2017-03-01

A Reconfigurable Trusted Platform Module

Matthew David James
Brigham Young University

A Reconfigurable Trusted Platform Module

Matthew David James

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

ABSTRACT

A Reconfigurable Trusted Platform Module

Matthew David James
Department of Electrical and Computer Engineering, BYU
Master of Science

A Trusted Platform Module (TPM) is a security device included in most modern desktop and laptop computers. It helps keep the computing environment secure by isolating cryptographic functions and data from the CPU. A TPM is usually implemented with a small microcontroller which is near the main processor. In addition to a microcontroller, it may employ hardware acceleration to assist in cryptographic computations. When vulnerabilities are found, or new algorithms developed, TPMs become obsolete because the hardware accelerators cannot be upgraded. This thesis presents a proof of concept implementation of a TPM on an FPGA. By using an FPGA, the TPM gains the ability to be upgraded or have new cryptographic modules added. This new design easily fits on the Zynq FPGA used in this work, with room left over for additional functionality. We explore the feasibility of this approach, including the added cost of the FPGA, and the added benefits of reconfigurable hardware.

Keywords: FPGA, TPM, Cryptography, Trusted Computing

ACKNOWLEDGMENTS

The author would like to thank his family for helping keep him motivated and on target. In addition, he extends a special thanks to his wife for her endless patience. He also extends his thanks to his professors, colleagues, and coworkers who helped inspire him to continue learning. Additionally, he extends his gratitude to the Science Mathematics and Research for Transformation (SMART) scholarship program for providing financial aid throughout his graduate school experience.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Keeping personal data secure is a constant game of cat and mouse. We hear about new vulnerabilities being discovered almost daily, and there are frequent patches and updates for our computer systems to keep us safe from them. Too many of us have been affected by data breaches or identity theft. Just as we use locks, keys, and safes to deter burglary in our homes, we have similar protections for our computers. Our lives in the internet-connected world are kept safe by authentication and encryption. However, we can trust these schemes only if we trust the computer they run on. Even if we trust the manufacturer to build secure hardware, malware and viruses make it hard to trust any device all the time. Enter the Trusted Platform Module; it acts as a safe place for us to authenticate and encrypt data.

The Trusted Platform Module (TPM) is a specification for a security device created by the Trusted Computing Group (TCG). The TPM contains three basic protected capabilities: key management, attestation, and integrity measurement [1]. These features are discussed further in Chapter 2, but their applications include drive encryption, identifying users, checking platform integrity, and security logging [1].

The TPM specification calls for various cryptographic algorithms, types of encryption, to keep data secure. However, just because those algorithms appear to provide security now, that does not mean they will continue to do so. Malicious actors are working around the clock to break these algorithms. In an effort to keep data protected, the TCG has issued several revisions to the TPM specification since the first version appeared in 2000 [2]. The first version of the specification used the Data Encryption Standard (DES) algorithm. Later, weaknesses were found in DES, which led to it being replaced by the Advanced Encryption Standard (AES). This occured in a 2005 revision to TPM version 1.2. Although the TCG has updated the specification in an attempt to stay on top

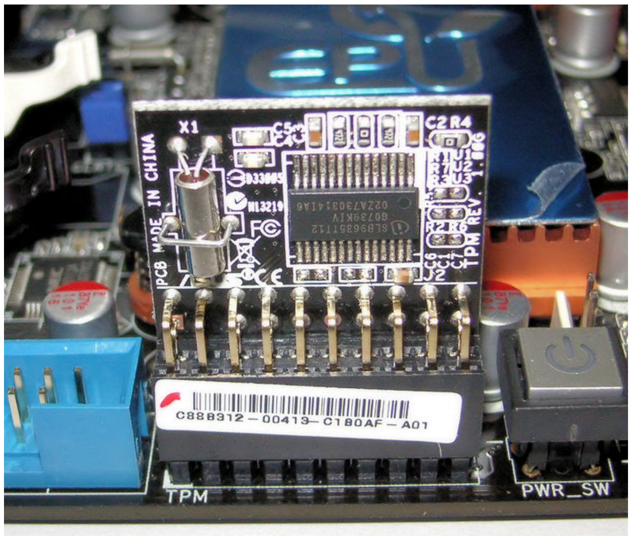


Figure 1.1: A commercial TPM. Photograph courtesy of FxJ [5] (Image is in the Public Domain).

of weaknesses and vulnerabilities, many existing TPMs embody implementations of version 1.2 of the specification, which was first released in 2003 [3]. The latest version of the specification is version 2.0, which was unveiled in 2014 [4]. This thesis will primarily deal with the 2011 revision of the version 1.2 specification, but most of the conclusions and research will also apply to later versions of the specification. Version 2.0 adds a few more algorithms and capabilities, but the algorithms used in TPM version 1.2 are still being used.

The TPM specification does not mandate a particular implementation method. Designers may use an ASIC, a microprocessor, or even a program running on the CPU. The details are left up to the developer, as long as the TPM meets the security requirements set forth in the specification. A common TPM implementation is a discrete chip that contains a microcontroller, I/O for communicating with the processor, a small amount of memory, and in some cases accelerators to speed up some of the cryptography [6, 7]. Figure 1.1 shows an example of a commercial TPM. The specification was designed to allow TPMs to be cheap and ubiquitous. Though they provide good security (algorithms used in the TPM are still widely used today), most implementations are limited in the ways that they can be upgraded in the event of the discovery of a security flaw. In a conventional TPM it may be possible to update the firmware, but if a new algorithm or update requires special hardware acceleration, the TPM itself would have to be replaced.

For example, the security of RSA (the main type of encryption used in the TPM) depends on the hardness of factoring a number that is the product of large primes. While current computers cannot solve this problem in any reasonable amount of time, consider what will happen as quantum computers become more ubiquitous. One proposed algorithm for quantum computers, Shor's algorithm, can factor these large numbers in polynomial time [8]. If a suitable quantum computer is ever built, all TPMs that depend on RSA will be rendered obsolete.

Breaking an encryption scheme is not just a theoretical problem. Encryption schemes have been broken before, without the assistance of quantum computers. Kumar et al. showed that a collection of Field-Programmable Gate Array (FPGA) devices costing less than \$10,000 could be used to crack DES and other similar algorithms in around 9 days [9]. What's more, even if an algorithm cannot be cracked in a reasonable time frame (RSA for example), a class of attacks called Side-Channel Attacks (SCAs) can compromise a specific implementation of an algorithm [10, 11]. These attacks work by measuring power usage, time taken, or electromagnetic emissions to gain information about what was encrypted. This is discussed in more detail in Section 3.8.

1.2 Contributions

1.2.1 TPM Implementation on an FPGA

FPGAs are reconfigurable devices that can be programmed much like one would write an application for a computer. Since their inception, they have become faster and more capable. Now, FPGAs are gaining attention for applications in cryptography. Not only can they be used to crack encryption, but they can also be used to encrypt and decrypt data. They have flexibility that fixed hardware cannot have, and speed that software cannot match. As they become more competitive in terms of cost and functionality, they are likely to show up more frequently in consumer systems.

Intel recently acquired FPGA maker Altera and announced that their FPGAs will be incorporated into the server-grade Intel Xeon processors. In fact, at the Open Compute Summit in early 2016 they showed a diagram of what that might look like [12]. Inevitably, these FPGA-supported microprocessors are likely to find their way into consumer desktop computers. As the price comes down and systems that include FPGAs become more common, it is not hard to imagine a scenario where it may be desirable to have a TPM or similar security processor that can be programmed into the FPGA fabric when needed.

In this work, we explore implementing a TPM on an FPGA, and the advantages and disadvantages that result. Eisenbarth et al. proposed such a design several years ago [13]. However, they were not able to implement their proposed design (see Chapter 4) due to constraints on their FPGAs. FPGAs are now available with the resources that are necessary to design and implement a reconfigurable TPM. Others have explored some of the security risks associated with reconfigurable TPMs, such as securing the bitstream and nonvolatile storage [14]. We hope to use this previous work as a stepping stone to allowing reconfigurable TPMs to become more useful.

1.2.2 Framework for Extending the Reconfigurable TPM

The TPM specification already provides an upgrade path for firmware through a special set of commands, but some TPM manufacturers include special hardware to accelerate algorithms [6, 7], and this hardware would be impossible to upgrade. Upgrading may be necessary in the event that vulnerabilities or side-channel attacks are discovered. If one of the included algorithms could not be upgraded, the security provided by the TPM would be undermined, making it effectively worthless.

While a reconfigurable TPM gains the ability to upgrade hardware acceleration, it is not without challenges. In addition to implementing a TPM on an FPGA, we also construct a framework that makes it easier to upgrade TPM components, or if need be add new cryptographic components. This framework aims to allow a TPM developer to abstract away the cryptographic components of a TPM, making it more ‘plug-and-play’. In this way, system designers can select the parts they want, make a few modifications to allow them to interface with the framework, and quickly have a working vendor-independent TPM.

The extensibility framework presented in this thesis simply allows easier management of the accelerators. If an accelerator needed to be upgraded, a new bitstream (the ‘firmware’ for an FPGA) could be generated. New algorithms or accelerators could be added in the same way. Additionally, the FPGA-based implementation could allow users to customize the TPM to their use cases. They may need to save space on the FPGA for additional applications (and so use software implementations of algorithms), or they may max out the FPGA with accelerators that provide high speed and high throughput. Although the FPGA may cost more at the outset, they will not have to replace the TPM chip as would be required with a more conventional TPM.

1.2.3 Transparent Operation

A goal of this thesis is to create a TPM implementation that will allow users or developers to conduct audits of the firmware. To this end, the components used to create the design in this thesis will be open source whenever possible. A frequent worry when dealing with closed source security or cryptography is that no one can see what is going on inside. With this open source design third parties can validate the security of the code. As it stands, conventional TPMs can be trusted only as far as one trusts the manufacturer, since the implementation details are hidden and proprietary. A second advantage is that upgrade timing may improve. If a bug is found, a user may be able to quickly patch the system herself rather than having to wait for a manufacturer (who may have already deprecated the product) to push out a patch.

Although using open source has many benefits, occasionally it may not offer the performance characteristics desired by the developer, or it may not be tested extensively. In this case, the framework presented in this work will allow a developer to choose components from any location.

1.3 Thesis Outline

The rest of this thesis will be organized as follows. Chapter 2 will focus on how what a TPM is, what it does and how it works. Chapter 3 will briefly detail the cryptosystems present in the TPM, their roles, and some background into how they work. Chapter 4 will discuss related work described in the literature. Chapter 5 presents an implementation for a reconfigurable TPM, along with a framework to help support new and upgraded modules. Chapter 6 will discuss testing and results from the design process, including a comparison against a conventional TPM implementation, FPGA resource utilization, and the speed of cryptographic algorithms. Finally, Chapter 7 will conclude the thesis and discuss potential future research.

Chapter 2

Trusted Platform Module

The Trusted Computing Group (TCG) is a consortium of companies such as Microsoft, Intel, AMD, and others involved in computer security and technology. The TCG created the TPM specification to enable access to trusted computing resources and make them more ubiquitous. Since the TPM specification is implemented by different manufacturers in different ways, in the rest of this thesis we will use the term TPM to mean an implementation of the TPM specification.

Other companies have seen the need for more security in their products and created similar devices. For example, Apple has a Secure Enclave for the iPhone, and ARM has the TrustZone. These types of devices are very common; TCG has been developing the TPM specification since 2003 [3]. For instance, there is a TPM in every Google Chromebook [15]. On Windows, running "tpm.msc" provides a quick way to see the status of the TPM if one is present on a machine. In fact, Microsoft now requires certain types of Windows-certified computers to come with a TPM [16].

This chapter will explain the basic components, capabilities, and limitations of a typical TPM implementation. We first explain how a conventional TPM is constructed. We then explain the main capabilities and uses of a TPM: measuring platform integrity, key management, drive encryption, and identification and signing. The chapter concludes by examining some of the limitations of typical TPMs.

2.1 Typical Implementation

Designers have great flexibility in how to implement the TPM specification. They may use software, firmware, or a discrete chip. However they are implemented, all TPMs must have storage, I/O, an execution engine, and several cryptographic algorithms. This is shown in Figure 2.1.

For purposes of discussion, when we refer to a conventional TPM, we mean a discrete TPM. This common implementation is a TPM that is on its own chip, connected to the processor

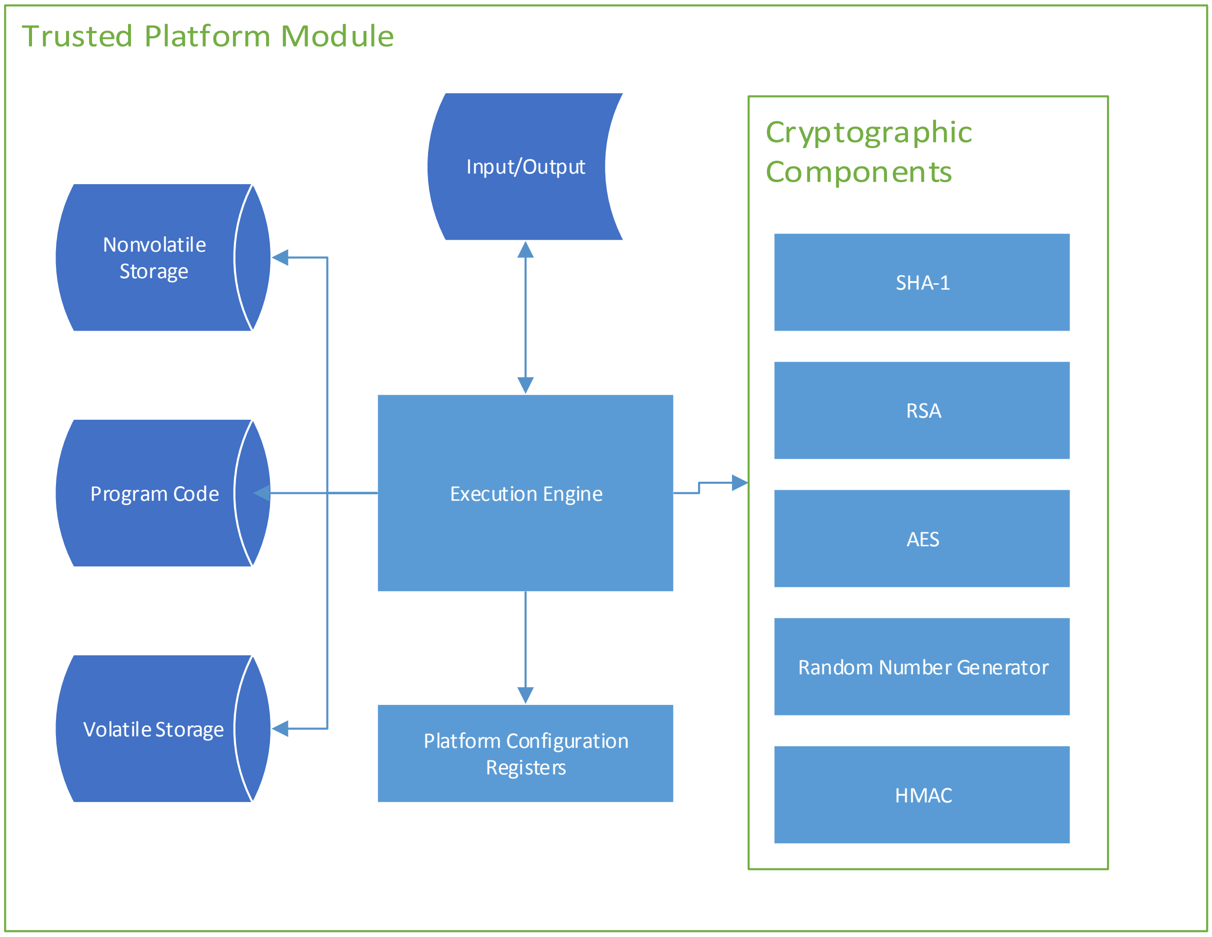


Figure 2.1: TPM Block Diagram

via the Low Pin Count (LPC) bus, an I2C bus, or similar [17]. The conventional TPM has the following hardware components:

- a Microcontroller,
- I/O,
- volatile storage (RAM),
- nonvolatile storage (flash),
- an entropy source, and
- a cryptographic accelerator for RSA.

Nonvolatile flash storage is used to hold sensitive configuration data that must persist when the TPM is off. This includes keys for encrypting data, the owner password, configuration flags, and other sensitive information. If this information were lost, the encrypted data protected by the keys would no longer be decryptable. Volatile RAM holds data about operations currently in progress, as well as the status of the TPM.

