EXPLORING TRUSTED PLATFORM MODULE CAPABILITIES:
A THEORETICAL AND EXPERIMENTAL STUDY
Vandana Gunupudi, M.S.

Dissertation Prepared for the Degree of
DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS
May 2008

APPROVED:
Stephen R. Tate, Major Professor
Armin R. Mikler, Committee Member
Ram Dantu, Committee Member
Roy T. Jacob, Committee Member
Krishna Kavi, Chair of the Department of
Computer Science and Engineering
Oscar Garcia, Dean of the College of Engineering
Sandra L. Terrell, Dean of the Robert B. Toulouse
School of Graduate Studies
Trusted platform modules (TPMs) are hardware modules that are bound to a computer's motherboard, that are being included in many desktops and laptops. Augmenting computers with these hardware modules adds powerful functionality in distributed settings, allowing us to reason about the security of these systems in new ways.

In this dissertation, I study the functionality of TPMs from a theoretical as well as an experimental perspective. On the theoretical front, I leverage various features of TPMs to construct applications like random oracles that are impossible to implement in a standard model of computation. Apart from random oracles, I construct a new cryptographic primitive which is basically a non-interactive form of the standard cryptographic primitive of oblivious transfer. I apply this new primitive to secure mobile agent computations, where interaction between various entities is typically required to ensure security. I prove these constructions are secure using standard cryptographic techniques and assumptions.

To test the practicability of these constructions and their applications, I performed an experimental study, both on an actual TPM and a software TPM simulator which has been enhanced to make it reflect timings from a real TPM. This allowed me to benchmark the performance of the applications and test the feasibility of the proposed extensions to standard TPMs. My tests also show that these constructions are practical.
ACKNOWLEDGMENTS

This dissertation would not have been written without the support of a number of people. First and foremost, I would like to acknowledge my advisor, Steve, for his guidance and support throughout these years. From the time that I took his first course, he has been a continual source of inspiration. He has always encouraged me to develop independent research and analytical skills. I also wish to thank him for ensuring that I presented my research at various conferences. Surely, I would not have traveled to so many interesting places if not for his kindness!

I was lucky to have a good committee and I wish to thank them for their support and encouragement over the years. I am very grateful to my colleague, Roopa Vishwanathan, with whom I had many useful discussions. She also helped me by developing some of the code used in this dissertation.

My sister, Neeraja, has supported me in this endeavor in more ways than I can acknowledge. She has been there for me, more so than anyone else, counseling me patiently during all the tough times. I wish to tell her how much I admire her generosity of spirit. This dissertation would certainly not have been written without her continual support.

My parents have been instrumental in my embarking on this journey in the first place. I wish to thank my mother for her unwavering support all these years. She has always made time for me and her advice and positive attitude have helped me stay the course. My Dad has always given me practical advice that helped me put things in perspective. Dad, this dissertation is for you!

Finally, a very special thank you to all my friends, who ensured that I did not take myself too seriously and provided me with plenty of distractions to take my mind off of work!
## CONTENTS

ACKNOWLEDGMENTS iii

LIST OF FIGURES viii

CHAPTER 1. INTRODUCTION 1

1.1. High-level Overview 1

1.2. The Problems Considered in This Dissertation 2

1.2.1. The Random Oracle Model 2

1.2.2. Oblivious Transfer and its Application to Secure Mobile Agents 3

1.3. Tools Used to Solve These Problems 4

1.4. Solution Approaches 6

1.5. Overview of Results 6

1.6. Organization of This Dissertation 8

CHAPTER 2. TRUSTED PLATFORMS 9

2.1. Introduction 9

2.2. Background on Trusted Platforms 10

2.3. Basic Components of a TPM 11

2.3.1. Cryptographic Functionality 13

2.4. Keys in a TPM 14

2.4.1. Special Types of Keys 15

2.4.2. Protected Storage 16

2.5. Various Secure Hardware and Trusted Computing Initiatives 16

2.5.1. Early Work 17

2.5.2. Microsoft’s NGSCB Architecture 18
2.5.3. LaGrande Architecture 20
2.5.4. Web Applications with Lots of Privacy and Security (WebALPS) 20
2.5.5. XOM Architecture 21
2.6. Theoretical Modeling of Hardware-assisted Security 22
2.7. Other TPM Capabilities 23
2.7.1. Roots of Trust 23
2.7.2. Integrity Measurement and Reporting 25
2.7.3. Miscellaneous TPM Features 25
2.8. Migration Functionality 27

CHAPTER 3. THEORETICAL MODELS 29
3.1. Definitions 30
3.2. Background on Random Oracles 34
3.2.1. Properties Required for a Random Oracle 35
3.2.2. Impossibility Results 36
3.3. Non-Interactive Oblivious Transfer and Secure Function Evaluation 37

CHAPTER 4. INSTANTIATING A RANDOM ORACLE USING A TPM 39
4.1. Introduction 39
4.2. Use of TPM Functionalities and Extensions 40
4.2.1. The Role of Trusted Third Parties 41
4.3. Implement a Random Oracle with a TPM 42
4.3.1. Establish a Shared Secret with CMKs 43
4.3.2. A TPM-Oracle 45
4.3.3. Security of the TPM-Oracle 46

CHAPTER 5. GENERALIZED NON-INTERACTIVE OBLIVIOUS TRANSFER 54
5.1. Virtual Monotonic Counters 54
5.1.1. System Model 55
5.2. Count-Limited Objects 56
7.6. Extensions to SAgent 96
7.7. Sample Applications 97
  7.7.1. Maxbid Application 98
  7.7.2. BuyAgent Application 98
  7.7.3. Efficiency Analysis of the GTX Protocol 99
  7.7.4. Parameters 99
  7.7.5. Variation of Times with Input Size 102

CHAPTER 8. CONCLUSION 106
  8.1. Conclusion 106
  8.2. Future Work 107
    8.2.1. Random Oracles 108
    8.2.2. TPM Simulator 108

BIBLIOGRAPHY 110
LIST OF FIGURES

2.1 Components of a TPM 12
2.2 Protected Storage Hierarchy 16
2.3 Transitive Root of Trust 24

6.1 SFE Applied to Agents 68
6.2 Core Interfaces of SAgent 74
6.3 Component Interactions in SAgent 76
6.4 Representation of Data in SAgent 78

7.1 TPM Command Execution 83
7.2 Times for CMK Operations on the Source TPM 90
7.3 Times for CMK Operations on the Destination TPM 92
7.4 Times for CMK Source Operations on the Simulator 94
7.5 Times for Destination Operations on the Simulator 94
7.6 HMAC Function 95
7.7 Comparison of Total Times for Security Protocols 101
7.8 Comparison of Circuit Creation Times 102
7.9 Comparison of Per Host Computation Times for All Protocols 103
7.10 Variation of Per Host Computation times for the GTX Protocol 103
7.11 GTX Protocol: Maxbid and Buyagent Applications 104
7.12 Comparison of Maxbid and Buyagent Applications 105
CHAPTER 1

INTRODUCTION

1.1. High-level Overview

With distributed computing becoming ubiquitous, there is a need to address the issue of security of the associated data and transactions. The increasing prevalence of e-services and the distributed nature of computer networks lead to various security concerns and alleviating these concerns is important. Typically, security is based on trust and trust is placed on the computing platform like a PC, business server or PDA to protect sensitive data and computations. The organization owning the computing platform is typically trusted by the end user and this trust is based usually on software-based checks of various kinds. However, the ability of a computing platform to protect sensitive information through software alone is limited at best. Software cannot vouch for its integrity and even if software modifications can be detected locally, there is no mechanism for a remote user to be assured that the software is trustworthy. Therefore, trust cannot be based in software alone; trusted hardware must be an integral part of the process of establishing trust in software.

To address some of these issues, an industry consortium of over 160 companies including Microsoft, HP, AMD, Infineon and Intel, called the Trusted Computing Group (TCG) [76] [77] was formed. The TCG is responsible for creating guidelines and specifications for adding trusted hardware to various computing platforms. The goal of trusted computing is to enhance trust in end-user systems via the use of trusted hardware and supporting software. This additional hardware functionality is called a trusted platform module (TPM), and these devices can be used to answer questions like: How can an end-user be sure that the private key used to sign a sensitive document was not compromised? Did the right entity authorize all the signatures created with the private key? How was the private key generated? Is there any verifiable proof that the key was randomly generated?
In this dissertation, I explore how various features of trusted platforms can be leveraged to realize useful cryptographic primitives that are outside the scope of the TCG’s design goals, like shared secret establishment and a non-interactive form of oblivious transfer. These primitives, in turn, can be used to realize useful applications like random oracles and non-interactive secure mobile agent protocols, which are impossible to implement with standard systems. In addition to a formal security analysis of these primitives and their applications, I examine the practicality of these constructions by performing an experimental analysis.

1.2. The Problems Considered in This Dissertation

We consider two main fundamental problems in this dissertation: instantiating a random oracle and performing non-interactive oblivious transfer in a secure manner. As a first step, I define basic cryptographic primitives to support these operations, and then show how the primitives can be applied to realize useful applications. In the following section, I describe these applications and outline why these problems were chosen.

1.2.1. The Random Oracle Model

The random oracle model is an idealized theoretical model that has been successfully used for designing many practical cryptographic algorithms and protocols such as the Probabilistic signature scheme (PSS) [10] and Optimal asymmetric encryption padding (OAEP) [66]. In this model, all parties, including the adversary, have oracle access to a truly random function. An instantiation of the algorithm in the standard model (or “real world”) is obtained by replacing the random oracle with a suitable function like a “good” cryptographic hash function.¹ Since the design goals for cryptographic hash functions result in functions that “look random”, this is intuitively reasonable and a proof of security in the random oracle model is then taken as evidence that a scheme is secure in the standard model if the hash function is secure. However, this reasoning has been found to be flawed, as will become evident in subsequent sections. As an example of how random oracles are used, here

¹Various real-world instantiations of random oracles have been proposed which are discussed in later sections.
is an efficient public key encryption scheme described by Bellare [10], which uses a random function, $H$.

Let $k$ be the security parameter and let $G : \{0,1\}^* \rightarrow \{0,1\}^\infty$ be a random generator. $H$ is a random hash function, specified as $H : \{0,1\}^* \rightarrow \{0,1\}^k$. Let $f$ be a trapdoor permutation and $f^{-1}$ is then its inverse. $G(r) \oplus x$ denotes the bit-wise XOR of $x$ with the first $|x|$ bits of $G(r)$ and $\|$ denotes concatenation. The message $x$ is then encrypted as follows:

$$E_{G,H}^G(x) = f(r)||G(r) \oplus x||H(rx)$$

for a random value $r$ from the domain of $f$. Typically, $f$ is the recipient’s public key and $f^{-1}$ is the secret key.

A promising candidate for a real-world instantiation of a random oracle is a pseudorandom function (PRF) which produces output indistinguishable from random to a polynomial-time algorithm. In order to compute with the PRF, knowledge of the seed used to key the PRF is required. However, to be able to use the PRF as a secure instantiation of a random oracle, the seed must be secret, but all parties should still be able to compute the PRF. In the standard model, if the seed is kept secret from any party (including the adversary), the PRF cannot be computed. Therefore, the fundamental problem with instantiating a random oracle using a PRF is that, in order for it to be secure, the seed used to key the PRF must be kept secret, which is impossible in the traditional model of computation.

1.2.2. Oblivious Transfer and its Application to Secure Mobile Agents

Oblivious transfer (OT) was introduced by Rabin [62] as a fundamental cryptographic primitive, and subsequently many variants have been studied and used in a variety of cryptographic protocols such as secure multi-party computation. In a 1-out-of-2 OT protocol, Alice (the sender) has 2 values $s_0$ and $s_1$, and Bob (the receiver) has a selection bit $c$. At the end of the protocol, Bob learns the value $s_c$ while obtaining no information about $s_{1-c}$, and Alice cannot determine which value Bob received. An OT protocol is called interactive if
Bob must communicate with Alice or some other party after selecting $c$, and non-interactive otherwise.

In the standard model of computation, non-interactive OT is clearly impossible: Bob can take a “snapshot” of his state immediately before picking a value of $c$, and then run his computation with $c = 0$ to learn $s_0$. Since this computation was non-interactive, no state external to Bob is affected, so Bob can roll back his state to the snapshot and re-run his computation with $c = 1$, thus learning $s_1$ as well.

Non-interactive oblivious transfer has applications in the construction of secure mobile agents. In the mobile agent paradigm, an agent owner, also called the originator, creates software agents that can perform tasks on her behalf. After creating the agents for some specific purpose, the originator sends them out to visit various remote hosts, where the agents perform computations on behalf of the originator. When the agents return home, the originator retrieves the results of these computations from the agents. The utility of this paradigm is based on the ability of the originator to go offline after sending the agents out, and, ideally, no further interaction between the agent and the originator or the host should be required.

Oblivious transfer is one of the cryptographic primitives used to provide security in agent computations. In particular, it is used to ensure that the host does not learn any “extra” information about the agent’s state other than what it is entitled to learn. The traditional implementation of OT requires the host to interact with the originator but this requires the originator to remain online, which violates the basic tenet of the agent paradigm. Typically, other entities are used as “stand-ins” for the originator. For example, trusted third parties or threshold cryptography techniques are used in place of the originator. However, the host must still interact with other entities to ensure that security is preserved.

1.3. Tools Used to Solve These Problems

As described in the previous section, these problems have no solution in a traditional model of computation. However, if we consider an enhanced model incorporating hardware, which performs some specific functions, I will show in this dissertation that these applications
become possible and even practical. This augmented model of computation involves the use of hardware modules that can be used to enhance security in end-user systems. Specifically, I show how to realize these applications using a combination of existing technology and simple modifications to existing standard hardware proposed by the TCG. Trusted platform modules are already appearing in desktop PCs and laptops, and the goal of TCG is to make this technology ubiquitous. In light of this, the contributions of this dissertation are timely and will enhance understanding of the security properties of trusted platform modules.

Trusted platform modules are tamper-resistant devices that have a processor (not programmable by the user), limited computational power and non-volatile storage with secure I/O interfaces and hardware accelerators for cryptographic operations. A TPM has various capabilities; and the ones I primarily use in this dissertation are protected storage, which refers to the ability of the TPM to store secrets on the chip in a form usable only by the TPM, and monotonic counters, that have the property that a counter value, once incremented, cannot be reverted to any previous value. Monotonic counters in the TPM are not designed to be used directly by user-level applications but do not require a trusted OS to be useful. A trusted OS would support other TPM features such as taking software measurements and maintaining these measurements in a meaningful way that can be vouched for and attested to for remote systems (described in detail in Chapter 2). However, implementing a trusted OS is a huge challenge and this is one of the reasons that Microsoft has dropped support for some of the TCG-specified features in their TPM implementation, NGSCB [56] in Windows Vista. Our constructions, however, do not require a trusted OS and utilize functionality that is commonly available in trusted platform modules and other common hardware architectures (see details in Section 2.5).

It should be noted that trusted platform modules do not add anything to stand-alone systems; however, in case of distributed applications, the ability of a TPM to keep secrets from its user has powerful implications and allows us to look at the security of distributed applications in a new way.
1.4. Solution Approaches

In this dissertation, I tackle these problems from a theoretical perspective and then examine their feasibility for practical applications. The theoretical standpoint involves the formal definition of these cryptographic primitives and their applications. After defining these primitives in a clear way, I formally analyze the security of these primitives and their applications.

Once a formal definition is in place, I study the practicality of these applications. Therefore, the second part of this study involves looking at the real-world applicability of these primitives by analyzing their efficiency. To that end, I benchmark the basic operations and measure the efficiency of these protocols. For constructions that use existing TPM functionality like certified migration, I measure the efficiency of those operations on actual TPMs. Since I also propose extensions to existing TPM functionality, I use a software-based TPM simulator [74] and modify it by adding new extensions to it. I enhance the simulator by developing timing-accurate “performance profiles” based on measurements on an actual TPM. This allows us to benchmark the proposed operations (for example, adding an HMAC interface to the TPM) and estimate the performance of the protocols. Because the performance profiles are derived from measurement of primitive operations on an actual TPM, this allows us to precisely estimate the cost of these proposed TPM extensions if they are implemented in future TPM chips.

1.5. Overview of Results

In this dissertation, I outline how to leverage specific functionality provided by TPMs to instantiate useful cryptographic primitives of “shared secret establishment” and “generalized oblivious transfer.” These primitives, in turn, are then used to instantiate useful applications like random oracles and secure mobile agents. One of the features provided by TPMs is certified migration. This process allows for one TPM to migrate an internal key to another TPM in such a way that this key is shared between the TPMs and yet is never available in a usable form outside the TPMs. The primary use for this kind of key is for one system to
be able to transfer keys and a protected storage hierarchy from one system to another, but in my work on TPM-instantiated random oracles I found it more useful to treat this as the more fundamental cryptographic operation of shared secret establishment between TPMs. Establishing shared secrets is a first step in many cryptographic protocols and transactions, and was one of the first basic cryptographic primitives considered in the introduction of public key cryptography by Diffie and Hellman [28]. Using this primitive and the standard HMAC function, I show how to instantiate a random oracle.

Another useful functionality of TPMs that I leverage is a monotonic counter. Recent work at MIT by Sarmenta et al. [64, 52] has introduced the idea of a virtual monotonic counter which can be used as a building block for various applications like digital cash, e-wallets, virtual trusted storage and digital rights management (DRM). A virtual monotonic counter is a trusted counter than can be incremented but not reset back to any previous value. This security property is enforced by the TPM alone and does not require a trusted OS for this purpose. Among other interesting applications, virtual monotonic counters allow us to realize count-limited objects or “clob”s which are tied to a particular virtual monotonic counter. These can be n-time use encryption or signature keys, for example. The use of this key is tied to the counter which enforces that the key is not used more than n times. I use count limited objects to instantiate another useful primitive, a non-interactive form of oblivious transfer. I formally define a new cryptographic primitive called “Generalized non-interactive oblivious transfer” (GNIOT), which is impossible to implement in standard computation models, but is possible in an augmented hardware model. I then apply the GNIOT primitive to secure mobile agent computation. I develop a provably secure mobile agent protocol, which I call the GTX protocol, which improves on previous secure agent protocols like the TX [75] and the ACCK [2] protocols by replacing the oblivious transfer step in these protocols with the non-interactive OT primitive.

In summary, contributions of this dissertation include

- construction of a shared secret establishment primitive and a TPM oracle using the migration functionality of the TPM;
• definition of a new cryptographic primitive called “Generalized non-interactive oblivious transfer” (GNIOT) which is a construction of a non-interactive form of oblivious transfer using count-limited objects based on the monotonic counter in the TPM;
• use of the non-interactive oblivious transfer primitive in secure agent protocols;
• careful security analysis and rigorous proofs of these primitives and their applications;
• experimental study and performance analysis of these primitives and the proposed applications.

1.6. Organization of This Dissertation

The rest of this dissertation is organized as follows: In Chapter 2, I present an overview of trusted computing concepts and features of trusted platform modules. Chapter 3 provides background information on various theoretical concepts utilized in this dissertation like random oracles, oblivious transfer, secure function evaluation and count-limited objects. The construction of TPM oracles, one of the contributions of this dissertation, can be found in Chapter 4. Chapter 5 discusses my next contribution, a new cryptographic primitive called generalized non-interactive OT. I apply this new primitive to outline a new secure agent protocol (GTX protocol), a new security provider for my SAgent secure agent framework. Details of this protocol can be found in Chapter 6. I present an experimental analysis of these constructions and the agent protocol in Chapter 7. Chapter 8 wraps up the dissertation with a summary of the contributions of this dissertation and discussion of future work.
2.1. Introduction

In the recent past, various initiatives have been announced to enhance security in computers via the use of special-purpose hardware. Most notable of these is an initiative by the Trusted Computing Group, which has created specifications and guidelines for developing hardware modules that can be used to augment security in end-user systems, and TCG-compliant chips are beginning to be included in many laptop and desktop systems. The idea of using secure hardware to enhance security in computers has been around for a long time and definitions of what constitutes secure hardware abound. These devices can range from simple devices with limited computational power, like cryptographic smart-cards, to sophisticated high-end devices like secure co-processors that come with powerful processors and provide a high-degree of tamper resistance [41, 70, 71, 80]. The trusted platform modules proposed by the TCG fall somewhere in between these two categories. These are tamper-resistant devices with limited computational power that have a processor (not programmable by the end-user), a limited amount of non-volatile storage, hardware accelerators for performing cryptographic operations, and secure I/O interfaces. In a typical personal computer or larger server, a trusted platform module is a secure microcontroller that is securely bound to the motherboard and connected to the system using a low pin count (LPC) bus. The ability of a modern computing platform (like a PC or a laptop) to protect sensitive information through software alone is limited at best. Software cannot vouch for its integrity and even if software modifications can be detected locally, there is no mechanism for a remote user to be assured that the software is trustworthy. Trusted platform modules were developed with this simple threat model in mind and enhance security in modern computing platforms via the use of trusted hardware.
Trusted platform modules have various features, as detailed in the TCG specifications [77, 78, 79] and some of these features like trusted boot and remote attestation [12] have already been studied. In this dissertation, I primarily utilize the sealed storage functionality which involves the cryptographic protection of keys and data and the monotonic counter functionality of TPMs. We present the salient features of TPMs in the following sections for background information.

2.2. Background on Trusted Platforms

Trust, in general, is defined as the expectation that a system will operate as it is expected to operate. In the context of trusted platforms, this is a guarantee by a known, trusted authority (usually the manufacturer of the platform) that a platform with a particular identity can be trusted to operate as expected. Trusted computing technology allows for execution of arbitrary software on the platform, while protecting information on the platform. Although executing programs have access to information protected by the trusted module, they cannot interpret the data since the data protected by the trusted module is encrypted. Pearson [61] defines a trusted platform as “a computing system that has a trusted component, in the form of built-in hardware which it uses to create a foundation of trust for software process.” The basic goal of trusted computing, as defined by the TCG, is to ensure trust in software processes executing on a computer, with the trust being rooted in hardware. Fundamentally, there is no way to distinguish arbitrary software from trusted software, other than to measure the executing software. Typically, this involves creating a hash of the executable and associating a secret with the measured hash. Trusted platforms provide a way to measure and store the resulting digest in a tamper-resistant environment. When queried by a remote party, a trusted platform has a way to provide proof of its capabilities as well as provide the measured digests to the remote party in an assured way.

The TPM architecture includes non-volatile storage for storing cryptographic keys, a random microcontroller-number generator, a cryptographic engine that can perform cryptographic operations like RSA and SHA-1 and an I/O component that manages communication
over the I/O bus. The added capabilities proposed for systems incorporating a trusted platform module include protected storage, cryptographic processing, integrity measurement and reporting, sealed storage, and remote attestation. As specified by the TCG, a trusted platform must have certain components and provide certain functionality in compliance with their specifications.

