Secure Mobile Association and Data Protection with Enhanced Cryptographic Engines

Matthew Areno
University of New Mexico
Department of Electrical and Computer Engineering
Albuquerque, New Mexico, USA
mareno@ece.unm.edu

Jim Plusquellic
University of New Mexico
Department of Electrical and Computer Engineering
Albuquerque, New Mexico, USA
jimp@ece.unm.edu

***Abstract*—Mobile devices are quickly becoming the most used and abundant electronics used today. The ability to positively identify a mobile device and associate it with a specific person or entity is critical to deter theft and provide enhanced security of stored data. This paper presents a novel method for providing device identification through the use of PUF technology that uniquely labels every supporting device. This method can alo be leveraged to create secure communications between devices and protection of on-board data.**

***Keywords*-Trusted Execution Environment, mobile security, physically unclonable functions, AES, authentication, association, cryptographic engines**

I. Introduction

Individuals are using mobile devices to accomplish an ever-increasing number of tasks ranging from simple games to financial transactions to controlling household appliances. Further, mobile processors are finding their way into an assortment of non-cellular applications, with one of the most prevalent being automotives. Regardless of the overall use of the electronic, there is almost always some need for interaction with other devices and for the sharing of data.

Regrettably the security of these devices are often not sufficiently considered until something goes wrong. A prime example of this was exposed by Koscher et.al. [1] and Checkoway et. al. [2] in their research on the security of embedded systems in automotive vehicles. Their research resulted in the discovery of a variety of serious security holes that would allow a hacker to control or bypass safety critical system, re-write firmware, or even steal the vehicle. Additionally, this research was only conducted on a small number of vehicles. It is anyone's guess as to how wide-spread this problem is in the automotive world at large!

Mobile devices, such as cellular telephones, are also susceptible to a huge number of attack methods that can result in loss of Personally Identifiable Information (PII), loss of service, unauthorized financial transactions, or financial charges due to data and/or cellular usage. This list is not intended to detail all possible consequences, nor are such attacks confined to mobile or automotive systems. Such concerns also exist in the areas of health-care system, cellular infrastructure, and Digital Rights Management (DRM). Each of these areas, as well as many more, are affected by the security mechanisms developed and implemented by mobile processor manufacturers and software designers.

Mobile security is further complicated by a new methodology known as ”Bring Your Own Device”, or BYOD. This concept describes the need of most people today to have some form of mobile communication available not only for home use, but also for use at work. Rather than requiring individuals to carry two devices, it is usually desirous to have a single device that can be used for both personal and work purposes.

The primary issue is that mobile devices are often personal, in that they contain information and/or applications that are personal to the user that may not fit well with corporate network requirements. Corporations often employ strict controls over devices connected to their networks. Such controls on a personal device being used outside or work is fairly impractical. While methods of enforcing corporate protocols on personal devices are available, they must work in a fashion that does not prohibit use of personal data or applications when the user is not at work, nor that would result in the loss of data if and when an employee leave the company.

In this paper, we present a novel application of a proven capability that can provide enhanced security of mobile and embedded devices while still providing the flexibility needed to support a BYOD environment. Physical Unclonable Functions (PUFs) provide a mechanism for generating unique-per-device values that can be used as secret, symmetrical keys, or can provide random seed values for asymmetric key generation engines. Incorporation of this functionality into a cryptographic unit for modern mobile processors can provide a mechanism for protecting sensitive data on a device, as well as supporting encrypted and secure communications between devices.

In Section II, we present supporting background research showing the viability of PUFs to create the necessary random values for key generation, as well as various research idea currently used in mobile security. In Section III, we present the design of an Enhanced Cryptographic Engine (ECE) that will facilitate the implementation of secure communications and data security in mobile processor applications. In Section IV, details of a software implementation of the ECE are presented along with results of how this worked and what functionality was possible. In Section V, we discuss future work in this area and briefly discuss additional applications and security concerns that can benefit from this new architecture.

II. Background

The concept of Physical Unclonable Functions (PUFs) was originally proposed in 1983 by D.W. Bauder at Sandia National Laboratory as a method for identifying counterfeiting, or alteration, of a chip design [3]. 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. [4] and then in 2007 by Suh and Devadas [5]. Their goal was to generate a key that could be used to support 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.