Most of the functionality for a conventional TPM is implemented in software running on the microcontroller. This software includes various cryptographic algorithms: RSA, SHA-1, HMAC, random number generation, and optionally AES. (These algorithms will be discussed in greater detail in Chapter 3.) Sometimes additional hardware components are included in the TPM. The conventional TPM we consider here has two: an entropy source, and an RSA accelerator. An entropy source helps provide better random numbers and is recommended by the specification [3]. While there is no requirement to do so, it appears that many TPM implementations also include a hardware accelerator for RSA [6]. This is included to help speed up computations.

By design, the host system cannot access the secure capabilities of the TPM directly. The specification defines the commands that may be sent, and when they may be used. Commands included in the specification include creating keys, hashing data, loading and unloading protected information, generating random numbers and many others. This command interface is implemented by an execution engine running on the microcontroller. Command packets are sent over the LPC bus from the CPU to the TPM. Each command must have a header consisting of a tag, a size, and an ordinal. The tag and size tell the TPM how large the command is, and how to interpret it. The ordinal is the specific action requested. The structure of a command packet is as follows:

1. a header which contains
    (a) a tag,
    (b) a size, and
    (c) an ordinal;
2. any key or data handles required by the command;
3. the command parameters; and
4. if required, any authorization sections.

Only the header is always required. If a command requires authorization, then the 'authorization section' and the 'key or data handle' section will be filled in. Some commands require additional information, included in the 'command parameters' section. The execution engine protects the TPM's secure data by ensuring that access to keys and configuration information is only permitted with respect to the policies set forth in the specification. After a command packet is successfully processed, a response packet is sent back to the host computer. The response packet has the same structure as the command packet.

This conventional TPM implementation is widely used because it is cheap, costing only tens of dollars. Part of the reason it has low cost is that it requires very little extra in terms of hardware. Figure 2.2 shows a possible view of the components, in terms of what is implemented in software and what requires dedicated hardware resources.

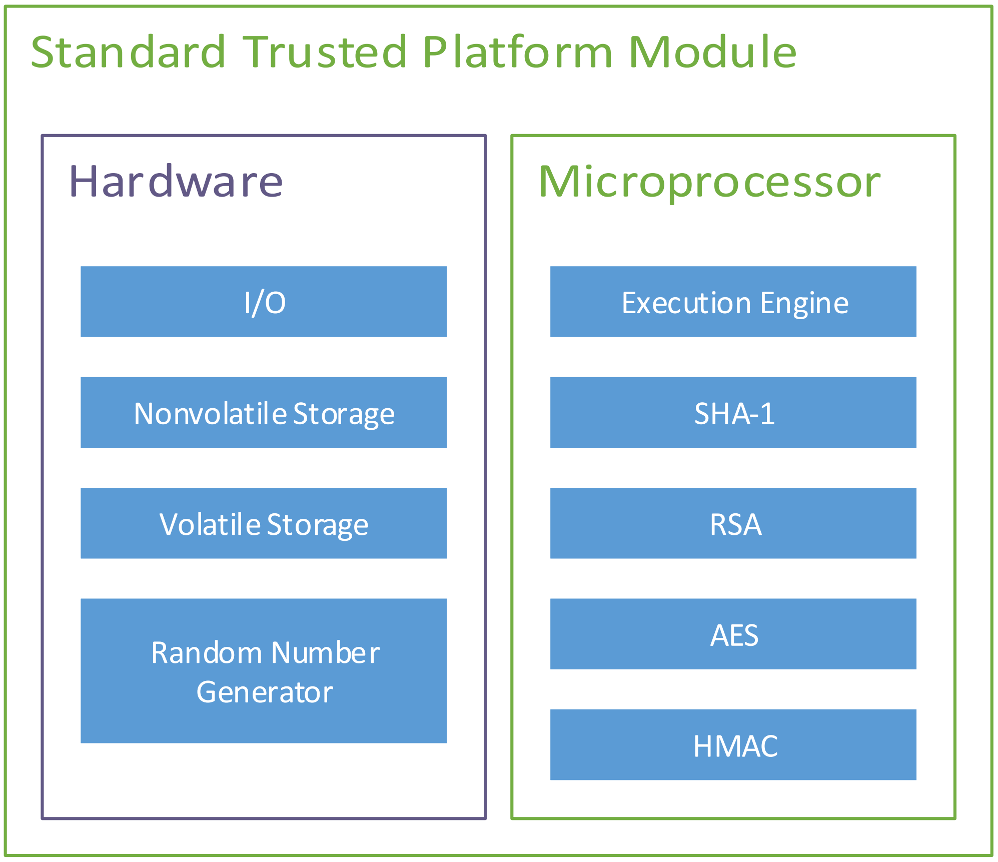


Figure 2.2: How a conventional TPM implementation may look in terms of components implemented in software, and components implemented in hardware.

Typical uses of the TPM include: generating and protecting drive encryption keys, facilitating identification and authentication of users and software, and checking and verifying platform integrity [18]. The next sections will go over the role of the TPM in each of these scenarios.

2.2 Platform Integrity

Platform integrity means that the system works as intended. The TPM provides the ability to help verify platform integrity through special registers called Platform Configuration Registers (PCRs). The PCRs are used to compare and store *hashes*, which are shortened representations of large messages or data objects. For example, when downloading a file, a hash of the file is often provided by the host. Once downloaded, another hash is taken. If the hashes match, it is reasonable to assume the download was not corrupted. A similar process using the TPM and PCRs can be used to verify that the configuration of a computer has not been corrupted. An example of this is *Measured Boot*, a process used by Microsoft Windows [19]. When the computer starts up, the PCRs are initialized to their default values; the BIOS completes this step by initializing the TPM. During each subsequent step of the boot, the operating system takes hashes of the hardware configuration, loaded drivers, and important operating system files. Each hash is sent to the TPM, which then concatenates this value with the existing PCR value and hashes it again [3]. The TPM specification calls this chained hash process *extending* a hash. In order for the Measured Boot to succeed, the final hash value must match the expected value (set during a clean install). Malware, newly installed hardware, or updated drivers can prevent the hashes from being the same. If a difference is detected along the way, the operating system can stop the boot and alert the user. This process works something like this:

1. The BIOS starts execution.
    (a) The BIOS initializes the TPM.
    (b) The TPM resets the PCRs to their default states.
2. The bootloader takes control from BIOS.
    (a) A hash of the bootloader is taken and extended into the PCRs.
    (b) If the PCRs differ from expected values, stop.

3. The Operating System takes control from bootloader.

    (a) Hashes of important drivers are extended into the PCRs.

    (b) Hashes of important system files are extended into the PCRs.

    (c) If the PCRs differ from expected values, stop.

In addition to this functionality, the PCRs can be used to implement a 'seal' operation. This operation ties encrypted data to a value in one or more of the PCRs. In other words the encrypted data cannot be decrypted unless the PCR value matches the expected value. During the encryption process, the user specifies which PCRs will be bound to the data. This is done by providing a bitmap selecting the desired registers inside the command packet. The user will then specify the hash values that must be present for decryption. Optionally, they can specify the value of the PCRs when the data was sealed. This optional feature allows the user to determine if the computer was in a legal state when the data was encrypted. During decryption, the TPM checks the PCRs that were specified and compares their current values against the stored values inside the sealed data. The values may have changed if different hardware was added, drivers were updated, malware was installed, etc. If the values do not match, the TPM does not return the decrypted data to the user. If the system has been compromised by malware, this protection ensures that the malware cannot access data shielded in this way [3]. This protection can be employed by disk encryption software like BitLocker [20].

2.3 Key Management and Use

One of the advantages of using a TPM is that it provides secure storage for encryption keys (for more detail on keys, see Section 3.4). Access to secure storage is subject to the policies explained in the specifications. Only the TPM itself is capable of decrypting data protected by one of its keys. Keys in the TPM are stored in a tree-like structure. A simple image depicting this structure is shown in Figure 2.3. When a new key is requested, several properties must be given:

- a parent key,
- the type of key (encryption, storage, signing, etc.),

- the algorithm to use,
- the encryption or signature scheme to use,
- a migration password, and
- a usage password.

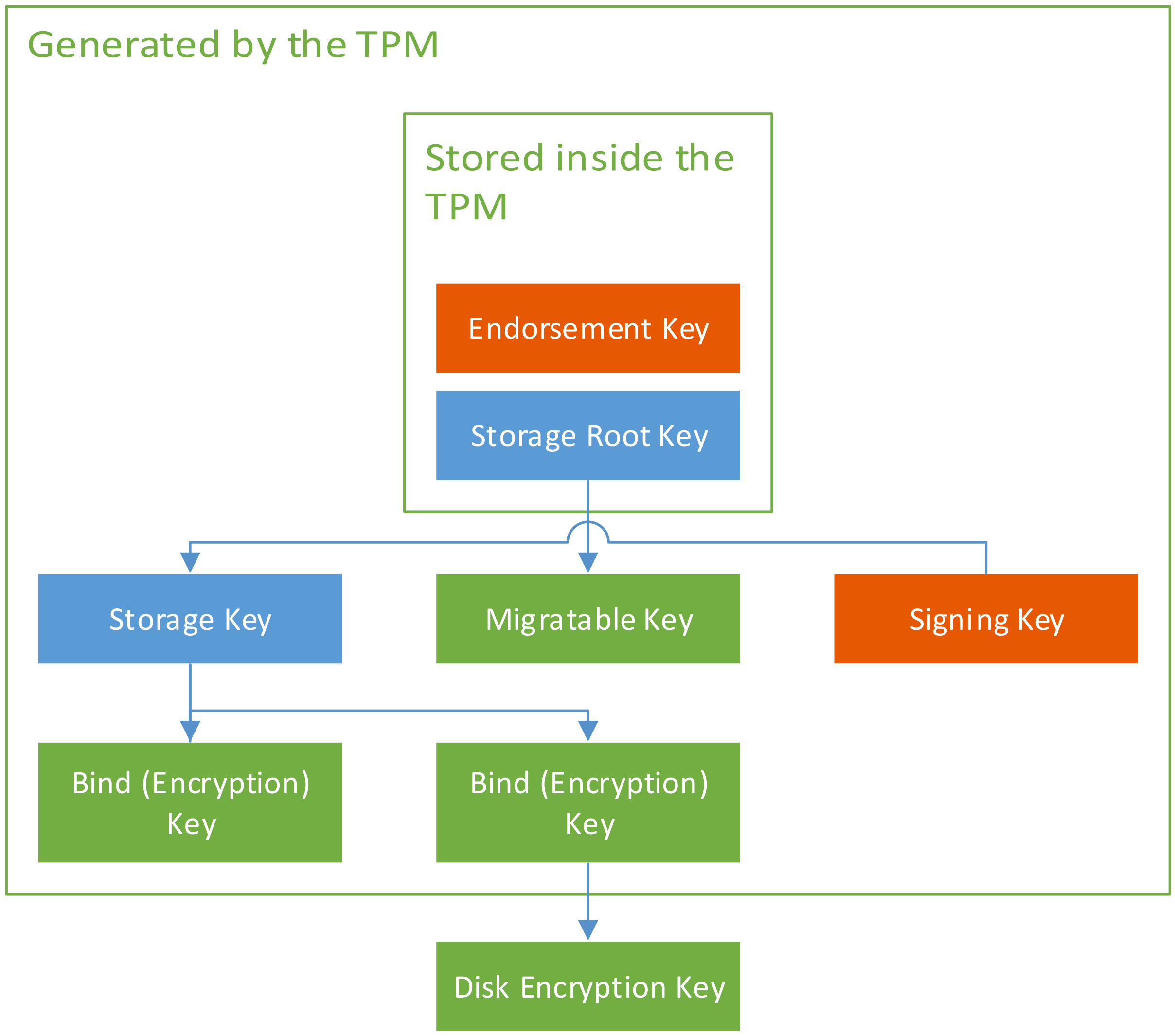


Figure 2.3: TPM Secure Storage

These items determine when and how a key can be used. Keys can be created for storing other keys, binding (encrypting) data, proving identity, or for signing. For each key, a parent key must be chosen. The parent key's job is to protect (through encryption) the child key's sensitive data. All keys created by a user are children of the Storage Root Key (SRK), but they may also be children of other storage keys as shown in Figure 2.3. The algorithm to be used is selected from a list of supported algorithms called TPM_ALG_ID. Additionally, there is usually more than one way to use a given cryptographic algorithm. AES for instance has multiple modes, including CBC, OFB, CTR, and others (see Section 3.6). The supported encryption and signature modes (or schemes) are determined by lists called TPM_ENC_SCHEME and TPM_SIG_SCHEME.

Keys stored in this tree are RSA keys. These keys are *asymmetric*, meaning they have a public and a private part (see Section 3.4). The private part of the key must never be exposed outside of the TPM. When the TPM creates a key, it encrypts the private portion of an asymmetric key with the chosen parent's public key. Each key is protected by its parent key all the way up to the SRK. The SRK is always stored in the TPM, and the private portion is never exposed to the outside world. Because each key is encrypted with the SRK (or encrypted by a parent which is encrypted by the SRK), this allows the keys to be stored outside the TPM on the disk drive. Since the keys are stored in encrypted form, whenever a key needs to be used it has to be loaded back into the TPM first. The entity desiring to load the key must provide the password for the SRK in order to use it to decrypt the child key. If this is successful, the key is loaded and can then be used. If the key to be loaded is further down the hierarchy, the password for the next parent key is given and the process continues.

2.4 Drive Encryption

Disk encryption programs can use this secure storage system in combination with the PCRs to bind encryption keys to just one computer. Consider these two scenarios.

Scenario 1 We use a program that does not use the TPM. We choose a password, and save a recovery password to a USB disk. The computer creates an encryption key, and saves it to disk. The key is protected, but only by the password. Anyone with the password can unlock the disk and access the data on it. Additionally, anyone with the recovery USB disk can also access the drive.

Scenario 2 We use a program that uses the TPM. We still choose a password and create a recovery key. This time, the TPM creates an asymmetric key. In addition to that key, the computer creates another key. The computer uses the second key to encrypt the drive, and the first key to encrypt the second key. The first key can only be decrypted by the TPM. In fact, only the TPM *inside this computer* can decrypt the key. While anyone with the password can still decrypt the drive, they cannot simply steal the drive to decrypt at their leisure. In order for decryption to succeed, they must also have the TPM used to encrypt the drive.

Microsoft BitLocker is one program that is capable of using the protection that a TPM provides. While these programs provide more security than the first approach, they may also cause problems when adding new hardware to the computer. This is because certain updates to the software or hardware can change the value of the hashes stored in the PCRs. In these cases the recovery key will be necessary to boot the computer, and the encryption key will have to be resealed with the new PCR values [20].

2.4.1 Key Migration

There are some instances where it may be desirable to move an encrypted drive between computers. The TPM can create special migratable keys that can be used in this case. If this type of key is created, a second password used for migration is also provided. This password keeps migratable keys from being moved without the consent of the key owner. This process is explained in the steps below:

1. The first TPM, Alice, creates a migratable asymmetric key, key A.
2. The second TPM, Bob, creates an asymmetric key, key B.
3. Bob sends the public portion of key B to Alice.
4. Alice encrypts key A with the public portion of key B.
5. Alice sends key A, now encrypted, to Bob.
6. Bob decrypts key A using the private portion of key B.
7. Bob re-encrypts key A using the SRK or another key in his hierarchy.

This process is secure because the public key cannot be used to decrypt data. If the encrypted migratable key and encryption key are both intercepted, the data remains secure. As long as the migration password is carefully controlled, and we trust the public key, the migratable key is never exposed to unauthorized entities [3].

2.5 Identification and Signing

Besides the SRK, one other root key is present in the TPM, called the Endorsement Key (EK). This unique key is usually created during manufacture of the TPM. The EK is an asymmetric *signing* key. Signing means that the key can be used to prove the origin and validity of an object. While the EK is a signing key, it is not used for general purpose signing operations. It is used to identify data that came from that particular TPM, and can also be used to identify the manufacturer of the TPM [3, 6]. The TPM can also create additional signing keys. Signing is a relatively straightforward process. In this process, a hash is taken of the data, and that hash is signed using the private key. The data and signature are sent to the recipient. The recipient can use this information, along with the signer's public key, to verify that it was not tampered with.

Another use of this capability is to enforce a policy of only allowing applications signed by certain developers to run. In a scheme like this, the operating system sends the application signature to the TPM, and the TPM checks to see if the signature was created by the selected signing key. If the signatures match, the key in the TPM must have been used to sign the data, so the application is permitted to run. There already exist tools that work in a similar way, for example, Microsoft AppLocker [21]. AppLocker allows administrators to control which applications, DLLs and installers can run. It can use signing and certificates, but it can also use simple blacklists to prevent the execution of unauthorized applications.