It is important to note that the TCG specifications are not constant or “set in stone”; they are evolving based on feedback from users with new functionality being added. To date, there have been two main versions, with various revisions of specifications with minor changes in each revision. Version 1.2 of the TCG specifications improved upon the first version 1.1 (also called 1.1b in various sources). The main changes between the two versions include support for certifiable migratable keys (CMKs) described in section 2.4, changes to the delegation model, improvement to security in the way data is sent to the TPM, specification of non-volatile (NV) storage and changes to address privacy concerns. We highlight the relevant changes in the appropriate sections.

2.3. Basic Components of a TPM

In this section, I describe the core functionality of trusted platform modules. A trusted platform must have special locations where sensitive data can be operated on. It should not be possible for user programs to access these locations. These are called shielded locations and can be accessed only with a certain set of commands called protected capabilities. The TPM implements protected capabilities for protecting shielded locations and for reporting integrity measurements. This can include additional functionality like random number generation, generation of cryptographic keys and sealing data to system state. Basic components, as illustrated in Figure 2.1, include the following:

- Secure I/O: This component manages information flow across the communication bus. It performs encoding and decoding of information passed over internal and external buses. It routes messages to the appropriate TPM component.
Figure 2.1. Components of a TPM

- Key generation: This function is a protected capability and manages the generation of keys and nonces.
- Cryptographic engine: This component includes the SHA-1 engine, the RSA engine and the HMAC engine.
- Opt-in: This component allows the TPM to be disabled if necessary. The TPM is disabled by default and the owner must enable it via the Opt-in.
- Random number generator (RNG): This is the source of entropy in the TPM.
- Platform configuration registers (PCRs): These are 20-byte values that are SHA-1 digests. The integrity metrics, obtained from measuring state, are stored in PCRs. Version 1.2 of the specification requires 24 PCRs.
- Monotonic counter: This hardware counter provides an ever-increasing value. Its value, once incremented, cannot be reverted to a previous value. The TPM must support at least 4 concurrent counters, although only one is active in any boot cycle.
- Non-Volatile storage: This holds persistent state information and identity information.
- Execution engine: This component runs the program code to execute TPM commands and ensures that operations are segregated and shielded locations are protected.
2.3.1. Cryptographic Functionality

TPMs provide cryptographic engines for several standard cryptographic operations, including the following:

2.3.1.1. SHA-1 Engine

The SHA-1 capability is a trusted implementation of the SHA-1 algorithm and is a capability primarily used by the TPM. The specification requires that the SHA-1 hash algorithm is implemented as defined by FIPS-180-1 [59]. The TPM does expose an external interface which is used for measurements during the boot process. Since the TPM is not a cryptographic accelerator, there are no specified minimum throughput requirements for this engine.

2.3.1.2. HMAC Engine

The HMAC algorithm must be implemented as specified in RFC 2104 [49]. The HMAC engine is used to provide proof of knowledge of the associated authorization data for an object as well as proof that a command is authorized and has not been modified in transit. The TCG specification specifies the use of SHA-1 in HMAC, resulting in a key length of 20 bytes and a block size of 64 bytes.

2.3.1.3. RSA Engine

The RSA algorithm is used for digital signatures and encryption. The TPM must support key sizes of 512, 768, 1024, and 2048 bits, though the TPM may support other key sizes as well. The minimum recommended key size is 2048 bits. The RSA public exponent must be $e = 2^{16} + 1$.

2.3.1.4. Random Number Generator (RNG)

The RNG component is the source of entropy in the TPM. The TPM uses random values provided by the RNG for nonces, key generation, and randomness in signatures. The RNG consists of a state-machine that accepts and mixes unpredictable data and a post-processor that uses a one-way function (e.g., SHA-1).
2.4. Keys in a TPM

The TPM labels keys with attributes that indicate allowed uses for keys and characteristics regarding the level of protection afforded the key. Storage keys are used only for encrypting/decrypting keys in the storage hierarchy, signing keys are used for signatures, binding keys are used to encrypt data created outside a TPM but decrypted only in the given TPM and legacy keys are used for both signatures and encryption (though their use is now discouraged).

The protection attribute indicates the conditions under which the private part of the key leaves the TPM: **Non-migratable** keys are generated within a TPM, and the private key can only leave the TPM in ciphertext form, encrypted using another non-migratable key in the storage hierarchy. Because of this protection, the private part of a non-migratable key is only usable within the same TPM that generated it.

**Migratable** keys can, with the authorization of both the TPM owner and the key owner, be migrated off a TPM — unfortunately, there is no way to guarantee the provenance of such a key, as it can be migrated to the control of a non-TPM environment just as easily as to another TPM-protected environment.

Version 1.2 of the TCG specification introduced a new protection level for keys protected by a TPM: **Certifiable migratable keys (CMKs)**. A CMK is a migratable key in which the migration is restricted — when the key is generated (inside a TPM), a list of public keys of “migration authorities” is committed to, and any migration of this CMK must be done by encrypting the private portion of the key using one of these previously-specified public keys. Since the authorized public keys are specified when the CMK is created, the CMK is bound to this set of migration authorities for the lifetime of the key, and the TPM can certify (sign) the key along with the migration authority list; therefore, any receiver of this certificate obtains assurance that the key exists only in the original TPM or in places authorized by one of this fixed set of migration authorities. The intended use of CMKs in the TCG specification is that a public, trusted migration authority would publish a migration practice statement
(similar to a certificate practice statement or CPS for an X.509 Certification Authority), so can provide assurance that CMKs are only migrated to TPM-protected environments.

2.4.1. Special Types of Keys

In addition to protection level, keys are marked with their intended use, and three particular kinds of keys associated with a TPM are especially important and so are defined below:

2.4.1.1. Endorsement Key (EK)

The EK is a public/private key pair, generated usually by the manufacturer or vendor of the TPM as part of the manufacturing process. The EK can attest to the authenticity of values produced by the TPM. The entity that creates the EK must provide a credential (certificate) certifying that this key pair is valid and was generated in the specified manner. Typically, this entity is the manufacturer of the TPM. This key is unique to each TPM. Therefore, there are privacy concerns associated with this key since it uniquely identifies the TPM and the platform. The EK is non-migratable and can only be used in carefully controlled ways, such as creating identity keys (see below).

2.4.1.2. Attestation Identity Key (AIK)

AIKs are non-migratable keypairs that are essentially aliases to the EK. Due to privacy concerns, the EK is not used to sign any information originating from a TPM. As a non-migratable key, the private portion of an AIK never leaves the TPM in plaintext and is used only for signing data originated by the TPM, such as certifications of other non-migratable keys or CMKs and measurement data during attestation operations. The EK is statistically unique, whereas multiple identity keys (acting as pseudonyms for the EK) can be generated by the owner of the TPM thus mitigating privacy concerns. The owner controls the generation and activation of the AIKs, unlike the EK which cannot be generated by the owner. The public part of an AIK is part of a certified AIK credential, which will be further described in Section 2.7.3.
2.4.1.3. **Storage Root Key (SRK)**

The SRK is a keypair that is generated internally to the TPM and has a private key which never leaves the TPM. The SRK is always a non-migratable key. This key is at the root of the secure storage hierarchy in the TPM as described below.

2.4.2. Protected Storage

This functionality refers to the ability of a trusted platform module to store secrets securely on the chip. The TPM has a limited amount of storage that allows it to store keys and other data that needs to be protected, and can extend this amount of storage by exporting keys in an encrypted form that can only be decrypted when loaded back into the TPM. Every key created in the TPM is then encrypted using either the SRK or some other storage key. The encrypted key is then stored on the untrusted host and is decrypted inside the TPM when it needs to be used. A tree of such keys, shown in Figure 2.2 can be maintained in such a way that each key is protected by its parent (the key under which it is encrypted). At the root of the tree is the SRK and the private key of the SRK never leaves the TPM.

![Figure 2.2. Protected Storage Hierarchy](image)

2.5. Various Secure Hardware and Trusted Computing Initiatives

With this background information, I now outline other prominent hardware-assisted security initiatives and describe how this work fits into the context of these architectures.
The architectures I discuss are either extensions of the TCG specifications (NGSCB and LaGrande) and include some variants of TPMs or are entirely different from the TCG specifications. The most famous of these initiatives is probably the Microsoft initiative called Next Generation Secure Computing Base (NGSCB) [56] (formerly known as Palladium). Other technologies include Intel’s LaGrande [43] initiative, and the XOM architecture [50] developed at Stanford University. This section begins with a description of the IBM 4758 co-processor [72] and the Dyad project [81] at Carnegie Mellon University, both of which were among the first prominent initiatives in trusted computing. It should be noted that the hardware functionality used by the constructions in this dissertation, namely, protected storage, migration and hardware monotonic counters, is available in most of the architectures described in this section. In particular, these constructions can be implemented in Microsoft’s NGSCB architecture, Intel’s LaGrande architecture and on any co-processor. Our constructions cannot be directly implemented in the XOM architecture, since it is significantly different from other architectures, although it is possible that similar techniques could be developed for XOM.

2.5.1. Early Work

Secure co-processors emerged from the idea of protecting software execution. This model was later generalized to encrypted storage via a tamper-resistant module. The Abyss project [82], developed in 1990, included architecture that allowed secure software that was protected in a hardware module to interact with software running on an unprotected host. This architecture was generalized into the Citadel project [73], which was used to develop several practical prototypes. Instead of simply protecting secrets in hardware, this architecture provided various security services to the host. Citadel was integrated into the Dyad project [81, 86] which was used for securing e-commerce applications. The IBM 4758 [31] secure co-processor (circa 2001) was the first architecture to use cryptographic accelerators with applications in the financial industry. The IBM 4758 architecture and the Dyad project formed the basis for the TCG’s specifications as well as for other prominent initiatives.
The Dyad co-processor was a high-end device, packaged with sensors that detect tampering, so that any unauthorized attempt to read the internal state of the co-processor resulted in a resetting of that state. This co-processor had its own key, which was certified by the manufacturer. A key feature of this co-processor was secure boot, with some hardware being a part of the bootstrap process. The secure co-processor interacted with the host operating system to provide various services which included integrity verification via checksums, cryptographic protection of data and secure communication channels.

This architecture used the Citadel co-processor with an Intel i386SX CPU running at 16MHz. This system used an IBM DES chip for performing cryptographic operations. The coprocessor had a modified Mach 3.0 kernel and used “cryptopaging”, which provided virtual memory for the internal applications of the co-processor. In 1994, Tygar and Yee also developed a series of applications [87] for the co-processor including secure audit logs, postal meters and other secure e-commerce applications.

The IBM 4758 co-processor was designed with some novel goals in mind. It had the ability to authenticate itself to remote parties, a feature called outbound authentication. It was designed to be configured at the user site, with the ability to download software updates securely. Notable features of this co-processor include secure boot, with each layer validating the next layer; persistent storage where keys and data can be stored; and hardware tamper-response which involves zeroing out of the state in response to any tampering. It also had a cryptographic accelerator that included a DES engine, a hash engine and a hardware RNG.

2.5.2. Microsoft’s NGSCB Architecture

The NGSCB architecture involves enhancements to the CPU, chipset, USB I/O and GPU hardware components [57]. The nexus and the Nexus Computing Agents (NCAs) are the primary components of the NGSCB architecture along with the Security Support Component (SSC), which is a version 1.2 trusted platform module. The nexus is a security kernel that establishes a protected OS environment by isolating certain memory locations (also called “curtained memory”). The nexus, which is authenticated during the secure boot process, manages the core NGSCB services. The NCAs are applications or services that run in the
curtained memory area, isolated from applications that run in regular RAM. Access to the protected memory is mediated by the NCAs.

The NGSCB architecture requires a special CPU with support for features like “curtained memory.” There is support for a new mode flag which allows the CPU to run in either the normal mode or a nexus mode. The CPU must support context switches between the normal and nexus modes.

2.5.2.1. Key Features of the NGSCB Architecture

Strong process isolation is a key feature of this architecture. NGSCB provides “curtained memory” which is a secure memory area where applications can run securely. Applications executing in this memory space are inaccessible to other applications.

The “sealed storage” feature involves the protection of keys and data via encryption. Data or keys associated with one NCA need not be shared even with other NCAs. Sealed storage is provided by the SSC.

Via encryption, a trusted path can be created from the I/O devices to trusted applications. Information sent across this path is thus protected. Secure channels allow data to move safely from the keyboard/mouse to nexus-aware applications, and for data to move from nexus-aware applications to a region of the screen.

This architecture also provides for attestation, which involves authentication of software or data to other entities.

The Security Support Component (SSC) is the name that Microsoft uses for the trusted platform module and this provides cryptographic storage of keys and data, while being part of the initialization process of the nexus. The SSC has a cryptographic engine that provides RSA public key operations as well as AES operations (which is optional in the TCG specification) and SHA-1 support. The SSC has a manufacturer-issued certificate for the RSA public key (the equivalent of a TCG EK). This key can uniquely identify the NGSCB computer, so other keys are generated for general use. The SSC key is used to certify other RSA keys generated by the SSC on a particular nexus. The private key of the SSC keypair is always protected and never accessible outside the SSC.
2.5.3. LaGrande Architecture

Intel’s LaGrande architecture, now renamed “Intel Trusted Execution Technology” [23], is defined as a “set of enhanced hardware components designed to help protect sensitive information from software-based attacks, where features include capabilities in the microprocessor, chipset, I/O subsystems, and other platform components” [43]. LaGrande architecture has an enhanced CPU, chipset as well as the keyboard mouse in addition to the graphics components. This architecture has a TPM version 1.2 device called the “fixed token.”

2.5.3.1. Key Features of the LaGrande Architecture

This architecture allows for “protected execution”. Applications can run in a protected execution space, separated from other processes. The protected space has dedicated resources.

LaGrande also provides “sealed storage”. Similar to other trusted hardware architectures, LaGrande provides for encryption of keys and data. These secrets can be bound to a certain configuration and decrypted only when the platform is in the same configuration.

Communication channels between the I/O devices and the protected applications is secured from unauthorized snooping. The keystrokes and the mouse clicks are encrypted with a key shared between the protected domain and the input device [43].

Similar to NGSCB and the TPMs, measurement of the software running in the protected environment is possible. This information can then be sent to the requesting party.

2.5.4. Web Applications with Lots of Privacy and Security (WebALPS)

WebALPS uses a secure co-processor (based on the IBM 4758) to secure web applications [46, 69]. WebALPS allows clients to have a greater degree of trust in web servers that run applications on their behalf. Trust in a web server is usually based on a Certification Authority (CA) which issues the certificate binding the public key of the server to its identity. The certificate, however, cannot certify the practices and policies of the server, merely its identity. A client often has no recourse but to trust that the server protected the integrity or the confidentiality of its application. The WebALPS project addresses this
problem by enhancing the server with the IBM 4758 co-processor and using its outbound authentication feature to attest to the client that the application (on the server) executed as it was supposed to. A small set of application programs run on the co-processor which is called a trusted co-server. Clients then use an authenticated SSL connection to the program running on the co-server called the “guardian” program. The client-server interaction then is really a client-co-server interaction. Data and computations in the trusted co-server are thus protected from the untrusted server.

2.5.5. XOM Architecture

XOM [50] is different from all the other architectures described in previous sections. The XOM trust model is also designed to protect against attackers who have physical access to the hardware. Trust is not based in the OS or the main memory, so data stored in memory are encrypted, along with their hashes of values and virtual addresses. To prevent applications from tampering, each application is placed in a separate compartment; this compartmentalization is enforced by hardware data tagging [50] and cryptographic protections. The OS cannot access programs running in the compartments. To accomplish this, XOM requires on-chip protection of memory caches and registers as well as confidentiality of code and data when transferred to untrusted memory. To support this, hardware cryptographic support is required as well as extensive modifications to the CPU.

XOM supports secure boot, with each layer in the boot process authenticating the layer above it. XOM utilizes hardware tamper-resistant properties to protect a master secret which is different for each XOM-enabled processor. This master secret is used to encrypt keys that, in turn, encrypt various software programs. All operations that use the master secret must be implemented on the processor. This architecture introduces new hardware instructions. Compartments are implemented using both asymmetric and symmetric encryption.

Therefore, XOM is different from the TCG specifications and other architectures described in that all accesses to memory are encrypted. Programs always execute in protected compartments and XOM uses encryption to manage untrusted memory. Additionally, XOM
does not include any separate hardware module - all protections are provided by the main platform CPU and its memory interface.

2.6. Theoretical Modeling of Hardware-assisted Security

Very recently, researchers have begun to look at how the existence of tamper-proof hardware affects the way researchers reason about the security of protocols from a theoretical perspective. Our construction of a TPM oracle [39] that is indistinguishable from a random oracle (to a polynomial-time attacker) is one of first such efforts. In the standard model of computing, a cryptographic model widely used to reason about the security of protocols is the universal composability (UC) model [19]. The notion of universally composable security was introduced by Canetti in order to provide strong security guarantees about protocols when used in arbitrary environments, including being composed with other protocols. Canetti and Fischlin also proved that some fundamental two-party protocols, including commitment [16] and general zero-knowledge proofs [34], cannot exist in the “plain model”, i.e., the standard model. Key questions in the UC framework, therefore, are which cryptographic tasks are realizable in this framework and under which set of assumptions are they realizable. This model is therefore augmented with certain setup assumptions like the common reference string (CRS) and protocols are proven secure in these augmented models. The current consensus is that all existing solutions in the UC model require some trusted party to initialize the setup in the “plain model”.

Recent work by Katz [47] (2007) proposes a new setup assumption regarding the existence of tamper-proof hardware. Katz proposes moving the trust from an actual trusted third party (TTP) to the physical assumption about the environment in which the execution is run, namely a tamper-proof environment. With this assumption, secure multiparty computation is possible in the UC model without the need for any trusted parties to initialize the setup.

The physical assumption of tamper-proof hardware is modeled in the following way: a user can create a hardware token that implements any polynomial-time computable functionality but an adversary cannot affect this execution in any way other than to observe the input/output behaviour of this token. The “real-world” sequence of events that this
ideal functionality tries to model is that a user can take some software process and seal it inside a tamper-proof hardware token and then give this token to other parties (including the adversary). These parties now have only “black-box” access to it.

Damgard et al. [27] extend Katz’s work with some modified assumptions about the behaviour of the hardware tokens. In particular, they assume that the hardware tokens are only partially isolated from the executing environment (in Katz’s work, the tokens are assumed to be totally isolated). They also present a scheme that is proven secure based on standard cryptographic assumptions and is secure against adaptive adversaries.

Chandran et al. [21] extended this model by modifying some of the assumptions in Katz’s work. In particular, they make no assumptions about how malicious parties may create the hardware token (in both the models described above, all parties are assumed to have knowledge of the program code inside the token). Additionally, the security of those constructions relies on the ability of the hardware token to maintain state. Chandran et al. relax this assumption and prove the security of their constructions with the assumption of resettable hardware tokens. These assumptions allow them to prove the security of their constructions in a more generalized model.

In contrast to these models, this work does not allow arbitrary functionality inside the hardware-protected environment, but rather keep functionality minimal and as close to TPM specifications as possible. Even with these restrictions I am able to build up significant new secure capabilities.

2.7. Other TPM Capabilities

In this section, I present background information on TPM functionality that is part of the TCG design but is not directly used in this dissertation.

2.7.1. Roots of Trust

A root of trust is a component that is trusted to operate as expected without any oversight is called a root of trust. Ultimately, the manufacturer is trusted to manufacture the
component to specifications. Common Criteria evaluation [42] can also provide some assurance that the component has been manufactured in a standard manner. A typical trusted platform has three roots of trust, each of which is trusted to function properly on its own.

The root of trust for measurement (RTM) is a computing engine that can make reliable integrity measurements. This is typically controlled by the “core root of trust for measurement” or (CRTM), which is the set of instructions executed when the platform acts as the RTM. The root of trust for storage (RTS) is a computing engine capable of maintaining accurate integrity measurements made by the RTM. Typically, this is a list of digests (hash values). The root of trust for reporting (RTR) reliably reports the information maintained by the RTS. The RTM is the root of the chain of transitive trust. When a TPM starts up, it must start at the CRTM and then it relies on other trusted building blocks like the RTR and RTS. On power-up, the RTM signals the TPM instructing it to start the initialization process, which includes a self-test that determines if the TPM is working properly. The TPM can create a chain of trust, starting from the hardware root of trust as shown in Figure 2.3. The CRTM measures the BIOS before passing control to it, and the BIOS in turn measures the bootloader. The PCRs are used to store the digests of each step. The

![Diagram](image.png)

**Figure 2.3. Transitive Root of Trust**
bootloader measures the OS and then passes control to it. The OS can measure software applications and store metrics about them.

2.7.2. Integrity Measurement and Reporting

Platform configuration registers are 160-bit registers, which are shielded locations in the TPM. A PCR is basically a SHA-1 hash of a value, typically a measurement value. To allow for an unlimited number of measurements to be stored, the value of a PCR is typically extended as follows: $PCR[i] = SHA-1(PCR[i]|| new \ value)$. Apart from storing integrity metrics, PCRs are also used by the protected storage functionality before revealing secrets bound to certain platform states and in the boot process to check if the boot process is proceeding correctly.

Integrity measurement is the process of obtaining quantitative data about platform characteristics. A TPM can be used to ensure that each platform will report its configuration parameters in a trustworthy manner. The TPM can measure the characteristics of hardware and software components of a platform, starting with the boot process, and securely store the results of these measurements in PCRs within the TPM. Integrity reporting involves attesting to the authenticity of these stored measurements and typically involves signing the measurements using an AIK (described in Section 2.4). The challenger can verify the AIK credential (signed by the Privacy CA) and the corresponding measurements by re-computing the hash.

2.7.3. Miscellaneous TPM Features

2.7.3.1. Authorization Protocols

The TPM specification distinguishes between a TPM owner and a TPM user. A TPM owner shares a secret with the TPM – the act of taking ownership of the TPM creates a 20-byte shared secret between the TPM and the entity taking ownership. The owner may then allow various users (who do not have knowledge of the owner secret) to use the TPM. The distinction between the user of a TPM and the owner can be illustrated with a simple example. In a company, laptops issued to employees may all have TPMs. In that case,
only authorized personnel in the IT department may have knowledge of the owner password, which is not passed on to the employee. The employee can still use the laptop with delegated authority from the owner for certain privileged commands. In version 1.1 of the TCG specifications, knowledge of the owner password was required for some special commands, so the TPM owner had to share the owner password with other entities that wanted to access these commands. In version 1.2, the delegation model was introduced, whereby the owner could delegate the permission for these commands without sharing the owner password with other entities. For certain privileged operations, the owner must provide proof of the shared secret to the TPM via an authorization protocol. There are various authorization protocols, but in general, they all work the same way using a challenge-response mechanism with randomly generated nonces to allow the user or the owner to prove knowledge of the authorization data. The authorization mechanism itself uses the HMAC function, with the authorization data as the key to the function. TPM-protected objects, namely both data and keys, may also require knowledge of 20 bytes of authorization data to be input before these objects can be used. The TPM also has certain “physical presence” commands which do not require any authorization data but require proof of physical presence by a user at a TPM. Commands that require physical presence include clearing a TPM owner and some operations on non-volatile storage.

