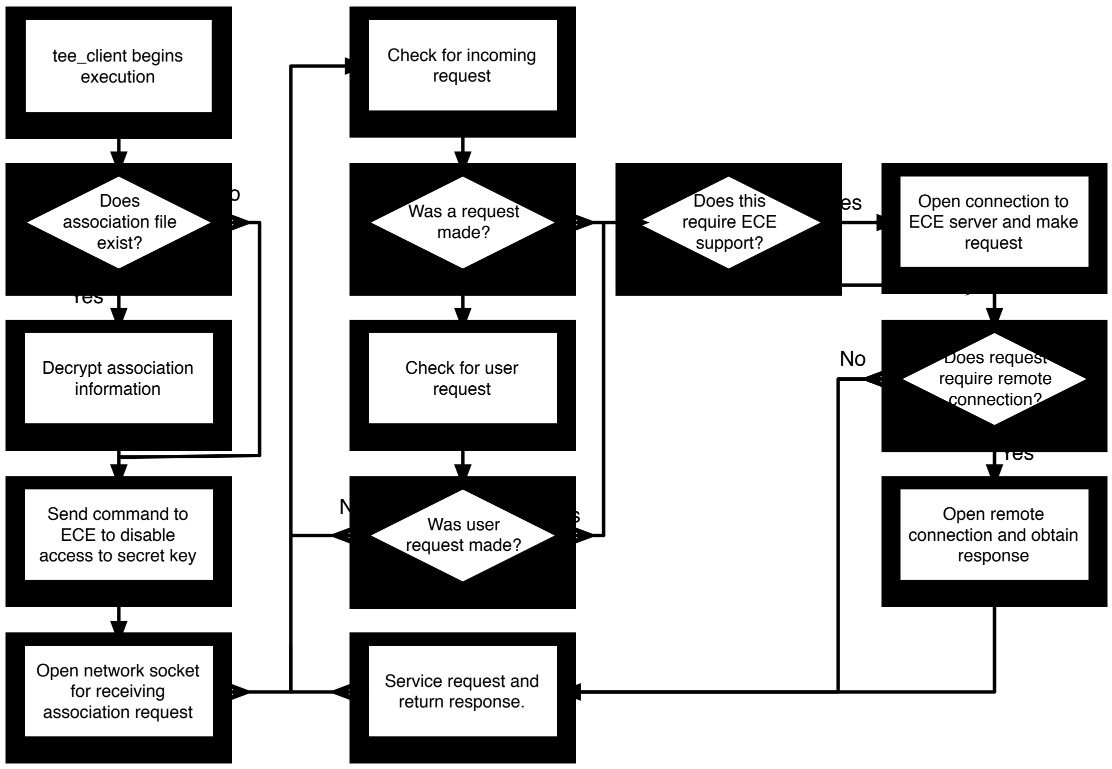

tee_client begins execution
Does association file exist?
Decrypt association information
Send command to ECE to disable access to secret key
Open network socket for receiving association request
Check for incoming request
Was a request made?
Check for user request
Was user request made?
Service request and return response.
Does this require ECE support?
Open connection to ECE server and make request
Does request require remote connection?
Open remote connection and obtain response
No
Yes

Figure 5.5: Execution flow of TEE client application

with regular cryptographic engines, as they typically have no method for detecting whether the key being used is the public or private key.

After both applications have been started, it is necessary to provide a communications mechanism in order for the tee_client to request certain operations by the ece_emulator. To facilitate this need, the ece_emulator opens a localhost only network socket that can be used by the tee_client to make requests. Once this port is open, the ece_emulator awaits connections and then services any requests as they arrive. Supported requests include en(de)cryption with the secret AES and RSA keys, AES and RSA key generation, and changing of the AES and RSA secret keys.

The tee_client has a more complex execution flow, as it has to listen for commands locally from the user, as well as from remote locations. Additionally, it must be able to initiate requests to remote devices as requested by the user. The full execution flow for the tee_client is shown in Fig. 5.5.

Execution starts by first looking for an existing association file inside the ECE disk file. Anytime an association is made with another device, there is a collection of values that must be stored in order to open a secure communication channel with an associated device. To support the storage of association information, the header shown in Table 5.2 is used. This header maintains all necessary association information. For the RSA keys, both the public and private local keys are stored. Although the local public key should never be used after it is transmitted to the associated device, it was decided to simply maintain it. This was mostly because the OpenSSL function PEM_write_bio_RSAPrivateKey, which is used to write the private key to a buffer, actually writes out both the public and private keys. Rather than hacking OpenSSL to only write the private key, it was determined to simply maintain both as doing so results in no serious security risk to the emulator.

In Discovery mode, the tee_client supports all of the necessary functionality to support association and authentication. In the case of the ece_emulator, the gen-

Section	**Size**
associationTag	4 bytes
headerLength	4 bytes
totalLength	4 bytes
addressType	4 bytes
associationAddress	20 bytes
offsetToLocalAESKey	4 bytes
localAESKeyLength	4 bytes
offsetToLocalRSAKeys	4 bytes
localRSAKeysLength	4 bytes
offsetToRemoteRSAKey	4 bytes
remoteRSAKeyLength	4 bytes

Table 5.2: Device association header

```
TEE_Error_Code TEE_Association_Request( remoteAddress )
  if( user does not approved )
    return TEE_ERROR;
  rsaKeys = ECE_RSA_Keygen();
  aesKey = ECE_AES_Keygen();
  nonce = Read_Random_Data();
  challenge = nonce + SHA_256_Hash( localAddress, remoteAddress );
  packet = ECE_RSA_Crypto( challenge, encChallenge, strlen(challenge), encrypt
      , rsaKeys->private );
  packet = packet + rsaKeys->public;
  Send_Request( packet, remoteAddress );
  Get_Response( response );
  if response < 0
    return TEE_ERROR;
  decResponse = TEE_Decrypt_Response( response );
  remoteKey = decResponse->pubKey;
  challenge = decResponse->challenge;
  TEE_Write_Association_To_File( remoteAddress, challenge, aesKey, rsaKeys,
      remoteKey );
  return TEE_SUCCESS;

TEE_Error_Code TEE_Association_Response( clientAddress )
  request = Get_Request();
  if( user does not approve request from clientAddress )
    return TEE_ERROR;
  rsaKey = ECE_RSA_Keygen();
  ECE_RSA_Crypto( request->data, decReq, strlen( request->data ), decrypt,
      request->pubKey );
  challenge = SHA_256_HASH( localAddress, clientAddress );
  if( challenge not same as decReq->challenge )
    return TEE_ERROR;
  ECE_AES_Crypto( challenge, encChallenge, length, encrypt, secretKey,
      secretIV );
  ECE_RSA_Crypto( encChallenge + request->nonce, encPacket, length, encrypt,
      rsaKey->private );
  ECE_RSA_Crypto( encPacket + rsaKey->public, encResponse, length, encrypt,
      request->pubKey );
  SendResponse( encResponse );
  TEE_Write_Association_To_File( clientAddress, decReq->challenge, decReq->
      aesKey, rsaKey, decReq->pubKey );
  return TEE_SUCCESS;

TEE_Error_Code TEE_Disassociation( remoteAddress )
  if( user does not approved )
    return TEE_ERROR:
  if( association does not exist for remoteAddress )
    return TEE_ERROR;
  TEE_Remove_Association_From_File( remoteAddress );
  return TEE_SUCCESS;
```

Figure 5.6: TEE association and disassociation

eration of AES and RSA keys are identical to what was shown in Figure 5.2 for generating the device secret keys. The only difference is that these keys are not stored in the ECE disk file. Instead, the ece_emulator calls the same functions and then returns the resulting keys over the network connection. For this reason, the pseudo code for these functions is not included. The pseudo code for the association function is shown in Figure 5.6. Because the authentication function operates in virtually the exact same manner, it is not included. It is important to note that functionality for association is split in two routines: one routine for making a request, and another routine for receiving a request. Authentication does not require two functions, as a mobile device should never request authentication; only a network should initiate such a procedure. Associated devices may use their stored challenge values as part of subsequent interactions to prove their identities and therefore have no need for an independent authentication routine.

Once all associations are retrieved, the tee_client sends a command to the ece_-emulator to disable use of the device secret key. To protect the key from being accessed directly by the tee_client, the ece_emulator program is run with root privileges and spawns the tee_client as a standard user application. The TEE disk file that is created by ece_emulator is therefore only accessible by root, so tee_client has no ability to access the file and read the key values. Since the ece_emulator application is run as root and opens a network port, a closed-network is highly recommended for any testing. Associated security risks should be mitigated by the fact that the connection accepts localhost connections only, but that opens the system up to malicious local applications. However, the purpose of doing this was to attempt to emulate the same protective environment that would exist on a mobile device. Both programs could also be run as a standard user and the claim could be made that the tee_client is assumed safe and would never attempt to access the key. Either approach is viable.

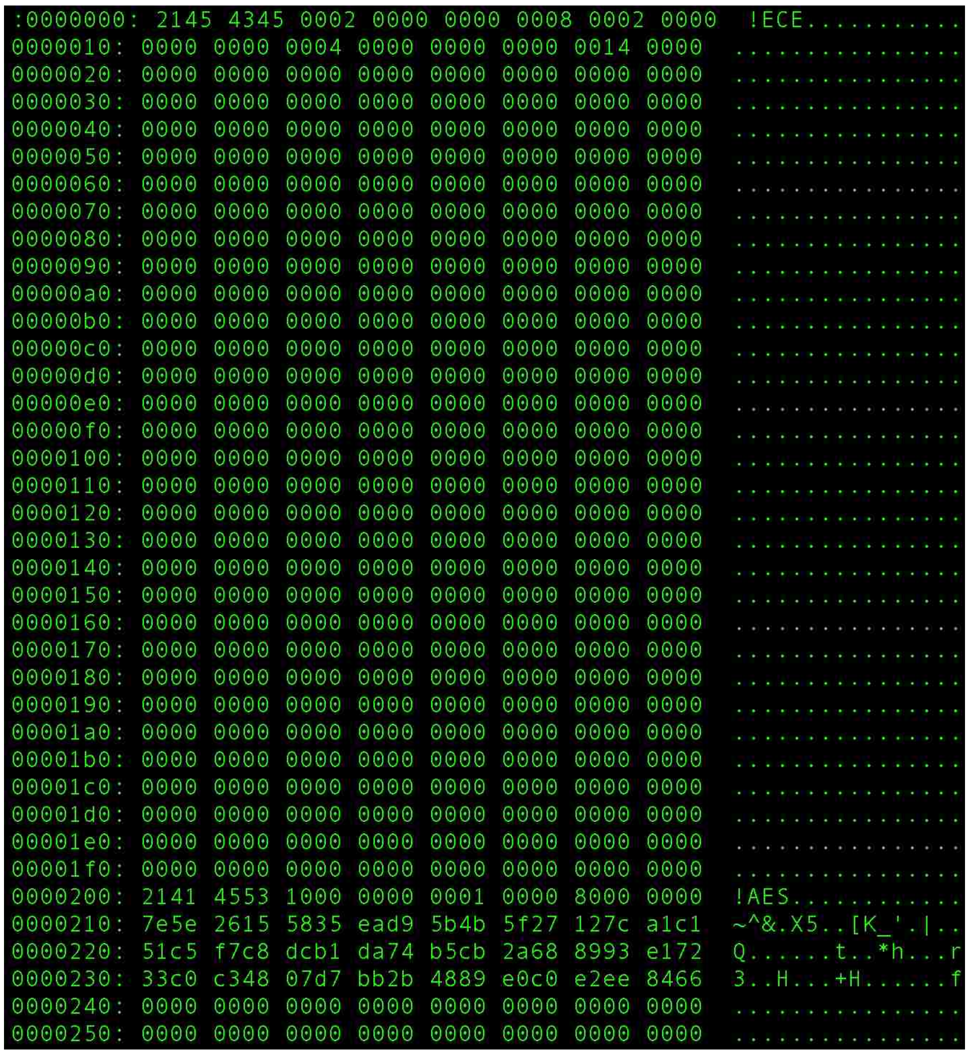
The tee_client also opens a network socket that may be used to receive requests from external devices. The tee_client monitors this connection for available clients.

```
TEE_Error_Code TEE_Change_Secret_Key()
  if( user does not approve )
    return TEE_ERROR;
  decTee = ECE_AES_Crypto( tee, length, decrypt, secretKey, secretIV );
  decAssociations = ECE_AES_Crypto( associations, length, decrypt, secretKey,
      secretIV );

  ECE_Change_Secret_Key();

  associations = ECE_AES_Crypto( decAssociations, length, encrypt, secretKey,
      secretIV );
  tee = ECE_AES_Crypto( decTee, length, encrypt, secretKey, secretIV );

  ECE_Write_Associations( associations );
  ECE_TEE_Initialize( tee );

TEE_Error_Code TEE_Change_RSA_Key_Pair()
  if( user does not approve )
    return TEE_ERROR;
  DeauthenticateWithNetwork();
  ECE_Change_RSA_Key_Pair();
  AuthenticateWithNetwork();
```

