SAND2012-4336C

Securing Trusted Execution Environments with PUF Generated Secret Keys

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*—Trusted Execution Environments are quickly becoming a preferred method for providing isolation between secure and non-secure execution environments. The protection of these environments, as well as their software structure, is still a primary area of interest and research. The abililty to use a Physically Unclonable Function to generate a unique-per-device AES key provides an excellent mechanism for protection of a Trusted Execution Environment at rest through encryption. These keys can also be used to manage modification of the TEE during execution. In this paper, we present an new methodology for how this protection can be achieved, as well as a framework for the incorporation of Physically Unclonable Functions into cryptographic engines.**

***Keywords*-Trusted Execution Environment, mobile security, physically unclonable functions, AES, system-on-a-chip, secure boot, data-at-rest encryption**

I. Introduction

The use of mobile processors continues to expand at an almost alarming rate. These processors are finding uses in a variety of applications, such as cellular, automotive, and health care. While none of these applications are the same, they do often share common requirements. One such requirement is the need to ensure isolated execution of specific code elements in order to perform critical tasks.

For instance, many users of smart-phones today perform an array of banking transactions, from transferring funds to making small purchases. Because the applications used to perform these tasks are not guaranteed to be running on a ”safe” system, banking institution are beginning to require some level of assurance that their respective applications execute securely on the target device. Not only does this protect the user from financial loss, but it lowers the liability taken on by the bank.

Additionally, while the owner of the device may not be concerned about all the internal security mechanisms in their device, there are several parties who are very concerned about this area. Cellular and Internet providers want to ensure that devices connected to their networks are always functioning in accordance with their required policies. This is especially true of corporate IT departments who want to provide secure functionality on these devices (such as VPN connections). Before providing these services, they need some form of assurance that not only is the device functioning according to policy, but that the information stored on the devices to facilitate these types of connections is not accidentally or maliciously modified.

To address the need for secure execution, the Global Platform group began creating a Trusted Execution Environment (TEE) specification that details the required element necessary to create and ensure an isolated execution environment [1]. The TEE can be used to run trusted applications that function outside the execution space of the on-device operating system, or Rich-OS. By providing this isolation, a vulnerability in the Rich-OS does not translate to compromise of secure data or unauthorized access to secure components.

While these TEEs do provide sufficient isolation in order to ensure secure execution, the code that runs inside the TEE must at all times be protected. The Secure-Boot process is a mandatory element for any TEE in order to verify that the code to be run inside the TEE has not been altered. Performing a full secure-boot requires the use of a secret or private key for encrypting the required software while on disk, and decrypting it prior to execution from memory. A popular approach at this time is to make use of eFuse or secureROM elements to store the necessary keys. Recent exploits however, such as GeoHot’s limera1n exploit for Apple A4 processors [2], have shown that these elements are not above attack. Although Apple does not appear to store any encryption keys inside the secureROM, the vulnerability allowed attackers to gain execution, decrypt firmware packages, and jail-break devices using the A4 processor.

The use of eFuses or secureROM to store encryption keys also presents another security concern. If the key for a single device is ever compromised, that could result in the compromise of all other devices in that family. This would be especially true if the key is stored in the secureROM. If the key is stored in the eFuses, the key value may be different from device to device, but this is not always the case. Is has often been recommended that the eFuses be used to store a hash of a needed public key, rather than the key itself. This would imply that software would need some means of retrieving the values from the eFuses in order to verify the validity of the key it received. This being the case, if an attacker could gain access on the device, it would potentially be possible for them to read all data stored in the eFuses from software. While

simply reading a hash may not be very helpful, it is unlikely that a hash would be the only useful information stored in the eFuse array.

To address these concerns, we present a methodology that can be used to leverage a PUF generated AES key to not only protect the TEE, but provide a mechanism for its expansion and customization. In section two, we will present information on background work that has been done to prove the feasibility of PUF generated AES keys and their applications in system security. In section three, we will present information on the structure of TEEs and the decisions that are necessary to determine what should, and should not, be included. In section four, well will present our method for bringing these two element together to provide enhanced security and expandability to TEEs. In section five, will present our conclusions and how we plan to proceed.