Both of these approaches made use of challenge/response pairs to positively identify given devices. The device is originally presented with a challenge that is then encrypted using the device specific key and returned to the challenger. This result is called a response and is maintained somewhere for comparison at a later time. To identify a device, one of the previously provided challenges is again issued and the result is compared with stored challenge/response pairs. If the response matches what is stored, the device is authenticated. If not, the device is not granted access to the requested resource, such as a Wi-Fi network.

A similar approaches was proposed by Ibrahim and Nair [6], except that the challenges where sent to a number of PUF enabled elements on the system. These responses are combined to provide a system-wide result, thus validating the system as a whole rather than a single element. This approach also utilized a third-party that would store challenge/response pairs for each device to use for comparison at a later time.

While the feasibility of these approaches is solid, the implementation can result is several problems. The storage requirements for challenge/response pairs can be considerable. To provide greater security, challenges should only be re-used if absolutely necessary. Therefore, each device would need to have a significant number of challenge/response pairs stored externally. There is then a question of what happens once all pairs have been utilized. Solutions vary from denial-of-service, which is not very sensible, to generation of new challenge/response pairs. While generation of new pair may sound easy, regaining access to the processor may be impossible. Further, leaving open access to a device such that anyone can provide a challenge and then record the response is impractical from a security standpoint. As a result, no feasible solution has yet been presented for how to support these devices once all available challenge/response pairs have been used.

A variety of software implementations and specification also exist that are directed at addressing issues of mobile security and providing protection for user data and transactions. ARM Limited first presented a new security suite known as TrustZone in 2003 and is currently on its 3^{rd} version [7]. TrustZone (TZ) encompasses a collection of modifications and support features that provide hardware enforced, software execution isolation on ARM core processors. This allows software to execute in one of two execution environments: the Secure world or the Non-Secure world. Each is provide with separate memory structures, interrupt support, debug isolation, and a variety of other features.

TZ can then be leveraged to implement a number of mobile security specification, such as Global Platform's Trusted Execution Environment (TEE) [8] [9] [10], the Trusted Platform Group's mobile Trusted Platform Module (mTPM) [11], and the Open Mobile Terminal Platform's (OMTP) Advanced Trusted Environment [12]. Hardware manufacturers have begun to pick up on these capabilities and have designed a number of supporting architectures over the last few years. Such architectures include Texas Instrument's M-Shield [13] and Qualcomm's SecureMSM [14]. SecureMSM is even used by a software company known as Giesecke & Devrient to support a TEE compliant software architecture known as MobiCore [15], which is found today in certain Samsung Galaxy S III devices [16].

Researchers have also started to develop supporting implementations that make use of these features to address mobile security issues. The first paper was written by Kurt Dietrich and presents a mTPM implementation using a Java-based application on a Suscriber Identity Module (SIM) card [17]. Johannes Winter presented a paper the following year that first introduced the concept of using ARM TrustZone to create a Trusted Execution Environment [18]. Building upon these two papers, Dietrich and Winter released another paper in 2008 [19]. In this paper, the authors presented a software-only mTPM architecture enforced via ARM TrustZone technology.

Not every researcher is convinced that these specification are sufficient in their current state. In a paper by Grossschadl et. al [20], the authors discussed three concerns with the mTPM specification that they felt needed to be addressed. The first concern was the use of a separate chip to support the TPM functionality used by mTPMs. A second concern is the use of out-dated cryptographic operation support, such as RSA-2048 and SHA-1. The final concern is with mechanisms for updating a TEE, which is a key issue that is addressed in this paper. The development of an official policy for updating a TEE is still in progress.

Two key issues that this research fails to address are protection and modification of a TEE, and secure device

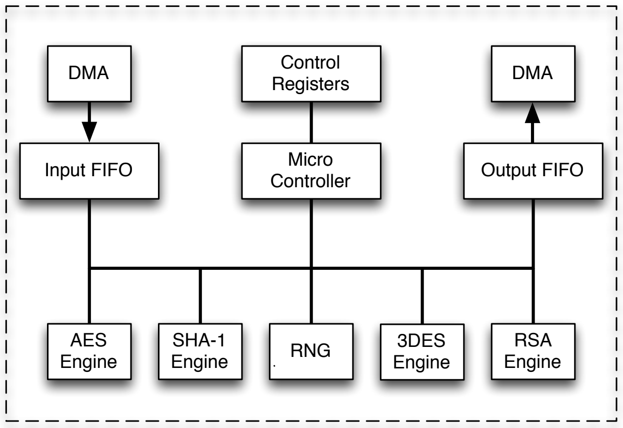


Fig. 1: Generic cryptographic unit.

association/authentication with minimal overhead. While these two problems may initially appear completely separate, a solution discovered for one had a surprising application to the other.