Figure 5.7: TEE secret key and key pair change

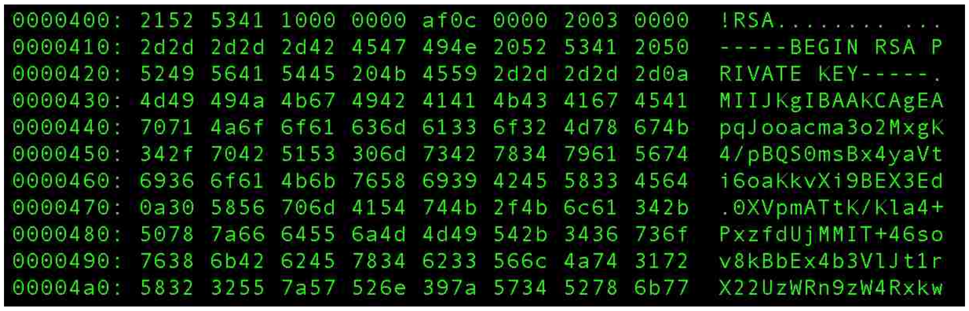
If a request has been made, it must be determined if this request requires interactions with the ece_emulator. For instance, association requests require generation of a new public/private RSA key pair and AES key, as well as encryption services during the transaction. Therefore, if communication is needed with the ece_emulator, a connection is made and the request(s) is issued. Other requests may require a connection to a remote device, such as an association request made locally. In this case, a connection must be opened to the remote device and the request issued. All association requests, whether received locally or remotely, require user approval in order to proceed.

After all remote requests have been serviced, the tee_client next looks for local request made via the command line. A user is able to issue local commands to the tee_client and may also provide input for certain requests that have been received. Anytime a remote device requests an association, the user must approve the transaction, as mentioned previously. The user can also issue the command to change the secret keys, disassociate with a device, or generate a new key. The pseudo code for changing the secret keys is shown in Figure 5.7.



```
:0000000: 2145 4345 0002 0000 0000 0008 0002 0000  !ECE............
0000010: 0000 0000 0004 0000 0000 0000 0014 0000  ................
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000130: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000160: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000170: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000180: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000190: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000200: 2141 4553 1000 0000 0001 0000 8000 0000  !AES............
0000210: 7e5e 2615 5835 ead9 5b4b 5f27 127c a1c1  ~^&.X5..[K_'.|..
0000220: 51c5 f7c8 dcb1 da74 b5cb 2a68 8993 e172  Q......t..*h...r
0000230: 33c0 c348 07d7 bb2b 4889 e0c0 e2ee 8466  3..H...+H......f
0000240: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000250: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Figure 5.8: ECE disk header and AES key in ECE disk file



```
0000400: 2152 5341 1000 0000 af0c 0000 2003 0000  !RSA.........  ...
0000410: 2d2d 2d2d 2d42 4547 494e 2052 5341 2050  -----BEGIN RSA P
0000420: 5249 5641 5445 204b 4559 2d2d 2d2d 2d0a  RIVATE KEY-----.
0000430: 4d49 494a 4b67 4942 4141 4b43 4167 4541  MIIJKgIBAAKCAgEA
0000440: 7071 4a6f 6f61 636d 6133 6f32 4d78 674b  pqJooacma3o2MxgK
0000450: 342f 7042 5153 306d 7342 7834 7961 5674  4/pBQS0msBx4yaVt
0000460: 6936 6f61 4b6b 7658 6939 4245 5833 4564  i6oaKkvXi9BEX3Ed
0000470: 0a30 5856 706d 4154 744b 2f4b 6c61 342b  .0XVpmATtK/Kla4+
0000480: 5078 7a66 6455 6a4d 4d49 542b 3436 736f  PxzfdUjMMIT+46so
0000490: 7638 6b42 6245 7834 6233 566c 4a74 3172  v8kBbEx4b3VlJt1r
00004a0: 5832 3255 7a57 526e 397a 5734 5278 6b77  X22UzWRn9zW4Rxkw
```

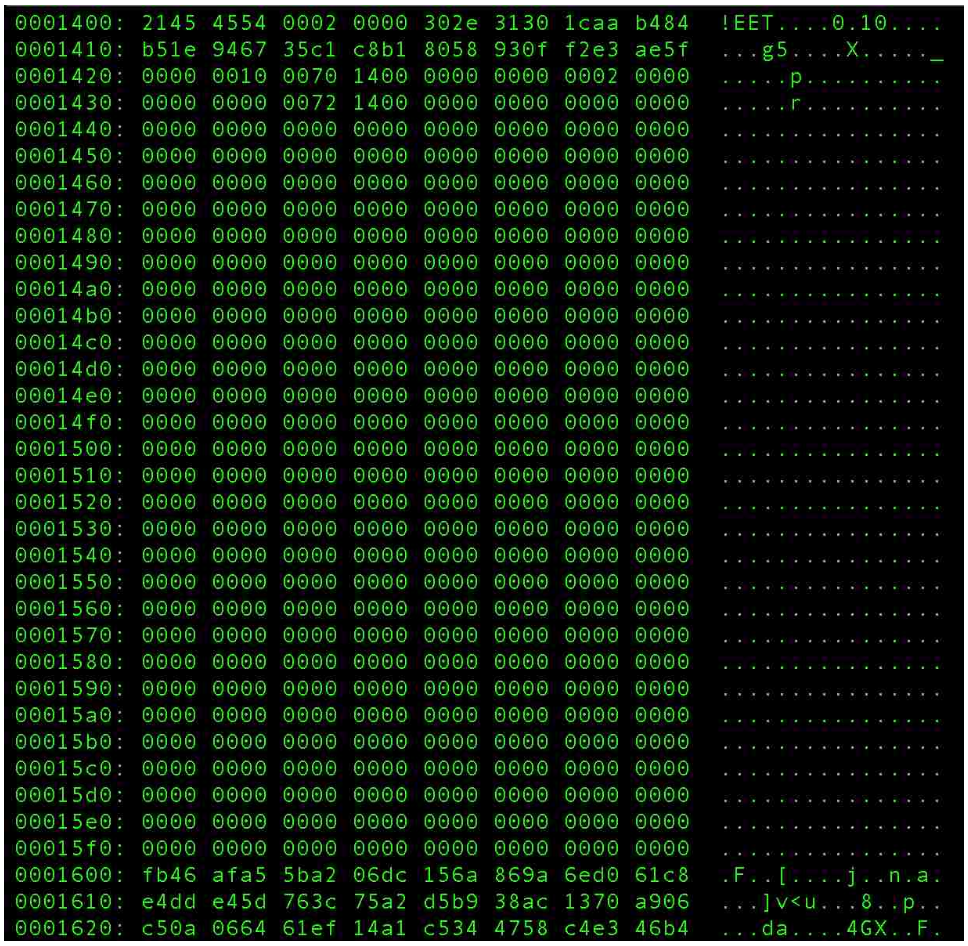
Figure 5.9: RSA key in ECE disk file

5.2 ECE Software Emulator Results

Using the pseudo code presented in the previous section as a basis, a full software emulator was developed and tested using three desktop devices. These devices included a 2012 Apple Macbook Pro laptop, a VMware virtual machine running 64-bit Ubuntu 12.04 Server, and a custom desktop with a quad-core AMD processor running 64-bit Ubuntu 11.10. Since timing analysis and overhead from cryptographic operations is highly implementation dependent and not the focus of this research, a detailed listing of the specifications for each desktop device is not included. Instead, the results focus on the implementation of the proposed API.

The first test conducted was the generation of ECE disk files along with the encryption and storage of a TEE executable. This was performed on all three devices using a natively compiled tee_client application as the TEE. The disk files generated contain an ECE header, along with the AES and RSA keys (Figure 5.8 and Figure 5.9), and the TEE header (Figure 5.10).

Next, it was necessary to prove that the secret key generated by each desktop device was unique and that the code properly encrypted each TEE. To show this, an entropy test was performed on the TEE section of each corresponding ECE disk file. The entropy results are shown in Table 5.3, which provides the entropy measurement,