2.6 Other Capabilities

The TPM offers additional capabilities to users who desire them. The SHA-1 hash function is exposed through a simple API. This allows a low resource system with a TPM, say a phone, to use hashes without needing its own implementation of the algorithm. Random number generation also has an API that allows a user to retrieve an arbitrary number of random bits, or to add random data into the random number generator to increase entropy. TPMs also offer transport sessions.

This capability allows a user to encrypt data that is being sent to and from the TPM. This could be useful in a situation where the user is operating remotely and needs to access a TPM on their home computer, or even when they are operating locally, if multiple users are logged in.

2.7 Limitations

There are a few problems with using a conventional TPM implementation. The first problem is that for a *Trusted* Platform Module, you may not actually be able to trust the implementation. Can a user who requires security really trust a company to not place backdoors into their products? Such backdoors have already been found in modems and routers, and it is likely that they are also in other devices [22]. Even if the company itself can be trusted, there is certainly motivation for state actors to know some of the properties of the keys your TPM may generate over its lifetime.

Another problem, as previously noted, is that while upgrading the software in a TPM is possible, upgrading the hardware is not. The TPM specification provides the TPM_FieldUpgrade command which allows a manufacturer to upgrade capabilities of the TPM. This command requires physical presence at the machine (proven by a hardware switch, or a BIOS prompt) if the TPM has no owner, or owner authorization to run if it does. To further protect the process, the contents of the upgrade patch can be kept secret by using a manufacturer key of some kind [3] (to prevent unauthorized patching or modifying of an authorized patch). A TPM that uses software for all of its functionality may be able to upgrade everything, but this is not always the case. For example, a TPM may use a hardware accelerator to assist in modular exponentiation for RSA. If it is determined at some future point that this core has a vulnerability, some latent error, or if it is incompatible with the new firmware, it becomes useless; the TPM would have to perform modular exponentiation in software, and performance will suffer.

Even if a TPM does all of its computations in software, and is completely upgradeable through TPM_FieldUpgrade, the hardware that the TPM is running on may leak information about the keys being used, the data being processed, or other sensitive information. This kind of attack is known as a side-channel attack, or SCA (see Section 3.8). If a side-channel vulnerability is discovered in existing hardware, a software fix is unlikely to fix the problem, because the leakage usually comes from the hardware. Thus, the entire device would have to be replaced.

Chapter 3

Cryptography Essentials

The TPM specification uses several different kinds of cryptography, and a brief background of the cryptographic functions supported in specification version 1.2 is presented here. As will be discussed in the next chapters, a goal of this project is to allow additional cryptographic functionality to be added to an existing TPM. To that end, it is useful to understand how some of these algorithms function. Each algorithm has a specific purpose inside the TPM; replacing or upgrading it requires one to understand why it is used, where it is employed, and the security benefits it provides.

3.1 Nonce

A nonce is simply a random value. In the TPM nonces are 20 bytes long. The general idea is that the nonce should be different every time one is required. Since the nonce changes with every use, it can be used to help prevent *replay attacks* that involve sending the same information multiple times. These attacks drain resources, and create a denial of service condition, so it is desirable to stop them. If the nonce does not change, there is an error.

TPM Uses In the TPM, nonces are used to prevent replay attacks, but are also used to form secret keys. The uses for these keys will be discussed in Section 3.3. The TPM creates nonces by asking the random number generator for 20 bytes of random data.

3.2 Hashing

Hash algorithms convert varying length messages into a fixed length representation called a *digest*. TPM version 1.2 uses Secure Hash Algorithm 1 (SHA-1) as its hash algorithm. SHA-1 takes a message up to 2^{64} bits and reduces it to a 160-bit digest [23]. The output of a good hash algorithm will look random for different inputs, even if those inputs are similar. For a hash algorithm to be useful in cryptographic applications, it should be resistant to *collisions*. A hash collision occurs when two messages hash to the same digest. A collision attack may be used to forge passwords or data. In this case if the data or password has the same hash value, the system will have no way of telling that the data is not legitimate. Research by Stevens et al. has shown a *freestart collision attack* on SHA-1 and so the authors recommend against using SHA-1 [24]. Normally, SHA-1 works by using a compression function iteratively on different parts of the message, with a constant value used to initialize the state. In a freestart attack, this constant value is changed. The susceptibility of the hash to this attack means that it may not be secure, and that a full hash collision attack on SHA-1 is likely. Because of research like this, in newer versions of the TPM specification, SHA-1 has been replaced with SHA-2.

TPM Uses The TPM specification does not use text passwords like most people are used to. Instead, authorization values are used. These values are 20 bytes long, exactly the length of a SHA-1 hash digest [3]. Applications that interface with the TPM will take a user's password as input, and then use the SHA-1 hash function to reduce or expand the arbitrary length password to a 20 byte digest. This digest is then used as the authorization value. Since hashed versions of passwords all take exactly 20 bytes, this simplifies processing of the TPM commands.

Hashes are also used in some commands to verify if the parameters were received successfully [3]. The host computer will hash the parameters to be sent in the command packet. When the TPM receives the command, it will also hash the parameters. If both hashes are the same, the message was received intact.

In addition to passwords and verification, the TPM also allows the operating system to use the hash function for its own purposes, which is particularly useful on low resource systems. These commands are also vital for setting the PCR values. The system uses TPM_SHA1Start to start a hash and TPM_SHA1Update to add information to it. When finished the system calls TPM_SHA1CompleteExtend, which finalizes the hash and copies it into one of the PCRs.

3.3 Keyed-Hash Message Authentication Code

In addition to SHA-1, the TPM also uses a Keyed-Hash Message Authentication Code (HMAC). An HMAC and a hash perform a similar function, but an HMAC is used with a secret key. The addition of a secret key means that an HMAC can prove that a message was not tampered with during transit. A hash function only proves that the received message is intact. An HMAC can be used with many different hash functions, but in the TPM the hash function is SHA-1. Equation 3.1 shows the basic idea of the HMAC. In the equation, $H(x)$ represents the hash of a value. SHA-1 uses a block length of 64 bytes, so $ipad$ is the value 0x36 repeated 64 times and $opad$ is the value 0x5c repeated 64 times. K is the secret key, and $text$ is the message being hashed [25]. The operator || represents concatenation.

$$HMAC(text, K) = H((K \oplus opad) \parallel H((K \oplus ipad) \parallel text)). \tag{3.1}$$

This scheme prevents someone from changing a message and impersonating the sender. Prior to beginning communication, the sender and recipient will need to exchange the secret key. When the message and accompanying HMAC digest are received, the recipient will calculate the HMAC of the message again and compare the computed value with the received value. If they match, the message was not tampered with. If they do not, someone or something has modified the message. Unless adversaries have the secret key, they will not be able to duplicate the HMAC digest that is sent, even if they can find another message that would cause a collision in the underlying hash. Adding the key builds another layer of protection.

TPM Uses The TPM uses HMAC when sending and receiving commands from the user. The key is either predetermined based on the command, or it is calculated by using nonces, or it is a hash of the owner password [3]. The message is a hash of some of the input parameters, as well as a nonce that changes with each command and response. These features are used to help guard against an application that may try to insert itself between the user and the TPM.

3.4 Asymmetric Encryption

Asymmetric encryption is based on the use of two different keys. One, the public key, is used for encryption and can be distributed to anyone who needs to send a secure message to the recipient. The recipient has a private key, which is used for decryption and which can decrypt any message encrypted by the public key. While there are several different types of asymmetric cryptography, the asymmetric algorithm used by the TPM is RSA.

RSA The Rivest, Shamir, and Adleman (RSA) algorithm is an asymmetric encryption algorithm [26]. In the TPM, RSA is used as the underlying algorithm protecting the key hierarchy, for sealing data, and for sign and verify operations. RSA should not be used to encrypt entire files, because it can protect only those messages smaller than the key size. If messages are larger than this, patterns may be spotted in the data. To avoid this, large data should be encrypted with another algorithm (usually symmetric), and RSA should be used to encrypt the key.

RSA is based on modular arithmetic; in particular, it uses modular exponentiation. Because of this, an RSA key consists of several different parts:

- a public exponent,
- a private exponent, and
- a public modulus.

The public exponent is generally chosen to be 65,537 or another small prime number. This exponent simplifies some of the math and code involved, but also makes encryption slightly faster than decryption. The public modulus is the product of at least two large prime numbers. The private exponent is calculated from the primes and the public exponent. The public key consists of the public modulus and the public exponent. The private key consists of the public modulus and the private exponent. RSA's security is determined by the number of bits used in the modulus. In the TPM, RSA uses 2,048 bits (256 bytes).

RSA is secure because there does not yet exist a polynomial time algorithm that can factor a number that is the product of two large primes. Finding the large primes used to create a key can be a slow process, because of the complexity of the primality tests. Thus, it can take several minutes to generate a key. The Fermat test shown in Equation 3.2 is often used as an initial test to determine if a number is *likely* to be prime. This test uses modular exponentiation, which is quite slow in software.

$$a^{p-1} \equiv 1 \pmod{p}. \tag{3.2}$$

Decryption and encryption are also slow operations for RSA. It turns out that the primality tests (including the Fermat test), encryption, and decryption all use modular exponentiation.

$$C(x) = x^n \pmod{m}. \tag{3.3}$$

Since the numbers used in RSA are very large, occasionally up to 4,096 bits, computing a modular exponentiation necessitates either an arbitrary precision arithmetic library (slow), or a hardware accelerator (fast). If the hardware accelerator is implemented in an ASIC, it cannot be changed after manufacture (say for different key sizes).

TPM Uses RSA is used in the TPM as the default algorithm for creating encryption keys, signing keys, and storage keys. Both the Storage Root Key and the Endorsement Key are RSA keys of length 2048. The TPM may have special requirements for the strength of other RSA keys depending on manufacturer and customer specifications. A conventional TPM typically has special modular exponentiation hardware, perhaps as a separate ASIC, to help accelerate RSA operations.

Padding Ciphers and hash functions often require that messages be a certain length before they can be processed. If a message is not the required length it must be 'padded' to achieve that length. There are many different padding schemes. For RSA, two common schemes (also in use in the TPM) are Optimal Asymmetric Encryption Padding (OAEP) and Public-Key Cryptography Standards (PKCS) version 1.5.

Padding has the advantage of obscuring the original message length, and also making processing easier (fewer corner cases). For example, say a three byte message is to be encrypted and sent using RSA with a 2048 bit key. After padding and encryption, the message will be 256 bytes long. Using padding also provides a way to check that decryption was successful. For example, the OAEP scheme used in the TPM includes a hash of the word ‘TCPA’ in the plaintext. Finding this value during decryption indicates that the process was successful.

3.5 Symmetric Encryption

In contrast to asymmetric encryption, symmetric encryption uses the same key for encryption and decryption. In RSA, the public key can be distributed widely, but in symmetric encryption the key must be kept secret. Anyone who has the key can both encrypt *and* decrypt the information. The symmetric algorithm for the TPM version 1.2 is AES. Older versions of the specification used DES, but this was deprecated in a 2005 revision to the version 1.2 specification [3].

AES There are several versions of the Advanced Encryption Standard (AES). AES-128, AES-192, and AES-256 are the most common, with the number referring to the key size in bits. As the keys increase in size, so does the security. In desktop computing, AES can be used to secure communications over the internet, and it can also be used in hard disk encryption. Since inadvertent exposure of the key is a problem, a secure process to exchange this key is required. Generally key exchange is accomplished by encrypting the key with an asymmetric algorithm before transmitting. Once the key is received on the other side, it is decrypted and can then be used.

A block diagram of AES is shown in Figure 3.1. The algorithm consists of several rounds; the exact number depends on the key size. The process during each round is fairly straightforward. The first step places the plaintext into a matrix. Then each byte in the matrix is substituted by a value in a special table. Then the matrix rows are shifted, and then each column in the matrix is multiplied by a fixed polynomial. Finally, an extended version of the key is mixed in and the process repeats [27]. When the rounds are complete, the result is a string of unrecognizable characters. The idea is that this new stream of characters will bear no resemblance to the original.

TPM Uses Symmetric encryption using AES is not required in the TPM specification, and so the keys are not usually generated or used. However, when AES is implemented, its addition allows encryption of command parameters through the use of transport sessions. Transport sessions allow a user to open an encrypted channel and send encrypted commands to a TPM from another computer (perhaps the user is operating remotely over the internet). Transport sessions can also

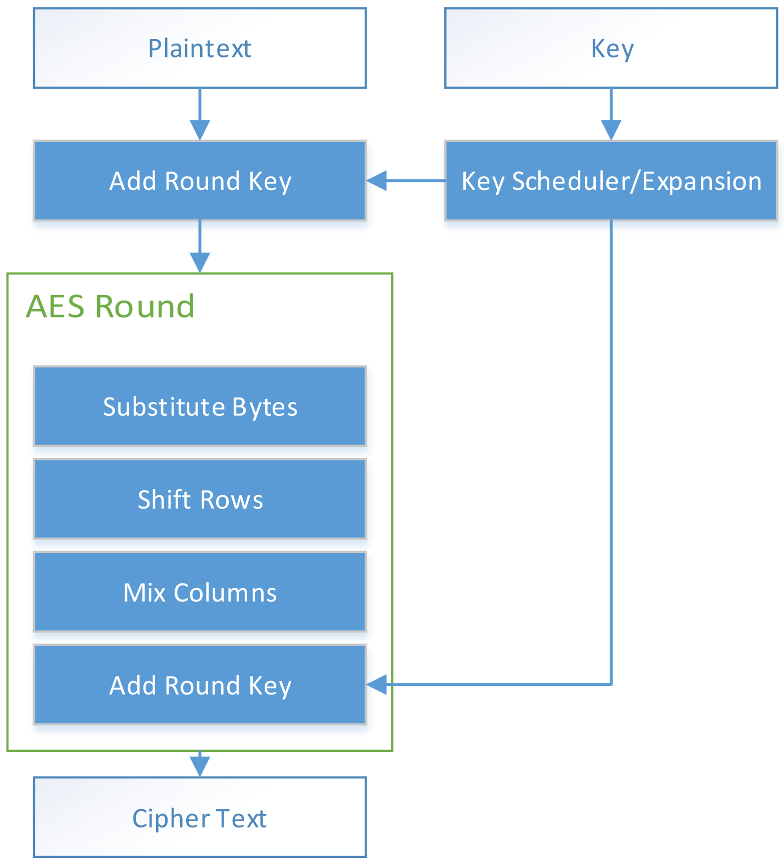


Figure 3.1: A block diagram showing how AES works. The AES Round section repeats a different number of times depending on the key length.

be used locally for added security. Compared with RSA, AES is much less resource intensive and even in software it has decent performance. Some high end processors from Intel can achieve tens of gigabits per second of throughput using multiple cores [28]. However, in cases where high throughput is desired it can still be beneficial to do hardware acceleration. Some hardware cores from commercial vendors such as Helion Technology can achieve speeds beyond 40 Gbps [29].

3.6 Symmetric Encryption Modes

Unlike RSA, AES and other symmetric algorithms are used to encrypt entire files. However, using these algorithms by themselves can leak information about what the plaintext looked like. In order to circumvent this problem, there are different 'modes' of operation. These modes encrypt data in blocks (16 bytes for AES-128), but allow longer messages to be linked together.

The TPM supports a few of these modes, including Cipher Block Chaining (CBC), Counter (CTR), and Output Feedback (OFB). Part of the strength of these modes is that they mix other information with the plaintext in order to form the cipher text. This mixing makes recovering information about the plaintext more difficult.

The figures that follow show how these modes operate. They each show three consecutive blocks of plaintext being processed. The CTR (Figure 3.3) and OFB (Figure 3.4) modes are designed so that they only need to use the encryption transform. In other words, they do not require separate hardware or software for decryption. The CBC mode (Figure 3.2) does use the decryption transform, so it may take up more space. In addition to only needing the encryption transform, the CTR mode can process blocks in parallel, as it does not depend on feedback from the result of the previous block. Notice that none of the modes feed unmodified plaintext to the encryption core. Instead, plaintext (or ciphertext) is mixed with other values, or an initialization vector (IV) is used. This reduces the risk that someone may spot patterns in the encrypted data.

Each mode uses an IV in some way. Like a nonce, this is simply a random value used to strengthen the encryption. This value must agreed upon in advance, and hence it must also shared when the key is given to the recipient of the message.