III. Enhanced Cryptographic Engine

In order to address the security issues presented in the Introduction, as well as a number of additional issues not mentioned, we propose the use of a modified cryptographic unit called the Enhanced Cryptographic Engine, or ECE. In addition to physical changes to the underlying hardware, a software interface must be provided that utilizes the features provided by the ECE to address these security issues. In this section, we present information on both of these elements and how they work together to provide enhanced security to mobile and embedded systems.

A. ECE Hardware Architecture

Modern cryptographic engines, such as the one shown in Fig. 1, are integrated into the majority of System-on-Chip (SoC) designs used for modern processors. Cryptographic accelerators were originally incorporated to provide performance enhancements over traditional software implementations, as well as allowing off-loading of computationally intensive algorithms to a separate device. The cryptographic engine generally includes a number of accelerators targeting specific cryptographic algorithms, as well as interface elements, such as control registers, FIFOs, or DMA controllers.

While these engines meet the purposes of their design, they also carry the potential to provide significant security benefits to mobile platforms. By incorporating the capability of a PUF to generate unique-per-chip data, a cryptographic engine can be used to provide a unique AES key and RSA or ECC key pair that can be utilized by the engine. However, these keys must never be directly accessible by any element in the SoC, including the cryptographic unit. Additionally, the crypto unit can also be used to generated random symmetric and asymmetric keys. Such an implementation is shown in Fig. 2 and is hereafter referred to as an Enhanced Cryptographic Engine (ECE).

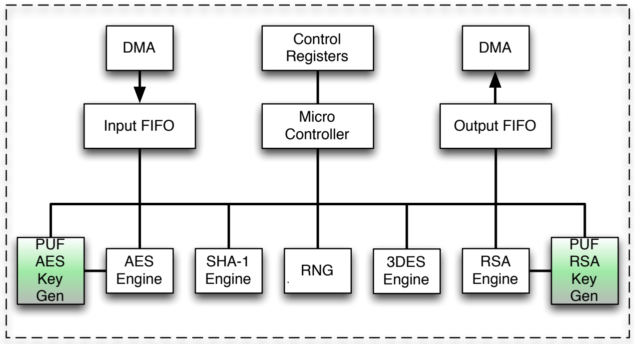


Fig. 2: Enhanced Cryptographic Engine.

As illustrated in Fig. 2, there are two primary elements that are added to any standard cryptographic unit: a PUF-generated secret key generator, and a PUF-generated Public/Private key pair generator. Both key generators receive input from the results of PUFs incorporated into the SoC design, as shown in Fig. 3. In the case of secret symmetric keys, such as AES, PUF-generated results can be be feed straight out as there are no mathematical requirements placed on the value of the key (other than randomness). For asymmetric keys, in this case RSA, the PUF-generated results are used as a seed for key generation logic. The resulting output is a public/private key pair that may be used in corresponding operations.

The ECE contains two separate output channels from the secret key generation modules, one channel that connects directly to the shared internal data bus, and a second channel that connects directly to the corresponding cryptographic engine. The reason for the separate channels is that while these key generator modules are meant to generate random keys that can be used by the system, they should also generate random, but repeatable, keys that are not accessible by any other element of the SoC. These keys should be generated every time the system boots and should be consistent across reboots, regardless of voltage or temperature variations.

The reason these keys must be regenerated and must be protected from access is because they are used to identify the system and to protect sensitive information. As such, they must not be directly accessible by any SoC elements, must

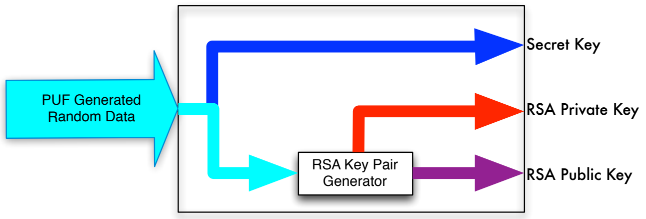


Fig. 3: Key generation method in ECE.

never be stored in any form of non-volatile storage, and must be usable only under a very strict set of requirements. Rather than being read or accessed directly, the value of each key can be muxed into the key input stream for each associated engine. Most modern cryptographic units allow multiple input vectors for cryptographic keys, such as regular memory, e-Fuse, Secure Read-Only Memory (SROM), or some other form of non-volatile storage. Therefore, inclusion of an additional input vector should require minimal changes to existing cryptographic engine implementations.