II. Using PUFs to Generate Crypto Keys

Physically unclonable functions provide an electrical 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 the 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 such results could be used as keys for cryptographic operations. The primary concern up to this point has been the stability of the result generated by the PUF, as it can be impacted in various ways, such as temperature and voltage levels. As the primary purpose of this paper is not to prove that the stability needed to consistently regenerate a cryptographic key is possible, such research will not be discussed in detail and it will be assumed that a stable and reliable 256-bit AES key can be provided. For more information on some of the methods used to provide this stability, please see [3] [4].

The concept of using a PUF to generate a crypto key is not new. In 2007, Suh and Devadas presented the concept of using a PUF to generate a secret key that could be used to provide device authentication [5]. Their research provided several methods for attempting to stabilize the results of a PUF in order to provide consistent and repeatable results that would be used to generate a cryptographic key. Once stabilized, the authors proposed the ability to use this key in device authentication mechanisms, 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.

Using a similar idea, Bohm et.al. proposed the use of a SRAM-PUF to generate a stable PUF that would then be read in by the processor and used for cryptographic purposes [6]. However, as the authors noted, the ability of software 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 solid 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.

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 [4]. In this paper, 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 application has also been discussed. To date, most of the applied uses involve using PUF generated results with RFID elements [7] [8]. 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.

III. TEE Architectures

A TEE is defined as an isolated execution space where code can be run outside the influence of the standard operating system, referred to as the Rich-OS Execution Environment (REE). Making use of hardware enforced barriers, the execution space for a system can be partitioned into a non-secure and secure world. A software API must then be included that provides access between these two worlds, allowing applications running the REE to interact with trusted elements in the TEE.

While the existence of a TEE does not guarantee the security of the code executing therein, it does attempt to guarantee isolation from the REE. It is still the responsibility of the developer of TEE elements and applications to ensure their code is sufficiently hardened against attacks, such as malformed input strings. Sanity checks of this kind are paramount to the security of the TEE, as the kernel executing in the REE cannot be trusted to provide the necessary screening of this information. Further, because of the enhanced privileges that come with execution in the TEE, an attacker gaining access inside the TEE could prove catastrophic to the overall security of the system.

The Trusted Platform Group has provided an architecture document to give both manufacturers and developers an overall

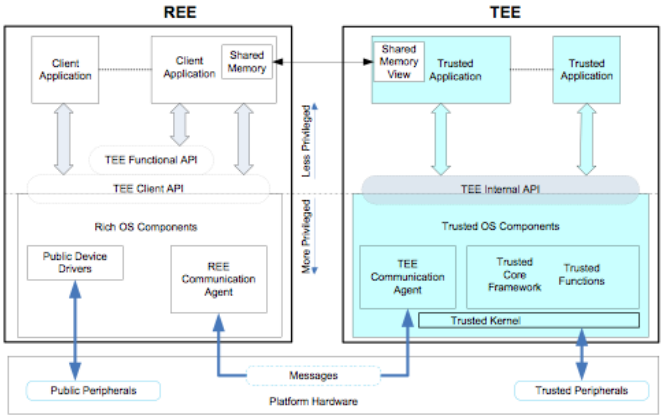

Fig. 1. TEE System Architecture

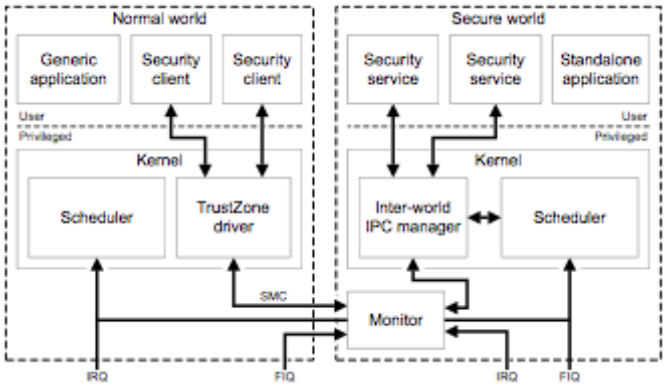

Fig. 2. ARM TrustZone Software Architecture

picture of how both hardware and software must be structured and how they should interact [1]. This is shown in Figure 1. Using this architecture as a guide, manufacturers have been able to begin designing and releasing chips that adhere to the TEE architecture specifications. An example of one such architecture is ARM TrustZone.