```
0001400: 2145 4554 0002 0000 302e 3130 1caa b484  !EET....0.10....
0001410: b51e 9467 35c1 c8b1 8058 930f f2e3 ae5f  ...g5....X....._
0001420: 0000 0010 0070 1400 0000 0000 0002 0000  .....p..........
0001430: 0000 0000 0072 1400 0000 0000 0000 0000  .....r..........
0001440: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001450: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001460: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001470: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001480: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001490: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00014a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00014b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00014c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00014d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00014e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00014f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001500: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001510: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001520: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001530: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001540: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001550: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001560: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001570: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001580: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001590: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00015a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00015b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00015c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00015d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00015e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00015f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001600: fb46 afa5 5ba2 06dc 156a 869a 6ed0 61c8  .F..[....j..n.a.
0001610: e4dd e45d 763c 75a2 d5b9 38ac 1370 a906  ...]v<u...8..p..
0001620: c50a 0664 61ef 14a1 c534 4758 c4e3 46b4  ...da....4GX..F.
```

Figure 5.10: TEE header and encrypted TEE in ECE disk file

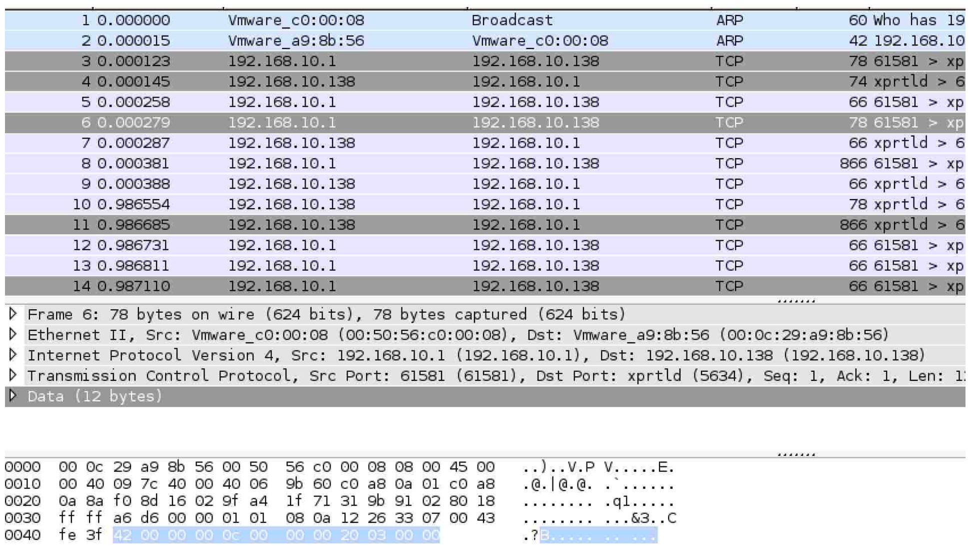
	Laptop	**VM**	**Desktop**	**Optimal**
File Size (bytes)	1266096	1333920	1339392	N/A
Entropy	7.999867	7.999876	7.999831	8.0
Arithmetic Mean	127.4433	127.4774	127.3839	127.5
Serial Correlation Coeff	0.001216%	-0.000163%	-0.000036%	0.0

Table 5.3: Entropy measures of platform specific, encrypted TEE

the arithmetic mean, and the serial correlation coefficient.

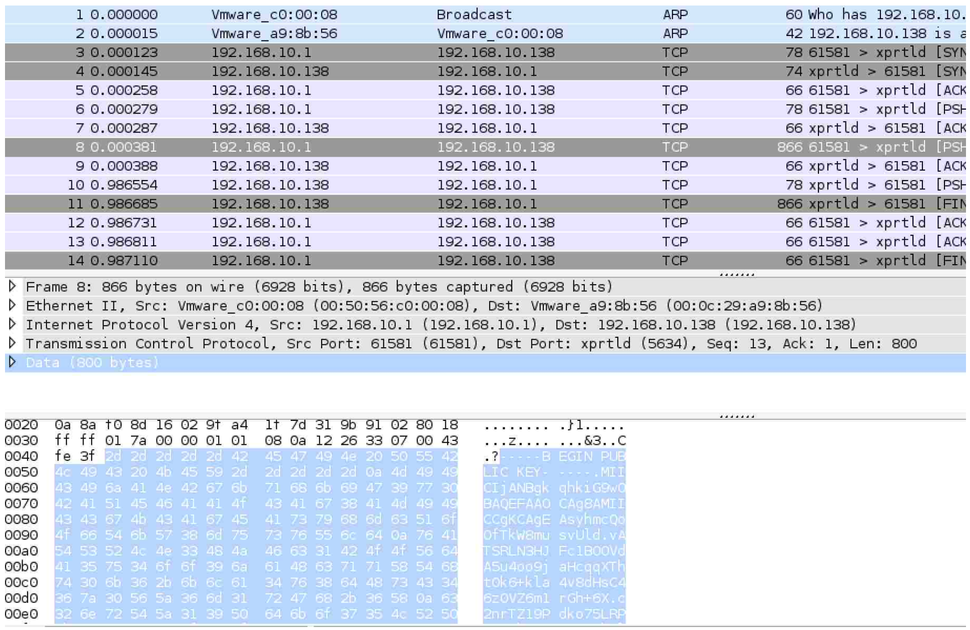
The entropy measurement is for each bit in a bytes. As there are eight bits in a bytes, the optimal value would be eight. The arithmetic mean is the average value of each byte in the file. Since bytes can contain values between 0 and 255, the median value is 127.5. Thus, the optimal arithmetic mean is 127.5. The serial correlation coefficient refers to the effect that the value of one byte will have on the subsequent byte. For encrypted or compressed data, the optimal value is zero. As shown in this table, each of the encrypted TEE sections are very close to the optimal values for their respective measurements. However, this is to be excepted since the data is encrypted using AES-256, which is known to produce data with such randomness values.

An additional item to note from Table 5.3 is the fact that the TEE file sizes are not identical. This is because the TEE for each device was compiled natively. Because each desktop device has a different version of the GCC compiler, and was compiled against a different kernel source, the resulting TEEs vary in size. To further prove that the device keys are unique, the same TEE was provided to each device



```
 1 0.000000   Vmware_c0:00:08    Broadcast          ARP    60 Who has 19
 2 0.000015   Vmware_a9:8b:56    Vmware_c0:00:08    ARP    42 192.168.10
 3 0.000123   192.168.10.1       192.168.10.138     TCP    78 61581 > xp
 4 0.000145   192.168.10.138     192.168.10.1       TCP    74 xprtld > 6
 5 0.000258   192.168.10.1       192.168.10.138     TCP    66 61581 > xp
 6 0.000279   192.168.10.1       192.168.10.138     TCP    78 61581 > xp
 7 0.000287   192.168.10.138     192.168.10.1       TCP    66 xprtld > 6
 8 0.000381   192.168.10.1       192.168.10.138     TCP   866 61581 > xp
 9 0.000388   192.168.10.138     192.168.10.1       TCP    66 xprtld > 6
10 0.986554   192.168.10.138     192.168.10.1       TCP    78 xprtld > 6
11 0.986685   192.168.10.138     192.168.10.1       TCP   866 xprtld > 6
12 0.986731   192.168.10.1       192.168.10.138     TCP    66 61581 > xp
13 0.986811   192.168.10.1       192.168.10.138     TCP    66 61581 > xp
14 0.987110   192.168.10.1       192.168.10.138     TCP    66 61581 > xp

▷ Frame 6: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
▷ Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Vmware_a9:8b:56 (00:0c:29:a9:8b:56)
▷ Internet Protocol Version 4, Src: 192.168.10.1 (192.168.10.1), Dst: 192.168.10.138 (192.168.10.138)
▷ Transmission Control Protocol, Src Port: 61581 (61581), Dst Port: xprtld (5634), Seq: 1, Ack: 1, Len: 1
▷ Data (12 bytes)

0000  00 0c 29 a9 8b 56 00 50  56 c0 00 08 08 00 45 00   ..)..V.P V.....E.
0010  00 40 09 7c 40 00 40 06  9b 60 c0 a8 0a 01 c0 a8   .@.|@.@. .`......
0020  0a 8a f0 8d 16 02 9f a4  1f 71 31 9b 91 02 80 18   ........ .q1.....
0030  ff ff a6 d6 00 00 01 01  08 0a 12 26 33 07 00 43   ........ ...&3..C
0040  fe 3f 42 00 00 00 0c 00  00 00 20 03 00 00         .?B..... .. ...
```

Figure 5.11: Transmission of TEE association request header



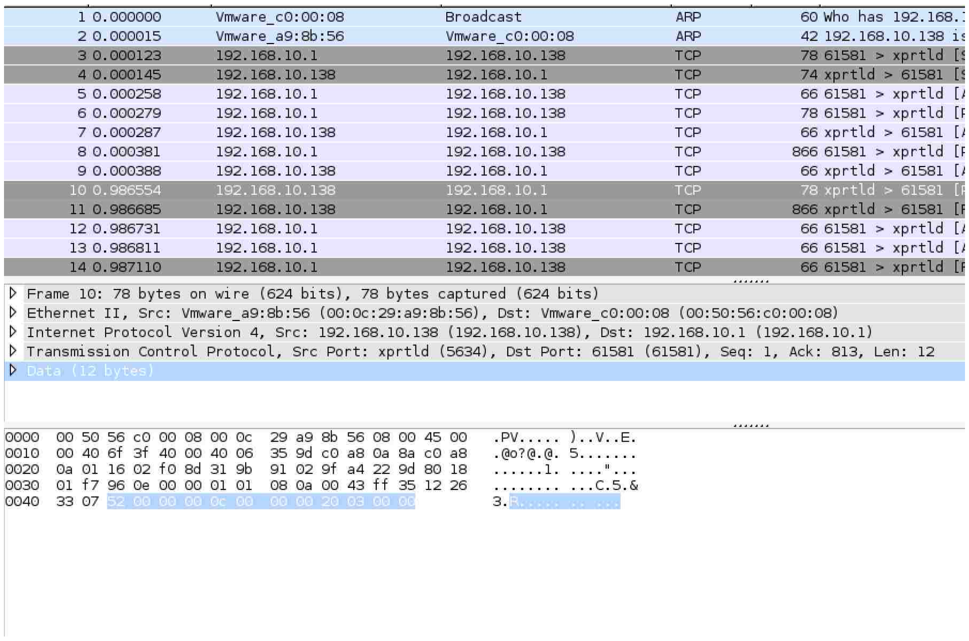
```
 1 0.000000   Vmware_c0:00:08    Broadcast          ARP   60 Who has 192.168.10.
 2 0.000015   Vmware_a9:8b:56    Vmware_c0:00:08    ARP   42 192.168.10.138 is a
 3 0.000123   192.168.10.1       192.168.10.138     TCP   78 61581 > xprtld [SYN
 4 0.000145   192.168.10.138     192.168.10.1       TCP   74 xprtld > 61581 [SYN
 5 0.000258   192.168.10.1       192.168.10.138     TCP   66 61581 > xprtld [ACK
 6 0.000279   192.168.10.1       192.168.10.138     TCP   78 61581 > xprtld [PSH
 7 0.000287   192.168.10.138     192.168.10.1       TCP   66 xprtld > 61581 [ACK
 8 0.000381   192.168.10.1       192.168.10.138     TCP  866 61581 > xprtld [PSH
 9 0.000388   192.168.10.138     192.168.10.1       TCP   66 xprtld > 61581 [ACK
10 0.986554   192.168.10.138     192.168.10.1       TCP   78 xprtld > 61581 [PSH
11 0.986685   192.168.10.138     192.168.10.1       TCP  866 xprtld > 61581 [FIN
12 0.986731   192.168.10.1       192.168.10.138     TCP   66 61581 > xprtld [ACK
13 0.986811   192.168.10.1       192.168.10.138     TCP   66 61581 > xprtld [ACK
14 0.987110   192.168.10.1       192.168.10.138     TCP   66 61581 > xprtld [FIN

▷ Frame 8: 866 bytes on wire (6928 bits), 866 bytes captured (6928 bits)
▷ Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Vmware_a9:8b:56 (00:0c:29:a9:8b:56)
▷ Internet Protocol Version 4, Src: 192.168.10.1 (192.168.10.1), Dst: 192.168.10.138 (192.168.10.138)
▷ Transmission Control Protocol, Src Port: 61581 (61581), Dst Port: xprtld (5634), Seq: 13, Ack: 1, Len: 800
▷ Data (800 bytes)