Use of these key generation modules is controlled via memory-mapped control registers for the ECE. Again, this requires little alteration to modern designs as this is a standard mechanism used to interact with cryptographic units. Anytime the Applications Processor (AP) needs a new key, the appropriate command can be written to the control registers to generate the key. Additionally, an output address can be provided where the ECE can write back the resulting key for use by the AP. A full list of which operations are needed and which are supported would be implementation dependent, with the only requirement being that a single symmetric key and a single asymmetric key pair must be generated and must never be directly accessible by any SoC element.

B. ECE Software Architecture

While the hardware enhancement of the ECE provide functionality that can be used to address the security concerns presented thus far, the proper access controls must be developed and specified to prevent the ECE from being used maliciously. Without proper controls, an attacker may be able to decrypt secure data or add unauthorized devices to a list of associated devices. To address this, we propose a software architecture that uses a TEE to control access to ECE functionality. Such a device then operates in one of four modes, as illustrated in Fig. 4. These modes are: Uninitialized, TEE_Initialization, Discovery, and Standard.

A brand new device starts execution in the Unitialized mode, typically executing out of the secure-ROM internal to the SoC. From this point, the device enter the TEE_Initialization mode where it scans non-volatile memory for the existence of a TEE. If no TEE is found, the manufacturer (presumed to be in control of the device at this point) uploads a TEE to the device. Once the TEE is loaded, the device can enter either Discovery or Standard mode. Discovery mode allows the device to associate itself with another device through a variety of interfaces, such as Wi-Fi or Bluetooth. Once Discovery is completed, execution moves to Standard mode. After the device has entered Standard mode, certain TEE functionality must be disabled to prevent unauthorized access by an attacker.

Although it is hoped that an attacker would never be able to gain execution inside the TEE, such an idea should never be considered absolute and should be guarded against from day one. Fig. 4 shows the ability to move back into Discovery Mode even after the device has transitioned to Standard mode. While this could be possible and would be implementation dependent, it is actually discouraged. Such a transition should

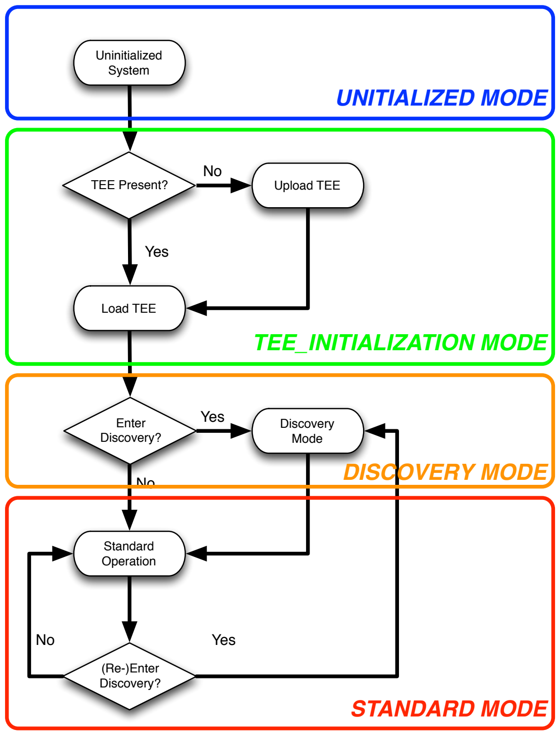


Fig. 4: Operational modes of device incorporating an ECE.

require a re-boot with the required capabilities being locked out until a re-boot or power cycle occur. Information is preserved across the re-boot that allows the device to enter Discovery mode and create the desired association prior to re-entering Standard mode.

To better understand what happens in each mode and what functionality is available, we created a software Application Programming Interface (API) to provide software functions for interfacing with the ECE. The functions listed in the API can be divided into two distinct collections: security sensitive and non-security sensitive. Security sensitive functions are those used in the Uninitialized, TEE_Initialization, and Discovery modes. Non-security sensitive functions can be used from any operating mode, but are unique in that they are still available in Standard mode. The security sensitive and non-security sensitive functions are shown in Listing 1 and Listing 2 respectively.