A. ARM TrustZone

Identifying the need to provide a TEE on mobile devices, ARM developed a set of hardware and software modifications to their ARM core architecture to support the use of a TEE. This suite is called TrustZone and is supported in all ARMv6 based architectures and newer [9]. Although TrustZone was originally released in 2003, it has only recently seen broad adoption with its 3.0 specification. The hardware architecture supported by this specification is shown in Figure 2.

TrustZone creates a TEE by making use of three changes made to the ARM core architecture [10]. The first change is partitioning all hardware and software elements on the SoC such that they can function in either a secure or non-secure mode. Alterations to the AMBA3 AXI bus fabric makes this possible in hardware, while development of a TrustZone API facilitates the isolation in software. The most evident change is the addition of the NS-bit, a 33rd bit added to each internal address and data element that labels it as belonging to either the secure or non-secure world (external memory must make use of a TrustZone-aware memory controller or wrapper in order to provide similar partitioning of memory between secure and non-secure areas). This bit was also added to the Secure Configuration Register (SCR) in CP15, the system control co-processor, and is used to indicate the current execution mode of the processor.

The second set of changes involve the addition of TrustZone extensions, allowing a single processor to execute in both modes, controlled in a time-sliced manner. The changes that were necessary to allow this to happen include the use of two independent interrupt controllers (one for each mode), a banked set of CP15 registers, and the addition of a new instruction called the Secure Monitor Call, or SMC. The SMC instruction can be used by the REE to trigger a transition into the secure world, thus allowing REE applications to request the services of secure elements. Execution can also move into the secure world via a secure world interrupt, enforced by the separate interrupt controllers. A manufacturer may also make use of a TrustZone aware interrupt controller that can securely maintain the interrupt vectors for each world, rather than requiring two independent controllers.

The last collection of changes affect the debug infrastructure. Debugging in the REE kernel would typically provide a user with a high degree of access rights to the underlying system. In order to maintain the necessary isolation, while still providing debugging capabilities, the debugging infrastructure is security-aware. This means that it is capable of restricting access to secure element even when the REE is running in debug mode.

B. Implementation Issues

Despite the great promise provided by a TEE, they are still almost entirely controlled by the manufacturer. This is mostly due to the standard implementation of the secure-boot process. Using secureROM inside the SoC as the initial root-of-trust, each subsequent element of code is measured and verified prior to being executed. Because the TEE is one of these elements, and because the measured values are typically stored in some form of one- time programmable memory, there is no mechanism available that provides for the ability to change the TEE.

Because of this restriction, all code that needs to be included in the TEE must be present at manufacturing time. As one might imagine, such a restriction is not feasible for broad application in the mobile arena where user needs vary dramatically. In such circumstances, it is virtually impossible to predict beforehand everything that a user will need that might make use of the TEE.

Further, device manufacturers often have special needs for a TEE that are dependent upon their specific application. Such applications include femtocells, automobiles, health-care systems, and military devices [**?**]. Because chip manufacturers are not going to roll out an entirely new design to support

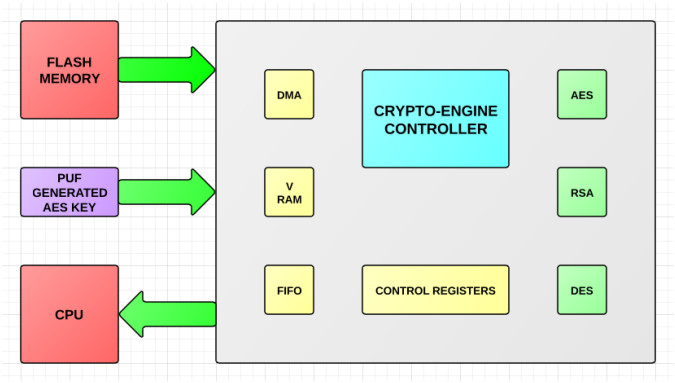

Fig. 3. PUF Supported Crypto-engine

changes in the TEE for each application, a method must be developed to provide flexibility and alteration of the TEE after the manufacturing process.