2.7.3.2. Sealed Storage

Sealed storage is a powerful functionality provided by the TPM whereby data is bound to a specific encryption key as well as to a particular platform configuration state. Data is encrypted using a symmetric key and then the platform configuration state (stored as SHA-1 digests) and the symmetric key are encrypted using a non-migratable asymmetric key. The symmetric key will now be decrypted by the TPM only when the platform configuration is in the state specified by the digests.
2.7.3.3. Attestation

Attestation is a mechanism to guarantee the authenticity of some information. A trusted platform can attest to the authenticity of its platform characteristics by signing measurements with keys that are guaranteed to only be usable within a TPM. Upon the owner’s request, the TPM can reliably communicate information about its capabilities to a remote platform. For privacy purposes, the attestation mechanism must not uniquely identify a TPM. The initial attestation mechanism specified by the TCG involved the use of a trusted party known as a “Privacy certificate authority (CA)” to preserve user privacy. Each TPM has a certificate that ties the public key of its EK to its identity. This certificate is provided by the manufacturer. Subsequently, all certificates for public keys originating from a TPM are issued by a Privacy CA who checks the manufacturer-issued certificate for the EK before issuing the new certificates. The Privacy CA is essentially a trusted third party. The Privacy CA’s knowledge ties each EK to each identity key used by the TPM. To allay the privacy concerns associated with the PrivacyCA, in version 1.2 of the specification this scheme was supplemented by the addition of the Direct anonymous attestation (DAA) method [13], a complex mechanism based on zero-knowledge proofs which obviates the need for trusted Privacy CAs.

2.8. Migration Functionality

In this section, I present details of how different types of keys can be migrated between TPMs and other platforms. I wrap up the section with a discussion of how CMKs can be migrated. As described in section 2.4, keys can be designated as migratable, non-migratable or as a CMK. Migration mechanisms were introduced to allow users to backup migratable keys protected by the TPM. CMKs can be migrated under stricter restrictions than migratable keys as described below. The TCG specification provides two methods for migration of migratable keys; both mechanisms require the TPM owner to authorize the migration destination.
Direct Migration involves encrypting the migratable key with the destination public key. The encrypted blob is migrated to the destination along with a plaintext description of the corresponding public key.

Indirect Migration involves the use of an intermediate party in the migration process. The migratable key is first OAEP encoded and XORed with a one-time pad and then encrypted using the public key of the intermediary. The resultant object is then unwrapped by the intermediary and then encrypted using the public key of the destination, where the key can be inserted into the storage hierarchy. The one-time pad must be made available to the destination in this case.

Certifiable migratable keys can also be migrated directly or indirectly; however the migration destination must be authorized by the migration authority (MA) as well as by the TPM owner. In each case, the MA must “sign-off” on the destination TPM, even if the CMK is being migrated directly to the MA. The MA creates a structure which contains the public keys of the CMK, the source TPM and the destination TPM and creates a signature ticket on this structure. This ticket is then verified at the destination before the CMK is stored there. The CMK can then be migrated directly to the destination TPM (if it is an MA) or it can be wrapped under the MA’s public key and then re-wrapped under the destination public key.
CHAPTER 3

THEORETICAL MODELS

In this chapter, I present background information on various theoretical concepts used in this dissertation. I present definitions of the random oracle model, oblivious transfer, secure function evaluation and virtual monotonic counters as well as formal definitions of other theoretical concepts used in this dissertation.

When reasoning about the security of a cryptographic scheme, it is common to create a series of experiments or attack games involving an adversary, $\mathcal{A}$. The adversary is a probabilistic polynomial-time Turing (PPT) machine, i.e., it always halts after a polynomial (in the length of the input) number of steps. There is an advantage function, $Adv$, associated with the adversary which is basically the probability that the adversary “breaks” the scheme. The game is typically played between the adversary and the challenger. Since different adversaries may have different advantages in breaking a scheme, it is common to take the maximum over all the possible adversaries. Informally, the advantage of the adversary in breaking the scheme should be small.

The adversary algorithm may be randomized and it is usually given oracle access to some function, $O$, denoted as $\mathcal{A}^O$. This means that $\mathcal{A}$ can only query the oracle at values in the function’s domain and obtain the output of the oracle at those values; it cannot directly examine the function.

Security proofs are based on reductions, with results of the form: If an adversary could win this game then it could be used to create algorithm to ”solve problem $X$” or to “break primitive $X$”, where problem $X$ is something that is believed to be fundamentally hard (like factoring or discrete log). In this work, the primitives are assumed to be unbreakable and then the higher-level composed functionality is shown to be unbreakable under those assumptions.
3.1. Definitions

In this section, I present formal definitions for the random oracle model and for common cryptographic concepts such as indistinguishability that I rely on in the following sections.

Definition 3.1. Random Oracle

A random oracle is a truly random function $H : \{0,1\}^* \rightarrow \{0,1\}^*$ which is universal and consistent across all parties, so that if Alice requests a value $H(x)$ from the oracle and Bob later also requests $H(x)$, they should obtain the same value.\(^1\) This function is called an oracle and access to this oracle for any protocol $\mathcal{P}$ is typically denoted as $\mathcal{P}O$. All parties including the adversary are allowed to query the oracle on inputs of their choice.

A core concept in security of pseudorandom generators and functions is that of “indistinguishability” — informally, this means that a polynomial algorithm cannot tell the difference between a pseudorandom output and a truly random output. The “game” here is somewhat reminiscent of the Turing Test in AI: the adversary is interacting with one of two possible other parties, either truly random or pseudorandom, and must try to determine which one.

Definition 3.2. Computational Indistinguishability

Let $l(k)$ and $m(k)$ be polynomially bounded length functions, where $k$ is interpreted as a security parameter, and let $\mathcal{X}_k(x)$ for $k \in \mathbb{Z}$ and $x \in \{0,1\}^{l(k)}$ be a collection of random variables (an “ensemble”) drawn from $\{0,1\}^{m(k)}$. $\mathcal{X}_k$ is used to denote the collection of random variables $\mathcal{X}_k(x)$ with $k$ fixed, $\mathcal{D}(\mathcal{X}_k)$ is used to denote an algorithm $\mathcal{D}$ that can sample random variables $\mathcal{X}_k(x)$ for any $x \in \{0,1\}^{l(k)}$. A distinguisher for ensembles $\mathcal{X}_k$ and $\mathcal{Y}_k$ is an algorithm $\mathcal{D}$ with boolean output such that for all sufficiently large $k$,

$$|\text{Prob}[\mathcal{D}(\mathcal{X}_k) = 1] - \text{Prob}[\mathcal{D}(\mathcal{Y}_k) = 1]| > \frac{1}{p(k)},$$

---

\(^1\)In practice, the domain and range of the random oracle are determined by the requirements of the cryptographic scheme using the random oracle, and are typically fixed-length binary strings where the length depends on a security parameter.
where $p(k)$ is some polynomial in $k$. Ensembles $\mathcal{X}_k$ and $\mathcal{Y}_k$ are said to be computationally indistinguishable if there exists no probabilistic polynomial time distinguisher for these ensembles.

I present a formal definition of a PRF and a negligible function as presented by Goldreich [35].

**Definition 3.3. Pseudorandom functions**

Let $d, r \in \mathbb{N} \rightarrow \mathbb{N}$. Then

$$\left\{ f_s : \{0, 1\}^{d(|s|)} \rightarrow \{0, 1\}^{r(|s|)} \right\}_{s \in \{0, 1\}^*}$$

is a pseudorandom function ensemble if the following conditions hold:

(i) Efficient evaluation: There exists a polynomial time algorithm that on input $s$ (also commonly called the *seed*) and $x \in \{0, 1\}^{d(|s|)}$ returns $f_s(x)$.

(ii) Pseudorandomness: For every probabilistic, polynomial time oracle machine $M$, every polynomial $p(\cdot)$, and all sufficiently large $n$'s,

$$\left| \text{Prob}[M^{F_n}(1^n) = 1] - \text{Prob}[M^{H_n}(1^n) = 1] \right| < \frac{1}{p(n)},$$

where $F_n$ is a random variable uniformly distributed over the multi-set $\{f_s\}_{s \in \{0, 1\}^n}$ and $H_n$ is uniformly distributed among all functions mapping $d(n)$-bit long strings to $r(n)$-bit long strings.

**Definition 3.4. Negligible Functions**

A function $\mu : \mathbb{N} \rightarrow \mathbb{R}$ is called negligible if for every positive polynomial $p(\cdot)$ there exists an $N$ such that for all $n > N$,

$$\mu(n) < \frac{1}{p(n)}.$$

**Definition 3.5. Public key encryption (PKE)**

A public key encryption scheme consists of the following probabilistic, polynomial time algorithms:
(pk, sk) ← PKE.Gen(1^\lambda): A probabilistic algorithm that, on input, security parameter, \lambda, generates public and private keys (pk, sk). The public key defines the message space \mathcal{M}.

c ← PKE.Enc_{pk}(m): A probabilistic encryption algorithm that encrypts message \(m \in \mathcal{M}\) into a ciphertext c.

m ← PKE.Dec_{sk}(c): A deterministic decryption algorithm that returns \(m \in \mathcal{M}\) as the decryption of c or a special symbol \(\bot \notin \mathcal{M}\).

One of the key properties required from any secure encryption scheme is that the ciphertext should not leak any partial information about the plaintext. The following definition of chosen ciphertext security captures this notion and intuitively, even if the attacker obtains decryptions of arbitrary ciphertexts of his own choosing, he obtains no partial information about other plaintexts.

Definition 3.6. Indistinguishability against chosen ciphertext security (IND-CCA)

This is also called IND-CCA2 or adaptive CCA security. For a probabilistic asymmetric key encryption algorithm, indistinguishability under chosen ciphertext attack is defined by the following game between an adversary and a challenger.

Game.PKE is defined as follows: Let O be the decryption oracle, PKE.Dec_{sk}() and let A be a PPT adversary that plays the following game.

(i) (pk, sk) ← PKE.Gen(1^\lambda)
(ii) (m_0, m_1, \rho) ← A^O(pk)
(iii) b $\xleftarrow{\$} \{0, 1\}, c ← PKE.Enc_{pk}(m_b)
(iv) \hat{b} ← A^O(\rho, c)

In Step (iv), A is restricted not to ask c of O. \(m_0\) and \(m_1\) must be of the same length.

Define \(Adv_{PKE,A} = |Pr[\hat{b} = b] - \frac{1}{2}|\) and \(Adv_{PKE} = max_A(Adv_{PKE}, A)\) where the maximum is taken over all machines.
A scheme is said to be secure against adaptive chosen ciphertext attacks (CCA-secure) if $Adv_{PKE}$ is negligible in $\lambda$.

Definition 3.7. Indistinguishability under chosen plaintext attack (IND-CPA security)

For a probabilistic asymmetric key encryption algorithm, indistinguishability under chosen plaintext attack (IND-CPA) is defined by the following game between an adversary and a challenger. Let $\mathcal{O}$ be the encryption oracle, $PKE.Enc_{pk}()$ and let $A_E$ be a PPT adversary that plays the following game.

(i) $(pk, sk) \leftarrow PKE.Gen(1^\lambda)$
(ii) $(m_0, m_1, \rho) \leftarrow A^\mathcal{O}_E(pk)$
(iii) $b \leftarrow \{0, 1\}, c \leftarrow PKE.Enc_{pk}(m_b)$
(iv) $\hat{b} \leftarrow A^\mathcal{O}_E(\rho, c)$

$A_E$ may request encryptions of unlimited number of messages, before and after the encryption oracle returns the target ciphertext.

Define $Adv_{pke, A_E} = |Pr[\hat{b} = b] - \frac{1}{2}|$ and $Adv_{pke} = max_{A_E}(Adv_{pke, A_E})$ where the maximum is taken over all machines. A scheme is said to be secure against chosen plaintext attacks (CPA-secure) if $Adv_{pke}$ is negligible in $\lambda$.

Definition 3.8. Symmetric key encryption (SKE)

In a symmetric key encryption scheme, the keyspace is defined as $\mathcal{K}_D = \{0, 1\}^\lambda$ and the message space is defined as $\{0, 1\}^*$. An SKE scheme consists of the following polynomial time algorithms:

$(dk) \leftarrow SKE.Gen(1^\lambda)$: A probabilistic algorithm that, on input, security parameter, $\lambda$, generates symmetric key $(dk)$.

$\chi \leftarrow SKE.Enc_{dk}(m)$: An encryption algorithm that encrypts $m$ into ciphertext $\chi$ using key $dk \in \mathcal{K}_D$.

$m \leftarrow SKE.Dec_{dk}(\chi)$: A decryption algorithm that returns $m \in \mathcal{M}$ as the decryption of $\chi$ or a special symbol $\bot \notin \mathcal{M}$. 

33
Definition 3.9. Indistinguishability against Chosen Ciphertext Security (IND-CCA) for SKE schemes

For a symmetric key encryption algorithm, indistinguishability under chosen ciphertext attack is defined by the following game between an adversary and a challenger.

Game.SKE is defined as follows: Let $O$ be the decryption oracle, $SKE.Dek()$ and let $A$ be a PPT adversary that plays the following game.

(i) $(m_0, m_1) \leftarrow A_O(1^\lambda)$
(ii) $b \leftarrow \{0, 1\}, \chi \leftarrow SKE.Enc_{dk}(m_b)$
(iii) $\hat{b} \leftarrow A_O(\rho, \chi)$

In Step (iii), $A$ is restricted not to ask $\chi$ of $O$. $m_0$ and $m_1$ must be of the same length.

Define $Adv_{SKE,A} = |Pr[\hat{b} = b] - \frac{1}{2}|$ and $Adv_{SKE} = \max_A(Adv_{SKE}, A)$ where the maximum is taken over all machines.

A scheme is said to be secure against adaptive chosen ciphertext attacks (CCA-secure) if $Adv_{SKE}$ is negligible in $\lambda$.

3.2. Background on Random Oracles

The random oracle model [9] is perhaps the most successful theoretical model used to design practical cryptographic algorithms and protocols. This model has been successfully used to design such cryptographic techniques as the probabilistic signature scheme (PSS) [10] and optimal asymmetric encryption padding (OAEP) [66], which are widely used in practice. The idea with this model is to prove a cryptographic scheme secure in the random oracle model, where all parties including the adversary have access to a random function, called a random oracle, and then replace the random oracle with a “good” cryptographic hash function in the standard model. A proof of security in the random oracle model is then taken as evidence that a scheme is secure in the standard model if the instantiation of the oracle by the hash function is secure, although this reasoning has some weaknesses as I describe below. The instantiation of the oracle in the real world requires certain properties; a simple hash function does not have the required properties. Various efforts [15, 20, 29] were
made to identify properties that are required for real-world instantiation of these oracles. These properties include pseudorandomness, collision resistance and one-wayness. Several combinations of these properties were used to realize various constructions, but none of these had all the properties required of a real-world random oracle. This real-world construction of a random oracle, encompassing all the properties required of a true random oracle, was termed a “pseudorandom oracle” [8] by Bellare and Ristenpart.

3.2.1. Properties Required for a Random Oracle

Various candidates for instantiation of random oracles in the real world have been proposed. Among these are pseudorandom functions [6, 5], hash functions [22] and verifiable (pseudo)random functions [55]. As defined in Section 3.3, a PRF is a function whose output is indistinguishable from random to a polynomial time attacker, as long as the seed is secret.

A construction called perfectly one way hash functions (POWHF) (or oracle hashing) was proposed by Canetti and others [20, 15] and this construction captured many of the properties required by a truly random function. POWHFs are one-way, pseudorandom functions with the properties of collision resistance and oracle secrecy. These functions, though randomized, are publicly computable.

Another proposed instantiation of a random oracle is the verifiable (pseudo)random function (VRF) proposed by Micali et al. [55]. VRFs are similar to PRFs in that their outputs look random, but are not publicly computable since they require a secret key or seed to obtain the pseudorandomness property. In order to keep the seed secret, the VRF instantiation of the PRF allows for the use of a separate trusted third party (TTP) to provide PRF functionality. Parties interact with the TTP to compute $PR_a(x)$ or to verify that all parties are using the correct value of $PR_a(x)$. In order to remove trust in a single point of failure, the authors suggest a distributed version of the VRF whereby trust is distributed in a number of parties using threshold cryptography. I note that this approach of using a TPM is similar to relying on a trusted party as in the use of VRFs. The TPM, while not an independent party, is a participant that is not completely under the control of the platform in which it resides, and is trusted to accurately implement its specification. Due to the tight interaction
between a TPM and the platform in which it resides, the strong verifiability properties of the VRF are not necessary in the TPM-based model.

Using the *indifferentiability* framework proposed by Maurer *et al.* [53], Coron *et al.* [22] formalized what it means to create a random oracle from a fixed-length compression function or a block cipher. The indifferentiability framework generalizes the concept of indistinguishability (as used in the definition of a PRF) and outlines the definitions required for comparing ideal objects with standard instantiations of these objects. The proposed secure constructions presented by Coron *et al.* are proven secure in this framework. Extending their work, Bellare and Ristenpart [8] use the term “pseudorandom oracle” to define constructions that are indifferentiable from random oracles. Bellare *et al.* also look at strengthening hash functions to obtain pseudorandom oracle-like properties. They propose a new construction that is a pseudorandom oracle while preserving the collision-resistance property of the hash function.

### 3.2.2. Impossibility Results

Amid efforts to outline properties required for secure instantiations of random oracles in the standard model, various authors [4, 18, 17] showed that there is a signature scheme that is secure in the random oracle model but is insecure in the standard model for any instantiation of the random oracle. Though the scheme in question was a contrived example, this cast doubts on other schemes as well, and various practical schemes were examined to see if they retained their security in the standard model. Bellare *et al.* [4] define a scheme as *uninstantiable* with respect to some goal if there is a secure implementation of the scheme in the random oracle model that meets this goal, but no instantiation of the scheme meets the goal in question in the standard model. When instantiating the scheme in the standard model, the random oracle is replaced by some family of functions like the examples described above. Canetti, Goldreich and Haveli [17, 18, 53] showed that there exist uninstantiable schemes for the cryptographic goals of IND-CPA secure encryption and digital signatures secure against chosen message attacks. Bellare *et al.* [4] showed that the Hash
ElGamal scheme, a hybrid encryption scheme, is uninstantiable for the goal of IND-CCA secure asymmetric encryption.

Not all schemes had negative outcomes: Canetti et al. proved that a hybrid encryption scheme proposed by Bellare and Rogaway [9] (secure in the random oracle model) retained its security in the standard model when the random oracle was instantiated with an oracle hash in the standard model. Boldyreva et al. [11] showed that it was possible to securely instantiate one of the two random oracles in a variant of the PSS-E encryption scheme (called PSS-I) [10] and either one of the two ROs in the Fujisaki-Okamoto hybrid encryption scheme [33] through oracle hashing. However, they also showed that it is not possible to instantiate either of the ROs in the OAEP encryption scheme [66] with POWHFs without losing chosen-ciphertext security. They also showed that replacing the random oracle in the FDH signature scheme [30] with a VPRF may result in an insecure scheme.

While these negative results don’t imply that all schemes designed using random oracles are insecure, they do show using security proofs that rely on the full power of random oracles to imply security in the standard model are inherently flawed, and security guarantees in such a situation are at best heuristic.

I now present background information on the theoretical concepts used in the construction of a new primitive, generalized non-interactive oblivious transfer (GNIOT).

3.3. Non-Interactive Oblivious Transfer and Secure Function Evaluation

1-out-of-2 oblivious transfer (OT) is a primitive between a sender Alice and a receiver Bob, defined as follows:

Definition 3.10. 1-out-of-2 oblivious transfer

Alice (the sender) has 2 values $s_0$ and $s_1$, and Bob (the receiver) has a selection bit $c$. At the end of the protocol, Bob learns the value $s_c$ while obtaining no information about $s_{1-c}$, and Alice cannot determine which value Bob received.

Two-party secure function evaluation (SFE) is a cryptographic primitive that allows two parties, Alice and Bob (with inputs $a$ and $b$ respectively) to compute a function $(A, B) \leftarrow$
\( f(a, b) \) such that Alice learns output value \( A \) and Bob learns output \( B \), and neither party learns anything more than what follows from its own values. Yao showed that for any polynomial-time computable function \( f \), there exists a polynomial time SFE protocol [85]. The function is represented as an \textit{encrypted circuit} where the values on the input wires are random strings (called signals) instead of the actual boolean values, and the mapping of the random signals to the real inputs is kept secret. Through carefully-specified truth tables that allow evaluation of gates without needing to know the semantics of the random signals, the encrypted circuit can be evaluated without any information being revealed to the evaluator. The result of the evaluation is in encoded form as well, and to decode the output, knowledge of the mapping of the random signals to the real outputs is required.

Typically, in a two-party protocol, Alice creates an encrypted circuit to evaluate the desired function. Then Alice sends the encrypted circuit as well as the semantics for the signals for the output wires for Bob’s output and the encrypted signals corresponding to her input to Bob. In order for Bob to receive the signals corresponding to his input, he engages in a 1-out-of-2 oblivious transfer protocol with Alice for \textit{each bit} of his input. Via this protocol, Bob obtains the signals for one input bit value, but not the other, while Alice remains unaware which value the signals correspond to. Bob never learns Alice’s input since the semantics of the signals corresponding to her input are never revealed to him. Bob now evaluates the encrypted circuit, having obtained both sets of inputs and retains his output and returns Alice’s output (still in encrypted form) to her. Bob uses the semantics of the signals for his output wires to decrypt his output. Similarly, Alice decrypts her output.
CHAPTER 4

INSTANTIATING A RANDOM ORACLE USING A TPM

4.1. Introduction

This chapter outlines one of the main results [39] of this dissertation, the construction of a random oracle using TPM functionalities. I prove that this construction is indistinguishable from a random oracle to a polynomial-time attacker. As a result, any protocol proved secure in the random oracle model will remain secure when the “TPM oracle” is used in place of the random oracle. I show how to utilize a CMK to share secrets across platforms and propose minor changes to existing TPM functionality in order to instantiate a TPM oracle.

One of the proposed instantiations of a random oracle in the real world is a pseudorandom function. A PRF produces output that is computationally indistinguishable from random to a polynomial-time attacker. PRFs are natural candidates for instantiating random oracles; however, this approach does not give the strong guarantees that one would like. The problem with instantiating a random oracle using a PRF is that, in order for it to be secure, the seed used to key the PRF must be kept secret. However, in protocols in which a party participating in the protocol can be corrupt, and yet must have access to the random oracle, it is impossible in a traditional model of computation to keep the seed from the adversary.