1) Software Security Sensitive Functions: The security sensitive functions consist of nine functions for interacting with the TEE. The first two functions are used to tell the ECE to generate, or rather store, the results of the PUF generated secret keys. Again, these values are never stored in non-volatile memory, but instead are latched into volatile memory and then muxed into the key input vector for their corresponding crypto accelerator. If at any point the overall security of the system become a concern, the next two functions allow the device to change its secret keys. It is strongly encouraged that such capability only be provided to the manufacturer. Further, the manufacturer should store a public key, or hash of the

Listing 1: ECE security sensitive functions

```
int32_t ECE_GENERATE_SECRET_KEY( void );
int32_t ECE_GENERATE_RSA_KEY_PAIR( void );
int32_t ECE_CHANGE_SECRET_KEY( void );
int32_t ECE_CHANGE_RSA_KEY_PAIR( void );
int32_t ECE_TEE_VERIFICATION( uint32_t *
    load_address );
int32_t ECE_TEE_INITIALIZATION( void );
int32_t ECE_DEVICE_AUTHENTICATION( uint8_t *key
    , uint32_t method, uint8_t *address );
int32_t ECE_DEVICE_ASSOCIATION( uint32_t method
    , uint8_t *address );
int32_t ECE_DEVICE_DISASSOCIATION( uint32_t
    method, uint32_t *address );
```

Listing 2: ECE non-security sensitive function

```
int32_t ECE_AES_CRYPTO( uint8_t *input, uint8_t
     *output, uint32_t length, uint32_t op_type
    , uint8_t *key, uint8_t *iv );
int32_t ECE_RSA_CRYPTO( uint8_t *input, uint8_t
     *output, uint32_t length, uint32_t op_type
    , uint8_t *key );
uint8_t * ECE_AES_KEYGEN( void );
uint8_t * ECE_RSA_KEYGEN( void );
```

public key, in non-volatile storage that will allow the device to authenticate the change request prior to execution. Even if authentication of the change request is not possible, there is no inherent security risk in the changing of the device secret keys. An attacker does not have the ability to deterministically set the value of the new secret keys, and all protected elements are decrypted with the original key and then re-encrypted with the new key, thereby preserving all data values across the change. However, in the instance that authentication is not performed, it is highly recommended that a cool-down time be implemented in order to prohibit an attacker from continually changing the secret keys in hopes of eventually guessing the right value.

There are also two functions that may be used to provide protection and authentication of the TEE on disk. In order for these functions to work, it is necessary to be able to identify a TEE when it resides on disk. To facilitate this, we propose the use of a TEE header as outlined in Table I. This header provides several pieces of information about the TEE, such as manufacturer, version, location, etc. It also provides the support for a fully encrypted TEE, for specifying the encryption algorithm used to encrypt the TEE, and for storing a hash of the unencrypted TEE. This header should be store at the first available sector on the disk (though not required) and should be encrypted in its entirety with the device secret symmetric key. The TEE may be encrypted using a randomly generated AES key that can also be stored in the header, rather than just using the device secret key. This is also implementation dependent, but for the purposes of this paper, encryption with the device secret key is used.

The TEE_Initialization mode consists of uploading a TEE to disk, encrypting it, and then generating and encrypting

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 I: Example TEE header

this header. Each time the device boots, it will first attempt to verify the TEE. If no TEE is present, it moves back to initialization. If it decrypts the first section and finds a TEE marker, it then decrypts the TEE and measures it, i.e. performs a hash of the TEE. If the resulting hash matches what is in the header, the boot process continues. Otherwise, the system moves back to initialization.

This process provides a capability that no other known TEE currently offers: upgrades and modification. Anytime a change to the TEE is needed, all that must be done is copying the new TEE and updating the TEE header. Because the measurements are stored on non-volatile, writable memory, they can be changed. Such measurements have often been stored in one-time programmable memory, such as e-Fuses, which do not provide for the ability to update or modify the value. In this implementation, the measurement are changeable and completely protected, whereas values in e-Fuses are typically unencrypted and unchangeable.