IV. PUF Protected TEE

Having now presented background information on both PUFs and TEEs, it is time to put these two technologies together. In order to better understand why this needs to be done, remember that the TEE is typically stored in external non-volatile memory. As such, if not properly protected, it is subject to attack. While standard encryption helps protect the TEE from these attacks, there is still a concern about insider attacks as well as damage control if the TEE is ever decrypted. Further, there must be a method provided to prevent an attacker from simply overwriting the entire TEE. As presented previously, if a TEE cannot be customized to the needs of the implementer or user, its functionality may be irrelevant.

To address these concerns, we propose the incorporation of a PUF inside the crypto-engine of modern SoC devices. By using a PUF to generate a secret AES key, we alleviate the concern of an insider-attack or a disgruntled employee revealing information about the location or value of the key. Further, because not even the manufacturer will know what the key is, no single entity will have greater control over the device than any other. Mobile devices typically have multiple entities of interest and they all want some element of control on the device without having to worry about being restricted by anyone else.

In addition to securing the TEE while on disk, the PUF generated key will be used to encrypt a measurement of the TEE that will also be stored on the disk. During each boot process, the TEE will be decrypted and measured. That measurement will then be compared with the measurement stored on disk. If these two do not match, then the system will know that the TEE on disk has been modified. Otherwise, it can assume the data is valid and continue the boot process.

This approach also provides a means of modifying the TEE without requiring any changes to the underlying hardware or software. Any time an element of the TEE is added or removed, a new measurement can be made, encrypted by the crypto-engine using the PUF generated key, and stored on disk. Because all measurements are encrypted by the PUF generated key, whose value is unknown to any entity and as such, any accidental or malicious alteration of this information is currently impossible.

To help visualize how this modification would fit into a typical SoC architecture, consider the design shown in Figure 3. In this figure, we present a fairly standard crypto-engine for an SoC. You have a primary controller which handles the flow of information through the engine, a set of control registers that are often memory-mapped by the CPU to provide a command interface, and then several cryptographic accelerators for specific operations, such as AES, RSA, and DES. Also included are interfaces for providing DMA and FIFO memory transactions.

The final piece added to this design in a small amount of volatile RAM, or V-RAM. This will serve as a storage location for the AES key generated by the PUF. This key should never be stored at any time in any form of non-volatile RAM. It should also not be stored in any way that would make it accessible by the Host CPU. In other words, this memory element should not be memory-mapped like the control registers. Rather, it should provide a direct input to the cryptographic accelerators that can be muxed together with any other key that the system might use. To support this, we will add bits to the control registers that may be used to determine exactly which key should be used during each operation.

A. Boot Process

The primary use of this implementation can be seen during the boot process. This process would start by loading and executing memory stored within the secureROM. Once the secureROM performs its standard initialization of the system, it would send the challenge to the PUF and inform the crypto-engine to lock in the result. Once the key is obtained, the first sector of the flash would be read and decrypted. This would provide the location, size, and measurement of the TEE on the flash device. From here, the secureROM can load, decrypt, and measure the TEE for the system. Once validated, the secureROM would transfer execution to the TEE and the secure-boot process would continue. (Use of an additional boot-loader could also be supported prior to loading the TEE if necessary to support additional system initialization)

B. TEE expansion

Because the measurement for the TEE is stored encrypted on the flash, the TEE has the ability to expand or contract without requiring any major modifications to the underlying system. The primary purpose in supporting such alterations is based upon the need of entities such as corporate IT departments to have a presence on mobile devices without impairing the usability of the device. When a new employee starts at a company, rather than having to use two separate mobile devices, one for personal and one for corporate, the employee could provide his mobile device to the company IT department who could then load authentication and policy

information onto the device. This code would most likely run as a trusted-application inside the TEE. As such, it would not affect the overall operation of the device, but rather would be accessed only at times when the user needed to connect to the corporate network or provide authentication of his identity. Should the employee ever leave the company, the IT department would simply remove the elements from the TEE and the device would continue to operate.