0020  0a 8a f0 8d 16 02 9f a4  1f 7d 31 9b 91 02 80 18   ........ .}1.....
0030  ff ff 01 7a 00 00 01 01  08 0a 12 26 33 07 00 43   ...z.... ...&3..C
0040  fe 3f 2d 2d 2d 2d 2d 42  45 47 49 4e 20 50 55 42   .?-----B EGIN PUB
0050  4c 49 43 20 4b 45 59 2d  2d 2d 2d 2d 0a 4d 49 49   LIC KEY- ----.MII
0060  43 49 6a 41 4e 42 67 6b  71 68 6b 69 47 39 77 30   CIjANBgk qhkiG9w0
0070  42 41 51 45 46 41 41 4f  43 41 67 38 41 4d 49 49   BAQEFAAO CAg8AMII
0080  43 43 67 4b 43 41 67 45  41 73 79 68 6d 63 51 6f   CCgKCAgE AsyhmcQo
0090  4f 66 54 6b 57 38 6d 75  73 76 55 6c 64 0a 76 41   OfTkW8mu svUld.vA
00a0  54 53 52 4c 4e 33 48 4a  46 63 31 42 4f 4f 56 64   TSRLN3HJ Fc1BOOVd
00b0  41 35 75 34 6f 6f 39 6a  61 48 63 71 71 58 54 68   A5u4oo9j aHcqqXTh
00c0  74 30 6b 36 2b 6b 6c 61  34 76 38 64 48 73 43 34   t0k6+kla 4v8dHsC4
00d0  36 7a 30 56 5a 36 6d 31  72 47 68 2b 36 58 0a 63   6z0VZ6m1 rGh+6X.c
00e0  32 6e 72 54 5a 31 39 50  64 6b 6f 37 35 4c 52 50   2nrTZ19P dko75LRP
```

Figure 5.12: Transmission of public key of initiating device

and then encrypted and stored inside the ECE disk file. The TEE sections were then extracted and compared with one another to determine if there was any correlation between to values of each byte in the encrypted TEE files. This test also resulted in a correlation of nearly zero, as expected.

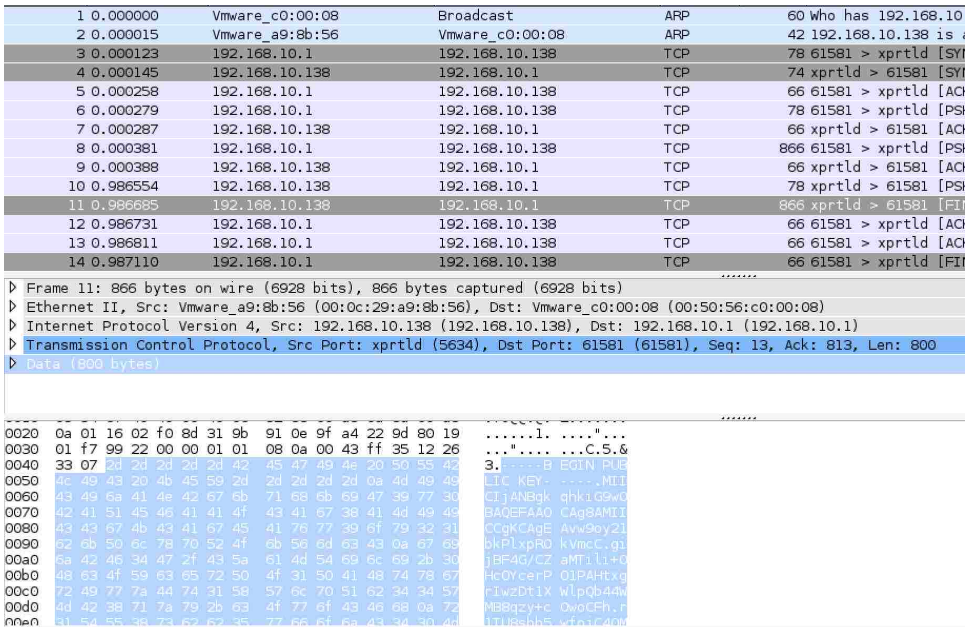
In addition to test on the randomness of the TEE, the full functionality of the software emulator was tested to ensure proper execution of the API functions. Each test conducted produced successful results. While it is difficult to quantify success for many of these features, a collection of network packet captures were generated that show a full association between two devices, followed by the transmission of a file encrypted using the public keys exchanged during the association.

The first step of the association is illustrated in Figure 5.11. In this step, the association request is sent to the remote device. All request and responses start with a 12 bytes header that contains the command, the header size, and the data size. Once the header is transmitted, the corresponding data is relayed. In this case, the



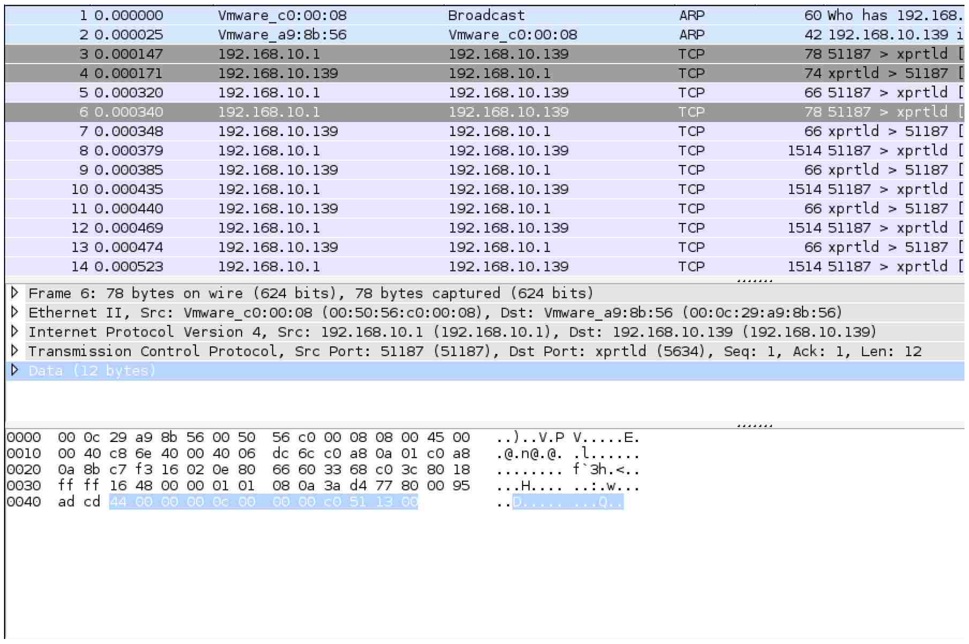
```
  1 0.000000   Vmware_c0:00:08    Broadcast          ARP    60 Who has 192.168.1
  2 0.000015   Vmware_a9:8b:56    Vmware_c0:00:08    ARP    42 192.168.10.138 is
  3 0.000123   192.168.10.1       192.168.10.138     TCP    78 61581 > xprtld [S
  4 0.000145   192.168.10.138     192.168.10.1       TCP    74 xprtld > 61581 [S
  5 0.000258   192.168.10.1       192.168.10.138     TCP    66 61581 > xprtld [A
  6 0.000279   192.168.10.1       192.168.10.138     TCP    78 61581 > xprtld [P
  7 0.000287   192.168.10.138     192.168.10.1       TCP    66 xprtld > 61581 [A
  8 0.000381   192.168.10.1       192.168.10.138     TCP   866 61581 > xprtld [P
  9 0.000388   192.168.10.138     192.168.10.1       TCP    66 xprtld > 61581 [A
 10 0.986554   192.168.10.138     192.168.10.1       TCP    78 xprtld > 61581 [P
 11 0.986685   192.168.10.138     192.168.10.1       TCP   866 xprtld > 61581 [F
 12 0.986731   192.168.10.1       192.168.10.138     TCP    66 61581 > xprtld [A
 13 0.986811   192.168.10.1       192.168.10.138     TCP    66 61581 > xprtld [A
 14 0.987110   192.168.10.1       192.168.10.138     TCP    66 61581 > xprtld [F

▷ Frame 10: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
▷ Ethernet II, Src: Vmware_a9:8b:56 (00:0c:29:a9:8b:56), Dst: Vmware_c0:00:08 (00:50:56:c0:00:08)
▷ Internet Protocol Version 4, Src: 192.168.10.138 (192.168.10.138), Dst: 192.168.10.1 (192.168.10.1)
▷ Transmission Control Protocol, Src Port: xprtld (5634), Dst Port: 61581 (61581), Seq: 1, Ack: 813, Len: 12
▷ Data (12 bytes)

0000  00 50 56 c0 00 08 00 0c  29 a9 8b 56 08 00 45 00   .PV..... )..V..E.
0010  00 40 6f 3f 40 00 40 06  35 9d c0 a8 0a 8a c0 a8   .@o?@.@. 5.......
0020  0a 01 16 02 f0 8d 31 9b  91 02 9f a4 22 9d 80 18   ......1. ...."...
0030  01 f7 96 0e 00 00 01 01  08 0a 00 43 ff 35 12 26   ........ ...C.5.&
0040  33 07 52 00 00 00 0c 00  00 00 20 03 00 00         3.R..... .. ...
```

Figure 5.13: Transmission of TEE association response header



```
  1 0.000000   Vmware_c0:00:08    Broadcast          ARP    60 Who has 192.168.10
  2 0.000015   Vmware_a9:8b:56    Vmware_c0:00:08    ARP    42 192.168.10.138 is a
  3 0.000123   192.168.10.1       192.168.10.138     TCP    78 61581 > xprtld [SYI
  4 0.000145   192.168.10.138     192.168.10.1       TCP    74 xprtld > 61581 [SYI
  5 0.000258   192.168.10.1       192.168.10.138     TCP    66 61581 > xprtld [ACI
  6 0.000279   192.168.10.1       192.168.10.138     TCP    78 61581 > xprtld [PSI
  7 0.000287   192.168.10.138     192.168.10.1       TCP    66 xprtld > 61581 [ACI
  8 0.000381   192.168.10.1       192.168.10.138     TCP   866 61581 > xprtld [PSI
  9 0.000388   192.168.10.138     192.168.10.1       TCP    66 xprtld > 61581 [ACI
 10 0.986554   192.168.10.138     192.168.10.1       TCP    78 xprtld > 61581 [PSI
 11 0.986685   192.168.10.138     192.168.10.1       TCP   866 xprtld > 61581 [FII
 12 0.986731   192.168.10.1       192.168.10.138     TCP    66 61581 > xprtld [ACI
 13 0.986811   192.168.10.1       192.168.10.138     TCP    66 61581 > xprtld [ACI
 14 0.987110   192.168.10.1       192.168.10.138     TCP    66 61581 > xprtld [FII

▷ Frame 11: 866 bytes on wire (6928 bits), 866 bytes captured (6928 bits)
▷ Ethernet II, Src: Vmware_a9:8b:56 (00:0c:29:a9:8b:56), Dst: Vmware_c0:00:08 (00:50:56:c0:00:08)
▷ Internet Protocol Version 4, Src: 192.168.10.138 (192.168.10.138), Dst: 192.168.10.1 (192.168.10.1)
▷ Transmission Control Protocol, Src Port: xprtld (5634), Dst Port: 61581 (61581), Seq: 13, Ack: 813, Len: 800
▷ Data (800 bytes)