The last three functions provide the methods necessary for (dis)associating with another device, or authenticating to a central location. In this paper, we are focusing on the ability to associate with other devices and pass information between them in a secure manner. To accomplish this functionality, the ECE generates a random public/private key pair that can be used to encrypt an initialization packet between the two devices. The private key is then stored on disk and the public key is then sent to the other device. The other device then does the same, allowing each device to maintain their own private key and the public key of the other device. These two keys, along with the communications information (interface, address, etc.) are encrypted with the device secret key and stored in non-volatile memory. Once the initial association has completed, all further communications can be encrypted from start to end without any need for an intermediary key server or key exchange. Although each device has its own secret public/private key pair, it is highly recommended that this pair only be used for authentication with a cellular network, rather than communications with with other mobile or embedded devices.

2) Software Non-security Sensitive Functions: The non-security sensitive functions shown in Listing 2 are first used in TEE_Initialization mode. However, these functions may still be utilized in the Standard operating mode. A number of applications would likely benefits from the ability to en(de)crypt data and generate random keys. Rather than having the capabilities be restricted at this point, it is the key usage that should be restricted. It is strongly recommended that the device secret keys never be used while in Standard mode. Instead, Standard mode makes use of keychains originating from the device secret keys. For instance, the TEE can generate a single key that is used to generate and protect all other keys. This key is encrypted and stored on disk and then decrypted each time the device boots, allowing for the decryption of subsequent keys without the need for constant access to the device secret key.

IV. Implementation and Results

There are two primary elements of the ECE architecture presented that had to be shown as viable in order for this work to succeed. The first element is the development of a PUF that can be incorporated into a cryptographic engine and is capable of generating the necessary values. This implementation has already been developed and is discussed in detail in [21]. The second element is showing that the software API will provide the proper functionality to support our claims. In this section, we cover a software implementation of the ECE described in Section III, show the practicality of this solution, and then present the results of our implementation.

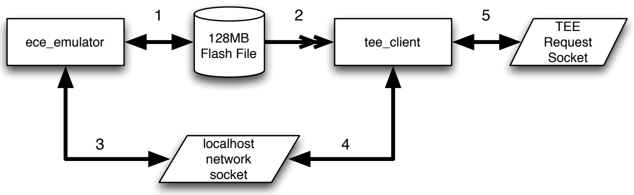
A. Implementation

To provide the initial proof of concept for this idea, we have created a software simulator based upon the openssl library that emulates the hardware ECE and provides a mechanism for implementing the full API. This software program consists of two processes that run on a device: ece_emulator and the tee_client. The ece_emulator supports the Uninitialized and TEE_Initialization modes of operation, while the tee_client implements the Discovery and Standard modes.

Upon startup, the ece_emulator is provided a file name for a file that mimics a 128MB flash storage chip on a mobile device. If the file does not exist, a new 128MB file is created and a secret AES key and RSA key pair are generated. The keys generated are store on the disk in order to maintain value between subsequent runs of the applications. This represents the only significant deviance from the proposed architecture. Once the file is created and the keys are generated and stored, the next step is initialization of the TEE.

On the initial run, no TEE is present. Therefore, the ece_emulator creates a generic TEE header and prompts the user for a TEE executable, in this case tee_client. It then reads in the tee_client application, encrypts it with the generated secret key, and stores it in the flash file. The TEE header is then updated with the appropriate information. The ece_emulator then launches the TEE application and opens a localhost network socket that can be used to communicate with the tee_client.

Once the tee_client is launched, it opens a network connection back to the ece_emulator that can be used to request generation of keys, en(de)cryption of data using the device secret keys, changing of the secret keys, or to provide updates to the TEE. Additionally, the tee_client opens its own network socket



STEP	DESCRIPTION
1	ece_emulator initializes disk file and load tee_client
2	tee_client is verified and spawned as a separate process
3	ece_emulator opens localhost network socket for receiving requests
4	tee_client sends requests to ece_emulator via localhost connection
5	tee_client open network socket for sending/receiving association requests

Fig. 5: ECE emulator flow.

that can be used to associate with other devices. The tee_client also has the ability to open a client connection to another device in order to request association. Each association made is recorded as discussed previously and is stored encrypted in a small file. This file is read when the tee_client first starts in order to retrieve information on existing associations. This entire flow is illustrated in Figure 5.

Although this software simulator makes certain assumptions due to the fact that it is primarily concerned with proving the feasibility of the architecture, a quick discussion on the association process is necessary. While it can rarely, if ever, be expected that a mobile device is always in a secure state or is operating in a trusted manner, there are almost always moments in which this is the case. For instance, it would be assumed that a new device that has not connected to any external communications channels and has a fully-functional TEE running is secure. This obviously ignores the possibility of any hardware trojan or supply-chain issues, which are too extensive to cover in this paper.