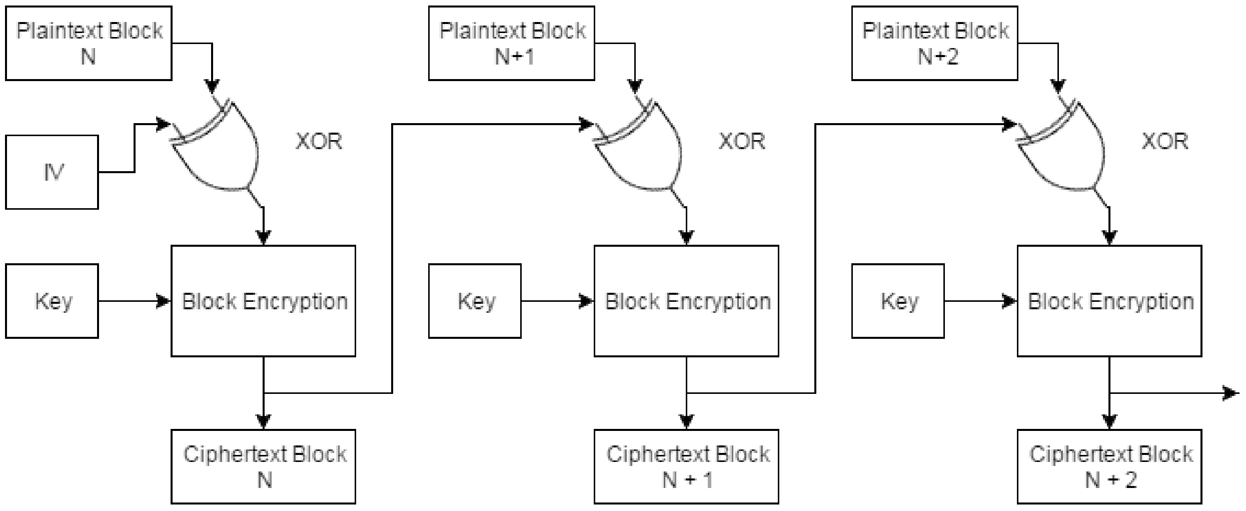


Figure 3.2: Encryption of three consecutive blocks of plaintext using the CBC mode, decryption is similar.

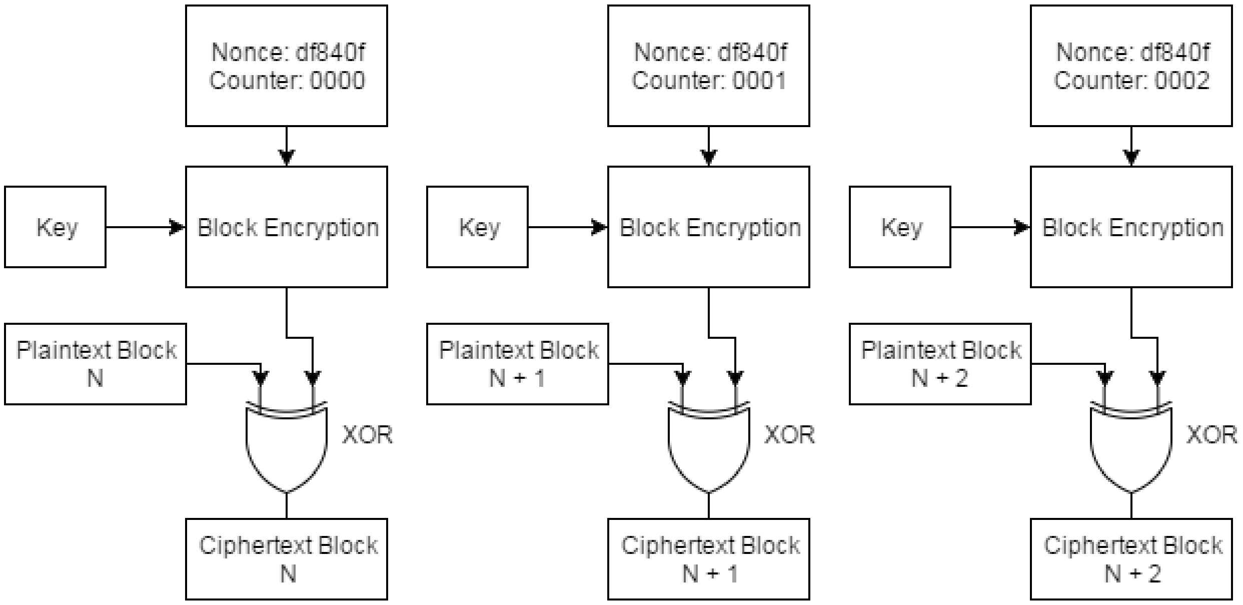


Figure 3.3: Encryption of three consecutive blocks of plaintext using the CTR mode. Encryption and decryption both use the encryption transform. The nonce stays the same for each block, but the counter increments by one each time. To decrypt, the ciphertext is used in place of plaintext and vice versa.

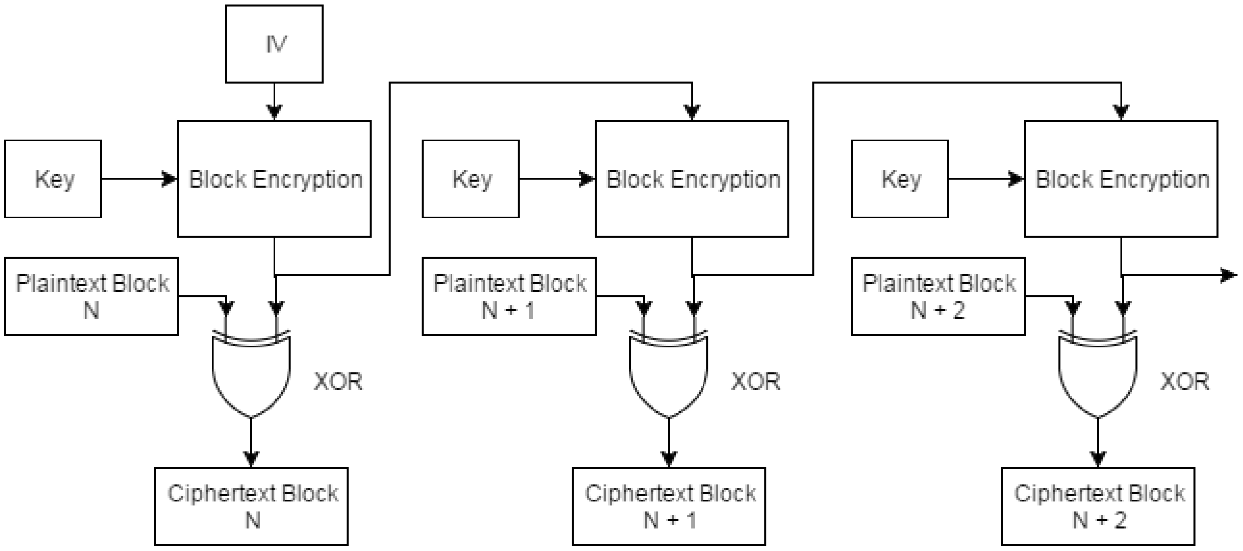


Figure 3.4: Encryption of three consecutive blocks of plaintext using the OFB mode. The output of the encryption transform is used as the input for the next block. Encryption and decryption both use the encryption transform. To decrypt, the ciphertext is used in place of plaintext and vice versa.

TPM Uses As mentioned previously, the TPM does not use symmetric encryption often, but it is important in some circumstances. Encrypted transport sessions can use the CTR and OFB modes. This allows the user to have more confidence that their messages are secure. CBC can be used to transfer sensitive data, such as owner or Storage Root Key authorization data, from the user into the TPM.

3.7 Random Number Generation

At the core of many cryptographic functions in the TPM is a random number generator. Random numbers are essential for generating secure secret keys. As such, it is important to have a good source of random numbers. If the numbers are not truly random, or have a short period and repeat frequently, it may be possible for an adversary to guess the generated key.

Pseudo-random number generators are commonly used when a source of true randomness is impractical or hard to come by. However, as their name implies, they do not generate 'real' random numbers. A pseudo-random number generator (PRNG) is an algorithm that generates numbers that look random, but are deterministic. They use a special value called a *seed* to initialize the algorithm. The seed determines all future numbers that will be generated. If two PRNGs use the same algorithm and have the same seed, they will generate the same numbers. Because of their determinism, these algorithms are normally considered insecure for applications in cryptography.

The TPM specification states that random numbers may come from a PRNG or a hardware source of entropy [3]. In order to increase the security and randomness of a PRNG, an alternative is to use a PRNG initialized by a true random seed [6]. Sources of true random data recommended by the specification include thermal noise, timing skew, or mouse and keyboard input. After it is collected, random data can be mixed in to the state machine controlling the PRNG. The specification recommends post-processing the data by using a hash function to further randomize it.

Creating a true random number generator is beyond the scope of this thesis. Even so, a better generator than the C library rand() function was desired. The rand() function is a commonly used random number generator, but it is a very simple generator and cryptographically insecure. In this project, we use the Mersenne Twister pseudo-random number generator [30]. This generator has a period of $2^{19937} - 1$. Even with a period this long, it is not cryptographically secure. However, Matsumoto and Nishimura suggest that using it with a hash function may help increase its usability for cryptography [30]. Although the generator used here is not ideal for applications in cryptography, it is sufficient to build a proof of concept. In the end, any acceptable random number generator can be used in its place. Ring oscillators are one such option for generating random numbers on an FPGA [31].

TPM Uses The TPM uses the random number generator for generating cryptographic keys and the nonces used for communication with the host. Several commands are available to allow the host computer to use the TPM's built-in generator. These commands allow mixing in more random data from the host system to add entropy, and also generating random numbers for use elsewhere [3].

3.8 Cryptanalysis

Whenever someone attempts to conceal a message through cryptography, someone else may be trying to break their code. This is called cryptanalysis. There are several different forms of cryptanalysis, including side-channel attacks, ciphertext analysis, and even using brute force to try to crack the cipher.

SCAs A side-channel attack involves getting information from an unusual channel. For example, measuring the amount of time an RSA private key operation takes can reveal information about the secret prime factors [11]. Other common side-channel attacks include measuring the power consumption of a circuit during a sensitive operation, or recording the output of an algorithm when specific faults are introduced [10]. If a device leaks information like this, it can drastically reduce the search space for a secret key, or even allow privilege escalation. *Rowhammer* is a recently uncovered vulnerability that affects the DRAM of a system [32]. In this attack, an attacker can alter memory that they would not otherwise be allowed to access. This is done by repeatedly reading rows of computer memory. After many successive reads, adjacent rows may accumulate bit errors. Researchers found that this attack could be used to gain kernel privileges and write access to all physical memory on 64-bit Linux systems. Software may be able to fix some of the effects of this vulnerability, but the hardware itself cannot be fixed. If a DRAM module is vulnerable to this attack, the only way to fix the hardware is to buy new DRAM that is not vulnerable.

Chapter 4

Related Work

TPMs have many practical applications. In [33] the authors propose using a TPM to facilitate trust between vehicles that are connected in a vehicular ad-hoc network. Cars in this network would communicate position, road conditions and other information to each other to help avoid accidents. In this situation a bad actor could potentially cause collisions, or cause cars to avoid sections of the road by presenting bad data. If the software in the car is verified and has not been tampered with, then messages from one car to another can be signed and those messages can be trusted.

The authors of [34] create a software TPM that is modular. They write the code for their TPM using Java. The design they propose has some similarities with the design that will be presented here. It was designed to allow components of the TPM to be dynamically loaded, swapped out, replaced or upgraded with relative ease, without affecting other components. Each of the separate modules can be packaged as a JAR file and loaded individually.

Eisenbarth et al. proposed creating a TPM on an FPGA [13]. In their proposed design the application resides on the FPGA with the TPM, and the TPM forms part of a chain of trust for the application. The application can be trusted because the TPM is trusted, and the TPM is trusted because it was included in a trusted bitstream, which presumably came from a trusted person. Since the TPM is included in the chain of trust, the entire design can be verified. The authors state that a TPM implemented on an FPGA would have advantages such as flexibility, customization, and openness. They did not implement their proposed design; instead they provided estimates of what it would look like. The work in this thesis will build on their ideas and demonstrate a proof of concept of a similar design. The main difference in this work is that the TPM is assumed to be on the FPGA, but the applications will be running on the computer's CPU.

Schellekens et al. expand on the work done by Eisenbarth et al. and add an external non-volatile memory to the reconfigurable TPM [14]. In particular they establish a method whereby the TPM can be secured even if components of it are upgraded. They use the nonvolatile memory to track revisions to the software and the bitstream to ensure that downgrading to a less-secure version is impossible.

Glas et al. discuss another FPGA based architecture, but rather than using a TPM implemented on an FPGA, they use a conventional TPM [35]. They use the TPM to check whether partial reconfigurations of the FPGA are allowed. During each boot, hashes of the FPGA configuration and software are stored in the TPM. On a subsequent boot, or during partial reconfiguration, the TPM can check these hashes against known good configurations. If a reconfiguration occurs that is not allowed, or if the partial reconfiguration is not in an allowed area of the FPGA, the TPM shuts down. Shutting down the TPM prevents an adversary from accessing secure data.

In [36], Huang et al. describe a system called TFSES that contains a TPM, or a device similar to a TPM called a Trusted Computing Module (TCM), and also an FPGA Controller. In their design, the TPM/TCM contains a RISC processor, a Data Encryption Standard (DES) module, and a random number generator [36]. At boot, the FPGA controller suspends the processor before the firmware for the processor loads. It sends the firmware image to the TPM where it is verified. If the TPM can verify that the firmware image has not been tampered with, then boot proceeds. In this state, the TPM functions as normal. If, however, the image was tampered with, the TPM shuts down and does not provide trusted services to the embedded processor.

A TPM is usually connected to the motherboard near the CPU. Zhang et al. propose creating a TPM that resides on a USB key [37]. It would have its own screen and button and could be used on different computers. Since the device connects over USB, it is possible that malicious drivers could compromise its security. It is also possible that other devices may be able to eavesdrop on the connection it has with the host computer. To mitigate some of the risks, their TPM begins operation before the OS loads. They propose using the Extensible Firmware Interface (EFI), similar to the BIOS on older systems, to begin communicating with the device at early boot time. This would allow the TPM to establish a chain of trust on the system before malware or unauthorized software loads. They created this portable TPM to make it easier to do remote authentication and attestation, but it additionally supports many of the other features of a standard TPM. This work shares some similarities with that described in this thesis, because the prototype I have built also connects via USB.

One of the security problems with FPGA based architectures is the potential for the insertion of malicious code into the bitstream. For example, some FPGAs allow adding and removing functionality at runtime in a process called dynamic partial reconfiguration. Johnson et al. discuss ways to authenticate different add-on modules for a FPGA-based Internet of Things (IoT) device [38]. One such method is to use a type of security primitive called a Physically Uncloneable Function (PUF). A PUF uses small manufacturing variations to generate a unique response to a challenge. When an administrator sends a challenge to an IoT device, the device sends the challenge to a PUF and then sends the response back to the administrator. If the response of the PUF matches the stored value, the add-on is approved for use and can be configured. With current technology, there is no way to make two PUFs produce the same response. PUFs can be created on an ASIC or on an FPGA. In [38], the authors use a PUF to authenticate different components of a design.

Although not integrated into a TPM, many people have done and are doing research on implementing cryptographic algorithms on an FPGA. This includes work on the algorithms currently used in the TPM, as well as their successors (see [39, 40, 41]). In addition, other work has been performed to help secure FPGAs and their bitstreams (see [14, 42, 43]). Although not directly addressed in this work, the topics discussed could be applied to make the implementation proposed in this thesis faster, more usable, and more secure.

Chapter 5

Implementing a Reconfigurable TPM

This chapter will discuss an implementation of the TPM specification on an FPGA. This TPM implementation aims to solve some of the problems of the conventional TPM. In particular, we seek answers to the questions listed below:

- Can we design a TPM whose inner workings are transparent?
- Can we provide an easy way to upgrade the hardware components of a TPM?
- Can we add new algorithms to an existing TPM if old ones become insecure?
- Can we build a functional open source TPM?

When we say we want to design a TPM whose inner workings are transparent we mean that the TPM is not proprietary, that is, its functionality is not obscured. The framework shown here, along with the open source execution engine, should help insure that the workings of a reconfigurable TPM can be subject to examination and security audits. The framework proposed should also allow us to easily add new algorithms to an existing reconfigurable TPM. The reconfigurable nature of the FPGA, in concert with the framework, allows older implementations of algorithms to be replaced with newer, faster ones. We also attempt to build a functional open source TPM. Having an open source example to study could help companies or users to build their own vendor-independent TPM. In addition, an open source, easily configurable TPM could benefit those without access to a built-in hardware TPM.