0020  0a 01 16 02 f0 8d 31 9b  91 0e 9f a4 22 9d 80 19   ......1. ...."...
0030  01 f7 99 22 00 00 01 01  08 0a 00 43 ff 35 12 26   ..."....  ...C.5.&
0040  33 07 2d 2d 2d 2d 2d 42  45 47 49 4e 20 50 55 42   3.-----B EGIN PUB
0050  4c 49 43 20 4b 45 59 2d  2d 2d 2d 2d 0a 4d 49 49   LIC KEY- ----.MII
0060  43 49 6a 41 4e 42 67 6b  71 68 6b 69 47 39 77 30   CIjANBgk qhkiG9w0
0070  42 41 51 45 46 41 41 4f  43 41 67 38 41 4d 49 49   BAQEFAAO CAg8AMII
0080  43 43 67 4b 43 41 67 45  41 76 77 39 6f 79 32 31   CCgKCAgE Avw9oy21
0090  62 6b 50 6c 78 70 52 4f  6b 56 6d 63 43 0a 67 69   bkPlxpRO kVmcC.gi
00a0  6a 42 46 34 47 2f 43 5a  61 4d 54 69 6c 69 2b 30   jBF4G/CZ aMTili+0
00b0  48 63 4f 59 63 65 72 50  4f 31 50 41 48 74 78 67   HcOYcerP O1PAHtxg
00c0  72 49 77 7a 44 74 31 58  57 6c 70 51 62 34 34 57   rIwzDt1X WlpQb44W
00d0  4d 42 38 71 7a 79 2b 63  4f 77 6f 43 46 68 0a 72   MB8qzy+c OwoCFh.r
00e0  31 54 55 38 73 62 62 35  77 66 6f 6a 43 34 30 4d   1TU8sbb5 wfojC40M
```

Figure 5.14: Transmission of public key of remote device

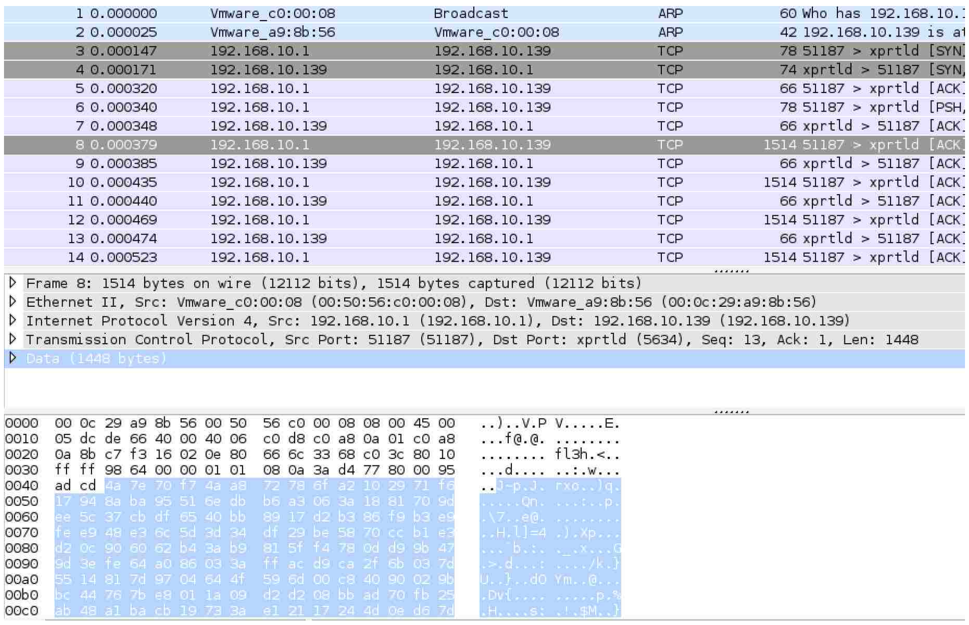


```
  1 0.000000   Vmware_c0:00:08   Broadcast         ARP     60 Who has 192.168.
  2 0.000025   Vmware_a9:8b:56   Vmware_c0:00:08   ARP     42 192.168.10.139 i
  3 0.000147   192.168.10.1      192.168.10.139    TCP     78 51187 > xprtld [
  4 0.000171   192.168.10.139    192.168.10.1      TCP     74 xprtld > 51187 [
  5 0.000320   192.168.10.1      192.168.10.139    TCP     66 51187 > xprtld [
  6 0.000340   192.168.10.1      192.168.10.139    TCP     78 51187 > xprtld [
  7 0.000348   192.168.10.139    192.168.10.1      TCP     66 xprtld > 51187 [
  8 0.000379   192.168.10.1      192.168.10.139    TCP   1514 51187 > xprtld [
  9 0.000385   192.168.10.139    192.168.10.1      TCP     66 xprtld > 51187 [
 10 0.000435   192.168.10.1      192.168.10.139    TCP   1514 51187 > xprtld [
 11 0.000440   192.168.10.139    192.168.10.1      TCP     66 xprtld > 51187 [
 12 0.000469   192.168.10.1      192.168.10.139    TCP   1514 51187 > xprtld [
 13 0.000474   192.168.10.139    192.168.10.1      TCP     66 xprtld > 51187 [
 14 0.000523   192.168.10.1      192.168.10.139    TCP   1514 51187 > xprtld [

▷ Frame 6: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
▷ Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Vmware_a9:8b:56 (00:0c:29:a9:8b:56)
▷ Internet Protocol Version 4, Src: 192.168.10.1 (192.168.10.1), Dst: 192.168.10.139 (192.168.10.139)
▷ Transmission Control Protocol, Src Port: 51187 (51187), Dst Port: xprtld (5634), Seq: 1, Ack: 1, Len: 12
▷ Data (12 bytes)

0000  00 0c 29 a9 8b 56 00 50  56 c0 00 08 08 00 45 00   ..)..V.P V.....E.
0010  00 40 c8 6e 40 00 40 06  dc 6c c0 a8 0a 01 c0 a8   .@.n@.@. .l......
0020  0a 8b c7 f3 16 02 0e 80  66 60 33 68 c0 3c 80 18   ........ f`3h.<..
0030  ff ff 16 48 00 00 01 01  08 0a 3a d4 77 80 00 95   ...H.... ..:.w...
0040  ad cd 44 00 00 00 0c 00  00 00 c0 51 13 00         ..D..... ...Q..
```

Figure 5.15: Transmission of TEE association request

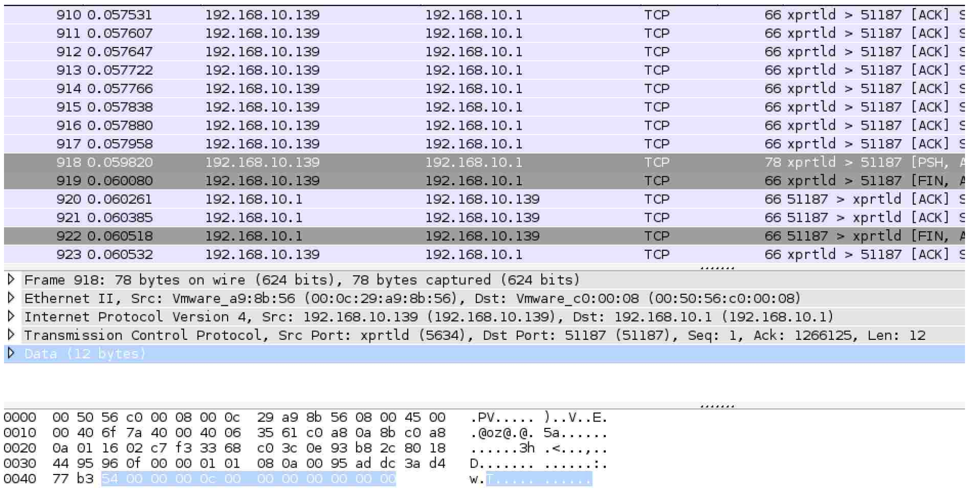
data is the public key of the initiating device, as shown in Figure 5.12. Once the key is received and the remote device has obtained approval for the association to proceed, a response is returned, shown in Figure 5.13. After the response header is transmitted, the public key of the remote device is transmitted, as depicted in Figure 5.14. While the public key of the remote device would normally be encrypted using the public key of the initiating device, the key was instead transmitted in the clear in order to more easily illustrate what is being sent during each step. A version that is fully compliant with the association algorithm presented previously has been developed and was tested to ensure proper operation.

After completing the association, an additional function was added to the ECE API that supported transmission of data files between two associated devices. This functionality was developed to prove to capability of this technology to provide a secure communication channel between two devices. This technique addresses the protection of DRM material that could be exchanged between devices registered with



```
 1 0.000000   Vmware_c0:00:08   Broadcast         ARP    60 Who has 192.168.10.
 2 0.000025   Vmware_a9:8b:56   Vmware_c0:00:08   ARP    42 192.168.10.139 is at
 3 0.000147   192.168.10.1      192.168.10.139    TCP    78 51187 > xprtld [SYN]
 4 0.000171   192.168.10.139    192.168.10.1      TCP    74 xprtld > 51187 [SYN,
 5 0.000320   192.168.10.1      192.168.10.139    TCP    66 51187 > xprtld [ACK]
 6 0.000340   192.168.10.1      192.168.10.139    TCP    78 51187 > xprtld [PSH,
 7 0.000348   192.168.10.139    192.168.10.1      TCP    66 xprtld > 51187 [ACK]
 8 0.000379   192.168.10.1      192.168.10.139    TCP  1514 51187 > xprtld [ACK]
 9 0.000385   192.168.10.139    192.168.10.1      TCP    66 xprtld > 51187 [ACK]
10 0.000435   192.168.10.1      192.168.10.139    TCP  1514 51187 > xprtld [ACK]
11 0.000440   192.168.10.139    192.168.10.1      TCP    66 xprtld > 51187 [ACK]
12 0.000469   192.168.10.1      192.168.10.139    TCP  1514 51187 > xprtld [ACK]
13 0.000474   192.168.10.139    192.168.10.1      TCP    66 xprtld > 51187 [ACK]
14 0.000523   192.168.10.1      192.168.10.139    TCP  1514 51187 > xprtld [ACK]

▷ Frame 8: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)
▷ Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Vmware_a9:8b:56 (00:0c:29:a9:8b:56)
▷ Internet Protocol Version 4, Src: 192.168.10.1 (192.168.10.1), Dst: 192.168.10.139 (192.168.10.139)
▷ Transmission Control Protocol, Src Port: 51187 (51187), Dst Port: xprtld (5634), Seq: 13, Ack: 1, Len: 1448
▷ Data (1448 bytes)