Being able to make the determination that a device is currently secure in terms of eavesdropping is critical for association. The easiest mechanism for accomplishing this is through the use of a wired, closed-network with all wireless interfaces being disabled. This provides the highest degree of certainty that no other entity can utilize any Man-In-The-Middle (MITM) techniques to alter data. Once this is implemented, we propose the key exchange methodology presented in Fig. 6.

In this approached, an initiator named Alice sends a packet to Bob requesting an association. This packet contains a nonce plus a challenge, for instance a hash of Alice and Bob's interface addresses. These two values are encrypted using Alice's private key then joined with Alice's public key and sent to Bob. As long as this is done over a closed-network, this should be secure. Further, this is a public key that is being sent, therefore there should be no real security risk with it being discovered. However, if this is not performed over a closed network, there is no method for Alice to verify the the received response originated from Bob and not an attacker performing a MITM attack.

$$Alice : C = E_{A_{priv}}(T_{nonce} + T_{challenge})) + A_{pub} \tag{1}$$
$$Bob : T_{nonce} + T_{challenge} = D_{A_{pub}}(C) \tag{2}$$
$$Bob : C = E_{A_{pub}}(B_{pub} + B_{secret} + B_{priv}(T_{nonce} + T_{challenge})) \tag{3}$$
$$Alice : B_{pub} + B_{secret} + E_{B_{priv}}(T_{nonce} + T_{data}) = D_{A_{priv}}(C)) \tag{4}$$
$$Alice : T_{nonce} + T_{challenge} = D_{B_{pub}}(E_{B_{priv}}(T_{nonce} + T_{data})) \tag{5}$$

Fig. 6: Public key exchange during association.

Bob then uses the received public key and decrypts the packet. Bob now has the public key for Alice and can verify the data packet received by simply performing the same hash. Bob then needs to return his public key to Alice. Additionally, Bob can take this opportunity to generate a shared secret key that can be used for all further communications. Bob then encrypts the original nonce and challenge with his private key, adds it to his public key and the generated shared key, and encrypts the entire blob with Alice's public key. This blob is then sent back to Alice where it is decrypted. After the full exchange, Alice and Bob both have copies of one another's public keys and a secret key that can be used for future communications. From this point on, all communications can be encrypted with the secret key regardless of the communications mechanism used, and a closed-network is no longer needed. All data generated and received during the key exchange process (public keys, symmetric key, etc.) is encrypted with the device's secret key and stored protected on disk.

The benefits of this approach are several. First, a secure communications channel can be created at anytime on any network between these two devices. Second, there is no need to store a massive number of challenge/response pairs. Instead, the nonce and data element exchanged between the two can be used at anytime to verify one another's identity if needed. Third, if for any reason communications using the secret key fail, or if it is simply preferred to use a different secret key for each communication, a new shared key can be generated at any time and exchanged securely since the public keys are already exchanged. And fourth, should there ever be a concern that a key is compromised, new keys can be generated and exchanged in a safe, secure manner without significant overhead or storage requirement. The stored data can be removed and the process shown in Fig. 6 can be redone.

B. Results

Using the design methodology presented previously, we developed the two C++ applications described and implemented them on a number of Apple and Linux based computers. We were able to successfully generate unique flash files for use by the ece_emulator on each machine. Further, we were able to successfully implement public key swaps between multiple devices and establish a completely encrypted communications network. This network was then used to swap files and issue commands. While initially setup on a wired, closed-network, we were able to continue operation even on a completely open network without any issues.

Unsuccessful attempts were make to altered data and MITM communications sessions. Because all communications were encrypted between devices and there was never an open request for a key exchange, we experienced no issues with the security of the network communications. While an attacker was able to maliciously attempt to initiate an association, it was simple to detect due to the fact that all association requests must by approved by the user.

Additionally, we were able to prove the ability to update or modify an existing TEE in a secure fashion. This has very strong implications for the BYOD arena discussed in the Introduction. With the capability of uploading, removing, or modifying a TEE, a corporation can install required software and policies, or even their own TEE, that can be removed when an employee leaves. This will not result in the loss of personal information or modifications to user data. The TEE specification details the use of Trusted Applications (TAs) that run inside the TEE [10]. A company need only install a TA, with its corresponding data, into the TEE of the employee's device. This app can then validate the phone and enforce required policies, but can also be easily removed when employment is terminated.