First we will explore creating an extensibility framework for the reconfigurable TPM. This framework will allow us to quickly and easily change the functionality of the TPM without having to edit huge portions of the source code. We show Zynq TPM, a functioning prototype TPM built on an FPGA development board. We show how it was created, and how it can be modified

Table 5.1: Example of defined interface for cryptographic modules.

Hash	HMAC	Symmetric	Asymmetric	Random Numbers
HashInit HashUpdate HashFinal	HMAC	Encrypt Decrypt LoadKey GenerateKey	Encrypt Decrypt LoadKey GenerateKey Sign Verify	StirRandom GetRandom

to fit the needs of the developer. The next chapter will compare the Zynq TPM (ZTPM) against a commercial desktop TPM, show FPGA resource statistics, and examine the benefits and limitations of the design.

5.1 Extensibility Framework

Allowing new cryptographic modules to be added – and old ones to be replaced – is central to the work in this thesis. If adding or upgrading modules requires extensive changes to the TPM's code, it will discourage developers from using the ZTPM, hence it is desirable for this process to be as simple as possible. This simplicity comes from interfaces. An interface is basically a list of common operations. For example, most televisions have a power button, channel up and down buttons, and volume up and down buttons. Similarly, encryption schemes have an encrypt operation, a decrypt operation, and a way to generate a key.

The TPM specification calls for the use of several different types of cryptographic algorithms, including hash functions and encryption schemes. We created interfaces for each of the different types: asymmetric algorithms, symmetric algorithms, hash functions, and keyed-hash (HMAC) functions. Since each type of algorithm has a different interface, this helps ensure correct behavior. It prevents us from say, accidentally attempting to encrypt data using a hash (because encryption does not exist in the hash interface). Table 5.1 shows one possible way of creating these interfaces. (The actual code has some additional functionality, for more details on how this is implemented, see Appendix A.)

Since the code for this framework was written in C, the interfaces take the form of structures. These structures (SymmetricEncRegistration, AsymmetricEncRegistration, HashRegistration, and HMACRegistration) contain function pointers that point to the module driver code (for

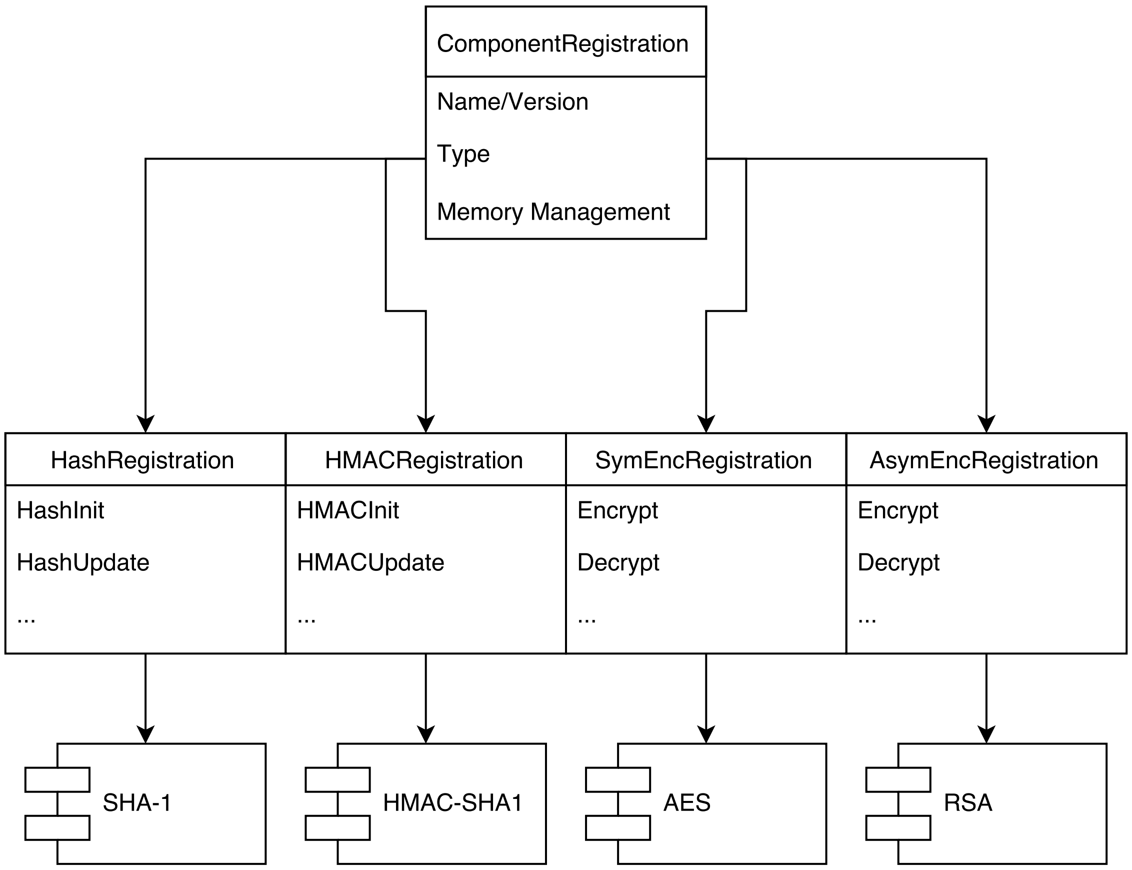


Figure 5.1: The shim layer abstracts away the details of the individual cryptographic algorithm.

example, calling the ‘Encrypt’ function pointer might call ‘RSAEncrypt’). In addition, each module also has a ComponentRegistration structure that tracks basic information like name, version, and memory management functions. Although random number generation also has an interface, it does not require its own structures, because it is is assumed that the TPM only needs one random number generator. However, this module can still be easily upgraded.

A shim layer, or set of wrapper functions, is responsible for selecting the requested algorithm and calling the appropriate function. Every call to a cryptographic routine goes through this shim layer that decides which algorithm should be used. Once the correct algorithm is determined, the shim layer calls down into the code for that specific module. Figure 5.1 shows how this is organized.

5.2 Zynq TPM

For this design, we built a TPM on a Digilent Zybo board. This board contains a Zynq chip with an dual core ARM processor, 16MB nonvolatile flash storage, 512MB RAM, and an Artix-7 FPGA [44, 45]. A high level block diagram for this design is shown in Figure 5.2.

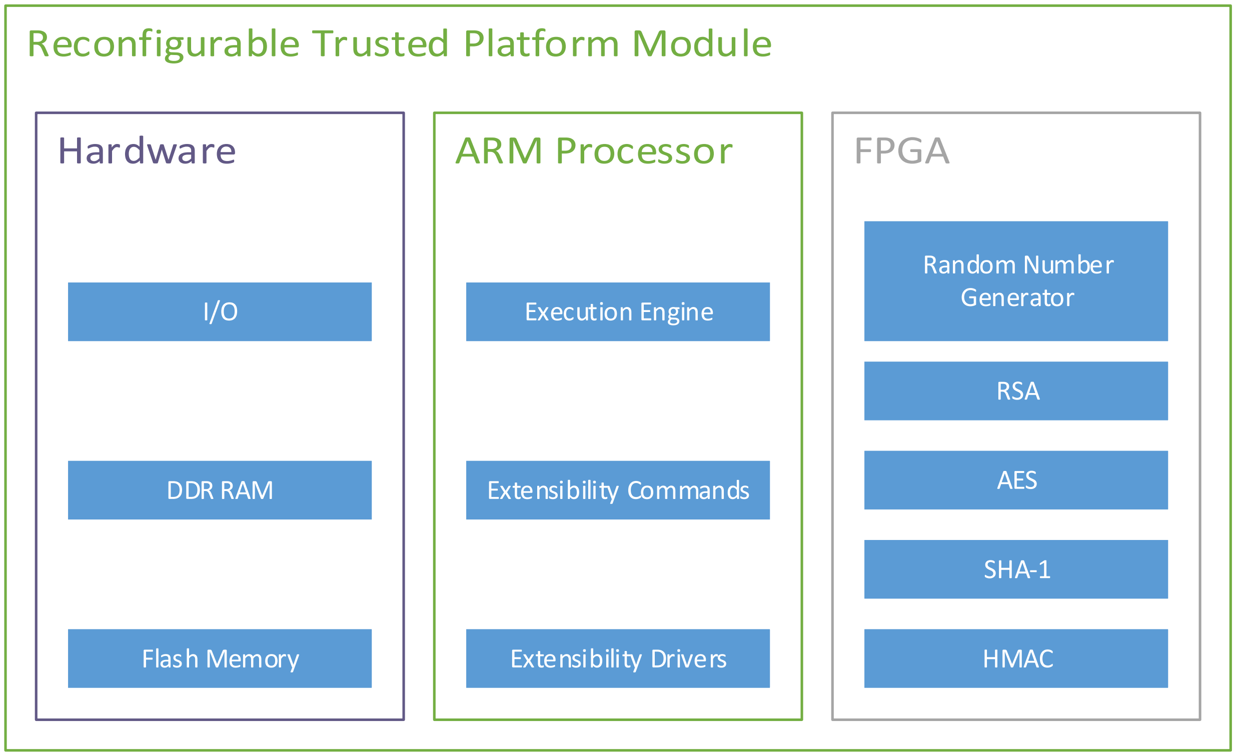


Figure 5.2: Components for the ZTPM are split between hardware, software, and the FPGA.

This TPM design differs from a conventional TPM in several ways. The software has been modified to support adding new cryptographic accelerators, and the cryptographic functions have been moved from software onto an FPGA. The FPGA provides flexibility, rapid development (relative to developing custom hardware), and speed. Normally, a TPM would reside on the LPC bus, or another serial bus located close to the CPU. Instead of using the LPC bus, this implementation uses a USB UART.

The USB UART provides convenience for the developer, because it does not require adding special hardware to their motherboard. In a commercial product, a USB UART could be a security risk, because it is much easier to eavesdrop on than the LPC bus. However, other research has shown techniques that could be employed to secure a USB-connected TPM, such as making use of functionality provided by the BIOS or EFI [37]. In this case, the TPM would have the additional advantage of being portable. Because of this, we consider the USB approach sufficient for this proof-of-concept design. With the proper tools, a well designed TPM could be moved to the LPC bus (or another serial bus) without many changes.

5.2.1 Design Firmware

A critical part of this work was selecting the software and FPGA cores that would be used for the TPMs. In an attempt to build a completely open source TPM, all the software and FPGA cores chosen for use in this design were open source. If another developer were building a reconfigurable TPM, they could of course choose commercial cores, or design their own. There is nothing special about any of the cryptography cores that were chosen. The tables below (Tables 5.2 and 5.3) show where the cores can be retrieved if they were downloaded.

Table 5.2 shows the software resources used, along with the approximate number of lines of code. All software that was selected had been written in C. For the software implementations of these algorithms in the design, I used existing code as much as possible. If a suitable module could not be adapted for use with GMP (the GNU Multiple Precision Arithmetic Library), I wrote my own simple implementation using the specification for the algorithm. I ended up porting an RSA implementation from Java to C (about 760 lines of code), and also a software AES module (about 540 lines). These modules were based on example code found in Wagner's book [46] and their respective specifications. Additionally I wrote the code for the AES modes (about 420 lines of code). The execution engine makes up the majority of the code for this project. The IBM software TPM was adapted for this purpose [47]. Changes were made to accommodate using a small embedded system version of GMP instead of the heavier OpenSSL that it originally used.

Table 5.2: Software Modules Used in the Reconfigurable TPM.

Module Name	Lines of Code	Website	Reference
Execution Engine (IBM Software TPM v1.2)	57321	sourceforge.net	[47]
GNU Multiple Precision Arithmetic Library	3618	gmplib.org	[48]
RSA Encryption/Decryption	1043	openssl.org[1]	[49, 46, 26]
AES Encryption/Decryption	541	(Original)[2]	[27, 46]
SHA-1 Hashing	235	(Original)[3]	[47, 23]
HMAC with SHA-1	141	(Original)[4]	[47]

Table 5.3: FPGA Cores Used in the Reconfigurable TPM.

Core Name	Language	Lines of Code	Website	Reference
Modular Exponentiation	Verilog	1668	cryptech.is	[50]
AES Encryption	Verilog	681	opencores.org	[51]
AES Decryption	VHDL	2148	opencores.org	[52]
Mersenne Twister PRNG	Verilog	177	github.com	[53]
SHA-1 Hashing	Verilog	639	github.com	[54]

The FPGA cores chosen for this project were modified to communicate using an AXI bus protocol. This usually required adding a wrapper around the existing HDL, as most cores I saw did not natively support AXI. Other modifications to the original HDL were avoided. However, in the case of the two separate AES cores, I added additional HDL to combine them into one functional unit (about 100 additional lines). Once this was done, driver software was written in order to interface with the FPGA modules from the execution engine and extensibility framework. Most of the FPGA cores were written using Verilog, but one was written in VHDL. Table 5.3 shows the cores, the language used, and the lines of HDL in the original core. The IDE used to synthesize and build all components of these designs was Vivado 2014.2.

As mentioned, the extensibility framework provides a module interface specification for each type of algorithm found in the TPM. The primary concern was to adapt the modules to conform to this interface specification. As long as this occurs, the modules can be used. Neither

[1]OpenSSL padding code was used for RSA. The RSA operation was ported to C from the example Java code in [46] with help the RSA specification.

[2]The AES operation was ported to C from the example Java code in [46] with help the AES specification.

[3]The SHA-1 software module uses the reference implementation found in RFC 3174. It was adapted to work with existing code in the IBM TPM.

[4]The HMAC software module was based on the reference implementation found in RFC 2104. It was adapted to work with existing code in the IBM TPM.

Table 5.4: Final Lines of Code Count for the ZTPM

Core/Module Name	Language	Lines of Code
ZTPM Firmware/Software	C	53965
AES	Verilog/VHDL	3240
Mersenne Twister PRNG	Verilog	422
Modular Exponentiation	Verilog	2075
SHA-1	Verilog	1330
Total	-	61032

FPGA nor software modules were optimized for speed, area, code size, or power consumption. However, these optimizations could and should be made by researchers in the future. To ensure that the modules were operating correctly, tests were run using small sets of known answer data. As they were incorporated into the TPM, they were tested again with a commonly used subset of TPM commands. Although not exhaustively tested, a central assumption in this entire body of work is that modules could be easily replaced if they did not function adequately. For example, the modular exponentiation FPGA core did not function properly. This was discovered during testing when prime number tests returned the wrong answer for numbers that are known to be prime. In order to build a working TPM, a new module was needed. A small software core was created, and added to the extensibility framework. After the core was created and tested, it only took a few hours to add it to the framework and integrate it into the ZTPM, showing that the framework is indeed making the process easier.

After completing all the work described in this thesis, the TPM firmware contains about 54,000 lines of C code, and almost 7,000 lines of HDL for the FPGA. Although the execution engine started with about 57,000 lines of code, the number of lines of software was reduced by removal of redundant code and dependencies. Additionally, replacing the software cryptographic libraries with FPGA versions contributed to this decline. Increases in line count for the hardware modules are due to adding AXI wrappers. Table 5.4 shows the final counts. The size of the compiled software for my prototype is 1.5MB. Since the ZTPM has 512MB of RAM, this is not a problem.

5.2.2 Implementation Details

A primary contribution of this thesis is the design of a framework that can easily allow the addition of cryptographic functionality, or to replace functionality without rewriting the firmware for the entire TPM. For this reason, and because it is advantageous for the TPM to be open source, the core of both implementations is the IBM software TPM [47]. The IBM software TPM is a simulator used to develop new applications for existing TPMs. It implements the entire version 1.2 specification, including optional parts like AES encryption. Because it was not built to run on its own hardware as a 'real' TPM, it depends on the underlying operating system, in this case Linux, for several features. To save resources the ZTPM does not run Linux, and so it lacks many of the Linux libraries. In order to adapt IBM's code to the FPGA, the first step was to remove all operating system dependencies, so that the software could run on the bare-metal Xilinx platform.

The two main dependencies were I/O and OpenSSL. The original software used sockets for I/O, and this code was rewritten to use a UART over USB. Xilinx makes this easy by providing UART drivers for use when there is no host operating system. Removing OpenSSL proved to be more challenging. Most Linux distributions come with OpenSSL, an open source library for secure communication. OpenSSL provides a wonderful cryptography library. However, it also depends on some operating system support, and it is fairly large. Most of the OpenSSL code was successfully removed and replaced with the GNU Multiple Precision Arithmetic Library (GMP). This library even comes with a lightweight implementation specifically designed for embedded systems.