In this chapter, I explore the possibility of utilizing the unique functionality of the trusted platform modules to instantiate a random oracle, when all parties have access to a TPM. A TPM has precisely the capability that is missing in the preceding discussion regarding the use of a PRF: the ability to keep secrets from the platform owner while allowing the use of those secrets in carefully controlled ways. Additionally, a CMK allows a set of TPMs to establish a shared secret in such a way that all parties have assurance that these secrets have never been available outside of a protected TPM environment.
Using such a shared secret as a seed to a PRF would then allow a platform to utilize this PRF with an unknown seed in place of a random oracle. What this does is allows us to move from heuristic arguments based on random oracles to rigorous proofs based on an assumption that TPMs securely implement their specified functionality (in addition to some standard complexity assumptions). While this means that this construction relies on an additional assumption, I feel that this is in many instances, a better foundation than the demonstrably flawed reasoning regarding random oracles being instantiated in the standard model.

In this chapter, I present a secure instantiation of random oracles in multi-party protocols using a combination of existing and easily provided TPM functionalities, outlined in Section 4.3. I present a formal definition and a security proof of a new cryptographic primitive called a “hybrid pseudorandom function” in Section 4.3.3 and finally outline the solution of an interesting subproblem, in Section 4.3.1, that of using CMKs to establish a shared secret without needing the participation of a migration authority as an active trusted third party.

4.2. Use of TPM Functionalities and Extensions

Ideally, I would like to design protocols that work with existing, standard TPMs. However, that is not possible in this case, and I require three modifications to a standard TPM. Two of these modifications are trivial, and the third would be easy to accomplish but is a more substantial change.

The first trivial change is to add the capability for using a TPM secret as a key to HMAC, which I will use as a PRF with an unknown seed. Since TPMs have protected storage for secrets and must support HMAC for other operations, this is simply a matter of adding the right command to the TPM’s command set. The second trivial change is the creation of a new key type, usable only as a secret key for HMAC in the command I just described — consistent with the TCG specification naming, I suggest a new TPM_KEY_USAGE value with the name TPM_KEY_HMAC. Note that this change is entirely optional, but I feel it is desirable for two reasons: First, as a valid key type sent to the TPM_CMK_CreateKey command, I could avoid the costly operation of generating a new asymmetric key, which is completely
unnecessary in this situation. Second, good cryptographic practice states that keys should be used for a single purpose — while I could, for example, create a signing key and use this as a shared secret (this is in fact what I do in Section 7.5.2 to benchmark these operations on an existing TPM), it unnecessarily complicates security arguments when I have to consider the impact of operations unrelated to my intended use for this key.

The third TPM modification is more significant, and is necessary due to the following chicken-and-the-egg problem: the overall goal is to instantiate a random oracle, and make rigorous security arguments which are free of the kinds of problems that arise in instantiating random oracles in the standard model. However, the standard TPM facility for migrating a CMK uses RSA encryption with OAEP, a random-oracle designed scheme that uses a standard hash function (SHA-1 in the TPM) for the random oracle. While such a system may be secure, it does not allow us to have a properly self-contained security proof. Fortunately, in order to perform CMK migration, I only need encryption and signature capability, and techniques are known for both of these that are secure in the standard model (not relying on random oracles). For concreteness, I assume that encryption is done using the Cramer-Shoup CCA-secure encryption scheme [25] and signatures are performed using Fischlin’s modification [32] to the Cramer-Shoup signature scheme [24]. Both of these schemes have been proven secure without the need for random oracles, with security that is based on standard complexity assumptions (the decisional Diffie-Hellman and strong RSA assumptions) and requiring operations of modular exponentiation, which must be already supported by TPMs due to the use of RSA in a standard TPM.

4.2.1. The Role of Trusted Third Parties

TPMs are not manufactured with any globally significant secrets, and the design is completely open. Because of this, a TPM simulator could easily be built which completely mimics the behavior and properties of a real TPM. The only thing that distinguishes a real TPM from a simulator or other fake TPM environment is the certificate for the TPM’s endorsement key, created by the manufacturer with the policy that endorsement key certificates will only be issued for keys embedded in and protected by actual, TCG-compliant TPMs. As
described above, the EK is used to establish the authenticity of a TPM interacting with a Privacy CA, which issues an AIK credential (certificate). If the Privacy CA is trusted to properly verify the EK certificate and only accept trustworthy EK CAs, then an AIK and its accompanying certificate can be used as assurance that you are interacting with a real TPM. Because of the necessity of trustworthy CAs for both the EK and the AIKs, it is impossible to gain any capabilities from a TPM without including a trusted third party (TTP) in the model of computation.

While the preceding argument shows that TTPs are necessary, it is important to distinguish between two different types of TTPs: active TTPs and key-certifying TTPs. An active TTP participates in interactive protocols between different TPMs. A migration authority, as described by the TCG specifications, is an example of an active third party, since key migration happens with the active participation of the migration authority. On the other hand, a key-certifying TTP is a much weaker requirement — a key-certifying TTP acts as a CA at some point during the lifetime of a key, but does not need to actively participate in protocols. The techniques I present in this paper only require this weaker form of TTP, the key-certifying TTP. In particular, this construction need trustworthy Privacy CAs for certifying AIKs, but don’t need a TTP for any other operation.

4.3. Implement a Random Oracle with a TPM

Since a PRF is a natural candidate for replacing a random oracle, I propose using a slightly modified TPM to compute the PRF while keeping the seed secret. Specifically, I use certifiable migratable keys, which are keys certified by the TPM and which can be used for computations at the request of the host, but are kept secret within the TPM. Since the TPM already possesses an internal HMAC engine for calculating and storing HMAC digests, it is simple enough to add an interface for performing HMAC operations using CMKs, as I described in Section 4.2. Since HMAC is proven to be a PRF under the assumption that the underlying compression function is a PRF [3, 5], I can use the CMK as the seed for an HMAC-based PRF when all parties have access to TPMs. While there are some concerns regarding whether HMAC is a PRF when using hash functions with recently discovered
weaknesses, such as MD5 and SHA1 [48], I assume that an appropriate underlying hash function is used.

4.3.1. Establish a Shared Secret with CMKs

In this section, I describe how standard TPM operations on CMKs can be used to establish a secret that is shared between the TPMs of a fixed set of parties with known public keys for non-migratable keypairs. As described earlier, when a CMK is created it is bound to a list of public keys of migration authorities (MAs), and any migration of this CMK must be coordinated by one of these migration authorities. In a typical TPM application, the MA would be a public entity, trusted to migrate keys in accordance with a published migration policy. In this case, I avoid having an independent MA by migrating keys “under the authority” of a non-migratable storage key on a destination platform — since this isn’t an actual MA key (a key of type TPM_KEY_MIGRATE), it cannot be further migrated, so is effectively contained within that particular destination platform.

In the following, denote parties as $A_1, A_2, \ldots, A_n$. Each party $A_i$ has a TPM $T_i$, a properly certified identity key $I_i$, and a non-migratable storage key $P_i$ which will be the parent of the shared secret in $A_i$’s protected storage hierarchy. The shared secret will be generated internally to $T_1$, and then transferred (migrated) securely to TPMs $T_2, T_3, \ldots, T_n$. The shared secret establishment process consists of three phases, destination certification, secret creation/migration, and destination secret installation, which I describe below by outlining the various TPM commands used in each phase.

**Destination Certification:** For each $i \in \{2, \ldots, n\}$, $A_i$ uses the TPM command `TPM_CertifyKey` to create a certification for key $P_i$ using identity key $I_i$. Note that this can be done in advance — any time after $P_i$ and $I_i$ have been created.

**Secret Creation/Migration:** $A_1$ collects all of the public keys corresponding to $P_2, \ldots, P_n$, along with their certifications and corresponding identity keys, and verifies that these are all certified as non-migratable storage keys using an identity key that is in turn certified by a trusted PrivacyCA. This list of public keys is then
used as the list of authorized migration authorities when the CMK $K$ is created using `TPM_CMK_CreateKey`. $A_1$ uses `TPM_CertifyKey2` to create a certification for key $K$ using identity key $I_1$. To transfer $K$, $A_1$ uses `TPM_CMK_CreateBlob` for each destination key $P_2, \cdots, P_n$ to create migration blobs (re-encrypted private keys\(^1\)) for each destination, and transmits the appropriate blob along with a copy of $K$’s certification and the list of certified parent keys $P_2, \cdots, P_n$ to each of $A_2, \cdots, A_n$.

**Destination Secret Installation:** When $A_i$ (for $i \in \{2, \cdots, n\}$) receives the migration blob and $K$’s certification, it uses `TPM_CMK_ConvertMigration` which installs $K$ under parent key $P_i$ in $A_i$’s storage hierarchy. Next, $A_i$ verifies that key $K$ is a CMK certified by an identity key which is in turn certified by a trusted PrivacyCA, and that its migration is restricted to keys $P_2, \cdots, P_n$, each of which is also verified as a non-migratable storage key certified by an identity key that is certified by a trusted PrivacyCA.

Theorem 4.1 If $T_1, \cdots, T_n$ are properly functioning, non-compromised TPMs, and if the PrivacyCAs certified only legitimate TPM-bound identity keys, then at the end of this protocol each TPM has a copy of $K$ which is internally usable and not available to other TPMs or outside of the TPM-protected environment.

**Proof:** The most important property of non-migratable TPM keys is that the private key can only be decrypted through a `TPM_LoadKey` command with another non-migratable key, and once decrypted and loaded there is no TPM command which exports the key in any form. Therefore, by induction from the SRK, the private portion of the non-migratable keys $P_1, \cdots, P_n$ can only exist in usable, unencrypted form inside the corresponding protected TPM.

---

\(^1\)Note that actual migration is slightly more complex, with each migration blob having an associated “random part” which is useful in certain scenarios — in this situation, I simply treat the migration blob and random part as a single unit of data, and transmit them together.
When CMK $K$ is generated, it is exported from the TPM encrypted by the non-migratable key $P_1$, and the TPM enforces the restriction that the parent key is non-migratable. Unlike a non-migratable key, there are actually two commands that can apply the parent key to decrypt a CMK: in addition to `TPM_LoadKey` there is also `TPM_CMK_CreateBlob`, but an exhaustive search through the TPM commands shows that these two commands are the *only* ones that can decrypt the private portion of the CMK $K$. Exactly like a non-migratable key, there is no way to export the private part after a `TPM_LoadKey`, so I concentrate on `TPM_CMK_CreateBlob`. This command *does* in fact export the private part of the key, but only after being encrypted by one of the keys that was approved when $K$ was created. Since $A_1$ ensures that all of these keys are in fact non-migratable storage (parent) keys, once loaded into the destination systems the same restrictions apply — they can only be exported using one of these pre-approved non-migratable storage keys.

Each party has assurance that the original key generation was done properly due to the certification provided by $A_1$, and that the TPMs on the destination systems are similarly restricted in their usage based on the certification of their parent keys. Therefore, $K$ can only be decrypted and put into usable form inside one of the TPMs $T_1, \cdots, T_n$. 

4.3.2. A TPM-Oracle

In the previous section, I described how parties equipped with TPMs can establish a shared secret in such a way that all parties have assurance that the secret is only available to a fixed list of parties and only within a TPM-protected environment. In this section, I will describe how to use this shared secret within a TPM to create a "TPM-Oracle" — a computation done inside the TPM that is polynomial-time indistinguishable from a random oracle. The desired random-oracle query is a call to a function $H: \{0,1\}^{a(k)} \to \{0,1\}^{b(k)}$, where $k$ is a security parameter, and $a(k)$ and $b(k)$ are polynomially-bounded functions — note that some random-oracle-designed protocols require multiple random oracles with different domain and range sizes, and in this work, these are treated as independent oracles with separate and independent shared secrets, but each with its own well-defined domain and range size. The TPM-Oracle is denoted $\mathcal{TO}: \{0,1\}^{a(k)} \to \{0,1\}^{b(k)}$. 

45
It is assumed that the HMAC engine uses a fixed Merkle-Damgård-style [26] hash function that has an underlying compression function that uses \(c(k)\)-bit blocks and produces \(d(k)\)-bit digests, defining a function \(HMAC : \{0,1\}^{c(k)} \times \{0,1\}^* \to \{0,1\}^{d(k)}\), and I use notation \(HMAC(s,m)\) to denote evaluating HMAC using secret \(s\) and message \(m\). Recall that the shared secret, established using CMK operations, is \(K \in \{0,1\}^{c(k)}\), and I use a TPM-Oracle to answer queries of the form \(H(x)\) for some party \(A_i\). Finally, let \(bin(b, x)\) denote the \(b\)-bit binary representation of \(x\) (assuming \(x < 2^b\)). The TPM-Oracle is then defined as follows:

\[
\begin{align*}
\TO(x) \\
m &\leftarrow \left\lceil \frac{b(k)}{d(k)} \right\rceil \\
b &\leftarrow \lceil \log_2(m) \rceil \\
A &\leftarrow HMAC(K, bin(b,0)||x)||HMAC(K, bin(b,1)||x)||\cdots||HMAC(K, bin(b,m-1)||x)
\end{align*}
\]

\text{return} The first \(b(k)\) bits of \(A\)

\(\TO\) is a standard secure extension of a PRF, and is itself a secure PRF if the underlying building block is a secure PRF. Based on [3], in which Bellare shows that HMAC is a PRF if the underlying compression function is a PRF, I get the following lemma regarding the TPM-oracle, \(\TO\).

Lemma 4.1. \textit{If the compression function used by HMAC is a pseudorandom function, then }\(\TO\) \textit{is a pseudorandom function.}

### 4.3.3. Security of the TPM-Oracle

Now I consider the security of the complete system, including the secret establishment and use of the TPM-oracle, and provide a proof to show how this system provides security guarantees equivalent to a true random oracle with respect to a polynomial-time adversary.

I define a new construction called a hybrid PRF system, modeled after standard hybrid encryption. In hybrid encryption schemes, the message to be protected is encrypted using a symmetric cipher with a randomly generated secret key. A public key encryption scheme is then used to encrypt the secret key. The KEM/DEM model for hybrid encryption schemes splits the scheme into 2 parts - an asymmetric key encapsulation mechanism (KEM) and a
symmetric data encapsulation mechanism (DEM). Typically, a CCA-secure public key encryption scheme is used to encrypt the randomly generated key for the symmetric cipher, which in turn, is also CCA-secure. While each of these pieces is individually secure, the composition of these pieces is not necessarily secure. In fact, issues due to the insecurity of cryptosystems built by composition of secure subsystems led to the development of the universal composability framework [14]. Therefore, a key part of developing secure hybrid encryption schemes involves proving that the composition of the individually secure encryption schemes leads to a secure hybrid scheme.

In this construction of the hybrid PRF, first, a key encapsulation mechanism is used to encrypt a random key. Then the key is used as input to a pseudorandom function to produce pseudorandom output. I first present definitions of key encapsulation mechanism and a pseudorandom function as given by Cramer and Shoup [1, 25] and Bellare and others [6, 5] respectively. Then I present a description of the new hybrid PRF scheme and prove its security.

Security for a key encapsulation mechanism is defined in terms of a game, similar to the games for CCA security and CPA security described in Chapter 3.

Definition 4.2. Key encapsulation mechanism (KEM)

A key encapsulation mechanism (KEM) consists of the following polynomial time algorithms:

- Key generation algorithm: A probabilistic algorithm that generates public and private keys \((pk, sk)\). The public key defines the key space \(K_K\).

  \((pk, sk) \leftarrow KEM.Gen(1^\lambda)\)

- Encryption algorithm: A probabilistic algorithm that generates \(K \in K_K\) and the encryption \(\phi\), of \(K\).

  \((K, \phi) \leftarrow KEM.Enc_{pk}()\)

- Decryption algorithm: \(K \leftarrow KEM.DEC_{sk}(\phi)\), output \(\perp\) if \(K \notin K_K\).

  A deterministic algorithm that returns \(K\) as the decryption of \(\phi\).
Security for a key encapsulation mechanism is defined in terms of a game, similar to the games for CCA security and CPA security described in Chapter 3.

Definition 4.3. Game.KEM

Game.KEM is defined as follows: Let \( O \) be the decryption oracle, \( KEM.DeC_{sk}() \), and let \( A \) be a PPT adversary that plays the following game.

(i) \((pk, sk) \leftarrow KEM.Gen(1^\lambda)\),
\((K_0, \phi) \leftarrow KEM.Enc_{pk}(), K_1 \leftarrow \mathcal{K}, b \leftarrow \{0, 1\}\)
(ii) \( \hat{b} \leftarrow A^O(pk, \phi, K_b) \)

In Step 2, \( A \) is restricted not to ask \( \phi \) of KEM.Dec.

Define \( Adv_{kem, A} = |Pr[\hat{b} = b] - \frac{1}{2}| \) and \( Adv_{kem} = \max_A(Adv_{kem, A}) \) where the maximum is taken over all adversaries. A scheme is said to be secure against adaptive chosen ciphertext attacks if \( Adv_{kem} \) is negligible in \( \lambda \).

Security for a PRF is defined in terms of a game, described below. The “game” here is somewhat reminiscent of the Turing Test in AI: the adversary is interacting with one of two possible other parties, either truly random or pseudorandom, and must try to determine which one. The set of PRFs is much smaller than the set of all functions, so the challenge in this game is to determine when the oracle comes from this much smaller set.

Definition 4.4. Game.PRF

This security game is defined as follows: Let \( F: \mathcal{K} \times \mathcal{D} \rightarrow \mathcal{R} \) be a family of functions, and let \( A \) be a PPT algorithm that takes an oracle for a function \( g: \mathcal{D} \rightarrow \mathcal{R} \), (set of all functions from domain \( D \) to range \( R \)), and returns a bit. Consider the following two experiments:

Experiment \( Exp^{prf - 1}_F(A) \)
\begin{align*}
K & \overset{\$}{\leftarrow} \mathcal{K} \\
b & \overset{\$}{\leftarrow} A^K \mathcal{F}K \\
\text{Return } b
\end{align*}

Experiment \( Exp^{prf - 0}_F(A) \)
\begin{align*}
g & \overset{\$}{\leftarrow} \text{Func}(D, R) \\
b & \overset{\$}{\leftarrow} A^g \\
\text{Return } b
\end{align*}
The advantage of $A$ is defined as

$$Adv_{PRF,A} = Pr[Exp^{prf^{-1}}_{F}(A) = 1] - Pr[Exp^{prf^{-0}}_{g}(A) = 1]$$

$$Adv_{PRF} = \max_A(Adv_{PRF,A})$$

Definition 4.5. Hybrid PRF

A hybrid-PRF construction is defined as follows:

(i) Key generation algorithm: A probabilistic algorithm that generates public and private keys $(pk, sk)$. The public key defines the KEM key space $\mathcal{K}$. 

$$(pk, sk) \leftarrow HPRF.Gen(1^\lambda)$$

(ii) Encryption algorithm: A probabilistic algorithm, that given $pk$, generates and encrypts a random key $K \in \mathcal{K}$ with the corresponding secret key, $sk$, and returns $\phi$, the encrypted key.

$$(K, \phi) \leftarrow HPRF.Enc(pk)$$

(iii) Decryption algorithm: A deterministic algorithm that returns $K$ as the decryption of $\phi$.

$$K \leftarrow HPRF.Dec_{sk}(\phi), \text{ output } \bot \text{ if } K \notin \mathcal{K}$$

(iv) PRF algorithm: A deterministic algorithm, that given a key and a message, $m$, computes a hash of the message using the key as input to a keyed PRF.

$$r \leftarrow HPRF.F_{K}(m)$$

The security of an HPRF system is defined in terms of an attack game between a challenger and an adversary. Informally, adversary $A$ interacts with one of the oracles: one oracle is the HPRF oracle $F_{K}$ (where $K$ is the key obtained from the encryption oracle) and the other oracle is a random function. The goal of the adversary is to guess which oracle it is interacting with.

Definition 4.6. Game.HPRF

Game.HPRF is defined as follows: Let $\mathcal{O}$ be the HPRF oracle and let $A$ be a PPT adversary that plays the following game.
(i) \((pk, sk, \delta) \leftarrow \text{HPRF.Gen}(1^\lambda),\)
\((K, \phi) \leftarrow \text{HPRF.Enc}_{pk}()\)

(ii) Adversary receives \((pk, \phi)\)

(iii) \(\hat{\delta} \leftarrow \mathcal{O}(pk, \phi)\) where \(\mathcal{O}\) takes pairs \((\psi, m)\) and based on the value of \(\delta\), returns \(r \leftarrow \text{HPRF.F}_{\text{Dec}_{sk}}(\psi)(m)\) or \(r \leftarrow f(m)\) where \(f\) is a random function. The oracle flips a bit and fixes the value of \(\delta\), with \(\delta = 1\) denoting the PRF and \(\delta = 0\) denoting the random function. Thereafter, all queries are answered either with the HPRF or the random function.

Success for an adversary \(A\) in this game is defined as follows:

\[
\text{Adv}_{\text{HPRF}, A} = |Pr[A^{\text{HPRF}}(m) = 1] - Pr[A^f(m) = 1]| \\
\text{Adv}_{\text{HPRF}} = \max_A(\text{Adv}_{\text{HPRF}, A})
\]

Theorem 4.7. If a given PRF is secure against the standard PRF game and the KEM used to encrypt the CMK, \(K\), is CCA-secure, then the hybrid construction HPRF is secure against the standard PRF game. In particular, \(\text{Adv}_{\text{HPRF}} \leq 2\text{Adv}_{\text{KEM}} + \text{Adv}_{\text{PRF}}\).

Proof: Consider a probabilistic, polynomial-time adversary, \(A\), that attacks the Hybrid-PRF construction by playing the HPRF game. Define game \(G_1\) as the HPRF game as defined in Definition 4.6, i.e., \(A\) tries to distinguish between the output of \(\text{HPRF.F}_{K}(\cdot)\) and \(f(\cdot)\). Now let us define a modified game \(G_2\), where instead of using the key, \(K_0\), given by the HPRF.Enc algorithm, the decryption oracle (in part 3 of the HPRF game) uses a different, independent, random key, \(K_1\), to compute \(\text{HPRF.F}_{K}(\cdot)\). Let \(T_1\) be the event that \(A\) wins in game \(G_1\) and \(T_2\) be the event that \(A\) wins in game \(G_2\).

\(A\) can be used to construct an adversary \(A'\) that attacks the CCA-security of the key encapsulation scheme. At the outset, \(A'\) fixes \(\delta = 0\) or \(\delta = 1\), choosing the value of \(\delta\) randomly. If \(\delta = 1\), \(A'\) obtains \((K_b, \phi)\) from the KEM game for later use.