0000  00 0c 29 a9 8b 56 00 50  56 c0 00 08 08 00 45 00   ..)..V.P V.....E.
0010  05 dc de 66 40 00 40 06  c0 d8 c0 a8 0a 01 c0 a8   ...f@.@. ........
0020  0a 8b c7 f3 16 02 0e 80  66 6c 33 68 c0 3c 80 10   ........ fl3h.<..
0030  ff ff 98 64 00 00 01 01  08 0a 3a d4 77 80 00 95   ...d.... ..:.w...
0040  ad cd 4a 7e 70 f7 4a a8  72 78 6f a2 10 29 71 f6   ..J~p.J. rxo..)q.
0050  17 94 8a ba 95 51 6e db  b6 a3 06 3a 18 81 70 9d   .....Qn. ...:..p.
0060  ee 5c 37 cb df 65 40 bb  89 17 d2 b3 86 f9 b3 e9   .\7..e@. ........
0070  fe e9 48 e3 6c 5d 3d 34  df 29 be 58 70 cc b1 e3   ..H.l]=4 .).Xp...
0080  d2 0c 90 60 62 b4 3a b9  81 5f f4 78 0d d9 9b 47   ...`b.:. ._.x...G
0090  9d 3e fe 64 a0 86 03 3a  ff ac d9 ca 2f 6b 03 7d   .>.d...: ..../k.}
00a0  55 14 81 7d 97 04 64 4f  59 6d 00 c8 40 90 02 9b   U..}..dO Ym..@...
00b0  bc 44 76 7b e8 01 1a 09  d2 d2 08 bb ad 70 fb 25   .Dv{.... .....p.%
00c0  ab 48 a1 ba cb 19 73 3a  e1 21 17 24 4d 0e d6 7d   .H....s: .!.$M..}
```

Figure 5.16: Transmission of TEE association request



```
910 0.057531   192.168.10.139   192.168.10.1     TCP   66 xprtld > 51187 [ACK] S
911 0.057607   192.168.10.139   192.168.10.1     TCP   66 xprtld > 51187 [ACK] S
912 0.057647   192.168.10.139   192.168.10.1     TCP   66 xprtld > 51187 [ACK] S
913 0.057722   192.168.10.139   192.168.10.1     TCP   66 xprtld > 51187 [ACK] S
914 0.057766   192.168.10.139   192.168.10.1     TCP   66 xprtld > 51187 [ACK] S
915 0.057838   192.168.10.139   192.168.10.1     TCP   66 xprtld > 51187 [ACK] S
916 0.057880   192.168.10.139   192.168.10.1     TCP   66 xprtld > 51187 [ACK] S
917 0.057958   192.168.10.139   192.168.10.1     TCP   66 xprtld > 51187 [ACK] S
918 0.059820   192.168.10.139   192.168.10.1     TCP   78 xprtld > 51187 [PSH, A
919 0.060080   192.168.10.139   192.168.10.1     TCP   66 xprtld > 51187 [FIN, A
920 0.060261   192.168.10.1     192.168.10.139   TCP   66 51187 > xprtld [ACK] S
921 0.060385   192.168.10.1     192.168.10.139   TCP   66 51187 > xprtld [ACK] S
922 0.060518   192.168.10.1     192.168.10.139   TCP   66 51187 > xprtld [FIN, A
923 0.060532   192.168.10.139   192.168.10.1     TCP   66 xprtld > 51187 [ACK] S

▷ Frame 918: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
▷ Ethernet II, Src: Vmware_a9:8b:56 (00:0c:29:a9:8b:56), Dst: Vmware_c0:00:08 (00:50:56:c0:00:08)
▷ Internet Protocol Version 4, Src: 192.168.10.139 (192.168.10.139), Dst: 192.168.10.1 (192.168.10.1)
▷ Transmission Control Protocol, Src Port: xprtld (5634), Dst Port: 51187 (51187), Seq: 1, Ack: 1266125, Len: 12
▷ Data (12 bytes)

0000  00 50 56 c0 00 08 00 0c  29 a9 8b 56 08 00 45 00   .PV..... )..V..E.
0010  00 40 6f 7a 40 00 40 06  35 61 c0 a8 0a 8b c0 a8   .@oz@.@. 5a......
0020  0a 01 16 02 c7 f3 33 68  c0 3c 0e 93 b8 2c 80 18   ......3h .<...,..
0030  44 95 96 0f 00 00 01 01  08 0a 00 95 ad dc 3a d4   D....... ......:.
0040  77 b3 54 00 00 00 0c 00  00 00 00 00 00 00         w.T..... ......
```

Figure 5.17: Transmission of TEE association request

a FairPlay account, as detailed in Section 4.3.2.

As shown in Figure 5.15, the file transfer starts with the transmission of a request header. This header identifies this request as a file transmission request, provides a header length of 12 bytes, and a data length of 1266112. If this file size is compared with the TEE files listed in Table 5.3, it may be noted that this file is only 16 bytes larger than the TEE compiled on the MacBook Pro. In fact, that is the file being transferred in this example. The additional 16 bytes are the result of padding during the AES algorithm. Figure 5.16 shows the beginning of the transmission of the encrypted TEE file. The transmission of the file is then completed and a response header is generated and returned by the remote device, as illustrated in Figure 5.17.

5.3 ECE API Implementation Results

Although it is difficult to illustrate the full functionality of this software emulator through figures and tables, the examples provided demonstrated each of the capabilities necessary to support the use cases presented in Chapter 4. The randomness values presented on the encrypted TEE files show the ability of this tool to uniquely protect a TEE resident on a mobile device, a need addressed in Section 4.3.1. Additionally, the use of the TEE header provides the functionality to support updates, authorized modifications, and additions to the TEE. The association performed illustrates how multiple devices can exchange device keys and authentication information, thereby allowing the device to uniquely authenticate itself not only to other device, but also to a network. This capability addresses the need for stronger authentication presented in Section 4.3.3. The association mechanism also facilitates the ability to securely transfer information between multiple devices, which can lead to the transfer of DRM protected materials between different devices on an account, per Section 4.3.2.

The results in this section have proven the strength of the ECE in addressing some of the most critical needs in the world of mobile security. The API presented provides all the functionality need to support authentication, secure communications, and unique protection of system critical code and data values. Incorporation of this architecture into any mobile framework will greatly enhance its ability to provide reliable mobile security that is far ahead of anything currently on the market today.

Chapter 6

Conclusions

Mobile security is a complex and ever changing research area. As detailed in Section 2, there are an immense number of solutions, specifications, protocols, and methodologies available which attempt to address a variety of areas in the realm of mobile security. These various implementations cover security related issues from device manufacturing all the way to communications and device interactions. As evidenced by the amount of research being done on mobile devices, this is definitely a growing area that will continue to face a number of challenges with its incredible growth.

The research presented in this paper was conducted first with a desire to understand all of the many security related research areas dealing with mobile devices. Once a solid pool of knowledge was generated, the research transitioned into a discovery process of how PUF technology might be leveraged to address some of the most pressing issues facing users, developers, cellular provides, and a host of other players. Does PUF technology provide any features or capabilities that are not already available? If so, could they be used to enhance current security technologies to provide greater protection of data and communications?

Over the course of conducting this research, these questions were answered. This has been accomplished by first conducting an in-depth analysis of modern mobile security technologies has been presented. Additionally, a novel new architecture has been proposed for providing enhanced security on mobile devices. The use of PUFs to generate unique-per-device keys is a substantial finding that provides researchers with a unique opportunity to strengthen mobile security in a way never done before. The API provided gives software developer a new set of tools for combating cyber crime in ways never before imagined. The findings of this research have shown that PUF technology can not only aid, but can greatly enhance, the mobile security world.

6.1 Additional Research

The results shown as a result of this research have proven that PUF technology has the required features and abilities to aid in the creation of significantly more secure mobile devices. This research has also lead to the discovery of a number of additional topics for further research. For instance, homes are starting to become more and more advanced with cellular accessible security devices, smart appliances, and remotely controlled environmental systems. Automobiles are also starting to come stock with 3G and 4G compatible computer systems that are able to monitor vehicle conditions, provide real-time traffic information, download media files for viewing on internal entertainment systems, and an array of other electronic enhancements. While one area of automotive security was addressed earlier, there are still many other areas that must be researched and may benefit from the capabilities provided by an ECE architecture.

Additional research can also be conducted in the same areas presented thus far. Although currently available hardware platforms have made a full hardware solution impractical at this time, this is a step that still must be performed. The software emulator discussed was the right first step, but it is not the only step necessary. As mentioned previously, Xilinx has created a new FPGA platform with an embedded dual-core ARM processor called Zynq. Although a bug currently prohibits access to the TrustZone secure world, and thus the cryptographic unit, a promised fix may provide researchers with a viable platform for testing of a full hardware implementation [85].

As additional research areas are explored, it may also be discovered that the ECE architecture provides additional benefits beyond those discussed herein. As mentioned in Section 4.3.3, processors like the baseband processor may also find a use for an ECE in device authentication, secure cellular communication, or for a full on-board authentication mechanism, such as the architecture described in [48]. While a determination of every possible application for the ECE is not possible without years of additional research, it has been shown through this research that PUF-enhanced cryptographic engines can be used for dramatically increased security.

6.2 Future of Mobile Security

In an almost constant “tug-of-war” battle, mobile security researchers attempt to counter attacks made by black-hat and white-hat hackers alike. Software security researchers are aided in their efforts by new hardware security features that are added to SoC designs with nearly every revision. The inevitability of new features is driven by an equally powerful certainty: their defeat.

Attackers will continue to find methods for circumventing, or out-right breaking, the latest and greatest security metric developed. The results of such exploits will continue to be accompanied with an ever increasing price tag, both in terms of financial lose and development time for updates. In a recent article, Christopher Soghoian describe the security issues with Android as being, ”..like a really dry forest, and it’s just waiting for a match [86].” As such, security researchers can no longer afford to wait around for current methods to be broken before developing enhanced security methods.

While developers may claim that current approaches are sufficient, this ideology has never worked in the past and will never work in the future. Current security simply is not sufficient. Use of the ECE architecture presented will allow device manufacturers and security researchers to take a significant leap forward in their battle against cyber criminals and hack-tivist. It does not provide an all-power, never to be defeated solution, but it has the potential to put us in the drivers seat with the foot on the gas!

Appendices

Appendix A

Enhanced Cryptographic Engine API

```
#ifndef _ece_interface_h
#define _ece_interface_h

typedef enum aes_op_type {
    AES_128_CBC     = 0x00,
    AES_128_CFB     = 0x01,
    AES_128_CFB1    = 0x02,
    AES_128_CFB8    = 0x03,
    AES_128_ECB     = 0x04,
    AES_128_OFB     = 0x05,
    AES_192_CBC     = 0x06,
    AES_192_CFB     = 0x07,
    AES_192_CFB1    = 0x08,
    AES_192_CFB8    = 0x09,
    AES_192_ECB     = 0x0A,
    AES_192_OFB     = 0x0B,
    AES_256_CBC     = 0x0C,
    AES_256_CFB     = 0x0D,
    AES_256_CFB1    = 0x0E,
    AES_256_CFB8    = 0x0F,
    AES_256_ECB     = 0x10,
    AES_256_OFB     = 0x11,
    AES128          = 0x12,
    AES192          = 0x13,
    AES256          = 0x14,
} aes_op_type;

typedef enum aes_key_size {
    AES_KEY_SIZE_128     = 0x0100;
    AES_KEY_SIZE_192     = 0x0200;
    AES_KEY_SIZE_256     = 0x0300;
```