V. Conclusions

In this paper, we have presented a novel architecture that utilizes PUF generated results to generate cryptographic keys and values for use inside an Enhanced Cryptographic Engine. We have proposed a software Application Programming Interface that can be used to interface with the ECE in order to provide stronger security for data and device association on mobile devices. We also presented the results of a software emulator of this architecture that was capable of establishing secure communication channels between multiple computers and using this secure channel to transfer commands and data. Further, this emulator showed the ability to uniquely protect TEEs and provide an mechanism for the modification, removal, or update of TEEs on embedded systems.

As a result of the work conducted in this paper, we have found that the architecture presented provides a viable solution for addressing issues with device association, TEE modifications, and protection of system critical data in a unique manner. As such, this research provides a solid starting point for further implementation work, such as a full hardware solution that can be used in a variety of different environments.

A. Future Work

Leveraging the research conducted in the paper, as well as other research involving PUF implementation, the logical path forward at this point is a full hardware platform implementation. The best platform currently available for this is the Xilinx Zynq 7000 FPGA which contains a dual-core ARM processor. In our previous work we were able to incorporate our PUF design directly into the bitstream used to program an FPGA [21]. Utilizing this same approach, we can incorporate our PUF into the programmable logic fabric of the Xilinx FPGA, and then interface to it via the ARM core processors inside the FPGA. Such an implementation will be dependent upon successful repair of a software bug on the first version of this board that prohibited access to the cryptographic unit on development kits [22].

Once a viable hardware platform can be found, it will be possible to demonstrate the applicability of this approach on a number of use cases. The first use case could be internal automotive networks on vehicles, thereby addressing security issues such as those discovered in [1] [2]. By creating a secure closed-network on the automobile, the ability of an attacker to connect to the network and overwrite firmware or issue unauthorized commands can be removed. A second use case for medical device networks in healthcare systems can also be explored. Similar to automotive networks, the idea is to prohibit snooping of network traffic and injection of malicious commands.

A third use case could involve exchanging DRM protected materials between authorized devices on a media account such as iTunes. In this instance, the public key of each device should be retrieved and stored on the iTunes servers for each user. When a device connects to the account, a listing of all device public keys associated with the account can be downloaded to the device. This would allow the device to transfer authorized media to any other device associated with the account. For instance, even when an internet connection is not available, a movie could be transferred between an iPod and an iPad in a way that does not violate copyright or DRM protections.

A final use case might be association of a device with a user on a cellular network. Cellular devices typically store user information on a SIM card that performs the authentication of the user to the cellular network. Although some devices provide unique hardware numbers for chip identification purposes, such as the International Mobile Station Equipment Identity (IMEI), these numbers may or may not be used by the carrier. Further, there is often a high chance these numbers can be changed or spoofed by software running on the device. Utilizing the device secret asymmetric key, it may be possible to provide a mechanism that will work with the SIM card to associate a user with the device on a network. This could greatly help to eliminate the ability of thieves to steal mobile devices and re-activate them on other networks.

References

[1] 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.

[2] 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*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 6–6.

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

[4] 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.

[5] 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.

[6] 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.

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

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

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

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

[11] Trusted Computing Group, *TCG Mobile Trusted Module Specification, 1.0*, 7th ed. TCG, 2010.

[12] "Advanced Trusted Environment: OMTP TR1, Version 1.1," 2009.

[13] J. Azema and G. Fayad, "M-Shield Mobile Security Technology: Making wireless secure," http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf, Texas Instruments, Tech. Rep., feb 2008.

[14] "SecureMSM security suite," http://www.gi-de.com/gd_media/en/documents/brochures/mobile_security_2/MobiCore-secure-os.pdf.

[15] 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.

[16] S. B. Gupta, "G&Ds Mobicore security platform to be integrated on the samsung galaxy s3," http://www.bgr.in/software/android/gds-mobicore-security-platform-to-be-integrated-on-the-samsung-galaxy-s3/, May 2012.

[17] K. Dietrich, "An integrated architecture for trusted computing for java enabled embedded devices," in *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, ser. STC '07. New York, NY, USA: ACM, 2007, pp. 2–6.

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

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

[20] 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*, ser. HST '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 84–90.

[21] J. Aarestad and J. Plusquellic, "A hardware-entangled delay puf based on delay variations in an fpga-based aes sbox macro," in *submitted to Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, June 2012.

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