For each query \((\psi, m)\), received from \(A\), if \(\delta = 1\) and if \((\psi = \phi)\), return \(r \leftarrow \text{HPRF.F}_{K_b}(m)\); otherwise if \((\psi \neq \phi)\), decrypt \(\psi\) by asking the decryption oracle (KEM.Dec)
for the decryption of $\psi$ and return $r \leftarrow HPRF.F_{Dec_{sk}}(\psi)(m)$ to $A$. If $\delta = 0$ and ($\psi = \phi$), return $r \leftarrow f(m)$; otherwise, if ($\psi \neq \phi$), decrypt $\psi$ by asking the decryption oracle (KEM.Dec) for the decryption of $\psi$ and return $r \leftarrow f(m)$. $A'$ simulates a random function $f$ in the normal way; namely, it keeps a lookup table of inputs and their corresponding outputs. So if $A$ queries $f$ on a new input, $A'$ generates an output at random and places the value in the lookup table. If $A$ queries $f$ on the same input more than once, $A'$ returns the output from the lookup table.

After a certain number of queries, $A$ outputs $\hat{\delta} = (0 \text{ or } 1)$ as its final guess as to which oracle it is interacting with. If $A$ outputs $\hat{\delta} = \delta$, $A'$ outputs $\hat{b} = 0$, guessing that GAME.KEM provided the correct key. Otherwise, if $\hat{\delta} \neq \delta$, $A'$ outputs $\hat{b} = 1$. Notice that $A$’s view is identical to game $G_1$ when $b = 0$ and to game $G_2$ when $b = 1$. Therefore, the probability of success of $A$ in these events can be bounded as follows:

$$Pr[T_1] = Pr[\hat{\delta} = \delta | b = 0] \text{ and } Pr[T_2] = Pr[\hat{\delta} = \delta | b = 1]$$

Now, if in any case, $\hat{\delta} = \delta$, say that $\hat{b} = 0$, else, say that $\hat{b} = 1$. Therefore,

$$Pr[T_1] = Pr[\hat{\delta} = \delta | b = 0]$$

$$= Pr[\hat{b} = 0 | b = 0]$$

and

$$Pr[T_2] = Pr[\hat{\delta} = \delta | b = 1]$$

$$= Pr[\hat{b} = 0 | b = 1]$$

$$Pr[\hat{b} = 0 | b = 0] = \frac{Pr[\hat{b} = 0 and b = 0]}{\frac{1}{2}}$$

Therefore,

$$Pr[\hat{b} = 0 | b = 0] = 2 \cdot Pr[\hat{b} = 0 and b = 0]$$

(1)

Now, $Pr[\hat{b} = 0 | b = 1] = 1 - Pr[\hat{b} = 1 | b = 1]$

$$Pr[\hat{b} = 0 | b = 1] = 1 - 2 \cdot Pr[\hat{b} = 1 and b = 1]$$

(2)
Subtracting equation (2) from equation (1), gives,

\[ Pr[T_1] - Pr[T_2] = 2 (Pr[\hat{b} = 0 \text{ and } b = 0] + Pr[\hat{b} = 1 \text{ and } b = 1]) - 1 \]

since these are the only 2 possible outcomes in the KEM game.

\[ Pr[T_1] - Pr[T_2] = 2 (Pr[A' \text{ wins}] - \frac{1}{2}) \]
\[ = 2 \text{ Adv}_{KEM,A'} \]
\[ \leq 2 \text{ Adv}_{KEM} \]

(3)

Now A can be used to construct an adversary A'' playing the standard PRF game. A'' does the following: It calls the HPRF.Gen algorithm to obtain (sk, pk). It then obtains (K, φ) from the HPRF.Enc_{pk} algorithm and runs A, providing (pk, φ) to A. For each query, (ψ, m) received from A, if (ψ = φ), return \( r \leftarrow \text{HPRF.F}_{K}(m) \) else if (ψ ≠ φ), decrypt ψ by calling HPRF.Dec_{sk}(ψ) and return \( r \leftarrow \text{HPRF.F}_{\text{Dec}_{sk}(ψ)}(m) \). After A outputs its final guess, A'' returns that value as its own guess. Now observe that the view of A in this simulation is identical to it’s view in game \( G_2 \), so the probability that A succeeds in this simulation is exactly \( Pr[T_2] \). Since A'' is playing the PRF game, the success probability is bounded by the PRF-advantage.

\[ Pr[T_2] \leq \text{ Adv}_{PRF} \]

(4)

Combining equation (3) and equation (4), gives

\[ Pr[T_1] - \text{ Adv}_{PRF} \leq 2 \text{ Adv}_{KEM} \]
\[ Pr[T_1] \leq 2 \text{ Adv}_{KEM} + \text{ Adv}_{PRF} \]

Therefore, \( \text{ Adv}_{HPRF} \leq 2 \text{ Adv}_{KEM} + \text{ Adv}_{PRF} \)

In summary, from Theorem 4.3.1, Theorem 4.7 and Lemma 4.1, I obtain the following result,
Theorem 4.8. Assume that there are properly functioning, secure TPMs that use the above technique for creating a TPM Oracle. If the hash function used by HMAC has a pseudorandom compression function, and the CMK transfer scheme uses a CCA-secure public key cryptosystem, then any polynomial time algorithm secure in the random oracle model is secure in the TPM oracle model.
CHAPTER 5

GENERALIZED NON-INTERACTIVE OBLIVIOUS TRANSFER

In this chapter, I present the second of our main results; the construction of a generalized non-interactive oblivious transfer primitive using the monotonic counter functionality of trusted platform modules. Oblivious transfer is a fundamental cryptographic primitive used to exchange information securely in various cryptographic protocols. However, most flavors of OT protocols require interaction between the parties. I show how this interaction requirement can be removed in a generalized OT protocol by using the functionality of trusted platform modules.

After presenting background information on count-limited objects, I give a generalized definition of the problem and outline the security properties required for any secure generalized non-interactive oblivious transfer scheme. Then I show how to instantiate a non-interactive form of oblivious transfer when all parties have access to trusted platform modules and conclude with a security analysis of the TPM-based scheme.

5.1. Virtual Monotonic Counters

Sarmenta et al. [64] outline how to create a potentially unlimited number of virtual monotonic counters from a physical monotonic counter. They model a virtual monotonic counter as a mechanism that stores a value and provides 2 commands to access this value: Read command, that returns the current value of the counter and Increment command that increments the value of the counter and returns the updated value of the counter. Fundamentally, a virtual monotonic counter must be:

(i) Non-volatile: the value of the counter must not change unless incremented in response to a command.

(ii) Irreversible: it must be infeasible for any adversary to reset the counter to any previous value.
(iii) Tamper-evident: at the very least, it must be possible to detect any unauthorized change in the value of the counter. Ideally, a counter should be tamper-resistant.

(iv) Produce verifiable output: it must be possible for a user to verify the output of the counter, through an execution certificate, for example. The counter produces a verifiable output message in response to the Read or Increment commands.

(v) Unforgeable: The execution certificates produced by the counter must be unforgeable. The execution certificates are typically signed using AIKs and random nonces are used to prevent replay attacks.

It should be noted that the use of the AIKs for the signature is to ensure that the operation is completed on an actual TPM and not a software emulator. AIKs, as pseudonyms to the EK, are associated with real TPMs and have certificates signed by Privacy CAs.

5.1.1 System Model

This section outlines the main functions associated with virtual monotonic counters.

Sarmenta et al. propose two different interacting systems, namely, the host and the client, when modeling a virtual monotonic counter. The host contains the virtual monotonic counters and the client runs an application that requires use of the virtual monotonic counters. The host has a trusted platform module and the virtual monotonic counter is modeled as a software component that must support the following functions:

- **CreateNewCounter(nonce):** This command creates a new virtual monotonic counter and returns a create certificate, which contains the counterID and the given nonce.

- **ReadCounter(counterID,nonce):** This command returns a read certificate which contains the current value of the virtual monotonic counter specified by the counterID. Along with the current value, the read certificate contains the counterID and the nonce.

- **IncrementCounter(counterID,nonce):** This command increments the virtual counter, specified by the counterID and returns an increment certificate containing
the new value of the virtual counter together with the counterID and the given nonce.

• DestroyCounter(counterID, nonce): This command destroys the specified virtual counter, i.e., ensures that the same CounterID cannot be used again. This command also returns a destroy certificate containing the counterID and the given nonce.

5.2. Count-Limited Objects

Building from virtual counters, Sarmenta et al. have proposed count-limited objects, or clobs, as an interesting and important primitive. These are proposed objects that utilize the ability of a TPM to encrypt data or keys into “blobs” such that they can only be decrypted when the TPM is in a specified state, which in current TPMs is limited to conditions based on the PCRs. In Sarmenta’s construction, these encrypted blobs are then linked to a virtual monotonic counter which is used to track/limit the usage of the blob. Sarmenta et al. proposed a hash-tree based scheme that allows the TPM to efficiently keep track of a large number of virtual monotonic counters, thereby enabling various count-limited objects, each having its own dedicated virtual monotonic counter. Briefly, each virtual monotonic counter is a leaf node in a Merkle hash tree [54]. This blob contains the virtual counter’s ID, its current value and other meta-data. The intermediate nodes of the tree are hashes of the concatenation of the left and right children. The root hash in this tree is guaranteed to change (with overwhelming probability) if the value of any of the descendant nodes changes. Updates and creation of the hash tree are efficient; these operations take \(O(\log n)\) time for \(n\) objects. Leaves and intermediate nodes of the hash tree are stored by the host, and only the root of this tree is stored in the TPM’s NV storage. All legitimate read and update requests of the counter go through the TPM which checks the integrity of the root hash before signing a certificate with the value of the counter and the output of the command.

Sarmenta et al. propose a new command called **TPM_ExecuteHashTree** which takes as input, the handle of an AIK which can perform signatures, the mode of the counter (read, execute, etc.), the counter blob, a random nonce and internal hash tree nodes. The TPM
returns the current value of the counter, along with an “execution certificate” signed by
the AIK. Other TPM commands and structures also require modifications to include the
count-limit condition associated with the object. For example, TPM\_KEY structure has a
variable length field to hold PCR-related information if needed. This field can be used to
hold the count-limit condition or a separate field can be added to the TPM\_KEY structure to
hold the count-limit condition. Additionally, the TPM\_Sign and TPM\_Unbind commands need
to be modified to check for the count-limit condition before executing the command. The
TPM\_LoadKey command needs to be modified to include the count-limit condition as part of
the information the TPM needs to check. The TPM\_Sign and TPM\_UnBind commands perform
the actual check before the command is executed.

5.2.1. Non-Interactive Oblivious Transfer Using Count-limited Objects

This section outlines new ideas on how count-limited objects can be used to implement a
non-interactive version of standard oblivious transfer. In an oblivious transfer protocol, two
parties can exchange information without learning anything about each other’s inputs.

5.2.1.1. 1-out-of-2 Oblivious Transfer

In the standard 1-out-of-2 OT, when Alice transmits one of $s_0$ or $s_1$ to Bob in an oblivious
manner, interaction between Alice and Bob is typically required. In a common solution, Bob
needs to supply Alice with keys to encrypt her strings and this is done only after he decides
which value he requires. Therefore, Alice cannot encrypt the strings unless Bob sends her
the keys, which he cannot do until he decides which string he wants. Using count-limited
objects, Bob can compute keys before making a decision of which $s_c$ he wants, and his later
use of that key is restricted by the count-limited property.

Note that Bellare and Micali [7] have previously introduced a related but different notion
of non-interactive oblivious transfer, but in their case Bob receives a randomly selected $s_c$
(he doesn’t get to choose which one). This is useful in some applications, but not in the
secure function evaluation problems examined in this dissertation.
5.2.1.2. Non-interactive OT using Count-limited Decryption Key

Alice has a TPM and 2 values \( s_0 \) and \( s_1 \). Bob also has a TPM and generates a one-time use non-migratable key pair, \( K_p, K_s \) and publishes the public key \( K_p \), which is certified using an AIK \( I_b \), which in turn is certified by a Privacy CA. This one-time use key pair is tied to a virtual monotonic counter which limits the private key \( K_s \) to being used no more than once. Alice encrypts both values \( s_0 \) and \( s_1 \) using \( K_p \), having verified that the key is indeed Bob’s via the accompanying certificate. At some later time, after receiving the ciphertexts, Bob can decide which value he wants. Then Bob decrypts only that value using \( K_s \), being restricted to do so by the virtual monotonic counter, which is incremented as soon as one of the values is decrypted.

This clearly solves the non-interactive OT problem, but in applications which use multiple oblivious transfers, a separate key must be generated for each OT, which is very inefficient. In the following section, I will show how a single clob can control multiple oblivious transfers.

5.3. Problem Definition

The 1-out-of-2 OT concept is generalized to a form where multiple independent oblivious transfers (of the general \( k \)-out-of-\( n \) type) are defined as part of a single operation. In many applications (such as secure function evaluation) multiple instances of OT must be run, so by defining this as a single operation I have the flexibility of creating solutions which can exploit improvements possible by aggregating multiple requests. I call this combined operation “generalized non-interactive oblivious transfer (GNIOT),” which I formally define in the following section.

I first define Generalized Oblivious Transfer (GOT), and I will subsequently define phases which will force this to be non-interactive, producing GNIOT.

Definition 5.1 (GOT). Define \( \lambda \) as the security parameter and \( l_d \) as the length of the data items being sent by Alice to Bob. Assume that Alice has \( n \) data sets \( S_1, S_2, \cdots, S_n \), with values \( x_{i,j} \in \{0,1\}^{l_d} \) for \( i \in \{1,2,\cdots,n\} \) and \( j \in \{1,2,\cdots,m_i\} \), and parameters \( k_1, k_2, \ldots, k_n, \)
where $1 \leq k_i \leq m_i$. At the end of the GOT execution, Bob will have either no result (represented by $\perp$) or a set of exactly $k_i$ values of his choice from each set $S_i$, for $i \in \{1, 2, \cdots, n\}$.

To refer to sets of indices into the data set, define index set $\mathcal{I}$ to be a set of indices $(i, j)$, and define $\mathcal{I}(i) = \{ j \mid (i, j) \in \mathcal{I} \}$. With respect to the parameters provided in an instance of GOT, I say that index set $\mathcal{I}$ is well-formed if $|\mathcal{I}(i)| = k_i$ for all $i \in \{1, \ldots, n\}$.

I define GNIOT as a set of operations which perform GOT, but accomplish this task without requiring any interaction between the receiver and another party after the receiver decides which values he wants. For maximum flexibility, allowing either batched or individual decryptions, I define the decryption operation as a stateful process which is called repeatedly — only at the very end are I required to have the actual plaintext values.

Definition 5.2 (GNIOT). Generalized non-interactive oblivious transfer consists of the following phases, which provide a solution to the GOT problem.

**Setup phase:** This phase involves key generation. Given security parameter $\lambda$, the key generation algorithm returns

$$(K_p, K_s) \leftarrow Setup(1^\lambda)$$

where $K_p$ is the public key information, and $K_s$ is the secret key information.

**Transmit phase:** This phase transforms the set of values $x_{i,j} \in \{0, 1\}^{l_i}$ for $i \in \{1, 2, \cdots, n\}$ and $j \in \{1, 2, \cdots, m_i\}$ into a data blob which can be transmitted to the receiver. Specifically,

$$C \leftarrow Transmit_{K_p}\left(\begin{array}{c}
\langle k_1, x_{1,1}, x_{1,2}, \cdots, x_{1,m_1} \rangle \\
\langle k_2, x_{2,1}, x_{2,2}, \cdots, x_{2,m_2} \rangle \\
\vdots \\
\langle k_n, x_{n,1}, x_{n,2}, \cdots, x_{n,m_n} \rangle
\end{array}\right).$$

**Decrypt phase:** In this phase, the receiver gives the indices $(i, j)$ of the $x_{i,j}$ values that he wishes to receive. The state-based process begins by calculating the initial state $S_0 \leftarrow InitialState(C)$, and then evolving the state and providing answers to
queries as

$$(t_k, S_k) \leftarrow \text{Decrypt}_{K_k}(S_{k-1}, C, i_k, j_k),$$

for $k = 1, 2, \ldots, q$ for some number of queries $q$. I require that index information be embedded in $t_k$ such that there is a function “$\text{ind}$” that extracts this information as

$$(i_k, j_k) \leftarrow \text{ind}(t_k).$$

**PostProcess phase:** This phase takes the results of the $\text{Decrypt}$ calls and either fails (giving $\perp$ as the result) or produces $q$ plaintext values as

$$\langle v_1, v_2, \ldots, v_q \rangle \leftarrow \text{PostProcess}(t_1, t_2, \cdots, t_q)$$

5.4. Desired Security Properties

A secure GNIOT scheme must satisfy the following properties:

**Correctness:** If the Alice and Bob follow the above steps in the prescribed way, and the index set defined by $\mathcal{I} = \{(i, j) \mid \text{ind}(t_k) \text{ for } 1 \leq k \leq q\}$ is well-formed, then the values produced by $\text{PostProcess}$ are exactly the requested plaintext values such that $v_k = x_{\text{ind}(t_k)}$ for $k = 1, \ldots, q$.

**Sender’s privacy:** Bob should not be able to obtain any information about the remaining $m_i - k_i$ elements in each set $S_i$.

**Receiver’s privacy:** Alice should not be able to determine which $k_i$ values Bob received from each set.

In a non-interactive process, where there is no communication with the sender in the $\text{Decrypt}$ or $\text{PostProcess}$ phases, the Receiver’s privacy property is trivially met. For the Sender’s privacy, I define a game played between a probabilistic, polynomial time (PPT) adversary $\mathcal{A}$ and an oracle, where the oracle runs the parts of the protocol associated with the Sender.
(i) The adversary supplies a plaintext input to the oracle where each input has two different possibilities:

\[
\langle (x_{1,1}^0, x_{1,1}^1), (x_{1,2}^0, x_{1,2}^1) \cdots, (x_{1,m_1}^0, x_{1,m_1}^1) \rangle \\
\langle (x_{2,1}^0, x_{2,1}^1), (x_{2,2}^0, x_{2,2}^1) \cdots, (x_{2,m_2}^0, x_{2,m_2}^1) \rangle \\
\vdots \\
\langle (x_{n,1}^0, x_{n,1}^1), (x_{n,2}^0, x_{n,2}^1) \cdots, (x_{n,m_n}^0, x_{n,m_n}^1) \rangle
\]

(ii) The oracle generates an independent random bit \( r_{i,j} \) for each pair. The oracle then creates a single GNIOT input by using inputs \( x_{i,j}^{r_{i,j}} \) for \( i = 1, 2, \cdots, n \) and \( j = 1, 2, \cdots, m_i \) and calls the Transmit function. The resulting \( C \) is sent back to the adversary.

(iii) (a) \( \mathcal{A} \) makes a series of calls to Decrypt, receiving values \( t_1, t_2, \ldots, t_q \).

(b) The adversary is free to perform any computation using the information it obtained, possibly calling the PostProcess function of the GNIOT scheme.

(c) The adversary finally outputs a guess \( g \) and an index \((a, b)\).

The adversary wins this game if \( g = r_{a,b} \), but I are only interested in when the adversary learns a value that it shouldn’t. Therefore, if \( \mathcal{I} \) is the index set for the queries made in Step 3a, I define the “advantage” for adversary \( \mathcal{A} \) as

\[
Adv_{GNIOT, \mathcal{A}} = \left| \Pr[ g = r_{a,b} | (a, b) \notin \mathcal{I} \text{ or } \mathcal{I} \text{ not well-formed}] - \frac{1}{2} \right|.
\]

The security of a GNIOT scheme is defined as the advantage of the best adversary,

\[
Adv_{GNIOT} = \max_{\mathcal{A}}(Adv_{GNIOT,\mathcal{A}}),
\]

and the scheme satisfies the Sender Privacy property if \( Adv_{GNIOT} \) is negligible.

5.5. TPM-based Solution

Our TPM-based solution makes use of both a standard symmetric cipher and a public key cryptosystem in which use of the private key is count-limited by the TPM. Based on previously defined parameters \( \lambda \) and \( l_d \) I define several additional parameters for our solution, as given below.
• \(l_b\) (Encrypted data length): Length of the data after encryption with the symmetric cipher.

• \(l_s\) (Symmetric key length): Length of the key for the symmetric cipher. Must be polynomial in \(\lambda\).

• \(l_p\) (Public key payload size): Length of data that can be encrypted with the public key scheme. Must be polynomial in \(\lambda\), and must satisfy \(l_p \geq l_b + l_s\).

The basic idea behind our GNIOT scheme is to doubly encrypt the values \(x_{i,j}\) with the symmetric scheme and the public key scheme so that the count-limit restriction ensures that not too many values are decrypted, and a secret sharing scheme is used to make sure that at least \(k_i\) are decrypted from each set to allow recovery of the symmetric key for the final plaintext decryption. As a result, exactly \(k_i\) values from each set must be decrypted. Our formal definition follows the phases defined in Section 5.3.

**Setup phase:** Bob creates an \(N\)-time use count limited key pair \((K_p, K_s)\), where \(N = (k_1 + k_2 + \cdots + k_n)\). For further assurance in subsequent key transfer, Bob can certify \(K_p\) using an Attestation Identity Key (AIK).

**Transmit phase:** The plaintext values \(x_{i,j}\) provided to the Transmit function will be first protected using a symmetric cipher (such as AES), using a session key \(R\) that is generated by selecting \(n\) partial keys \(R_i \in_R \{0,1\}^{l_s}\) and letting \(R = R_1 \oplus R_2 \oplus \cdots \oplus R_n\). Next, for each \(i\) I compute \(m_i\) shares of each \(R_i\) using a threshold-\(k_i\) secret sharing scheme, such as the polynomial interpolation based scheme due to Shamir [65], and I denote the shares of \(R_i\) by \(f_i(j)\), for \(j = 1, \ldots, m_i\). By using threshold \(k_i\) in the secret sharing scheme, I will be able to compute \(R_i\) given any \(k_i\) of the \(f_i(j)\) values. Using \(\mathcal{PKE}_{K_p}\) and \(\mathcal{SKE}_{R}\) to denote the public key and symmetric encryption schemes with keys \(K_p\) and \(R\), respectively, I doubly encrypt each \(x_{i,j}\) along with a share of \(R_i\) to give

\[
C_{i,j} = \mathcal{PKE}_{K_p}((\mathcal{SKE}_R(x_{i,j}), f_i(j))).
\]

The collection of ciphertexts \(C_{i,j}\), for \(i \in \{1,2,\ldots,n\}\) and \(j \in \{1,2,\ldots,m_i\}\), is then the output of the Transmit function.
Decrypt phase: The only state used in our implementation is in the virtual monotonic counter maintained by the TPM, so all state operations are implicit in the use of count-limited keys. \( \text{Decrypt}_{K_s}(S,C,i_k,j_k) \) then just uses \( K_s \) to decrypt \( C_{i_k,j_k} \), and bundles the resulting values with the index \((i_k,j_k)\) to give

\[
t_k = \langle i_k, j_k, SKE_R(x_{i_k,j_k}), f_{i_k}(j_k) \rangle.
\]

PostProcess phase: For the final PostProcess stage, let \( I = \{(i_k,j_k)|1 \leq k \leq q\} \) be the index set of requests made in the Decrypt phase. Then Bob extracts the shares \( f_{i_k}(j_k) \) from each \( t_k \), and for each \( i \in \{1,\ldots,n\} \) combines the shares corresponding to \( I(i) \) to recover each \( R_i \). These values are then exclusive-ORed together to recover the symmetric key \( R \), which is used to decrypt the plaintexts \( x_{i_k,j_k} \).