```
} aes_key_size;

typedef enum rsa_op_type {
    RSA_STANDARD            = 0x0,
    RSA_NO_PADDING          = 0x2,
    RSA_SSL2_PADDING        = 0x1,
    RSA_PKCS1_1_5_PADDING   = 0x3,
    RSA_PKCS1_OAEP_PADDING  = 0x4,
} rsa_op_type;

typedef enum rsa_key_size {
    RSA_KEY_SIZE_1024       = 0x0000,
    RSA_KEY_SIZE_2048       = 0x0100,
    RSA_KEY_SIZE_4096       = 0x0200,
} rsa_key_size;

typedef enum rsa_block_size {
    RSA_BLOCK_SIZE_1024     = 0x0000,
    RSA_BLOCK_SIZE_2048     = 0x0100,
    RSA_BLOCK_SIZE_4096     = 0x0200,
} rsa_block_size;

typedef enum crypto_key_type {
    CRYPTO_KEY_AES_INTERNAL         = 0x000000;
    CRYPTO_KEY_AES_CUSTOM           = 0x100000;
    CRYPTO_KEY_RSA_INTERNAL         = 0x200000;
    CRYPTO_KEY_RSA_PUBLIC_INTERNAL  = 0x300000;
    CRYPTO_KEY_RSA_PRIVATE_INTERNAL = 0x400000;
    CRYPTO_KEY_RSA_PUBLIC_CUSTOM    = 0x500000;
    CRYPTO_KEY_RSA_PRIVATE_CUSTOM   = 0x600000;
} crypto_key_type;

typedef enum crypto_method {
    CRYPTO_ENCRYPT  = 0x00000000,
    CRYPTO_DECRYPT  = 0x80000000,
    CRYPTO_GENERATE = 0xC0000000,
} crypto_method;

typedef enum association_method {
    ASSOCIATION_WIFI        = 0x0;
    ASSOCIATION_BLUETOOTH   = 0x1;
    ASSOCIATION_USB         = 0x2;
    ASSOCIATION_NFC         = 0x3;
}

/*  Function:   ECE_AES_CRYPTO
    Parameters: input - pointer to a buffer containing the data to be processed
                output - pointer to a buffer inwhich to store the result
                length - number of bytes to process
                op_type - number representing the requested AES operation, keysize, key type, and crypto method
                key - pointer to a buffer containing the key to be used, if applicable
                iv - pointer to a buffer containing the iv to be used, if applicable
    Result:     int32_t, 0 representing success; otherwise negative value representing error
    Purpose:    The purpose of this function is provide the capability of performing and AES operation using
          either a custom key or the internal secret key.
*/
int32_t ECE_AES_CRYPTO( uint8_t *input,
                        uint8_t *output,
                        uint32_t length,
                        uint32_t op_type,
                        uint8_t *key,
                        uint8_t *iv );

/*  Function:   ECE_RSA_CRYPTO
    Parameters: input - pointer to a buffer containing the data to be processed
                output - pointer to a buffer inwhich to store the result
                length - number of bytes to process
                op_type - number representing the requested RSA operation, keysize, key type, and crypto method
                key - pointer to a buffer containing the key to be used, if applicable
    Result:     int32_t, 0 representing success; otherwise negative value representing error
    Purpose:    The purpose of this function is provide the capability of performing an RSA operation using either
           a custom key or the internal secret key.
 */
int32_t ECE_RSA_CRYPTO( uint8_t *input,
                        uint8_t *output,
                        uint32_t length,
                        uint32_t op_type,
                        uint8_t *key );

/*  Function:   ECE_AES_KEYGEN
    Parameters: key_size - value representing the size of the AES key to generate
    Result:     uint8_t *, a pointer to a valid key; otherwise NULL
    Purpose:    The purpose of this function is to provide a mechanism for generating ad-hoc AES keys that can be
          used for protection of communications or system data.
*/
uint8_t * ECE_AES_KEYGEN( uint32_t key_size );

/*  Function:   ECE_RSA_KEYGEN
 Parameters:    key_size - value representing the size of the RSA keys to generate
 Result:        uint8_t **, a pointer to a valid public/private key-pair; otherwise NULL
 Purpose:       The purpose of this function is to provide a mechanism for generating ad-hoc RSA key pairs that
      can be used for protection of communications or system data.
 */
uint8_t ** ECE_RSA_CRYPTO( uint32_t key_type );

/*  Function:   ECE_DEVICE_AUTHENTICATION
    Parameters: key - the proposed public key of the device being authenticated
                method - the protocol used to authenticate
                address - the address of the device requesting authentication
    Result:     int32_t, 0 representing success; otherwise a negative value representing error authentication
          response for the device.  This will result in a direct exchange of information with an onboard SE, such
          as the SIM card.  No information is passed to this call as control of this functionality should mostly
          exist outside the device.
*/
int32_t ECE_DEVICE_AUTHENTICATION( uint8_t *key,
                                   uint32_t method,
                                   uint8_t *address );

/*  Function:   ECE_DEVICE_ASSOCIATION
    Parameters: device_address - address of device with which to associate
    Result:     int32_t, 0 representing success; otherwise a negative value representing error
    Purpose:    The purpose of this function is to provide a method for associating, i.e. exchanging public keys,
          with another device.  This can happen over any supported communications interface, which may be an added
          parameter in later versions.  This method will fail if the system is not in "discovery" mode.
 */
int32_t ECE_DEVICE_ASSOCIATION( uint32_t method,
                                 uint8_t *device_address );

/*  Function:   ECE_DEVICE_ASSOCIATION
 Parameters: device_address - address of device with which to associate
 Result:     int32_t, 0 representing success; otherwise a negative value representing error
 Purpose:    The purpose of this function is to provide a method for associating, i.e. exchanging public keys,
       with another device.  This can happen over any supported communications interface, which may be an added
       parameter in later versions.  This method will fail if the system is not in "discovery" mode.
 */
int32_t ECE_DEVICE_DISASSOCIATION( uint32_t method,
                                   uint8_t *device_address );

/*  Function:   ECE_TEE_VERIFICATION
    Parameters: load_address - address at which to load the decrypted TEE
    Result:     int32_t, 0 representing success; otherwise a negative value representing error
    Purpose:    The purpose of this function is to verify that a TEE exist and to decrypt it from memory and load
          it into the provided address.  This is done by looking for a TEE header at a predefined address.  If the
          header exist, its information will be used to decrypt the TEE and then load it into memory.  If not, an
          error is returned.
*/
int32_t ECE_TEE_VERIFICATION( uint32_t *load_address );

/*  Function:   ECE_TEE_INITIALIZATION
    Parameters: None
    Result:     int32_t, 0 represents success; otherwise a negative value representing error
    Purpose:    The purpose of this function is to initialize a TEE environment.  When the system boots for the
          first time, no firmware or TEE will typically be loaded. This function will accept a new TEE, encrypt it
          at the correct location, and  then generate a corresponding TEE header that may be used to boot the TEE.
*/
int32_t ECE_TEE_INITIALIZATION( void );

/*  Function:   ECE_GENERATE_SECRET_KEY
    Parameters: None
    Result:     uint8_t *, pointer to the location of the new secret key
    Purpose:    The purpose of this function is to instruct the cryptographic unit to generate the system secret
          key by executing the PUF and storing the result.  This is done to ensure that the key is not loaded
          automatically in case access to the secureROM is somehow bypassed.  Operations with the system secret
          keys should only be granted when the system is in a known "good" state.
*/
uint8_t * ECE_GENERATE_SECRET_KEY( void );

/*  Function:   ECE_GENERATE_RSA_KEY_PAIR
    Parameters: None
    Result:     uint8_t *, pointer to the location of the new RSA key pair
    Purpose:    The purpose of this function is to instruct the cryptographic unit to generate the system RSA key
          pair by executing the PUF and storing the result.  This is done to ensure that the key is not loaded
          automatically in case access to the secureROM is somehow bypassed.  Operations with the system secret
          keys should only be granted when the system is in a known "good" state.
*/
uint8_t * ECE_GENERATE_RSA_KEY_PAIR( void );

/*  Function:   ECE_CHANGE_SECRET_KEY
    Parameters: None
    Result:     int32_t, 0 represents success; otherwise a negative value representing error
    Purpose:    The purpose of this function is to provide a mechanism for changing the system AES secret key in
          the event of suspected disclosure of the current key, or simply as a precautionary measure on a periodic
          basis.
*/
```

```
int32_t ECE_CHANGE_SECRET_KEY( void );

/* Function:   ECE_CHANGE_RSA_KEY_PAIR
   Parameters: None
   Result:     int32_t, 0 represents success; otherwise a negative value representing error
   Purpose:    The purpose of this function is to provide a mechanism for changing the system RSA key pair in the
          event of suspected disclosure of the current keys, or simply as a precautionary measure on a periodic
         basis.
 */
int32_t ECE_CHANGE_RSA_KEY_PAIR( void );