After the operating system and OpenSSL dependencies were removed, the next step was to extract each cryptographic algorithm into its own module to make it conform to the interface defined by the structures in the extensibility framework. During this process, generalizing function parameters made it possible to remove entire functions that became redundant. When all of this was completed, five separate and independent modules remained: random number generation, RSA, SHA-1, HMAC, and AES.

Each remaining software module was removed and replaced by the corresponding hardware core and its driver (except the malfunctioning modular exponentiation core, which was replaced by its software counterpart). This process was straightforward due to the extensibility framework.

All that was needed was to ensure that the drivers for the FPGA cores conformed to the interfaces created in the extensibility framework.

In addition to these code changes, several vendor-specific commands were added. The specification states that a manufacturer may create and use these commands, as long as they do not expose the secrets of the TPM. The TPM_GetCapability command reads settings and other information from the TPM and sends them to the host CPU. While not a vendor-specific command itself, manufacturers may add their own capabilities to this command. We added the capability to read out the names of the new modules, and the algorithms that they support. Another capability may report which algorithm is selected as the 'primary' algorithm. Through TPM_SetCapability, we can pass a value to the TPM to allow a change. We can also set these changes to be allowed only if authorized by the TPM's owner password.

Adding a New Core

These changes allow a user to upgrade existing cryptographic modules and add new modules. In a conventional TPM, a user or company cannot add new functionality to an existing TPM. Conventional TPMs are only capable of firmware (software) upgrades. The framework presented in this section will allow the hardware of an existing TPM to be customized during its lifetime. The framework is designed to be as simple as possible; below we consider the process for adding a new hash algorithm:

1. The developer creates or acquires an IP to add to the TPM.
2. An AXI wrapper/driver is created if necessary.
3. The developer creates a HashRegistration structure.
4. The developer creates a ComponentRegistration structure.
5. The new module is added to the installed module list.
6. New firmware and a new bitstream are uploaded to the board.

Adding completely new algorithms is a similar process, but there are a few other considerations. Fortunately, the TPM specification includes many features that make this process easier.

Recall that the TPM contains lists of supported algorithms and modes to determine how keys and data should be used (TPM_ALG_ID, TPM_ENC_SCHEME, and TPM_SIG_SCHEME). If we need to add a new encryption or decryption algorithm, we can simply add entries to these lists to support our new modes. There are checks inside the software to ensure that modes are paired with the correct algorithms. If an existing mode can be used with a new algorithm, those checks must be updated to support the new algorithm.

As an example, consider adding a new asymmetric algorithm, NTRU (Nth Degree Truncated Polynomial Ring, a new quantum computer resistant algorithm). We would follow the steps above to add this module to our TPM, in addition to adding a new TPM_ALG_NTRU entry to the list of supported algorithms, TPM_ALG_ID. Once the new module is fully installed, we can use it to create keys for that algorithm simply by specifying the algorithm type to be TPM_ALG_NTRU. This key would then be inserted into the hierarchy and protected just like any other key. Fortunately, no code changes would be required to interface with another TPM. Trying to use this key on a standard TPM would simply report use of an unsupported algorithm.

It may be that the developer of the system does not require backward compatibility, or the ability to communicate with systems using a traditional TPM. Perhaps a developer decides to set the default cryptographic modules to something other than those in the original specification. It should be noted that support for such a change was not the intent of this thesis work, so although possible, it would require more development to remove dependencies between modules. RSA for example, depends on SHA-1 for message padding, and it will not work correctly without it.

Chapter 6

Results and Analysis

The previous chapter described an implementation of a reconfigurable Trusted Platform Module. This chapter contains test results, reports and statistics for the ZTPM and compares it against a desktop TPM. We will discuss FPGA resource utilization and power usage for the ZTPM, we will examine the performance of the cryptographic modules on the FPGA relative to their software counterparts, and we will compare the performance of the ZTPM and a conventional TPM. Following the presentation of the results, the benefits and limitations of a reconfigurable TPM will be discussed.

6.1 FPGA Resource Utilization

Tables 6.1 and 6.2 show the slice LUTs and registers (respectively) used by the ZTPM. The system components row refers to the LUTs (or registers) used for the AXI bus, the memory interfaces, etc.

We can see that the component using the most slice LUTs is the AES module (Table 6.1). If we dig a little deeper into the data, the most expensive component was actually the AES decrypt core. It would be possible to save space on the FPGA by using modes of AES (OFB, CTR) that do not require a separate decryption core and can use the encryption core for both operations.

Even without this simple optimization, about half of the FPGA is left unused, with the ZTPM using about 47%. This is important because additional capabilities like new asymmetric algorithms or hash functions can be added without removing any existing functionality. It could also be possible to add redundant modules to speed up calculations or reduce side-channel vulnerabilities.

The Zynq chip used in this design has about 270KB (2.1Mb) of BRAM tiles (Block RAM, distributed around the FPGA), where each tile is 36Kb [44]. Table 6.3 shows that the modules

Table 6.1: Slice Look Up Table Utilization

Module	Slice LUTs	Percent
AES	4487	25.49%
SHA-1	1516	8.61%
Modular Exponentiation	1371	7.79%
Mersenne Twister PRNG	348	1.98%
System Components	612	3.48%
LUTs Used	8334	47.35%
Remaining	9266	52.65%
Total	17600	100.00%

Table 6.2: Slice Register Utilization

Module	Slice Registers	Percent
AES	1162	3.30%
SHA-1	1587	4.51%
Modular Exponentiation	838	2.38%
Mersenne Twister PRNG	205	0.58%
System Components	699	1.99%
Registers Used	4491	12.76%
Remaining	30709	87.24%
Total	35200	100.00%

for AES and modular exponentiation use most of the BRAM tiles in this design. AES uses many of them for the byte substitution step of the cipher, and exponentiation requires a lot of RAM for dealing with large numbers. If more BRAMs are required for use with a newer or faster algorithm, a different FPGA with sufficient resources could be used. If money is not an issue, Xilinx does sell Zynq chips that have many more BRAMs (up to several megabytes), but they are more expensive than the chip used in this work (costing up to thousands of dollars).

6.2 Power Usage

Table 6.4 shows power estimates from the synthesis tool for the ZTPM. It shows that the largest power user is by far the ARM core. In fact, the estimate for all the FPGA cores combined shows them using less than 10% of the total power. For comparison, a commercial Infineon TPM uses less power. A data sheet from Infineon for a TPM designed for Chromebooks provides a maximum current of 25 mA at 3.3 V, giving a maximum power usage of 0.083 W [17]. The true

Table 6.3: Block RAM Tiles Utilized

Module	BRAM Tiles	Percent
AES	11	18.33%
SHA-1	0	0.00%
Modular Exponentiation	9	15.00%
Mersenne Twister PRNG	2	3.33%
System Components	0	0.00%
BRAMs Used	22	36.67%
Remaining	38	63.33%
Total	60	100.00%

Table 6.4: ZTPM Power Usage

Module Type	Power (Watts)
AES	0.076
Modular Exponentiation	0.048
SHA-1	0.007
Mersenne Twister PRNG	0.005
Zynq ARM Core	1.349
Total Dynamic Power	1.485
Total Static Power	0.126
Total Power	1.611

power usage of the ZTPM is likely lower than these estimates when considering the duty cycle and frequency of use of the accelerators. It could be lowered further by allowing the processor to enter into an idle state while waiting for packets. Infineon makes these assumptions in their estimates as well.

6.3 Module Performance

Table 6.5 shows the speedup of the cryptographic modules implemented on the FPGA relative to those running in software on the Zynq processor. It is important to mention here that the speed of each individual module is important. For example, almost every command will use the SHA-1 hash module in some way. If it were slow, the entire TPM would slow down.

The performance of the SHA-1 modules was tested in several steps. First a value was hashed, then the hash context was saved. Another value was then hashed. The original context

Table 6.5: Speedup of Individual Modules. FPGA Versus Software.

Module	SW Module Clock Cycles	FPGA Module Clock Cycles	Speedup
SHA-1	69,584	40,936	1.7
AES	54,142	1,820	29.8
Mod Exp	113,958,720	52,004,479	2.2
RNG	9,508	538	17.7

was reloaded and another message was added to the hash. In this test the FPGA core is about 1.7x faster than the reference software implementation.

The FPGA core for the Mersenne Twister PRNG is 17.67x faster than a Mersenne Twister software implementation. For comparison, the FPGA core is 1.97 times slower than the C library rand() function. However, recall that the rand() function should not be used for cryptography, because the random values have a short period (see Section 3.7).

Both the FPGA and software AES modules were tested by loading a key, and then encrypting and decrypting a block of data (128 bits). The test was repeated five times and averaged. Results varied less than 1% between runs for both cores. Only the modules themselves were tested, that is, only the raw AES algorithm was tested. The different encryption modes (OFB, CTR, etc.) were not included for this test. In this test, the hardware implementation was about 40x faster than the software version. When the test excludes loading the key, the speedup is about 30x. Note that loading a key is done only once before using it on any amount of data, so this initial cost is amortized. The throughput of the FPGA core is approximately 43 Mbps, while the software core only achieves 1.5 Mbps. Cores from Helion Technology start around 20 Mbps for a small FPGA core, but can go beyond 25 Gbps [29].

It is important to note that the modular exponentiation core chosen for the design did not produce correct values in initial testing. If this were discovered in an environment where this TPM was being built or compiled for manufacture it would require the developer to simply drop in a new core. In this case, the RSA algorithm was switched over to a software implementation that works, but is much slower. In order to test the speed of these modules, ten exponentiations of random 1,024 bit values of the form $2^{n-1} \pmod{n}$ were performed. The resulting time is the average of those ten runs. For this type of RSA operation, the (nonworking) FPGA core was about 2.2 times faster than the software version. It is unsurprising that it was faster, but a greater speedup had been expected. Common commercial modular exponentiation cores such as the Helion Modular Exponentiation family of cores are capable of up to 43 RSA operations per second on an Artix-7 FPGA [55]. The hardware core on this system achieves 6.4 RSA operations per second, and the software system achieves 2.9 RSA operations per second (on the 667 MHz Zynq CPU). For reference, the slowest core listed on Helion's website achieves 3.2 RSA operations per second. It is likely that if the FPGA core were working correctly, it would achieve higher speeds.

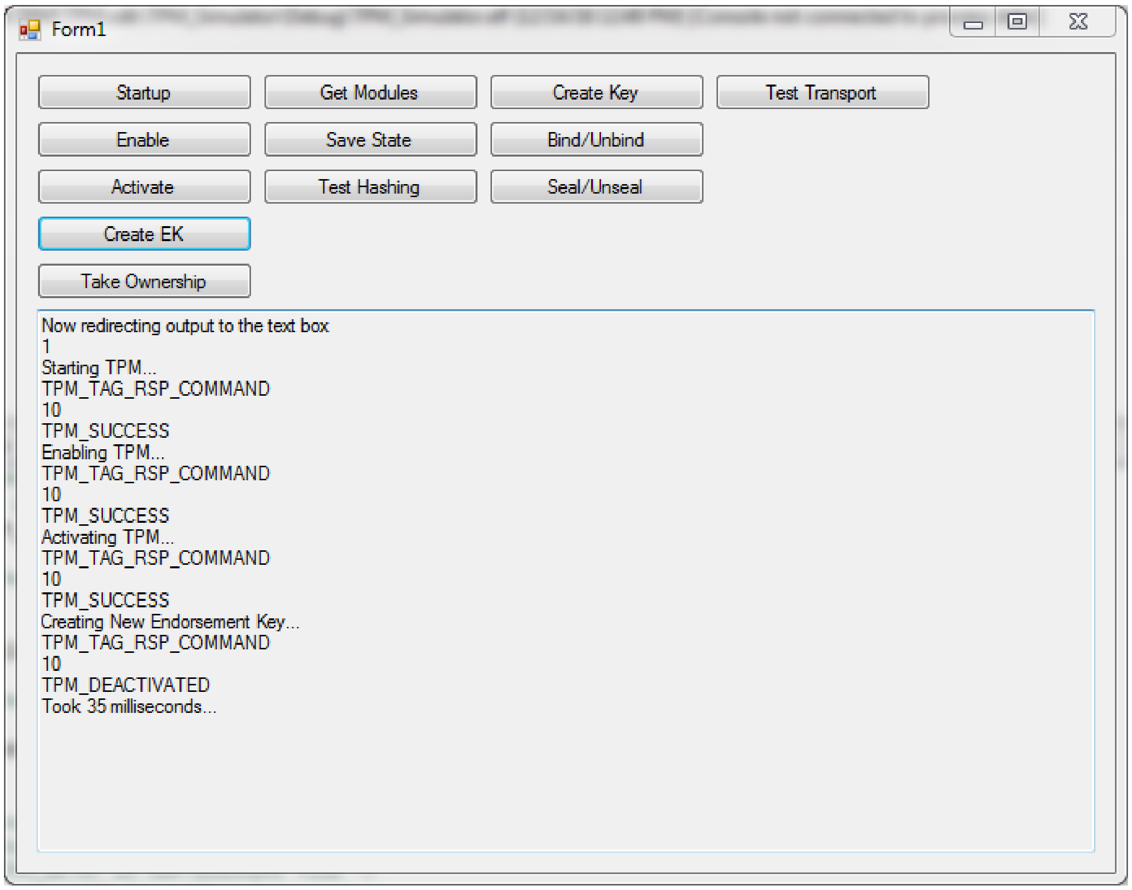
Although the cores that were used in this work were not particularly fast, there are faster cores available. If a TPM developer had the funds to purchase IP or the time to develop their own, they could expect higher speeds. If it really mattered, there are larger and faster FPGAs available as well. This work was done primarily to prove that such a TPM could be built; building high speed hardware accelerators is beyond the scope of this work.

6.4 System Performance

To compare performance against an existing TPM, I wrote an application to interface with both the ZTPM and the Infineon TPM inside my computer (an HP Compaq 6200). It was written in C# and contains 4,365 lines of code. This application (shown in Figure 6.1) allows for manual execution of TPM commands, including starting up the TPM, taking ownership of the TPM, creating RSA keys, decrypting, sealing, unsealing, and hashing data.

If new encryption schemes, hashes or other functionality are added, support must be integrated into the TPM application running on the host system. For an experienced developer, it should take less than a week to implement a subset of the functionality needed for a properly running application. Because the new framework does not require changes that would break the TPM specification, any applications written for a reconfigurable TPM should work correctly on any existing hardware TPM. This is the case with my application. It contains commands that can request information about the installed modules in the ZTPM, but when those commands are sent to the TPM in my desktop, they have no effect. However, it is important to note that if algorithms not originally in the specification are used, data will not be able to be shared between the ZTPM and a conventional TPM (as it would have no way to decrypt or read the data).

A subset of TPM operations was selected to be tested, including common operations and operations required to properly set up a TPM. This was done to compare performance, and also



```
Now redirecting output to the text box
1
Starting TPM...
TPM_TAG_RSP_COMMAND
10
TPM_SUCCESS
Enabling TPM...
TPM_TAG_RSP_COMMAND
10
TPM_SUCCESS
Activating TPM...
TPM_TAG_RSP_COMMAND
10
TPM_SUCCESS
Creating New Endorsement Key...
TPM_TAG_RSP_COMMAND
10
TPM_DEACTIVATED
Took 35 milliseconds...
```

Figure 6.1: TPM Test Application

to ensure that the basic functionality required by the TPM specification was operating correctly. Table 6.6 shows the time taken by these operations on the prototype ZTPM and the TPM inside my desktop computer. Times are given in milliseconds.

The time required for TPM_Startup, TPM_CreateEK, and TPM_TakeOwnership on the desktop TPM are not available, because the operating system executes these functions before the test application can load. The tests focus on exercising the cryptographic modules on the FPGA. The RSA module is exercised by TPM_TakeOwnership, TPM_CreateEK, TPM_CreateWrapKey, and TPM_Unbind. We test the random number generator by using TPM_GetRandom, a command that returns a requested number of random bytes. The SHA-1 module is tested using a combination of TPM_SHA1Start, TPM_SHA1Update, and TPM_SHA1CompleteExtend. These commands start a hash, add blocks of data to the ongoing hash and then terminate it. The final command, TPM_SHA1CompleteExtend, moves the result of the hash into a selected PCR. The AES module

Table 6.6: Speedup of system tasks. ZTPM versus conventional TPM.

TPM Function	Conventional TPM (ms)	ZTPM (ms)	Speedup
TPM_Startup	N/A	1788	N/A
TPM_CreateEK	N/A	53400	N/A
TPM_TakeOwnership	N/A	110000	N/A
TPM_CreateWrapKey	1416	60600	0.02
TPM_Unbind	534	4695	0.11
TPM_Seal then TPM_Unseal	896	5030	0.18
TPM_GetRandom	74	75	0.99
SHA-1 (Start, Update, Complete)	223	110	2.03
Transport Session (Get Version)	114	109	1.05
Transport Session (Get Random)	122	120	1.02

is tested by retrieving the TPM version number in an encrypted transport session. Since this information is relatively short, 128 bytes of random data was retrieved in a separate transport session test in order to fully exercise the AES module.