A scenario such as this is currently non-existent in any environment that we have discovered. Most corporate environments require a device to be completely re-flashed with their own version of the operating system. This method does not require the device to be re-flashed, or does it alter in any way the user's data or applications. While this is only one use case for this new architecture, we can easily see this fitting into other applications, such as femto-cells, health-care devices on a internal hospital network, vehicular systems, and may more. Even on the mobile platform, this has ramifications for financial institutions, cellular providers, device manufacturers, and application developers. And because the key generated by the PUF is unique, even if an attacker was able to determine the key being used on a specific devices, that key would not transfer to any other device. This would make discovery of keys extremely expensive, if at all possible.

C. Implementation Platform

Because the creation of a custom ASIC is very costly, we propose the use of an FPGA platform to provide the initial proof-of-concept for this design. Utilizing the ability to instantiate soft-core processors on Xilinx FPGA, we will create a design that implements two independent CPUs that mimic the isolation provided by TEE compliant hardware. We will then create a custom crypto-engine and interface this logic to CPU-0, which will serve as the secure world. In addition, based upon prior work demonstrating the ability to instantiate a PUF inside an FPGA [11], we will create a PUF and tie its results into the crypto-engine. The final component will be an external sd-card that contains a theoretical TEE.

Using this hardware platform, we will develop software to run on both the secure and non-secure processors. The secure processor will interact with the crypto-engine and sd-card to provide access to the TEE, as well as making all modifications and updates to the TEE. The non-secure world will be responsible for interfacing with the secure world to making requests for alternations to the TEE. It will then be the responsibility of the secure world to re-measure the update TEE and provide new measurement data.

Follow-on work will include a detailed analysis of the appropriate mechanisms necessary for implementing the updating process. Simply allowing any application to load any amount of code into the TEE is obviously not a feasible solution. A certification process will be needed to validate that requested TEE code in compliant with some security metric. Such a process could be used in a manner similar to a certificate authority wherein the secure-OS requests confirmation from a centralized authority that the code is compliant prior to installation. This is still an area of research we are planning to pursue and show not be considered the defacto method that will be used.

V. Conclusion

In this paper, we presented a new idea for the use and incorporation of PUF generated AES keys on mobile processors. We discussed how modern advances have made it possible to create a trusted and isolated execution environment. By protecting this environment with a unique-per-device key, we can alleviate concerns about manufacturer attacks and device family vulnerabilities. This approach also provides a strong method for the expansion and contraction of the TEE as needed by user applications. Dynamic expansion provides manufacturers, carriers, OS providers, and even employers, a method for securely storing code and data on a users device without altering the underlying operating system or making drastic changes to the device.

Using the results of this research, we can ensure the security of the TEE. However, this leaves the question of how the TEE should be structured. The best structure for the TEE is our next research step. Identifying how the device should respond if no TEE is found, how the TEE should support adding and removing trusted elements, and how to store verification information within the TEE are some of the questions we intend to address over the next four months. Our final area of research will involve creating an actual TEE kernel and porting an Android build to our platform. We will use this platform to prove the feasibility of this approach and show how it provides a greater security framework than any current implementations.

References

[1] GlobalPlatform, "Tee system architecture, version 1.0," http://www.globalplatform.org/specificationform.asp?fid=7763, December 2011.

[2] Geohot, "limera1n," http://limera1n.com.

[3] 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.

[4] 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.

[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] 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.

[7] 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.

[8] 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.

[9] ARM, *TrustZone API Specification*, 3rd ed. ARM Limited, February 2009, no. PRD29-USGC-000089 3.1.

[10] A. S. Technology, "Building a secure system using trustzone technology," http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, April 2009.

[11] 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.