#endif
```

References

[1] GlobalPlatform *TEE system architecture, version 1.0* http://www.http://www.globalplatform.org/specificationsdevice.asp, December 2011.

[2] Trusted Computing Group, *TCG Mobile Trusted Module Specification, 1.0*, 7.02 edition, TCG, 2010.

[3] R. York *A new foundation for CPU systems security* Technical report, ARM Limited, 2003.

[4] ARM Limited *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, April 2009.

[5] Cisco *Cisco visual networking index: Global mobile data traffic forecast update, 2011-2016* http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html.

[6] ABI *Users will download 44 billion mobile apps by 2016* http://techcrunch.com/2011/04/28/users-will-download-44-billion-mobile-apps-by-2016/.

[7] IDC *Appcelerator/idc 1q11 mobile app developer survey report* http://www.appcelerator.com/company/survey-results/.

[8] A. Vuong *Southwest airlines iphone app vulnerable to hackers, study says* http://blogs.denverpost.com/techknowbytes/2012/02/09/southwest-airlines-iphone-app-vulnerable-to-hackers-study-says/3264/.

[9] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, *Experimental security analysis of a modern automobile*, in Security and Privacy (SP), 2010 IEEE Symposium on, may 2010, pp. 447 –462.

[10] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, *Comprehensive experimental analyses of automotive attack surfaces*, in Proceedings of the 20th USENIX conference on Security, SEC'11, USENIX Association, Berkeley, CA, USA, 2011, pp. 6–6.

[11] U. S. G. A. Office *Fda should expand its consideration of information security for certain types of devices* http://gao.gov/assets/650/647767.pdf.

[12] J. Radcliffe, *Hacking medical devices for fun and insulin: Breaking the human scada system*, in Black Hat USA, July 2011.

[13] R. Borgaonkar, N. Golde, and K. Redon, *Femtocells: A poisonous needle in the operator's hay stack*, in Black Hat USA, July 2011.

[14] J. Lee, D. Lim, B. Gassend, G. Suh, M. van Dijk, and S. Devadas, *A technique to build a secret key in integrated circuits for identification and authentication applications*, in VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on, june 2004, pp. 176 – 179.

[15] G. Suh and S. Devadas, *Physical unclonable functions for device authentication and secret key generation*, in Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE, june 2007, pp. 9 –14.

[16] GlobalPlatform *Frequently asked questions* http://www.globalplatform.org/mediafaq.asp.

[17] GlobalPlatform *Why the Mobile Industry is Evolving Towards Security.*

[18] GlobalPlatform *The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market* http://www.globalplatform.org/documents/GlobalPlatform_TEE_White_Paper_Feb2011.pdf, February 2011.

[19] ARM Limited *TrustZone API specification* http://www.lcs.syr.edu/faculty/yin/teaching/CIS700-sp11/TrustZone_API_3.0_Specification.pdf, February 2009.

[20] GlobalPlatform *TEE client API specification, version 1.0* http://www. http://www.globalplatform.org/specificationsdevice.asp, July 2010.

[21] GlobalPlatform *TEE internal API specification, version 1.0* http://www. http://www.globalplatform.org/specificationsdevice.asp, December 2011.

[22] *Trusted environment: OMTP TR0, Version 1.2*, 2009.

[23] *Advanced trusted environment: OMTP TR1, Version 1.1*, 2009.

[24] *The history of wac* http://www.wacapps.net/history.

[25] D. A. Dai Zovi, *Apple iOS 4 security evaluation*, in Black Hat USA, July 2011.

[26] Microsoft *Bitlocker drive encryption overview* http://windows.microsoft.com/en-us/windows-vista/BitLocker-Drive-Encryption-Overview.

[27] Microsoft *Protecting the pre-os environment with uefi* http://blogs.msdn.com/b/b8/archive/2011/09/22/protecting-the-pre-os-environment-with-uefi.aspx.

[28] T. C. Group, *TCG Mobile Reference Architecture, 1.0*, TCG, 2007.

[29] Trusted Computing Group, *Mobile Trusted Module 2.0 Use Cases 1.0*, TCG, 2011.

[30] *Open mobile alliance* http://www.openmobilealliance.org.

[31] *3rd generation partnership program* http://www.3gpp.org.

[32] J. Azema and G. Fayad *M-shield mobile security technology: Making wireless secure* Technical report, Texas Instruments, feb 2008.

[33] *Securemsm security suite* http://www.gi-de.com/gd_media/en/documents/brochures/mobile_security_2/MobiCore-secure-os.pdf.

[34] G. . Devrient *Mobicore: Giesecke & devrient's secure os for arm trustzone technology* http://www.gi-de.com/gd_media/media/en/documents/brochures/mobile_security_2/MobiCore_EN.pdf, 2010.

[35] Giesecke & Devrient *Mobicore - securing smartphone applications* `http://www.gi-de.com/gd_media/media/documents/brochures/mobile_security_2/MobiCore.pdf`, 2012.

[36] J. Winter, *Trusted computing building blocks for embedded linux-based arm trustzone platforms*, in Proceedings of the 3rd ACM workshop on Scalable trusted computing, STC '08, ACM, New York, NY, USA, 2008, pp. 21–30.

[37] K. Dietrich and J. Winter, *Secure boot revisited*, in Proceedings of the 2008 The 9th International Conference for Young Computer Scientists, ICYCS '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 2360–2365.

[38] K. Dietrich and J. Winter, *Implementation aspects of mobile and embedded trusted computing*, in Proceedings of the 2nd International Conference on Trusted Computing, Trust '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 29–44.

[39] K. Dietrich and J. Winter, *Towards customizable, application specific mobile trusted modules*, in Proceedings of the fifth ACM workshop on Scalable trusted computing, STC '10, ACM, New York, NY, USA, 2010, pp. 31–40.

[40] J. Winter, *Experimenting with arm trustzone; or: How i met a friendly piece of hardware*, in Proceedings of the 11th IEEE International Conference on Trust, Security, and Privacy in Computing and Communications, TrustCom 2012, 2012.

[41] J. Grossschadl, T. Vejda, and D. Page, *Reassessing the tcg specifications for trusted computing in mobile and embedded systems*, in Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust, HST '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 84–90.

[42] O. Aciicmez, A. Latifi, J.-P. Seifert, and X. Zhang, *A trusted mobile phone prototype*, in Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE, jan. 2008, pp. 1208 –1209.

[43] X. Zhang, O. Acimez, and J. pierre Seifert, *A trusted mobile phone reference architecture via secure kernel*, in In Proceedings of the ACM workshop on Scalable Trusted Computing, 2007, pp. 7–14.

[44] S. Choi, J. Han, J. Lee, J. Kim, and S. Jun, *Implementation of a tcg-based trusted computing in mobile device*, in Proceedings of the 5th international conference on Trust, Privacy and Security in Digital Business, TrustBus '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 18–27.

[45] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger, *Measuring integrity on mobile phone systems*, in Proceedings of the 13th ACM symposium on Access control models and technologies, SACMAT '08, ACM, New York, NY, USA, 2008, pp. 155–164.

[46] M. Kim, H. Ju, Y. Kim, J. Park, and Y. Park, *Design and implementation of mobile trusted module for trusted mobile computing*, Consumer Electronics, IEEE Transactions on **56** (2010), 134 – 140.

[47] D. Bauder, *An anti-counterfeiting concept for current systems*, in Research report PTK-11990 Sandia National Laboratory, 1983.

[48] O. Al Ibrahim and S. Nair, *Cyber-physical security using system-level pufs*, in Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International, july 2011, pp. 1672 –1676.

[49] S. Devadas, E. Suh, S. Paral, R. Sowell, T. Ziola, and V. Khandelwal, *Design and implementation of puf-based unclonable rfid ics for anti-counterfeiting and security applications*, in RFID, 2008 IEEE International Conference on, april 2008, pp. 58 –64.

[50] W. Choi, S. Kim, Y. Kim, Y. Park, and K. Ahn, *Puf-based encryption processor for the rfid systems*, in Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, 29 2010-july 1 2010, pp. 2323 –2328.

[51] C. Bohm, M. Hofer, and W. Pribyl, *A microcontroller sram-puf*, in Network and System Security (NSS), 2011 5th International Conference on, sept. 2011, pp. 269 –273.

[52] B. Gassend, *Physical random functions*, in Master's Thesis, Massachusetts Institute of Technology, january 2003.

[53] C. Bosch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls, *Efficient helper data key extractor on fpgas*, in Proceeding sof the 10th international workshop on Cryptographic Hardware and Embedded Systems, CHES '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 181–197.

[54] C.-E. Yin and G. Qu, *Lisa: Maximizing ro puf's secret extraction*, in Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on, june 2010, pp. 100 –105.

[55] M. Majzoobi, F. Koushanfar, and S. Devadas, *Fpga puf using programmable delay lines*, in IEEE Workshop on Information Forensics and Security, 2010.

[56] R. Maes, P. Tuyls, and I. Verbauwhede, *Low-overhead implementation of a soft decision helper data algorithm for sram pufs*, in Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems, CHES '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 332–347.

[57] M.-D. Yu and S. Devadas, *Secure and robust error correction for physical unclonable functions*, Design Test of Computers, IEEE **27** (2010), 48 –65.

[58] Z. Paral and S. Devadas, *Reliable and efficient puf-based key generation using pattern matching*, in Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on, june 2011, pp. 128 –133.

[59] GeoHot *limera1n* `http://limera1n.com`.

[60] G. Heiser, *The role of virtualization in embedded systems*, in Proceedings of the 1st workshop on Isolation and integration in embedded systems, IIES '08, ACM, New York, NY, USA, 2008, pp. 11–16.

[61] J. Matthews *Virtualization and componentizationi in embedded systems and how it will change the way you engineer* Technical report, Open Kernel Labs, Inc., 2008.

[62] G. Heiser *Virtualization for embedded systems* Technical report, Open Kernel Labs, Inc., 2007.

[63] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, *sel4: formal verification of an os kernel*, in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, ACM, New York, NY, USA, 2009, pp. 207–220.

[64] G. Heiser, *Secure embedded systems need microkernels*, Usenix **30** (2005), 9–13.

[65] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters, *Towards trustworthy computing systems: taking microkernels to the next level*, SIGOPS Oper. Syst. Rev. **41** (2007), 3–11.

[66] G. Heiser *The motorola evoke qa4, a case study in mobile virtualization* Technical report, Open Kernel Labs, Inc., 2009.

[67] Trusted Logic *Trusted logic's trusted foundatioins first to incorporate the latest globalplatform tee specification* `http://www.tl-mobility.com/IMG/pdf/Trusted_Foundations_FINAL.pdf`, 2011.

[68] Texas Instruments *Ti achieves first nexflix hd certification* http://investor.ti.com/releasedetail.cfm?ReleaseID=589023, 2011.

[69] Trusted Logic *Gemalto and texas instruments collaborate to deliver secure mobile financial services in mobile devices* http://www.tl-mobility.com/IMG/pdf/Gemalto_TI_Integration_CTIA10_Final.pdf, 2010.

[70] Giesecke & Devrient *Digital rights management - white paper* http://www.gi-de.com/gd_media/media/en/documents/brochures/mobile_security_2/MobiCore-Digital-Rights-Management_White-Paper.pdf, 2011.

[71] G. . Devrient *Digital rights management* http://www.gi-de.com/gd_media/media/en/documents/brochures/mobile_security_2/MobiCore-Digital-Rights-Management.pdf, 2011.

[72] AuthenTec *Mobicore secured digital rights management on mobile devices* http://www.gi-de.com/gd_media/media/documents/brochures/mobile_security_2/MobiCore_Secured_Digital_Rights_Management_on_Mobile_Devices.pdf, 2012.

[73] AuthenTec *Mobicore secured mobile vpn for android* http://www.gi-de.com/gd_media/media/documents/brochures/mobile_security_2/MobiCore_Secured_Mobile_VPN_for_Android_Devices.pdf, 2012.

[74] Charbax *Samsung galaxy s2 may be the first smartphone with full arm trustzone support for enabling 100% security in everything online* http://armdevices.net/2012/05/04/samsung-galaxy-s3-may-be-the-first-smartphone-with-full-arm-trustzone-support-for-enabling-100-security-in-everything-online/, May 2012.

[75] T. Alves and D. Felton, *TrustZone: Integrated hardware and software security*, Information Quarterly **3** (2004), 18–24.

[76] ARM Limited *ARM Architecture Reference Manual*, armv7-a and armv7-r edition, 2010.

[77] IBM *Ibm introduces chip morphing technology: Self-managing semiconductors physically reconfigure themselves* http://www-304.ibm.com/jct03001c/press/us/en/pressrelease/7246.wss, 2004.

[78] M. Braga *How efuses work and why they're not as bad as you think* `http://www.tested.com/tech/585-how-efuses-work-and-why-theyre-not-as-bad-as-you-think/`, July 2010.

[79] R. Torrance and D. James, *The state-of-the-art in semiconductor reverse engineering*, in Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE, june 2011, pp. 333 –338.

[80] P. Simons, E. van der Sluis, and V. van der Leest, *Buskeeper pufs, a promising alternative to d flip-flop pufs*, in Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on, june 2012, pp. 7 –12.

[81] Wikipedia *Fairplay* `http://en.wikipedia.org/wiki/FairPlay`.

[82] S. Jobs *Thoughts on music* `http://web.archive.org/web/20080517114107/http://www.apple.com/hotnews/thoughtsonmusic`.

[83] E. Barkan, E. Biham, and N. Keller, *Instant ciphertext-only cryptanalysis of gsm encrypted communication*, J. Cryptol. **21** (2008), 392–429.

[84] OpenSSL *Openssl homepage* `http://www.openssl.org`.

[85] I. Xilinx *Zynq-7000 ap soc devices - silicon revision differences errata* `http://www.xilinx.com/support/answers/47916.htm`, 2012.

[86] C. Ratcliffe *Fragmentation leaves android phones vulnerable to hackers, scammers* `http://www.washingtonpost.com/business/technology/android-phones-vulnerable-to-hackers/2013/02/01/f3248922-6723-11e2-9e1b-07db1d2ccd5b_story.html`, February 2013.