As shown in Table 6.6 the ZTPM does not perform well in any test that uses RSA, since all such tests use the modular exponentiation core. We can see the benefit of a working hardware exponentiation core by looking at the raw data. In the case of TPM_CreateWrapKey the desktop TPM provides a standard deviation of about 23ms. The ZTPM gives a standard deviation of about 53300ms! This clearly shows the value of having an optimized key generation algorithm, as well as a fast modular exponentiation core. Although the hardware accelerated modular exponentiation core does not work on the ZTPM, the tests in Table 6.5 show that even without optimizing the key generation algorithm, we could expect a little better than a 2x speedup with a working core. However, even with the slow software core, the ZTPM still provides acceptable performance. There is no requirement in the TPM spec for high speed key generation. Key generation is usually performed infrequently, so it may be acceptable to wait a minute or two for the extra security the TPM provides.

Random number generation was nearly identical between the two TPMs. For this test 32 bytes of random data were requested 15 times, then the results were averaged. Software running on the host system may use the TPM's generator as a source of good random numbers, so it is important that the TPM can provide them quickly.

The ZTPM outperforms the desktop TPM in the SHA-1 test with a speedup of 2x. The API for the hash algorithm is exposed primarily for low resource systems. An embedded control system or a mobile phone might see an even larger speedup due to a lower speed host CPU. On those kind of systems this speedup may be a larger advantage.

The ZTPM also does slightly better in the AES transport session tests. Since the version information only takes up a few bytes, another test was conducted using TPM_GetRandom as the wrapped command. For this test TPM_GetRandom was set to retrieve 128 bytes of random data, and this was repeated 10 times. Again, the ZTPM was slightly faster. This is likely due to the advantage the hardware accelerator provides, especially considering that the conventional TPM was faster in the TPM_GetRandom test. The desktop TPM did not support the OFB mode for encrypted transport sessions. For both tests CTR mode was used instead. The ZTPM shows its greater flexibility here, because it supports both OFB and CTR modes.

6.5 Analysis

When Eisenbarth et al. conceived of the idea of an FPGA-based TPM they were not able to implement it due to limitations in FPGA technology [13]. The results in the previous sections show that a TPM implementation on an FPGA is now feasible. The reconfigurable TPM in its present state should be a useful tool for developers, but it would need more work before it is ready for end-users. Since security is a race between adversaries and users trying to protect their data, the flexibility of the reconfigurable TPM may well outweigh the disadvantages. The remainder of this chapter will compare these new implementations against a conventional TPM. It will summarize the benefits of both designs, as well as their limitations.

6.5.1 Limitations

Bitstream Security

While the proposed system provides many advantages over a standard TPM, it is not without its drawbacks. There needs to be a way to ensure that a particular IP loaded onto the FPGA can be trusted. If an adversary can upload their own full, or partial, bitstream to the TPM, it may be possible to compromise user data. Other research has shown some methods of authenticating an FPGA bitstream that may be possible here. Eisenbarth et al. discuss ways to include the bitstream of the TPM in the chain of trust [13]. Since the TPM specification leaves the upgrade procedure largely up to the manufacturer, it may be possible to use the existing upgrade command, TPM_FieldUpgrade to verify that the bitstream comes from a trusted source. This command requires physical presence or owner authorization before it will upgrade the firmware. Just as with the conventional TPM, the ZTPM might make use of a key to encrypt the upgrade and prevent unauthorized changes. But to make this process even safer a 'secure boot' type scheme could be implemented where the TPM checks that each module has been signed with the manufacturer or user's key before proceeding. A scheme using PUFs, perhaps the scheme proposed by Johnson et al. or Schellekens et al., is another possibility [38, 14].

Cost

One of the main disadvantages associated with this approach, compared to a traditional TPM, is the cost. While a commercial TPM module like the one in Figure 1.1 costs just $15, the Digilent Zybo board used in this work costs around $100. This high price limits its utility in the broad market. While the entire board may not be necessary, the Zynq core itself still costs a little over $60 from a distributor like Digi-Key. Using the ZTPM will incur at least 4x the cost. In the future as prices drop, this may be less of an issue. Additionally, if Intel begins adding FPGAs to desktop computers [12], or if a company already uses an FPGA in their application or product, the only cost to add a ZTPM would be the cost to develop and test it.

Working Open Source Cores

These reconfigurable TPMs use open source components. This is a benefit, but finding a working modular exponentiation core was definitely problematic. Many of the cores that I found were not chosen, because they were built to work with smaller RSA keys (512 bit) that did not provide sufficient security. When a core that supported the proper length key was finally found, it did not produce correct results. To get around this, a software implementation was used, but it was slow. If a budget is available for building a ZTPM, commercial cores could be used, or perhaps a core could be developed in-house.

6.5.2 Benefits

Potential to Upgrade Existing Modules

The reconfigurable nature of the FPGA provides the user the opportunity to customize core components. A user could use the fast FPGA fabric to speed up cryptographic algorithms, or use software to save on FPGA space for other algorithms. The components can also be upgraded. With a traditional TPM implementation a side-channel attack might completely compromise it due to being unable to upgrade hardware accelerators. Once compromised, the purchase of a new TPM may be necessary. Since in this design it becomes possible to upgrade hardware accelerators as well as software, we now have the capability to defend against side-channel attacks, once they are discovered. For example, if a timing attack is discovered on a particular algorithm, we might load a new core that performs operations in constant time. Rather than having to purchase a new TPM, we can simply upload a new bitstream. While a traditional TPM could receive a software update, it would not be possible to upgrade any hardware accelerators it contained. This could prohibit fast encryption or decryption using the upgraded algorithms. Again, the open source nature would provide an upgrade path whereas a proprietary TPM may never provide updates.

Extensible

Even if users do not desire to upgrade old algorithms, the ZTPM gives them the ability to add new ones. Perhaps in the future, one of the existing algorithms will be compromised, and quantum computers can decipher its encrypted messages quickly. Instead of buying a new TPM, a state of the art algorithm can be added. New modules can be added for symmetric encryption, asymmetric encryption, and hashing, and this would prolong the useful life of the ZTPM. Additionally, even with all the TPM functionality in place there is plenty of room on the FPGA for more algorithms. The Zynq chip used in this work is a midrange chip with an Artix-7. Xilinx sells much larger Zynq chips, including one that has a Kintex-7 and 3.3MB of Block RAM [44]. This chip also contains over 15x the amount of logic cells. If the increased cost is not an issue, the capabilities of the reconfigurable TPM can vastly improve.

Transparent Design

Recall that manufacturers of conventional TPMs are not required to disclose their implementation details to users. Although many users are not concerned with these details, there are cases where it is very important to know about the inner workings of a system. The framework proposed in this thesis allows security specialists or developers to create their own custom TPM. Depending on their requirements, they may be able to use freely available open source FPGA cores, or they may want the speed provided by commercial FPGA cores. In the design advocated in this thesis, security experts can look at the code and verify that there are no backdoors. In the future, this framework and the concept of a reconfigurable TPM could even be used by TPM manufacturers to improve their own designs.

Portability

A limitation specific to the use of the development board is that it connects with the computer over USB. Although the current USB interface might be considered a limitation, there are circumstances in which this limitation becomes an advantage. It allows the TPM to be used with platforms that otherwise would not have TPM support (Macintosh computers for example). Additionally, since the platform can be easily moved, it becomes possible to share data between platforms and operating systems. If a person owns both a Macintosh and a Linux PC, they could use the reconfigurable TPM on both platforms. As Zhang et al. observe, even if a computer already has a TPM, since this device connects over USB, it can also be used as a backup TPM [37].

Chapter 7

Conclusion

Too many people have been affected by identity theft and compromised personal information. The TCG has created the TPM to help secure data and keep it safe, even in an untrusted environment. Unfortunately the algorithms used in existing TPMs will only become more susceptible to hacking as time goes on. Older algorithms previously used by the TPM have already been phased out. By creating a TPM that can adapt and gain new functionality over time, it becomes possible to extend the life of a TPM beyond that of the algorithms it started with. This will become especially important as research into quantum computing brings further challenges to existing cryptography.

7.1 Contributions

The work in this thesis concentrated on building and analyzing a proof of concept TPM on an FPGA. The main contributions of this thesis are summarized below:

- creating an open source FPGA-based TPM,
- creating ZTPM, an effective portable TPM, and
- creating a framework for adding additional functionality to a TPM.

The TPM is clearly useful for key storage, platform integrity and other applications, but it has limited capacity for upgrades. To overcome this hurdle it is clear that more flexibility is required. FPGAs can provide this flexibility. In this thesis, an FPGA-based reconfigurable TPM called the ZTPM was proposed. The ZTPM is similar to conventional TPM designs, it contains a microprocessor, RAM, and flash, but it in addition it contains an FPGA to allow for hardware acceleration. While commercial TPMs may use hardware accelerators, using an FPGA allows the ZTPM to upgrade those accelerators as vulnerabilities are found.

This thesis also proposes a framework that makes it easier for developers to upgrade existing functionality or add new functionality to the TPM. Our results showed that, while the ZTPM is more flexible, a conventional TPM retains a large cost advantage and some speed advantages. Recall that no optimizations were performed for these cores, but there is plenty of research on cryptography on FPGAs (see [39, 40, 41]) and this work could be used with the ZTPM. At that point it is likely to handily outperform a conventional TPM. Unfortunately at the present time the cost of these prototypes is much greater than the cost for a conventional TPM. The development board used in this work retails for about 10x the cost of a conventional TPM. As time goes on, this cost will decrease. As this happens, and FPGAs become more common, the approach taken in this thesis should become more feasible. If FPGAs begin to be included alongside processors as Intel has proposed [12], it could mean that the only cost for an FPGA-based TPM would be the development cost.

Although the cores used in this work were open source and readily available online (see Tables 5.3 and 5.2), there is no reason that commercial IP could not be used in its place. This thesis used open source cores to show that the proposed framework could be used to create a vendor-independent TPM, and also to save money. Since the work in this thesis uses open source cores, anyone should be able to duplicate this work, or conduct audits on the security of reconfigurable designs built using this framework. For those users who may require more speed or have other more stringent requirements, there are plenty of commercial cores available, and users could of course build their own. Using commercial cores may allow the speed of the ZTPM to more closely match the speed of a conventional TPM.

7.2 Future Work

Some of the limitations mentioned in the previous chapter, including use of USB, lack of RAM/flash, and slow speeds provide good opportunities for future research. One larger topic that could be explored is the impact of allowing dynamic partial reconfiguration for the different cores installed on a reconfigurable TPM. Allowing this could provide even more flexibility, but may require further countermeasures to prevent adversaries from installing malicious bitstreams onto a reconfigurable TPM. Schellekens et al. proposed using PUFs and version counters in nonvolatile storage to reduce this risk [14]. However, there are likely to be additional challenges besides bitstream security. For example, a method of dynamically loading and unloading cores will need to be added to the extensibility framework.

Related to this, one possible area for future research would be to build a system that has many different encryption hardware and software modules to choose from. As the workload changes, the TPM could allocate more, or fewer hardware resources. With these extra resources it might be possible to allow the host computer to use the FPGA for things unrelated to trusted computing. For example perhaps a machine with an FPGA may need to use a large network packet processing module in the FPGA. It could use partial reconfiguration to switch the TPM to a primarily software mode of operation, which would be slower, but take up less FPGA area. The network processing module could then be loaded into the FPGA. Or, in a scenario where the TPM was being used more frequently, the system could give it more resources and speed up the operations it performs.

Another issue that could be explored is the extent to which a change in the default encryption or hash algorithm changes other parts of the TPM. For example, changing the default hash algorithm from SHA-1 to SHA-256 would break backward compatibility with other TPMs, but would it have other effects on the same device? The size of a SHA-1 digest is used in multiple places in the specification to determine other data type sizes. RSA is another example; in fact, some features of the TPM may not work if RSA is ever completely removed. One workaround could be adding a TPM_SetCapability command to change the default algorithm only for some commands. By adding a command to select the replacement algorithm, we could make the new algorithm available for those commands, but leave it intact for compatibility.

Individual modules may have a significant impact on the security of the TPM. The security perimeter of the TPM must remain intact even in the presence of new functionality. Side-channel vulnerabilities should be carefully analyzed. For example, if new modules are added to the TPM, can we be sure that those modules are not leaking private or sensitive information to the outside? Similarly, what vulnerabilities impact conventional TPMs? Are there undiscovered side-channel vulnerabilities in existing TPM designs?

Bibliography

[1] Trusted Computing Group Incorporated, *TCG Specification Architecture Overview*, Aug. 2007, version 1.4. [Online]. Available: https://www.trustedcomputinggroup.org/wp-content/uploads/TCG_1_4_Architecture_Overview.pdf 1

[2] ——, *Trusted Computing Platform Alliance (TCPA) Main Specification Version 1.1b*, Feb. 2002. [Online]. Available: https://trustedcomputinggroup.org/tcpa-main-specification-version-1-1b/ 1

[3] ——, *TPM Main Specification 1.2*, Mar. 2011, revision 116. [Online]. Available: http://www.trustedcomputinggroup.org/tpm-main-specification/ 2, 6, 8, 10, 11, 15, 16, 18, 19, 22, 26, 27

[4] ——, *TPM Library Specification 2.0*, Oct. 2014, revision 01.16. [Online]. Available: http://www.trustedcomputinggroup.org/tpm-library-specification/ 2

[5] FxJ, "Trusted Platform Module on Asus motherboard P5Q PREMIUM," Online, Aug. 2009, image is in the Public Domain. [Online]. Available: https://commons.wikimedia.org/wiki/File:TPM_Asus.jpg 2

[6] S. L. Kinney, *Trusted Platform Module Basics: Using TPM in Embedded Systems*, ser. Embedded Technology Series. Elsevier Inc., 2006. 2, 4, 8, 15, 26

[7] Infineon, "Product brief TPM 1.2 SLB9635 TT 1.2 the Trusted Platform Module solution," Online, 2006. [Online]. Available: https://media.digikey.com/pdf/Data%20Sheets/Infineon%20PDFs/SLB%209635%20TT1.2.pdf 2, 4

[8] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Review*, vol. 41, no. 2, pp. 303–332, Jan. 1999. [Online]. Available: http://dx.doi.org/10.1137/S0036144598347011 3

[9] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler, "Breaking ciphers with COPACOBANA –a cost-optimized parallel code breaker," in *Lecture Notes in Computer Science*, L. Goubin and M. Matsui, Eds. Berlin, Heidelberg: Springer Nature, 2006, pp. 101–118. [Online]. Available: http://dx.doi.org/10.1007/11894063_9 3

[10] A. P. Fournaris and O. Koufopavlou, "Protecting CRT RSA against fault and power side channel attacks," in *2012 IEEE Computer Society Annual Symposium on VLSI*. Institute of Electrical and Electronics Engineers (IEEE), Aug. 2012. [Online]. Available: http://dx.doi.org/10.1109/ISVLSI.2012.54 3, 27

[11] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems," in *Advances in Cryptology — CRYPTO '96*. Springer Nature, 1996, pp. 104–113. [Online]. Available: http://dx.doi.org/10.1007/3-540-68697-5_9 3, 27

[12] N. Hemsoth, "Intel marrying FPGA, beefy broadwell for open compute future," *The Next Platform*, 2016. [Online]. Available: https://www.nextplatform.com/2016/03/14/intel-marrying-fpga-beefy-broadwell-open-compute-future/ 3, 49, 53

[13] T. Eisenbarth, T. Güneysu, C. Paar, A.-R. Sadeghi, D. Schellekens, and M. Wolf, "Reconfigurable trusted computing in hardware," in *Proceedings of the 2007 ACM workshop on Scalable Trusted Computing*, ser. STC '07. New York, NY, USA: Association for Computing Machinery (ACM), 2007, pp. 15–20. [Online]. Available: http://dx.doi.org/10.1145/1314354.1314360 4, 28, 48, 49

[14] D. Schellekens, P. Tuyls, and B. Preneel, "Embedded trusted computing with authenticated non-volatile memory," in *Trusted Computing - Challenges and Applications*. Springer Nature, 2008, pp. 60–74. [Online]. Available: https://doi.org/10.1007/978-3-540-68979-9_5 4, 29, 30, 49, 53

[15] W. Drewry and S. Gwalani, "Chromebook security: browsing more securely," July 2011. [Online]. Available: https://chrome.googleblog.com/2011/07/chromebook-security-browsing-more.html 6