5.6. Security Analysis

In this section, I formally prove that our scheme has the required security properties. The proof uses definitions and security games for public key and symmetric key encryption schemes, which were defined in Chapter 3.

Theorem 5.3. If \( PKE \) is an IND-CCA2 secure public key scheme and \( SKE \) is a IND-CCA2 secure symmetric cipher, then the GNIOT game can be won by a probabilistic, polynomial time adversary \( A \) if and only if \( I \) is a well-formed index set and \((a,b) \in I\).

Proof: Case 0: The GNIOT game against the TPM-based scheme can be won by a PPT adversary \( A \) if \( I \) is a well-formed index set and \((a,b) \in I\).

This is easy to see as follows: If \( I \) is a well-formed index set, \( A \) can obtain exactly \( k_i \) values from set \( S_i \), by calling the decrypt function, which returns \( t_{i,j} \) values as the decryption of the corresponding \( C_{i,j} \) values in each set. If \((a,b) \in I\), then \( A \) can call the PostProcess function to correctly obtain corresponding value \( x_{a,b} \).

Case 1: \((a,b) \notin I\), where \( I \) is a well-formed index set.
Let \( A \) be a PPT adversary that wins the GNIOT game with non-negligible probability, i.e. \( A \) distinguishes between the encryptions of \( x^0_{i,j} \) and \( x^1_{i,j} \) with non-negligible probability. I
can use \( A \) to construct a PPT adversary \( A' \) that attacks the CCA security of the PKE. When \( A' \) is given a query \( x_{i,j}^{r} \), from \( A \), it calls GNIOT.Transmit and returns the output of GNIOT.Transmit, \( C_{i,j}^{r} \), to \( A \). If \( A \) requests the decryption of any value \( C_{i,j}^{r} \), \( A' \) calls \( t_{i,j}^{r} \leftarrow GNIOT.Decrypt(C_{i,j}) \) and returns \( t_{i,j} \) to \( A \). Optionally, if \( A \) requests decryption of \( t_{i,j}^{r} \), \( A' \) calls the PostProcess oracle, \( A' \) returns the result of GNIOT.PostProcess, \( x_{i,j}^{r} \) to \( A \).

After a number of queries, when \( A \) outputs its guess, \( g = (0, 1) \) and the index \((a, b)\) \( A' \) outputs that guess as its own. Since the GNIOT game is perfectly simulated in this construction, if \( A \) succeeds with non-negligible probability, \( A' \) also succeeds with non-negligible probability.

**Case 2**: \((a, b) \in \mathcal{I} \) but \( \mathcal{I} \) is not a well-formed index set.

Let \( A \) be a probabilistic, polynomial time (PPT) adversary that plays the GNIOT game and attacks the TPM-based scheme. The intuition behind this case is that in order for \( A \) to win the GNIOT game in this case, it must either break the SKE scheme to decrypt \( \text{SKE}_R(x_{a,b}) \) without knowing \( R \), or must break the PKE scheme to gain additional information about \( R \).

Define game \( G_1 \) as the GNIOT game as defined in definition 5.4, i.e., \( A \) tries to distinguish between the encryptions of \( x_{i,j}^0 \) and \( x_{i,j}^1 \) for some \((i, j)\). Now let us define a modified game \( G_2 \), where instead of using the real symmetric key \( R \), the transmit oracle (in part 3 of the GNIOT game) uses a different, independent, random key, \( \tilde{R} \), to encrypt the values in each set. Let \( T_1 \) be the event that \( A \) wins in game \( G_1 \) and \( T_2 \) be the event that \( A \) wins in game \( G_2 \).

\( A \) can be used to construct a PPT adversary \( A' \) that attacks the CCA security of the PKE scheme. In particular, since \( \mathcal{I} \) is not well-formed, there must be some set \( i \) such that \( |\mathcal{I}(i)| < k_i \), so \( R_i \) and hence \( R \) is independent of the decrypted shares of \( R_i \). Therefore, unless \( A \) can get some information from the non-decrypted \( C_{i,j} \) values it gets no information about \( R \) and so must break the SKE scheme.

\( A' \) gets public key \( K_p \) from the PKE game. \( A' \) picks random key \( R \) and computes all \( R_i \) values and shares \( f_i(j) \). Next, \( A' \) picks a random index \((a', b')\), and for all \((i, j) \neq (a, b)\)
computes $C_{i,j}$ for random selection $r_{i,j}$ exactly as our GNIOT algorithm. For index $(a', b')$, $A'$ substitutes a random share $\tilde{f}_{a'}(b')$ in place of the real $f_{a'}(b')$ for one alternative:

$$P^0_{a',b'} = \langle SKER(x^0_{a',b'}), f_{a'}(b') \rangle \quad P^1_{a',b'} = \langle SKER(x^1_{a',b'}), \tilde{f}_{a'}(b') \rangle .$$

These two plaintexts are then passed along to the PKE game as the challenge plaintexts, and the PKE oracle sends a ciphertext $C_{a',b'}$ back, which is the encryption of one of these. Note that if $P^0_{a',b'}$ is chosen, the key used is the correct key constructed from the share $f_{a'}(b')$, so the GNIOT game (game $G_1$) is perfectly simulated. On the other hand, if $P^1_{a',b'}$ is chosen then the fake share $\tilde{f}_{a'}(b')$ makes the symmetric key $R$ independent of the key reconstructed from the shares, and so $G_2$ is perfectly simulated. Let $\delta \in \{0, 1\}$ represent the choice made by the PKE game.

When $A$ produces an index $(a, b)$ and guess $g$, if $(a, b) = (a', b')$ output “fail” and quit. When $(a, b) \neq (a', b')$, if $g = r_{a,b}$ (i.e., the guess is correct), output $\hat{\delta} = 0$ as our guess in the PKE game; otherwise output $\hat{\delta} = 1$. Analyzing the probability that output $\hat{\delta}$ is correct,

$$Pr[\hat{\delta} = \delta] = Pr[g = r_{a,b}|\delta = 0]Pr[\delta = 0] + (1 - Pr[g = r_{a,b}|\delta = 1]) Pr[\delta = 1]$$

$$= \frac{1}{2} Pr[T_1] + \frac{1}{2} (1 - Pr[T_2])$$

$$= \frac{1}{2} (Pr[T_1] - Pr[T_2]) + \frac{1}{2} .$$

Since $\hat{\delta} = \delta$ means $A'$ wins the PKE game,

$$Pr[T_1] - Pr[T_2] = 2 \left( Pr[\hat{\delta} = \delta] - \frac{1}{2} \right) \leq 2 Adv_{PKE} .$$

(5)

Next $A$ is used to construct an adversary $A''$ playing the standard SKE game. $A''$ selects $R_i$ values and computes $R$ and the shares $f_i(j)$ as in the algorithm, and also generates a public keypair $(K_p, K_s)$. $A''$ initiates the SKE game, which causes the SKE oracle to select a symmetric key that is random and independent of $R$, and which will be used for all symmetric encryptions that are provided to $A$ — this means that $A$ is actually playing game $G_2$. Next,
\( \mathcal{A}'' \) selects a random index \((a', b')\), picks a random bit \(r_{i,j} \) for each \((i, j) \neq (a', b')\), and uses the SKE encryption oracle to compute plaintexts \( P_{i,j} = \langle \text{SKE.Encrypt}(x_{i,j}^{r_{i,j}}), f_i(j) \rangle \). \( \mathcal{A}'' \) then passes both \( x_{a', b'}^0 \) and \( x_{a', b'}^1 \) as the challenge plaintexts to the SKE game, and receives a ciphertext \( c \) back, which it uses to compute \( P_{a', b'} = \langle c, f_{a'}(b') \rangle \). Now \( \mathcal{A}'' \) uses it’s public key \( K_p \) to compute \( C_{i,j} = \text{PKE}_{K_p}(P_{i,j}) \) for all \((i, j)\).

Finally, \( \mathcal{A} \) will produce index \((a, b)\) and a guess bit \( g \). If \((a, b) \neq (a', b')\) output “fail” and quit; otherwise, pass along the guess \( g \) as \( \mathcal{A}'' \)’s guess in the SKE game. \( \mathcal{A}'' \) wins exactly when it’s index \((a, b)\) is correct and when \( \mathcal{A} \) wins (in game \( G_2 \)), so

\[
\text{Adv}_{\text{SKG}, \mathcal{A}''} = \frac{1}{N} \text{Pr}[T_2].
\]

This means that \( \text{Pr}[T_2] \leq N \cdot \text{Adv}_{\text{SKG}} \). Combining with equation (4), gives

\[
\begin{align*}
\text{Pr}[T_1] - N \cdot \text{Adv}_{\text{SKG}} & \leq 2 \text{Adv}_{\text{PKE}} \\
\text{Pr}[T_1] & \leq 2 \text{Adv}_{\text{PKE}} + N \cdot \text{Adv}_{\text{SKG}}
\end{align*}
\]

Therefore, \( \text{Adv}_{\text{GNIOT}} \leq 2 \text{Adv}_{\text{PKE}} + N \cdot \text{Adv}_{\text{SKG}} \), and since PKE and SKE allow only negligible advantage, \( \text{Adv}_{\text{GNIOT}} \) is also negligible.

\[ \blacksquare \]
CHAPTER 6

NON-INTERACTIVE SECURE AGENT PROTOCOL

In this chapter, I show how the generalized non-interactive oblivious transfer primitive developed in the previous chapter can be applied to the mobile agent paradigm. I outline a secure agent protocol called the GTX protocol and outline how the GTX protocol is more efficient and requires less communication and synchronization between systems when compared to existing ACCK [2] and TX [75] security protocols, while providing the same security guarantees. I then present an outline of our SAgent [36][37] security framework for mobile agents. SAgent was developed as a generic security framework for mobile agents where new security protocols can be easily plugged into the framework, so I show how the GTX protocol can be integrated into SAgent.

6.1. Mobile Agent Paradigm

In the mobile agent paradigm, an agent owner, also called the originator, creates mobile agents to perform some task on her behalf. After creating the agents for some specific purpose, the originator sends them out to visit various remote hosts, where the agents autonomously perform computations on behalf of the originator. When the agents return home, the originator retrieves the results of these computations from the agents. The utility of this paradigm is based on the ability of the originator to go offline after sending the agents out, and, ideally, no further interaction between the agent and the originator or the host should be required. However, various agent security protocols require some input from the originator once the agent reaches a host; for example, due to security reasons, the originator and the host should not obtain information about each other’s inputs for the computation. Secure function evaluation and oblivious transfer (presented in Section 3.3 and 5.2.1 respectively) are primitives that are utilized in various agent security protocols, and techniques used in existing protocols remove the need for direct interaction with the
originator, including using trusted third parties and using threshold cryptography as stand-ins for the originator. However, while direct interaction with the originator is avoided, interaction with other entities is still an integral component of these protocols. In the following sections, I outline a new non-interactive agent protocol called the GTX protocol which uses the GNIOT primitive to provide the same security guarantees as the existing ACCK [2] and the TX [75] protocols.

6.1.1. Application of SFE to Mobile Agents

The SFE primitive is used in various mobile agent protocols, including the ACCK protocol, which provides security utilizing a trusted third party (TTP), and the TX protocol where use of multiple agents guarantees security. The agent-host interaction can be modeled as a multi-party computation. At each host, the agent performs some computation and typically, this computation must be protected. The agent-host interaction at each host is an application of SFE (shown in figure 6.1) as follows: The agent provides its input to the host and receives the host’s input in return. The secure computation takes place in an oblivious manner due to the use of encrypted circuits on the agent input, the host input and the agent state. As a result of this evaluation, the agent state is updated and any output for the given host is returned to the host. Note that this figure refers to parts of the SAgent framework, which is described in later sections.

Figure 6.1. SFE Applied to Agents

A core component of this secure interaction is how the host gets the signals corresponding to its inputs from the originator, who is offline, making a direct bit-by-bit oblivious transfer impossible.
I break secure agent computation into 3 basic phases. I describe the phases below and outline the differences between the ACCK and the TX protocol.

(i) Initialization: The originator creates an encrypted circuit for each sensitive computation to be carried out at a host — the square box in Figure 6.1. As outlined in section 3.3, encrypted circuits are special boolean circuits where the signals on the wires are random strings instead of 0 or 1. Since the encrypted circuit can be evaluated with encoded signals, the agent state and inputs are encoded and incorporated into the agent. The method used to encrypt these signals is described in the following sections.

(ii) Evaluation: Before computing the sensitive agent function, which is in the form of an encrypted circuit, the host obtains the signals for its input according to the techniques of either the ACCK or TX protocol. These protocols ensure that only a single input can be retrieved, so malicious hosts cannot reset the agent state and try repeated execution with different inputs. If a malicious host can repeatedly evaluate the circuit with multiple inputs, it can potentially extract information related to the agent’s state (which is the private information being protected).

With the right signals for all three inputs, the encrypted circuit can be evaluated resulting in a new agent state and an output to the host, both in the form of signals. Because of its oblivious nature (due to fact that the computation is being performed on encrypted signals), the computation is tamper resistant. The originator can include in the agent the semantics of the strings that encodes the host output, so that the host is able to decrypt the value of its output.

(iii) Finalization: When the agent returns to the originator, its state will be uncovered by the originator, who holds the semantics for the corresponding signals.

Now, I outline how the signals are encoded in the ACCK and the TX protocols.

**ACCK protocol:** The ACCK protocol takes a straight-forward approach to signal encryption and decryption based on the assumption that a trusted third party (TTP) is available. The TTP possesses a keypair for some public key algorithm,
and makes its public key known to all participants (if necessary, some form of PKI/certificates can be used to ensure authenticity of the keys). The originator then bundles each signal with identifying information and encrypts each bundle using the TTP’s public key. At a remote host, encrypted bundles corresponding to the host or agent input are retrieved from the agent and sent to the TTP. The TTP decrypts the signals while ensuring (using the bundled identifying information) that only a single input is decrypted for a particular host before returning the decrypted values to the host/agent.

**TX protocol:** To eliminate the TTP, the TX protocol makes use of a threshold cryptosystem [68] to encrypt the signals for the host input and the public agent input. The TX protocol requires multiple agents, each carrying a share of the decryption key, visiting mutually disjoint sets of hosts. The required number of agents (more than the threshold) work together to serve the decryption requests from a host or agent in Step 2 above. Tate and Xu [75] introduced a new cryptographic primitive called oblivious threshold decryption (OTD) which combines threshold decryption with oblivious transfer [7] in such a way that the host learns the appropriate decrypted signal value, and the agents which help in the decryption do not learn which input the host has retrieved.

The GNIOT primitive can be applied to provide security in agent computations. The GTX protocol has the same phases described for other secure agent protocols and I outline these in section 6.4.

### 6.2. SAgent Security Framework

SAgent is a general-purpose agent security framework for the JADE platform that is designed to protect the computations of mobile agent applications in potentially hostile environments. The JADE (Java Agent DEvelopment) platform [44] is a popular mobile agent platform that enables the development and deployment of multi-agent applications using Java, and conforms to FIPA standards for software agents. SAgent supports the development
of secure mobile agent applications, where data is protected from compromised or malicious hosts. Various theoretical methods [2, 75] have been proposed to protect agent data and computations from malicious hosts and the aim of SAgent is to bring these theoretical explorations into practice so that various methods can be evaluated for their practicability.

SAgent was designed keeping in mind a basic tenet of software engineering, that definition of proper abstractions is essential for software reusability. The key abstraction behind SAgent mirrors that of public/private key pairs in public key cryptography in which a private key is known only to the owner of the key, while the corresponding public key is available to everyone. Similarly, a mobile agent application has public functionality and information that it needs to perform computations at remote hosts and private information and functionality that is kept by the originator of the agent and not exposed to other entities. Note that while I refer to the information that travels as “public,” this does not mean it is understandable to an outside viewer — encrypted data (ciphertext) can be public even if the corresponding plaintext remains unintelligible. The private information is necessary only for the final interpretation of the results and does not need to be available in any way during the agent’s travels. To that end, our design separates the public and private functionality of the agent into distinct pieces.

There are two different views of these pieces in SAgent — one from the point of view of a programmer that develops protection techniques for SAgent and another from the point of view of a developer who writes applications for SAgent. The architecture of SAgent is designed so that the security provider and the application-developer can remain unaware of each other and develop protocols and applications independently of one another. Data transfer and usage in SAgent revolves around a generic, intermediate data format that is not secure but also not application-dependent. The application developer is responsible for providing translation routines that convert application-specific data into the intermediate format, and the security provider provides routines which convert this intermediate format into a secure but protocol-specific representation. Data can then be operated on in this
secure format by going through methods in the generic public interface, which resolve to the appropriate protocol-specific methods.

6.2.1. Security Model

Keeping in mind the fundamental secure design principle of economy of mechanism, agent applications are decomposed into protected and unprotected computations, with only the sensitive data and computation being run in the protected environment. For instance, a shopping agent may browse through a store using unprotected computations, but could use SAgent to protect the portions of the agent which make purchasing decisions. The protected portion could include very sensitive information, such as payment authorization values, credit card numbers, or electronic cash tokens. The protected agent computation is modeled as a 3-input, 2-output function, as shown in Figure 6.1. The three inputs are the current agent state, the input from the host, and an input provided by any unprotected computations performed by the agent on that host. The two outputs are an updated agent state and an output that is provided to the current host. Specific security providers guarantee slightly different security properties, but in general the agent state is protected so that it is unintelligible to an outside observer (either an attacker or a malicious host), and the host’s input is protected so that it can only be used in a manner consistent with the agent functionality, which can be determined or negotiated as a contract between the originator and the host. In our sample applications I have experimented with encoding credit card numbers both as constants within the protected computation and as part of the agent state. From an efficiency standpoint there is very little difference, so the application designer is free to choose how this private information is included. As an example of an unprotected agent input, one of our applications provides the current time as an input to be used within the protected computation as a timestamp.

6.3. Core Interfaces of SAgent

SAgent uses sound principles of object-oriented design and achieves abstraction by specification of several general interfaces, which are extended by security providers to supply
protocol-specific implementations. The design is similar to some well-designed cryptography libraries, such as Sun’s Java Cryptography Architecture [45], in which generic “encryption” and “decryption” interfaces are defined, and these are bound to specific algorithms and implementations when the cryptography objects are allocated. The design of SAgent separates the private and public functionality of the agent into distinct pieces, which correspond roughly to the agent originator’s secret information and the information which can travel with the agent. The core of our design, therefore, includes secure private and public interfaces, along with a secure data interface. An application designer calls a provider-supplied constructor with appropriate parameters in order to set up the agent application, and from that point on all operations are performed through the generic interfaces. To illustrate this, I point out that SAgent supports both configuration files and command-line parameters that allow protocols and protocol parameters to be changed on the fly. For example, to change from the TX protocol to the ACCK protocol in the Maxbid application, starting the originator can be done with the following command:

```java
java jade.Boot -container Orig:MaxbidOriginator(SAgent.protocol:ACCK)
```

While there is a conditional test (an if statement) in the MaxbidOriginator code to call the appropriate constructor, not a single line of code in the MaxbidMobileAgent class depends on the protocol being used, and this on-the-fly change of security provider is completely transparent to the mobile agents and remote hosts.

In this rest of this section I describe these core interfaces in detail, and describe the functionality they specify. The views and use of these interfaces from both the security provider and application developer are described in more detail in the following section. For reference in the following discussion, the private and public interfaces defined in SAgent are shown in Figure 6.2.

6.3.1. SecureFnPrivate Interface

The SecureFnPrivate interface encapsulates the private functionality in our mobile agent model, defining the methods for interacting with the agent that are restricted to
the agent originator. The originator creates an agent for a particular application using a constructor given by a particular security provider, with the result being a SecureFnPrivate object. Through this interface, the originator sets the initial state of the agent, and retrieves the “public part” (a SecureFnPublic object) which is then sent out in the mobile agents for this application. Each mobile agent then visits various remote hosts, utilizing its public functionality to perform computations on behalf of the originator. Note that the agent can perform both unprotected and protected computations at remote hosts, and good design would suggest that only the sensitive parts of the agent application be performed within the SAgent framework. The SecureFnPrivate interface also defines the functionality that allows the originator to decode the state of the agent after the agent returns home. After successful decryption of the agent state, the state of the agent is converted from a secure data format to a generic object, understood by the application. Thus, it can be seen that the SecureFnPrivate interface provides exactly those methods that allow the originator to perform operations that no remote host should be allowed to do.

6.3.2. SecureFnPublic Interface

The SecureFnPublic interface encapsulates the functionality required by the agent to perform protected computations at remote hosts, modeled as described earlier and shown in Figure 6.1. The public functionality provides means for encoding the agent’s input (obtained from unprotected agent computations) as well as the ability to interact with a visited host and encode the host’s input in an appropriate manner. The encoding is a conversion from
a generic object into an opaque SecureFnData object, which involves a two-step conversion behind the scenes: the application-specific data is first converted into the intermediate, generic data format, and then the protocol-specific conversion of the intermediate data into a protected format is done. The SecureFnPublic interface also defines the function to evaluate the protected agent functionality on these inputs. Below, I describe how the public functionality of an agent is used to perform computations at a visited host, while preserving the security of the agent data. The details of the interactions are illustrated in Figure 6.3 as well.

**Initialization:** Once the originator creates a mobile agent and initializes it with the public part of the protected functionality, the mobile agent should call the initializeSecurity method to initialize any structures that are tied to the mobile agent. In some protection techniques, this might not do anything, but some techniques (such as the TX protocol) require some specific per-agent initialization. However, whether or not a protocol requires this step is not something the application developer should consider, and should simply call this method as the first step of the mobile agent’s actions. Following this initialization, the mobile agent can use the secureMove method to transfer itself to the first remote host.

**Encoding of agent and host inputs:** Upon successful contact with the host (through the host agent), the agent supplies its public part to the host and asks the host for its input. The public part of the agent includes the encodeHostInput method, which is then used by the host to encode its input in an appropriate format. Specifically, this involves reading in the host’s input in a generic form and converting it into an object of type SecureFnData, where the specifics of this transformation depend on the particular protocol which implements this interface. Only the encoded input is returned from the host to the agent, which is what enforces that the host data is used only as specified by the particular agent functionality. Similarly, if the agent has unprotected computation results that need to be included
in the protected computation, then it encodes its input using the \textit{encodeAgentInput} method.

\textbf{Evaluation of the protected function:} Once the agent obtains the host and agent inputs in a secure format, it calls the \textit{evaluate} method to perform the protected computation. This method updates the agent state and returns a \texttt{SecureFnData} object representing the output to be provided to this host, which can be converted to plaintext using the \textit{decodeHostOutput} method. This evaluation obviously depends on the specifics of the agent application, but the evaluation is protected in a uniform way by the particular security provider. This is one of the key features of SAgent, whereby the evaluation function operates only on \textit{secure} objects, where security is defined by the particular protocol or application and can refer to integrity and/or privacy of the data.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{ComponentInteractions.png}
\caption{Component Interactions in SAgent}
\end{figure}
Agent movement: Once a mobile agent has completed its computation at a particular host, it decides to either migrate to another host or to return home. Movement is requested through the secureMove method in the SecureFnPublic interface. When the agent returns home, the final state is communicated to the originator as a SecureFnData object, and the originator can decode this using the decodeAgentState method in the SecureFnPrivate object that it has retained from when it created the agents.

The key to all of the mobile agent computations is that computations are only performed on SecureFnData objects, and the privacy and/or integrity of these operations is guaranteed by the security provider and the security protocol. Any protected information stored in the agent state can be operated on through the evaluate method, but can not be decoded or decrypted except through the SecureFnPrivate object, which only the originator possesses.

6.3.3. SecureFnData Interface

SAgent is designed to protect the security of the mobile agent data. When data are operated upon within SAgent, the data are in a secure format. Therefore, the key idea with SAgent is that only protected data objects should be allowed within the protected execution. The SecureFnData type is simply a generic, opaque type to represent protected data, and does not provide any methods for operating on this data — all operations are performed using either SecureFnPublic or SecureFnPrivate methods, including conversion to/from insecure data representations from/to SecureFnData objects.

Different applications have different data requirements, so SAgent defines an application-independent intermediate format that can be used within SAgent. This is just a binary representation of the data, and makes it simple for application-developers and security providers to develop their products independently of one another. The security provider just deals with bits which he encrypts, decrypts, and operates on as binary data, and is unaware of the actual form or meaning of the data in any application that may use his protocol. Once protected as a SecureFnData object, the actual encrypted representation of the data remains
opaque to the user of the protocol. The application-developer, on the other hand, provides conversion routines between generic binary data and application-specific data. Figure 6.4 shows the different formats that data can be represented in in the SAgent framework. Data comes into the SAgent framework in an application-dependent format, is converted by the application developer to a generic unprotected format, and is then converted into the secure format by routines given by the security provider. All computations with data in SAgent are performed in this secure format. When data is to be returned to the application, it is converted to a generic format (by the protocol) and then passed to the application. The application then converts this generic format to a form specific to the application. This abstraction and automatic conversion between unprotected and protected forms is one of the key ideas of SAgent, which gives the framework an elegant way of separating application developer and security provider interests.

6.4. Design of a Non-interactive Secure Agent protocol in SAgent

SAgent allows programmers to develop agent protection techniques which may be used by a wide variety of applications. SAgent borrows the idea of “security providers” from the Java Cryptography Extension (JCE) [45] to allow programmers to independently develop their own implementations of various security protocols. Implementations of protocols for SAgent may have different characteristics, and a user can select a protocol which satisfies her needs. In this section I describe all of the steps required by our non-interactive secure agent protocol called the GTX protocol. I break down the required operations into three

**Figure 6.4. Representation of Data in SAgent**
phases, initialization, evaluation, and finalization, corresponding to the three phases of the SAgent software framework for secure mobile agents [38, 63]. While all steps are described here, readers unfamiliar with previous work in secure agents may want to refer to earlier papers in this area [2, 75, 84].

(i) Initialization: The originator creates an encrypted circuit for each sensitive computation to be carried out at a host — the square box in Figure 6.1. As outlined in section 3.3, encrypted circuits are special boolean circuits where the signals on the wires are random strings instead of 0 or 1. Since the encrypted circuit can be evaluated with encoded signals, the agent state and inputs must be encoded and incorporated into the agent.

For the GTX protocol, the participating hosts are assumed to have TPMs, with unambiguous identities which can be verified by an agent originator. Each host willing to accept agents and supply \( n \)-bit inputs executes the Setup phase of GNIOT to generate \( n \)-time use keys that are made available to users wishing to send agents. Then at some later time, when an originator wants to send out agents, the originator executes the Transmit phase of the TPM-based GNIOT scheme, where \( m_i = 2 \) and \( k_i = 1 \) for all \( i \in \{1, \ldots, n\} \), and let \( x_{i,1} \) and \( x_{i,2} \) be the two signals corresponding to boolean values 0 and 1 for host input bit \( i \). Note that the output of the Transmit phase of GNIOT is exactly what the hosts will need to be able to decrypt one (and only one) random signal for each of its \( n \) input bits. In creating the agent, the originator bundles together the encrypted circuit, the output \( C \) of the GNIOT Transmit phase, and the host’s output-to-boolean mapping and includes all of this information in the agent. The originator keeps the final state signal-to-boolean mapping for use in decrypting the final agent state when it returns after having visited the hosts.

(ii) Evaluation: In the evaluation phase, the host has received an agent, which carries with it the encrypted circuit to use for execution at this host, the random signals for the current state input, and the output \( C \) from the GNIOT Transmit phase that
was executed when the agent was created. If the host’s input is made up of bits \( \langle b_1, b_2, \ldots, b_n \rangle \), the host calls the GNIOT \emph{Decrypt} function with indices \( \langle i, b_i + 1 \rangle \) for \( i = 1, \ldots, n \). Running \textit{PostProcess} on the results of these \textit{Decrypt} calls will provide \( \langle x_{1,b_1+1}, x_{2,b_2+1}, \ldots, x_{n,b_n+1} \rangle \), which are exactly the random signals needed to evaluate the encrypted circuit. Note that if the host tries to cheat either by requesting both signals corresponding to a single input bit or by requesting more than the allowed number of decryptions, the GNIOT protocol guarantees that the host learns nothing at all about the random signals used by this encrypted circuit. After evaluation of the encrypted circuit, the host uses the output signal-to-boolean mapping supplied by the originator (and carried by the agent) in order to decrypt its input.

(iii) \textit{Finalization}: When the agent returns to the originator, its final state will be decrypted by the originator.

In this chapter, I presented the GTX protocol that uses the GNIOT primitive to ensure security in agent protocols in an efficient manner. In the next chapter, I turn to an experimental study of the efficiency of our TPM-based solutions.
7.1. Introduction

This chapter presents experimental results, both from experiments on the TPM simulator as well as measurements on an Atmel version 1.2 TPM to estimate the performance of the TPM oracle. We also present results for the GTX protocol, which has been integrated into the SAgent framework. In this dissertation, I propose various modifications to the TPMs based on other researchers’ work as well as my own (such as the migration-related commands), the idea being that the TCG specification is designed to be extensible and not static. If enough useful applications make use of proposed modifications like count-limited objects, then the TCG specifications could be modified in the future. For example, the specifications changed from version 1.1b to version 1.2, adding the functionality for certifiable migration and monotonic counters. To that end, I carefully benchmark the proposed modifications as well as the new protocols.

Since experimentation for these modifications with a 1.2 version TPM is not directly possible, I use the TPM simulator\footnote{Note that Strasser and others involved in this project call it an emulator, but as standard terminology used by embedded system designers uses “emulator” for systems that involve specific hardware support, and “simulator” is used for software-only simulations, I use the term “simulator” in this dissertation.} developed by Strasser [74] to measure the performance of these additional operations. This simulator, now an open-source project under the GPLv2 license, is only a partial simulator, emulating only some of the required TPM commands. However, at this time, the commands related to migration of keys are not implemented. Also, since the work on virtual monotonic counters is very recent, that is also not part of the TPM simulator. We use the simulator to implement the `TPM_ExecuteHashTree` command...
used for creating count-limited objects. In Section 7.2, I present an overview of the TPM simulator and describe my extensions to it.

For performing the timing experiments, I use TPM/J, a Java interface to the TPM, developed at MIT [51]. Additional code that was written for standard TPM functionality such as the migration-related commands will be offered back to the main projects (the simulator or TPM/J) for incorporation into the official release. We added the commands and structures related to the migration functionality and count-limited objects to TPM/J, so I present a brief overview in Section 7.3.

7.2. TPM Simulator

The TPM simulator consists of a kernel module, an simulator engine and a user-space daemon that implements the TPM simulator. The kernel module, called tpmd_dev, simulates the hardware TPM by providing a char device /dev/tpm. This interface is the same as for the physical TPM, so application software written for a real TPM will run under the simulator without modification, and vice-versa. When the simulator is loaded, all commands destined for this device are written to the TPM simulator daemon, tpmd, which implements the actual simulator. This consists of the daemon application, the TPM simulator engine and the cryptographic module. The simulator daemon, tpmd, listens for requests on a Unix socket, which has the default name /var/tpm/tpmd_socket:0. Received commands are processed by the simulator engine and responses are returned to the socket and thus back through the kernel driver to the application that sent the command.

The public interface of the simulator engine has 3 main functions. The first function, tpm_emulator_init(), initializes the simulator. To handle various commands, the tpm_handle_command() can be used. Finally, a call to the tpm_emulator_shutdown() shuts the simulator down. At startup, all TPM-related data is set to their default values. The simulator can be started in various modes, as specified in the TCG specifications. All persistent data is written to a specific file (/var/tpm/tpm_emulator-1.2.x.y). If the simulator is started in the “save” mode, all relevant data is restored from that file. If the simulator shuts down normally, all persistent data is written to the same file. Otherwise, in case of
abnormal shutdown or if the simulator is started in the mode “clear”, all internal data of the simulator are destroyed.

7.2.1. TPM Commands and Execution

TPM commands and responses are byte arrays which consist of three main parts: a request or response header, some command-specific input/output parameters, and an optional authorization trailer. Presence or absence of the authorization trailer depends on the used command (response) type, defined by the tag field. There are three different types of commands that vary in how the command is authorized. Figure 7.1 taken from Strasser’s thesis [74]^2, shows the execution of a TPM command in the simulator. The execution of a

![Diagram of simulator and TPM command execution](image.png)

(a) Overview of the simulator  (b) TPM command execution

**Figure 7.1. TPM Command Execution**

TPM command is initiated by calling the function `tpm_handle_command()` with the marshaled TPM command (converted into a byte array) as input parameter. The command is then processed in three steps:

(i) the command is first decomposed into its three main components: the request header, the command parameters, and the (optional) authorization trailer. If it is a legal command, the input parameter digest is computed and the parameters are further unmarshaled according to the specific TPM command.

---

^2Figures reproduced with permission of the author, Mario Strasser
(ii) The command is executed and the command response parameters are setup. For commands that require authorization, the authorization trailer is verified in order to guarantee command authorization and integrity of the input parameters.

(iii) Finally, the response authorization is computed and combined with the response parameters and the response header. The response is then marshaled into its byte representation and returned to the caller.

7.3. TPM/J: Java-based API for the Trusted Platform Module (TPM)

TPM/J is an object-oriented Java interface to the TPM, developed at MIT by Luis Sarmenta [51]. TPM/J has a modular structure, which allows new commands to be added easily to the existing software. Various base classes and interfaces form the core of TPMJ; these classes are then extended by subclasses to provide the required functionality. Different base classes are provided for different types of commands and structures outlined in the TCG specifications.

TPM/J abstracts the low-level interface to the TPM into a TPM driver object. It is a cross-platform API that works on multiple platforms since platform-specific drivers can be provided for any desired platform. TPMJ currently provides driver objects for Linux, Windows XP, Mac OS and Windows Vista.

Additionally, the commands directly sent to the TPM and their corresponding structures are also abstracted as Java objects. Each command and its associated structures are defined as separate classes, allowing new classes and commands to be added easily. For example, any command sent to the TPM consists of a well-defined structure. This structure is represented by the `TPMCommand` class which is the superclass that must be extended by all the command subclasses. The same class is extended by classes that represent the response received from the TPM. Similarly, there are base classes defined for the various types of structures specified in the TPM specifications. For example, structures which have tags associated with them can implement the `TaggedTPMStruct` interface.

TPM/J also provides higher-level classes that represent higher-level functionality like authorization functions, transport functions, administrative functions and identity functions.
These classes group together related functionality in a well-defined manner. TPMJ also provides various utility functions for performing cryptographic operations.

The following table provides an overview of the main packages in TPM/J.

<table>
<thead>
<tr>
<th>Package</th>
<th>Package Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>edu.mit.csail.tpmj</td>
<td>Low-level declarations, including exceptions and constants</td>
</tr>
<tr>
<td>edu.mit.csail.tpmj.commands</td>
<td>Structures representing TPM commands and responses</td>
</tr>
<tr>
<td>edu.mit.csail.tpmj.drivers.*</td>
<td>Communication Interfaces between Java and OS-specific TPM drivers</td>
</tr>
<tr>
<td>edu.mit.csail.tpmj.funcs</td>
<td>High-level functions grouped by function</td>
</tr>
<tr>
<td>edu.mit.csail.tpmj.structs</td>
<td>Java representations of the TPM structures</td>
</tr>
<tr>
<td>edu.mit.csail.tpmj.tools</td>
<td>Various tools to perform basic tasks</td>
</tr>
<tr>
<td>edu.mit.csail.tpmj.util</td>
<td>Utility structures and functions for cryptographic support</td>
</tr>
<tr>
<td>edu.mit.csail.tpmj.tests</td>
<td>Basic tests of the TPM functions</td>
</tr>
</tbody>
</table>

7.4. Implementation Details

In this section, I present details of the extensions to the simulator. I added functionality to support migration of regular keys as well as CMKs. To enable creation of count-limited keys, I extended the simulator with the TPM\_ExecuteHash\_Tree command. Finally, I added the GTX protocol to the SAgent framework.

7.4.1. Extensions to the TPM Simulator

The TPM simulator had support for creation and use of various types of keys; however a big piece of the functionality of migrating keys was missing. The TCG specifications allow for migration of regular keys as well as CMKs and the TPM oracle construction uses CMKs to establish a shared secret between parties. To test the feasibility of creating a TPM oracle, I added all the CMK-related commands to the simulator. Since this construction requires a new key type specifically for use as a seed to HMAC, I add a new key type, which requires very little modification. There is already an algorithm type specified in the TPM for HMAC, so I use this in the TPM\_KEY structure. Since this structure simply keeps track of the public
key information in byte array, no further changes are required. The only modification is to simply define a new key flag called `TPM_KEY_HMAC` to ensure that any key with this flag is used only as a seed to HMAC.

In order to be able to create and use count-limited objects, I added a new command called `TPM_ExecuteHashTree` (as outlined by Sarmenta et al. [64]) to the simulator. I store the count-limit condition in the PCRInfo field in the `TPM_KEY` structure and add a flag to the `TPM_KEYFLAGS` structure. In addition, I modified the `TPM_LoadKey`, `TPM_Sign` and the `TPM_UnBind` commands to check the count-limited condition of a key (only if the blob key flag is set) before performing the desired operation.

To test the implementation of the above functionality in the simulator, I migrated a CMK from an actual TPM to the simulator and vice-versa. Along with providing a validation of my methodology, this allowed us to check that the migration functionality was implemented correctly. It should be noted that I did not use a globally trusted CA, so certificates for keys are not generated or checked.

7.4.2. Extensions to TPM/J

Taking advantage of the modularity of TPMJ, I added new structures and commands corresponding to the migration functionality of the TPM. In order to add new commands, the specific `TPMCommand` class is extended. For example, `TPM_CMK_CreateBlob` extends the `TPMKeyAuth1Command` command which is the base class for commands that take as input parameters, a key handle and an authorization handle. The migration functions are all grouped in the `TPMMigrationFuncs` class in the `edu.mit.csail.tpmjfuncs` package, so high-level access to the migration functionality can be directly provided. Finally, a test class was added to the `edu.mit.csail.tpmjtests` package to perform timing experiments on the various migration functions.

7.5. Benchmarks for Creating a TPM Oracle

In this section, I report some benchmarks on basic operations performed on an existing TPM, and use these benchmarks to estimate the performance of the new protocol. I must
extrapolate from basic benchmarks due to this protocol’s requirement of currently unavailable functionality (specifically, the use of HMAC with an internal TPM secret key).

Apart from benchmarking the operations, my goal was to add timing information taken from an actual TPM to the simulator. All my constructions are composed of basic TPM operations like encryption, signatures, etc., so I incorporate timing information for these operations into the simulator. This creates a timing-accurate simulator, which can be used to estimate the efficiency of these constructions as well as those proposed in the future. This is an innovative tool that can be used to predict the performance of any new operations before they are added to future TPMs.

For my tests, I used a desktop machine which incorporates a version 1.2 TPM made by Atmel (firmware version:13.5). I ran each of the timed operations at least 4 times, and give the overall times for each operation, the times in the TPM and the overhead for each operation in the following table. Using the strace tool, I pinpointed the exact system calls where TPM/J interacts with the simulator (via the write/read calls) and measured the times for these calls. This allowed us to estimate the actual time for the operation inside the TPM. This time is reported as the TPM time. The total time for the operation then includes the overhead for the output to filter back through the layers of TPM/J and includes operations performed on the host. An example of this is the authorization protocols associated with each command, which involve the generation of nonces and secrets on the host. Therefore, the total times reported include the overhead of such computations performed on the host and are host-dependent.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Time on TPM (sec)</th>
<th>Total time (sec)</th>
<th>overhead(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_LoadKey2</td>
<td>0.9</td>
<td>1.05</td>
<td>16.7</td>
</tr>
<tr>
<td>TPM_CMK_CreateKey</td>
<td>0.2 to 22.23</td>
<td>0.25 to 22.25</td>
<td>25</td>
</tr>
<tr>
<td>TPM_CMK_CreateBlob</td>
<td>0.91</td>
<td>1.01</td>
<td>11</td>
</tr>
<tr>
<td>TPM_CMK_ConvertMigration</td>
<td>0.94</td>
<td>0.96</td>
<td>2.1</td>
</tr>
<tr>
<td>TPM_CertifyKey</td>
<td>0.82</td>
<td>0.92</td>
<td>12</td>
</tr>
<tr>
<td>TPM_AuthorizeMigrationKey</td>
<td>0.04</td>
<td>0.27</td>
<td>575</td>
</tr>
<tr>
<td>TPM_CMK_ApproveMA</td>
<td>0.04</td>
<td>0.36</td>
<td>800</td>
</tr>
</tbody>
</table>

Only the TPM_CMK_CreateKey showed any significant variability in the time between different tests, which is understandable due to the probabilistic nature of generating RSA keys. While creating a new asymmetric key is currently the only way to establish a CMK and hence a shared secret, which is why I include that time above, the new protocol does not require an asymmetric key as a CMK — as I discussed in Section 4.2 I suggest the use of a simple randomly generated secret, in which case the time for creating such a CMK would be dominated by the time to wrap (encrypt) this secret using the parent key. I assume that verification of certificates by the host platform takes insignificant time compared to these TPM operations — RSA keys used by the TPM use a fixed public exponent of 65,537, and on a modern PC-class processor such a verification can be done in less than a millisecond.

7.5.1. Performance on an Actual TPM

In this section, I show the timing results for creation, migration and conversion of a CMK on the Atmel TPM, where the host is an Intel Core 2 Duo 6700 running at 2.66 GHz.

I describe the various phases on the source and destination TPMs and estimate the performance of the “shared secret establishment” construction.

7.5.1.1. Performance of Secret Creation and Migration

This section presents the timing results for the secret creation and migration phase of the TPM-oracle construction.
(i) The first step in the CMK creation process involves putting the CMK under the control of a migration authority. This process involves creating a SHA-1 digest of the public key of the MA (storage and signing). This step corresponds to the `TPM_CMK_ApproveMA` command and takes 0.04 seconds.

(ii) CMK creation involves the execution of the `TPM_CMK_CreateKey` command which creates a key which is then encrypted using a parent key. As explained in Chapter 2, every key in a TPM is encrypted under a parent key when it is created. The root of this storage hierarchy is the SRK. Since this operation involves the generation of a random prime of the required bitlength, there is considerable variability in the times, from 0.22 seconds to 22.25 seconds. I use the lowest time reported to estimate the performance of the protocol, since this technique does not require the costly RSA generation.

(iii) The newly created CMK must now be certified using an AIK by the source. In order to perform the certification, the CMK and the AIK must both be loaded and this takes 1.80 seconds to load both keys. While I include these times in these estimates, it may not be necessary to load every parent key since the SRK could be the parent of some of these keys and is always loaded in the TPM. The certification itself takes 0.82 seconds.

(iv) Before the CMK can be migrated to the destination, the public key of the destination must be approved as a suitable destination using the `TPM_AuthorizeMigrationKey` command. This step takes 0.04 seconds.

(v) The next step in the migration process at the source TPM is executing the `TPM_CMK_CreateBlob` which decrypts the CMK (encrypted using a parent key) and then re-encrypts the CMK using the destination (MA) public key (of type storage). This step takes 0.91 seconds.

A migration blob is created for each of the \( n \) destinations, taking a total of \( 0.91(n - 1) \) seconds. Combining these results, the total time for the secret creation/migration phase
Figure 7.2. Times for CMK Operations on the Source TPM

is $2.01 + 0.91n$ seconds, as shown in the following table. In particular, for a two-party protocol, the total time at the source is 3.83 seconds.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time on TPM (sec)</th>
<th>Total time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_CMK_CreateKey</td>
<td>0.22</td>
<td>0.25</td>
</tr>
<tr>
<td>TPM_CMK_CreateBlob</td>
<td>$0.91(n - 1)$</td>
<td>$1.01(n - 1)$</td>
</tr>
<tr>
<td>TPM_AuthorizeMigrationKey</td>
<td>0.04</td>
<td>0.27</td>
</tr>
<tr>
<td>TPM_CMK_ApproveMA</td>
<td>0.04</td>
<td>0.36</td>
</tr>
<tr>
<td>TPM_LoadKey2 (2 keys)</td>
<td>1.8</td>
<td>2.1</td>
</tr>
<tr>
<td>TPM_CertifyKey</td>
<td>0.82</td>
<td>0.92</td>
</tr>
<tr>
<td>Total time at the source</td>
<td>$2.01 + 0.91n$</td>
<td>$2.89 + 1.01n$</td>
</tr>
</tbody>
</table>

7.5.1.2. Performance of Destination Secret Installation

This section presents the timing results for the phase of the CMK migration at the destination.
(i) The first step on the destination is to load a signing key into the TPM and create a signature over a structure called the TPM_CMK_AUTH which contains the digests of the public keys of the CMK, the destination and the MA. This signature, along with the structure, is passed to the TPM_CMK_CreateTicket command, which verifies the signature on the structure and creates a ticket, upon successful verification. This step ensures that the MA has “signed off” on the destination.