[16] Microsoft Corporation, *Windows Certification Program*, Dec. 2014. [Online]. Available: https://msdn.microsoft.com/en-us/library/dn423132(v=vs.85).aspx 6

[17] Infineon, "TPM Trusted Platform Module SLB9645 TCG rev. 116 data sheet," Online, Feb. 2014. [Online]. Available: http://www.infineon.com/dgdl/Infineon-TPM+SLB+9645-DS-v02_14-EN.pdf?fileId=5546d4625185e0e201518b83d0c63d7c 7, 42

[18] Trusted Computing Group Incorporated, *Trusted Computing*, 2016. [Online]. Available: http://trustedcomputinggroup.org/trusted-computing/ 10

[19] Microsoft Corporation, *Measured Boot*. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/hh848050(v=vs.85).aspx 10

[20] ——, *BitLocker Drive Encryption Overview*, 2016. [Online]. Available: https://technet.microsoft.com/en-us/library/hh831507(v=ws.11).aspx 11, 14

[21] ——, *Overview of Windows AppLocker*, 2008. [Online]. Available: https://technet.microsoft.com/en-us/library/dd759113(v=ws.11).aspx 15

[22] S. Gold, "Backdoors to the future? [cyber security]," *Engineering & Technology*, vol. 9, no. 9, pp. 59–63, Oct. 2014. [Online]. Available: http://dx.doi.org/10.1049/et.2014.0921 16

[23] D. Eastlake and P. Jones, "US secure hash algorithm 1 (SHA1)," Internet Engineering Task Force, Tech. Rep. RFC 3174, Sept. 2001. [Online]. Available: http://dx.doi.org/10.17487/rfc3174 17, 36

[24] M. Stevens, P. Karpman, and T. Peyrin, “Freestart collision for full SHA-1,” in *Advances in Cryptology – EUROCRYPT 2016*. Springer Nature, 2016, pp. 459–483. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-49890-3_18 18

[25] H. Krawczyk, M. Bellare, and R. Canetti, “HMAC: Keyed-hashing for message authentication,” Internet Engineering Task Force, Tech. Rep. RFC 2104, Feb. 1997. [Online]. Available: http://dx.doi.org/10.17487/rfc2104 19

[26] RSA Laboratories, “PKCS #1 v2.2: RSA cryptography standard,” EMC Corporation, Tech. Rep. PKCS #1, Oct. 2012. 20, 36

[27] National Institute of Standards and Technology, “Announcing the Advanced Encryption Standard (AES),” National Institute of Standards and Technology, Tech. Rep. Federal Information Processing Standards 197, Nov. 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf 22, 36

[28] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guildord, E. Ozturk, Gilwolrich, and R. Zohar, “Breakthrough aes performance with intel aes new instructions,” Intel Corporation, Tech. Rep., 2010. [Online]. Available: https://software.intel.com/sites/default/files/m/d/4/1/d/8/10TB24_Breakthrough_AES_Performance_with_Intel_AES_New_Instructions.final.secure.pdf 23

[29] Helion Technology, “AES cores,” Mar. 2014. [Online]. Available: http://www.heliontech.com/aes.htm 23, 44

[30] M. Matsumoto and T. Nishimura, “Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, Jan. 1998. [Online]. Available: http://dx.doi.org/10.1145/272991.272995 26, 27

[31] “Hardware random number generator for FPGAs,” Source code, May 2015. [Online]. Available: https://github.com/teknohog/rautanoppa 27

[32] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors,” *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 361–372, June 2014. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665726 27

[33] G. Guette and C. Bryce, “Using TPMs to secure vehicular ad-hoc networks (VANETs),” in *Lecture Notes in Computer Science*, J. A. Onieva, D. Sauveron, S. Chaumette, D. Gollmann, and K. Markantonakis, Eds. Berlin, Heidelberg: Springer Nature, 2008, pp. 106–116. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-79966-5_8 28

[34] K. Dietrich and J. Winter, “Towards customizable, application specific mobile trusted modules,” in *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing*, ser. STC ’10. New York, NY, USA: ACM, 2010, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1867635.1867642 28

[35] B. Glas, A. Klimm, O. Sander, K. Müller-Glaser, and J. Becker, "A system architecture for reconfigurable trusted platforms," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 541–544. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403505 29

[36] H. Huang, C. Hu, and J. He, "A security embedded system base on TCM and FPGA," in *2009 2nd IEEE International Conference on Computer Science and Information Technology*. Institute of Electrical and Electronics Engineers (IEEE), Aug. 2009, pp. 605–609. 29

[37] D. Zhang, Z. Han, and G. Yan, "A portable TPM based on USB key," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 750–752. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866419 29, 35, 51

[38] A. P. Johnson, R. S. Chakraborty, and D. Mukhopadhyay, "A PUF-enabled secure architecture for FPGA-based IoT applications," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 1, no. 2, pp. 110–122, Apr. 2015. [Online]. Available: http://dx.doi.org/10.1109/TMSCS.2015.2494014 30, 49

[39] A. Rezai and P. Keshavarzi, "High-throughput modular multiplication and exponentiation algorithms using multibit-scan–multibit-shift technique," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 9, pp. 1710–1719, sep 2015. [Online]. Available: https://doi.org/10.1109/tvlsi.2014.2355854 30, 53

[40] K. Rahimunnisa, P. Karthigaikumar, S. Rasheed, J. Jayakumar, and S. SureshKumar, "FPGA implementation of AES algorithm for high throughput using folded parallel architecture," *Security and Communication Networks*, vol. 7, no. 11, pp. 2225–2236, oct 2012. [Online]. Available: https://doi.org/10.1002/sec.651 30, 53

[41] M. Rao, T. Newe, I. Grout, and A. Mathur, "High speed implementation of a SHA-3 core on virtex-5 and virtex-6 FPGAs," *Journal of Circuits, Systems and Computers*, vol. 25, no. 07, p. 1650069, jul 2016. [Online]. Available: https://doi.org/10.1142/s0218126616500699 30, 53

[42] S. Trimberger, "Trusted design in FPGAs," in *Proceedings of the 44th annual conference on Design automation - DAC '07*. Association for Computing Machinery (ACM), 2007. [Online]. Available: https://doi.org/10.1145/1278480.1278483 30

[43] J. Guajardo, S. S. Kumar, G. Schrijen, and P. Tuyls, "Physical unclonable functions and public-key crypto for FPGA IP protection," in *2007 International Conference on Field Programmable Logic and Applications*. Institute of Electrical and Electronics Engineers (IEEE), aug 2007. [Online]. Available: https://doi.org/10.1109/fpl.2007.4380646 30

[44] Xilinx Incorporated, *Zynq-7000 All Programmable SoC Overview*, Online, Sept. 2016, version 1.10. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf 34, 41, 50

[45] Digilent Incorporated, *ZYBO FPGA Board Reference Manual*, Apr. 2011. [Online]. Available: https://reference.digilentinc.com/_media/zybo:zybo_rm.pdf 34

[46] N. R. Wagner, "The laws of cryptography with java code," 2003. 35, 36

[47] K. Goldman, *Software TPM Introduction*, Source code, IBM Thomas J. Watson Research Center, 2010. [Online]. Available: http://ibmswtpm.sourceforge.net/ 35, 36, 38

[48] Free Software Foundation, "The GNU multiple precision arithmetic library," 2016. [Online]. Available: https://gmplib.org/ 36

[49] OpenSSL Software Foundation, "OpenSSL cryptography and SSL/TLS toolkit," 2016. [Online]. Available: https://www.openssl.org/ 36

[50] NORDUnet, "Modular exponentiation core," Source code, June 2016. [Online]. Available: https://trac.cryptech.is/browser/core/math/modexpa7 36

[51] H. Hsing, "Tiny AES 128," Source code, 2012. [Online]. Available: http://opencores.org/project,tiny_aes 36

[52] Hegde, "AES decryption IP," Source code, 2015. [Online]. Available: http://opencores.org/project,aes_decry_ip_128bit 36

[53] A. Forencich, "Verilog implementation of Mersenne Twister PRNG," Source code, Sept. 2016. [Online]. Available: https://github.com/alexforencich/verilog-mersenne/ 36

[54] J. Strombergson, "Verilog implementation of the SHA-1 cryptographic hash function," Source code, June 2016. [Online]. Available: https://github.com/secworks/sha1 36

[55] Helion Technology, "Modular exponentiation core family for FPGA," Ash House, Breckenwood Road, Fulbourn, Cambridge CB21 5DQ, England, Mar. 2014. [Online]. Available: http://www.heliontech.com/modexp.htm 45

Appendix A

Code Reference for Extensibility Module

The TPM implementation described in this thesis uses a shim layer, or set of wrapper functions, to abstract away the details of the cryptographic algorithm in use. Every algorithm that is to be used on the TPM must conform to the interface created by the structures described in this appendix. Instances of these structures are stored in an array in memory. When the TPM's user requests a cryptographic operation, the TPM searches the array for the appropriate algorithm. If the algorithm is found, the appropriate function is called. If the algorithm is not found, the TPM returns an error.

The following structures show the function pointers necessary to implement each type of algorithm. Structure tdAsymEncRegistration should be used for an implementation of an asymmetric encryption algorithm. Structure tdHashRegistration should be used for an implementation of a hash function. Structure tdSymEncRegistration should be used for an implementation of a symmetric encryption algorithm. Finally, all algorithms must implement tdComponentRegistration. This structure contains the algorithm ID, name, self test function, and pointer to the registration structure.

```
typedef struct tdAsymEncRegistration
{
  /// Returns the block size in bytes for this algorithm.
  uint32_t BlockSize;

  /// Decrypts encryptData of size encryptDataSize. Caller frees
     decryptData, size of the new buffer is decryptDataSize
  TPM_RESULT (* Decrypt)(unsigned char** decryptData, uint32_t
     *decryptDataSize, const unsigned char* encryptData, uint32_t
     encryptDataSize, void* key);

  /// Encrypts decryptData of size decryptDataSize. Caller frees
     encryptData, size of the new buffer is encryptDataSize
  TPM_RESULT (* Encrypt)(unsigned char** decryptData, uint32_t
     *decryptDataSize, const unsigned char* encryptData, uint32_t
     encryptDataSize, void* key);

  /// Generates a new keypair. Caller must free the tpm_pub buffer.
  /// The key cache in tpm_priv will be set and also must be freed.
  /// \param tpm_pub Generated public key.
  /// \param tpm_priv Generated private key.
  /// \param tpm_parms Specified parameters for key generation.
  TPM_RESULT (* GenerateKeyPair)(TPM_SIZED_BUFFER* tpm_pub,
     TPM_STORE_ASYMKEY** tpm_priv, TPM_KEY_PARMS* tpm_parms);

  /// Converts the TPM formatted key to a key format that the module
     can understand.
  /// Caller initializes the keyPtr to NULL and frees the keyPtr when
     finished.
  /// \param pubKey Contains the public key for the algorithm.
  /// \param privKey Contains the private key for the algorithm. (May
     be NULL if not needed)
  TPM_RESULT (* LoadKey)(TPM_SIZED_BUFFER* pubKey, TPM_STORE_ASYMKEY*
     privKey, TPM_KEY_PARMS* keyParms, void** keyPtr);

  /// Signs the message of length messageSize.
  /// Caller frees signature, size of the new buffer is in sigLength
  TPM_RESULT (* Sign)(unsigned char** signature, uint32_t
     *signature_length, const unsigned char* message, uint32_t
     message_size, void* key);

  /// Frees the key, and any resources allocated by the key.
  void (* UnloadKey)(void* key);

  /// Signature contains the supposed signature of the message.
  /// \return TPM_SUCCESS if the signature is valid.
  TPM_RESULT (* Verify)(unsigned char* signature, uint32_t
     signature_size, const unsigned char* message, uint32_t
     message_size, void* key);
} AsymEncRegistration;

typedef struct tdHashRegistration
{
  /// Hashes a group of buffers, including an optional buffer
     prepended to everything else.
  TPM_RESULT (* Hash)(TPM_DIGEST md, uint32_t length0, unsigned char
     *buffer0, va_list ap);

  /// Loads the Hash Context
  TPM_RESULT (* HashContext_Load)(void **context, unsigned char
     **stream, uint32_t *stream_size);

  /// Stores the Hash Context
  TPM_RESULT (* HashContext_Store)(TPM_STORE_BUFFER *sbuffer, void
     *context);

  /// Deletes the context allocated in HashInit
  void (* HashDelete)(void** context);
  /// Returns the computed hash. The context should not be reset or
     freed.
  /// \param context The hash context.
  /// \param digest The computed hash value.
  TPM_RESULT (* HashFinal)(void* context, uint8_t* digest);

  /// Initializes the hash function. Allocates and creates a new
     context.
  TPM_RESULT (* HashInit)(void** context);

  /// Returns the size of the output (in BYTES) of the hash function.
  uint32_t HashSize;

  /// Updates the hash function with new information. This should not
     reset the hash function.
  TPM_RESULT (* HashUpdate)(void* context, const uint8_t* data,
     uint32_t data_len);

} HashRegistration;

/// This structure should be used for an implementation of a HMAC
typedef struct tdHMACRegistration
{
  ///HMAC a variable argument list.
  TPM_RESULT (* HMAC)(TPM_HMAC tpm_hmac, const TPM_SECRET key, va_list
     ap);

  /// Returns the size of the output (in BYTES) of the hash function.
  uint32_t HMACSize;
} HMACRegistration;

typedef struct tdSymEncRegistration
{
  /// Returns the block size (in BYTES) for this algorithm.
  /// Future: if the algorithm has separate input and output sizes a
     parameter could be added.
  uint32_t BlockSize;

  /// Returns the size in bytes for the key of this algorithm.
  uint32_t KeySize;

  /// Decrypts the ciphertext and places it into the plaintext.
  /// Both arrays are assumed to be preallocated.
  /// They should both be BlockSize bytes.
  TPM_RESULT (* Decrypt)(unsigned char* ciphertext, unsigned char*
     cleartext);
  /// Encrypts the plaintext and places it into ciphertext.
  /// Both arrays are assumed to be preallocated.
  /// They should both be BlockSize bytes.
  TPM_RESULT (* Encrypt)(unsigned char* cleartext, unsigned char*
     ciphertext);

  /// Generates a new symmetric key.
  TPM_RESULT (* GenerateKey)(TPM_SYMMETRIC_KEY* key);

  /// Set the Key, which should be KeySize bytes.
  TPM_RESULT (* SetKey)(const TPM_SYMMETRIC_KEY* key);
} SymEncRegistration;

typedef struct tdComponentRegistration
{
  /// The Algorithm ID associated with this algorithm.
  /// If there isn't one for your algorithm create a new one.
  TPM_ALGORITHM_ID algorithmType;

  /// A short name to determine which modules are installed.
  /// I imagine this will mostly be used for debug.
  char ComponentName[32];

  /// Test the component for proper operation.
  /// \return TRUE if the component works, false if it does not.
  int (* SelfTest)();

  /// The enumeration corresponding to the type of module being added.
  RegistrationType cryptoType;

  /// A pointer to one of the registration structs.
  void* Registration;

  /// Checks that the key parameters are valid for creation of a new
     key.
  TPM_RESULT (* ValidateKeyParms)(TPM_KEY_PARMS *tpm_key_parms,
     TPM_KEY_USAGE tpm_key_usage, uint32_t keyLength, /* in bits */
     TPM_BOOL FIPS);

} ComponentRegistration;
```

Appendix B

List of Abbreviations

AES	Advanced Encryption Standard 3.5
BRAM	Block Random Access Memory
CBC	Cipher Block Chaining 3.6
CTR	Counter 3.6
DES	Data Encryption Standard
EK	Endorsement Key 2.5
FPGA	Field-Programmable Gate Array
GMP	GNU Multiple Precision Arithmetic Library
GNU	GNU is Not Unix
HMAC	Keyed-Hash Message Authentication Code 3.3
IoT	Internet of Things
IV	Initialization Vector
LPC	Low Pin Count
LUT	Look Up Table
OAEP	Optimal Asymmetric Encryption Padding 3.4
OFB	Output Feedback 3.6
PKCS	Public-Key Cryptography Standards 3.4
PRNG	Pseudo-Random Number Generator 3.7
PUF	Physically Uncloneable Function 4
RSA	Rivest Shamir Adleman 3.4
SCA	Side-Channel Attack 3.8
SHA	Secure Hash Algorithm 3.2
SRK	Storage Root Key 2.3
TCG	Trusted Computing Group 2
TPM	Trusted Platform Module 2
UART	Universal Asynchronous Receiver Transmitter