The time to load the signing key is 0.90 seconds. The time for the signature computation is 0.80 seconds and the time for the TPM_CMK_CreateTicket command is 0.1 seconds.

(ii) The next step is to execute the TPM_CMK_ConvertMigration command, which takes the encrypted private key of the CMK and decrypts and re-encrypts it under its destination storage key.

This step takes 0.94 seconds. However, in order to perform this step, the destination key must be loaded into the TPM. The TPM_LoadKey operation takes 0.9 seconds on average, giving us a total destination time for steps (i) and (ii) of 3.64 seconds.

Figure 7.3 shows the breakdown of the times as measured in the TPM and the total time for the operations on the host.

Therefore, the total protocol time (excluding the host-related overhead) for a two-party protocol for migrating a CMK from a source TPM to a destination TPM is 7.47 seconds.

The times presented above were the times for the operations inside the TPM only. Next, I estimate the performance of the protocol by adding the host-related overhead for these operations. On the source, a migration blob is created for each of the $n$ destinations, taking a total of $1.05(n - 1)$ seconds. Combining these results, the total time for the secret creation/migration phase is $2.81 + 1.05n$ seconds. For a two-party protocol, the total time at the source (including the host overhead) is 4.91 seconds. At the destination, the total time is 4.09 seconds, giving a total protocol time of 9.0 seconds.
Figure 7.3. Times for CMK Operations on the Destination TPM

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time on TPM (sec)</th>
<th>Total time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_LoadKey2 (for signing key and parent)</td>
<td>1.8</td>
<td>2.1</td>
</tr>
<tr>
<td>TPM_Sign</td>
<td>0.8</td>
<td>0.82</td>
</tr>
<tr>
<td>TPM_CMK_CreateTicket</td>
<td>0.1</td>
<td>0.16</td>
</tr>
<tr>
<td>TPM_CMK_ConvertMigration</td>
<td>0.94</td>
<td>1.01</td>
</tr>
<tr>
<td>Total time at the destination</td>
<td>3.64</td>
<td>4.09</td>
</tr>
</tbody>
</table>

7.5.2. Benchmarks from the TPM Simulator

In this section, I present benchmarks from experiments performed on the TPM simulator which has been augmented with the migration functionality. The measured times from the operations on the actual TPM are added to the simulator to generate “performance profiles” for the basic operations. Since operations on an actual TPM typically take longer than operations performed on a desktop-class CPU, I added `sleep` statements to various basic operations in the simulator. For example, I added a delay of several milliseconds to simulate the TPM_LoadKey command which involves an RSA decryption. Since all TPM commands
are composed of basic cryptographic functionality like hashing, signatures, encryption and decryption, I add `sleep` to these basic commands. By making the simulator timing-accurate, I have developed a novel tool using which the performance of any proposed extensions to a standard TPM can estimated. When new commands like the `TPM_ExecuteHashTree` command are added then to the simulator, I get an accurate estimate of the performance of the particular command, since these commands build on basic operations.

Only one machine containing the Atmel TPM was available to us and I used that machine to measure the times on an actual TPM. The simulator was tested on a cluster of Pentium IV 2 GHz machines, running Fedora Core 4 Linux, since I wanted to test the migration of CMKs in a distributed environment. Notice that there is considerable overhead for some of the operations tested on the simulator. The Intel Core Duo 2 is about 2-3 times faster than the machines which ran the simulator, and since the overhead is entirely dependent on the host CPU, I expect the overhead to be half to a third more on the older machines. This is indeed reflected in the measured results.

I give the times for migrating a CMK from a source to a destination. In this case, I generate a key of type HMAC, which is a new key type proposed in this work. Notice that the overhead for some of the operations is considerably more in this case since that is dependent on the processor speed. Since the simulator is designed to be timing accurate, the TPM times do not vary. In fact, that is the reason I measure the times on the simulator - to be able to predict the performance of this extensions.

I present the times in seconds for the operations with the most overhead below:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Total time (Pentium IV)</th>
<th>Total time (Intel Core Duo)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_AuthorizeMigrationKey</td>
<td>0.67</td>
<td>0.27</td>
</tr>
<tr>
<td>TPM_CMK_ApproveMA</td>
<td>0.54</td>
<td>0.36</td>
</tr>
</tbody>
</table>
Figure 7.4. Times for CMK Source Operations on the Simulator

Figure 7.5. Times for Destination Operations on the Simulator
7.5.3. Performance of TPM-Oracle Queries

There is no current user interface to the HMAC engine within a TPM, but there is a direct way to use the SHA1 engine. Since the HMAC computation consists entirely of multiple calls to the SHA1 function, this is a good way to estimate the performance of HMAC in the TPM. I tested SHA1 on inputs of size 16k, 32k, 48k, and 64k, and found a very consistent increase of 1.15 seconds per 16k increment.

Here’s how I derive the estimates. For the SHA-1 estimation: the SHA-1 block size is 512 bits, or 64 bytes. Therefore there are $16k/64 = 256$ blocks in each 16k increment. Therefore, I obtain $\frac{1.15}{256} = 4.5$ milliseconds as the estimate for one application of the SHA1 function. Note that the since the input data to be hashed had to be passed into the TPM over a relatively slow interface, this timing may be as much constrained by platform-to-TPM communication speed as it is by the computational speed of the SHA1 engine, which would not be as much an issue with this usage. The ultimate goal is to estimate the performance of HMAC, not just the SHA1 function. As seen in Figure 7.6, taken from [83], notice that the input key extension adds 1 block to the hashing, and the output hash adds an extra 2 blocks.
Based on these benchmarks, I see that if the TPM-oracle is answering queries from a domain that requires \( b_i \) input blocks to hash, and I am generating an output that needs \( b_o \) output blocks, then the time would be at most \( 4.5(b_i + 3)b_o \) ms. To make this concrete, note that for 1024-bit inputs (\( b_i = 3 \)), with 160-bit outputs (\( b_o = 1 \)), each random oracle query would take no more than 27 milliseconds.

In summary, the TPM-oracle using current hardware takes around 8 seconds of setup time for one party, and less than 30 milliseconds for most realistically sized TPM-oracle queries. I feel that this is quite practical, and future improvements to TPMs which reduce the setup time would only make it more clearly practical.

7.6. Extensions to SAgent

I create a new security provider, GTX, for SAgent [38] by extending the key interfaces of SAgent. In the previous chapter, I outlined the phases of the GTX protocol. In this section, I present implementation details of how this protocol fits into SAgent.

**Initialization:** The initialization phase of this protocol involves the GNIOT *Setup* phase. In this phase, all hosts generate count-limited RSA keys, with the count-limit condition corresponding to the number of input bits \( n \), and send the corresponding public keys to the originator. These keys are certified using an AIK corresponding to each host’s TPM. Then at some later time, when an originator wants to send out agents, the originator executes the *Transmit* phase of the TPM-based GNIOT scheme, where \( m_i = 2 \) and \( k_i = 1 \) for all \( i \in \{1, \ldots, n\} \), and I let \( x_{i,1} \) and \( x_{i,2} \) be the two signals corresponding to boolean values 0 and 1 for host input bit \( i \). As part of the GNIOT *Transmit* phase, the originator chooses random \( n \) 128-bit values \( R_1, R_2, \cdots, R_n \) and the computes an AES key as \( R = R_1 \oplus R_2 \oplus \cdots \oplus R_n \). Since this scheme uses a 1-threshold secret-sharing scheme for each input bit, the same secret is associated with each of the two bits. The originator then encrypts the host inputs using the AES symmetric key. The encrypted inputs and the
shares of the secret key are then encrypted using the corresponding host’s count-
limited public key. This corresponds to the private functionality of the agent and is
implemented in the GTXFnPrivate class which implements the SecureFnPrivate
interface of SAgent.

Visit to a host: When the agent is at a remote host, it retrieves a bid from the host
and as part of the GNIOT Decrypt phase, decrypts its inputs using the clob. This
involves a call to the TPM_UnBind command which checks the count-limit condition
before decrypting any signal.

Since these computations must be performed by the agent using informa-
tion from its public part, this phase corresponds to the public functionality of
the agent and is implemented in the GTXFnPublic class which implements the
SecureFnPublic interface. In particular, the GNIOT Decrypt computations cor-
respond to the encodeHostInput method of the public interface.

The inputs are then evaluated (evaluate method of GTXFnPublic is called) and
the state of the agent is updated as a result of the evaluation of the inputs.

Finalization: When the agent returns, the originator decodes the state of the agent,
converting the protected state into an unprotected object. This phase corresponds
to the private functionality of the agent and is implemented in the decodeAgentState
method of the GTXFnPrivate class.

7.7. Sample Applications

I developed two sample applications for SAgent, which I describe in this section. Both
applications are in the general area of e-commerce agent applications. I use the simpler
Maxbid application to test the protocols with various keysizes and use the BuyAgent appli-
cation to compare the ACCK and the TX protocols for the typical asymmetric keysize of
1024 bits. For the GTX protocol, I use an asymmetric keysize of 2048 bits.
7.7.1. Maxbid Application

In this application, the originator wants to sell an item and sends one or more agents out to visit remote hosts to gather offers for the item. Each host gives its bid and the agent keeps the *maximum* bid it has seen so far in its state. I would like to keep the current maximum bid secret so that hosts cannot cheat by bidding slightly over the current maximum. When the agents returns to the originator, the originator decodes each agent’s maximum bid and computes the overall maximum bid. The protected function in this application is a comparison of two integer values (current maximum bid and bid offered by current host). In this application there is no unprotected agent input and there is no output to the host. This is often called a “blind auction” or a “sealed bid auction.”

7.7.2. BuyAgent Application

This application is slightly more complex than the *Maxbid* application, using all three of the protected functionality inputs and providing the host with one of the outputs. In this application, the originator wishes to buy an item, so the agent maintains the *minimum* bid it has seen so far. Furthermore, the minimum bid is time-stamped, with the time being provided as the unprotected agent input (unprotected since the current time is not sensitive information). In addition, the originator programs a threshold price called the “buy now” value into the agent, whereby, if any host offers a bid lower than the threshold, the agent provides the host the originator’s credit card number to immediately buy the item. Clearly, both the “buy now” value and the credit card number are sensitive pieces of data, and should be protected. The function in this application is a comparison of two integer values, with the evaluation resulting in an update of the minimum value and the corresponding timestamp. The protected function also provides a comparison with the “buy now” threshold, revealing the sensitive information only if the threshold value is met. Next, I present results comparing the new non-interactive GTX protocol with interactive agent security protocols.
7.7.3. Efficiency Analysis of the GTX Protocol

In this section, I present results from experiments with the GTX protocol in the SAgent framework. Additionally, I compare the performance of the GTX protocol to existing agent protocols. The experiments were performed on a cluster of six 2GHz Pentium IV machines, connected to a 100 Mbps network, all running Fedora Core 4 Linux. I used Sun’s Java SDK 1.5 with JADE version 3.2 and an instrumented version of SAgent 0.9. One machine was designated as the originator and there were 5 visited hosts. For comparing the times of the ACCK protocol, I present times for 1 agent visiting the 5 remote hosts. For the TX protocol, the threshold property requires at least 3 or 4 agents, so for comparison, I present times for 4 agents visiting the remote hosts. For detailed results comparing the performance of the ACCK and the TX protocols, refer to these previous work [40].

7.7.4. Parameters

Asymmetric key sizes vary dramatically depending on the algorithm used, or more precisely on the cyclic group that algorithms operate over. The simple modular arithmetic groups, which I refer to generically as $Z_p$, require much larger key sizes than those based on Elliptic Curve Cryptography, which I refer to generically as ECC. For example, comparable key sizes (in bits) between these two classes of algorithms, as specified by the National Institute of Standards and Technology (NIST) and the IEEE standard on Public Key Cryptography [60], show that a keysize of 2048 bits over $Z_p$ is comparable to 224 bits over ECC. Since the TPM always creates 2048 bit RSA keys, I set the asymmetric keysize over $Z_p$ for all protocols to 2048 bits. From previous experiments, I discovered that the signal size does not affect the overall protocol times. Hence, I picked the signal size as 80 bits. Finally, for the AES encryption in the GTX protocol, I chose a key size of 128 bits.

In addition to cryptographic parameters for the protocols, the applications have tunable parameters as well. Both applications work with “bids,” which are integer values that can be any number of bits. The BuyAgent application includes a timestamp and a credit card number as the agent input and part of the agent state, respectively, so varying the
size of these parameters affects the size of the inputs to the encrypted circuits. In these experiments, I varied these parameters by drawing from a fixed set of possibilities, shown in the following table. Note that, based on previous experiments which showed that there was no significant variability in times with the size used for the credit card number, I fixed that size to 16 bits for experiments with the GTX protocol.

| Bid (i.e., host input) size (in bits): | {16, 24, 32} |
| Credit card number size (in bits):    | {16}         |
| Timestamp (i.e., agent input) size (in bits): | {32, 48, 64} |

I instrumented the standard SAgent distribution in order to measure the following values.

- **Initialization time:** The initialization step involves the originator creating the encrypted circuits, setting the initial agent state, and then sending the agents out to visit the hosts. Creating the encrypted circuits includes encrypting the input signals for each circuit, which dominates the initialization time.

- **Agent Computation time:** This is time the agent spends on remote hosts, measured from when the originator sends the agents out to the time when they return.

- **Per-Host Computation time:** This is the agent computation time, divided by the number of visited host. This metric gives a measure of the average amount of time an agent spends performing computations (and waiting on results from other parties) on each host that it visits.

- **Total Protocol time:** This is the total time from start to finish for each of the protocols. This includes the initialization time as well as the total agent computation time.

I present overall times in seconds for all the security protocols below. See Figure 7.7 for a comparison of the times. For the TX protocol, I present the times for when the computation is performed over elliptic curves (ECC with 224 bits) and \( \mathbb{Z}_p \) (comparable directly to the
GTX protocol) with a threshold of 4 agents.

<table>
<thead>
<tr>
<th>Times (seconds)</th>
<th>ACCK</th>
<th>GTX</th>
<th>TX-ECC-4</th>
<th>TX-Zp-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>17.22</td>
<td>175.20</td>
<td>325.88</td>
<td>1541.51</td>
</tr>
<tr>
<td>Circuit Creation</td>
<td>1.78</td>
<td>6.76</td>
<td>27.75</td>
<td>421.88</td>
</tr>
<tr>
<td>Per Host</td>
<td>14.65</td>
<td>33.4</td>
<td>63.62</td>
<td>295.79</td>
</tr>
</tbody>
</table>

**Figure 7.7.** Comparison of Total Times for Security Protocols

Notice that there is a significant improvement in efficiency over the TX protocol, even when the threshold group for the TX protocol is the more efficient ECC group. The time for just the initialization phase of the protocols is reported in Figure 7.8. These results mirror, on a reduced scale, the total protocol time very closely.

Figure 7.8 shows the per host times for each visited host for all the security protocols, when the time for the initialization phase is removed. As expected, the per host times for the ACCK protocol are low, with the agent spending less than 15 seconds at each visited host.
when using 2048-bit keys. The times are higher for the TX protocol, with the ECC times being significantly faster than those over corresponding $\mathbb{Z}_p$ key sizes. For keysizes considered secure into the future (2048-bit $\mathbb{Z}_p$ and 224-bit ECC), the computation time per visited host when using elliptic curves) is around 1 minute. Notice that for the GTX protocol, this time is even lower, around 33 seconds or half a minute per visited host. Considering that I am using a strong keysizes of 2048 bits, these times for the GTX protocol are quite reasonable for the innovative features and level of provided by the protocol.

Figure 7.8. Comparison of Circuit Creation Times

7.7.5. Variation of Times with Input Size

I present times showing how the perhost times and total times vary with host input size for the maxbid application for the GTX protocol. Note that the number of decryptions performed by the count-limited key is tied to the number of bits in the input, and as expected, the times increase as the host input size increases. Next, I present the times showing how the variation in host input size affects different applications. The host input size is varied from 16 bits to 32 bits for the maxbid and buyagent applications. Since the buyagent application includes agent input (set to 32 bits here), there is an increase in times
Figure 7.9. Comparison of Per Host Computation Times for All Protocols

Figure 7.10. Variation of Per Host Computation Times for the GTX Protocol when compared to the maxbid application. Here I show how the increase in agent input affects the buyagent application.
<table>
<thead>
<tr>
<th>Host Input</th>
<th>Agent State</th>
<th>Agent Input</th>
<th>Number of signals to be recovered</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>64</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>24</td>
<td>72</td>
<td>32</td>
<td>56</td>
</tr>
<tr>
<td>32</td>
<td>80</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

Note that for the maxbid application, there is no agent input and the agent state is simply the host input (16, 24, 32). Notice that the addition of the agent input when the host input is 16 bits leads to a threefold increase in the number of signals that need to be recovered for the buyagent application. The results are given in Figure 7.11, showing that the increase in signal decoding requirements is closely mirrored in the increase in total time, verifying that this is the dominant cost. Notice in Figure 7.12, where the agent input size is 64 bits for all protocols, that a similar increase is noticed for the ACCK and TX protocols. As seen from the results, the GTX protocol shows considerable improvement in efficiency over the TX protocol. Although the GTX protocol is less efficient than the ACCK protocol, the GTX protocol is more scalable than the ACCK protocol, where all the hosts contact one TTP for decryption of signals, which can often be a bottle-neck. So, I conclude that the GTX

**Figure 7.11.** GTX protocol: Maxbid and Buyagent Applications
Figure 7.12. Comparison of maxbid and buyagent applications

protocol, in addition to having the significant advantage of being a non-interactive protocol and therefore, being less sensitive to network delays, is more efficient than the TX protocol.

In the next chapter, I wrap up this dissertation with a summary of the aforementioned results and a discussion of future work.
CHAPTER 8

CONCLUSION

8.1. Conclusion

In this dissertation, I have studied the capabilities of trusted platform modules from a theoretical as well as an experimental standpoint. I have shown how a random oracle can be instantiated in a multi-party setting if each party has access to a trusted platform module (TPM). Utilizing a special kind of key called “certifiable migratable key” that can be certified by a TPM and migrated to other TPMs, I show how to instantiate a random oracle using HMAC-based pseudorandom functions. This method requires a few modifications to the TPM specification, but none of the modifications are particularly difficult or unreasonable. I provided rigorous proofs that under an assumption of secure TPMs (and standard complexity assumptions) this construction is computationally indistinguishable from a random oracle to a polynomial-time attacker. In the process, I formally define and prove properties about a new cryptographic primitive which I call a “hybrid pseudorandom function” that may be of independent interest.

Additionally, I have shown how to remove interaction requirements in the fundamental cryptographic primitive of oblivious transfer to create a new cryptographic primitive called “generalized non-interactive oblivious transfer” (GNIOT). This primitive is based on current research which shows how to instantiate count-limited objects using the monotonic counter in trusted platform modules. I provided rigorous proofs that under an assumption of secure TPMs (and standard complexity assumptions), this construction provides the same security properties as those of standard oblivious transfer. In addition, I showed how to apply the GNIOT primitive to develop a secure mobile agent protocol (called the GTX protocol) where strong security guarantees can be achieved without the interaction requirements necessary in previous secure agent protocols.
On the experimental front, I enhanced the functionality of a software TPM simulator [74], an open-source project under the GPLv2 license, to include functionality for creating count-limited objects as well as for migrating various types of keys. I also integrated the GTX protocol into the SAgent [38] mobile agent framework and performing experiments to evaluate the performance of this protocol. Apart from enhancing the functionality of the simulator, I also added timing information taken from an actual TPM to it. This creates a timing-accurate simulator, which allows us to estimate the efficiency of the GTX protocol and the TPM oracle accurately. This is an innovative tool that can be used to predict the performance of any new operations before they are added to future TPMs.

Our experimental analysis shows that the GTX protocol is considerably more efficient than the TX protocol, while providing the same security guarantees and having the advantage of being a non-interactive protocol. Our experiments with constructing a TPM oracle also show that instantiating this construction would be reasonably efficient.

8.2. Future Work

In this dissertation, I explored some features of TPMs that I used in a manner outside of the TCG’s original design goals. This work can be used to build a more comprehensive model for TPMs, creating a layered approach that has a strong theoretical component as well as an experimental part. Functionality found in the TPM could form the bottom layer, with the middle layers providing abstractions based on cryptographic primitives like shared secret establishment and the top layer then corresponds to applications. The benefits to such a layered structure are well-known in software development; for analysis the benefit is similar. By abstracting lower level functionality as basic operations, proofs are simplified, much like using well-defined lemmas to add structure and simplify proofs of complex theorems. Various other theoretical constructs like those based on homomorphic encryption techniques could be developed in this layered model.

In the following section, I outline possible future work specifically for random oracles and extensions to the TPM simulator.
8.2.1. Random Oracles

An interesting open problem is to improve upon the secret establishment process. Currently, if I wish to avoid having an active trusted migration authority, then public keys for all parties must be known at the beginning of the computation, which is particularly difficult in the case of non-interactive zero-knowledge proofs where I don’t know who the verifier may be. This is also a problem for dynamic multi-party settings, where parties may join or leave the group during the execution of the protocol. Unfortunately, this seems to be a very difficult problem if the parties are mutually untrusting — there is no obvious way to have a key that can migrate multiple hops without a transitive trust relationship, which is not directly available with the current TPM design. An additional problem related to secret establishment is that, even in a static case where all keys are known, the secret distribution scheme does not scale well as the originating party must send the packaged secret to every participant, taking $\Theta(n)$ time. A tree-structured distribution scheme with depth $O(\log n)$ would be significantly better, but the same problem with transitive trust and forwarding migrating keys is encountered.

8.2.2. TPM Simulator

Currently, the simulator does not include any randomization for generating any of the keys, in particular the EK is simply hard-coded values read from a file. Future work will involve adding randomization to the simulator to estimate timings of probabilistic key generation. In this dissertation, I developed performance profiles for the operations necessary for my work. Future extensions to the simulator will involve adding accurate timing information to all the basic operations and common functionality used by most applications. Performance profiles could be then developed for various features like operations using the monotonic counter. In my work, I used measurements from one particular Atmel TPM to derive measurements for the simulator. In the future, performance profiles could be generated for various types of TPMs. These performance profiles could be “loadable” in the sense that a performance profile for a specific TPM could loaded on demand. Currently, I implemented
the \texttt{TPM\_ExecuteHashTree} command in the simulator from the work on virtual monotonic counters that allows for creation of count-limited objects. Sarmenta \textit{et al.}\cite{52} have proposed other schemes that are more efficient and support for these can be added to the simulator.


[53] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Naor [58], pages 21–39.


