DEVELOPING SECURE APPLICATIONS USING HARDWARE SECURITY MODULES

BY

LODOVICO L. MINNOCCI

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree of Master of Science

Southern Connecticut State University

New Haven, Connecticut

August 2007
DEVELOPING SECURE APPLICATIONS USING HARDWARE SECURITY MODULES

BY

LODOVICO L. MINNOCCI

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Lisa Lancor, Department of Computer Science, and it has been approved by the members of the candidate's thesis committee. It was submitted to the School of Graduate Studies and was accepted in partial fulfillment of the requirements for the degree of Master of Science.

Lisa Lancor, Ph.D.
Thesis Advisor

Hrvoje Podnar, Ph.D.
Second Reader

Winnie Yu, MS.
Department Chairperson

Sandra C. Holley, Ph.D.
Dean, School of Graduate Studies

August 10, 2007
Date
ABSTRACT

Author: Lodovico L. Minnocci
Title: DEVELOPING SECURE APPLICATIONS USING HARDWARE SECURITY MODULES
Thesis Advisor: Dr. Lisa Lancor
Institution: Southern Connecticut State University
Year: 2007

Cryptography is a mathematical science used to secure the confidentiality and authentication of data by replacing it with a transformed version. Once transformed, the data can be re-converted to reveal the original information only by someone holding the proper cryptographic algorithm and key. It is a discipline that embodies the principles, means, and methods for transforming data in order to hide its information content, prevent its undetected modification, and/or prevent its unauthorized uses. Procedures and protocols that meet some or all of the above criteria are known as cryptographic systems. The goal of this research is to provide a framework for an Application Programming Interface (API) that can be used to develop secure applications using Hardware Security Modules (HSM). This framework will broker communications between an application and the HSM. Providing a comprehensive API will enable application developers to implement HSM cryptography without extensive knowledge of the underlying cryptographic hardware.
To my parents.
ACKNOWLEDGEMENTS

This work is the synergistic product of many minds. I am grateful for the inspiration of those close to me and the technical knowledge acquired over the years from professors, colleagues, and fellow students. For the development of this work I feel a deep sense of appreciation to:

Dr. Lisa Lancor for her advisement. Her knowledge of computer security has been of great value for me. Her understanding, encouragement and guidance has provided a basis for the present thesis.

The happy memory of my grandfather whose name I bear.

My friend Paul Carbone. I would like to thank him for his support, mentoring, and friendship through the years.

My wife Michelle for supporting me. I would like to thank her for her love, encouragement, and endless editorial suggestions.
# TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION ........................................................................... 1  
  Information Security .................................................................................. 1  
  Cryptographic Systems ............................................................................. 2  
  Purpose ..................................................................................................... 5  

CHAPTER 2: COMPUTER SECURITY ............................................................... 7  
  Security Goals .......................................................................................... 7  
  Vulnerabilities .......................................................................................... 9  
  Countermeasures ...................................................................................... 10  

CHAPTER 3: CRYPTOGRAPHY ..................................................................... 11  
  History of Cryptography .......................................................................... 11  
  Public Key Cryptography ........................................................................ 13  
  Encryption ............................................................................................... 14  
  Public Key Encryption ............................................................................. 14  
  Encryption Algorithms ............................................................................ 15  
  Hash Algorithms ...................................................................................... 18  
  Message Authentication Code ................................................................. 20  
  Digital Signatures .................................................................................... 20  
  Digital Signature Scheme ....................................................................... 22  

CHAPTER 4: HARDWARE SECURITY MODULES ........................................... 25
HSM Functionality ................................................................. 26
Common Implementations of an HSM ........................................ 27
Positive Attributes of an HSM .................................................. 28
Drawbacks to Using HSMs .......................................................... 30
Public Key Cryptography Standards #11 (PKCS#11) ......................... 32

CHAPTER 5: THE CRYPTOGRAPHIC SYSTEM FRAMEWORK .......... 34
Design Overview ....................................................................... 34
Public Interfaces ...................................................................... 37
Configuration ........................................................................... 40
Secure Store Utilization .............................................................. 41

CHAPTER 6: CONCLUSION ........................................................ 51
Decreased Programming Complexity Across Implementations .......... 51
The Ease of Extending the Interface .............................................. 53
The Simplicity of HSM Initialization ............................................. 53
The Simplicity of Data Stores Initialization .................................. 54

APPENDIX – SOURCE CODE ...................................................... 56
ICryptoEngine Class ................................................................. 56
IStore Class ............................................................................. 57
StoreFactory Class ................................................................. 58
CryptoFactory Class ................................................................. 60
DesEncrypter Class ................................................................ 62
HSM Class .............................................................................. 64

REFERENCES .......................................................................... 72
LIST OF TABLES

Table 1. *Lines of Code Across Implementations* .............................. 52
LIST OF FIGURES

Figure 1. Software Based Cryptosystem ................................................................. 3
Figure 2. Dallas Java iButton™ ............................................................................. 4
Figure 3. Safenet Borderless Security iKey™ 2032 USB ........................................... 4
Figure 4. Hardware Based Cryptosystem ............................................................... 5
Figure 5. Relationship Between Confidentially, Integrity, and Availability .............. 8
Figure 6. Hash of plaintext ..................................................................................... 19
Figure 7. Message Authentication Code (MAC) lifecycle ....................................... 20
Figure 8. Hashing ................................................................................................. 23
Figure 9. Signing ................................................................................................. 24
Figure 10. Appending a signature ......................................................................... 24
Figure 11. Basic design for a hardware security module ........................................ 26
Figure 12. Power consumption of DES operation (Kocher, Jaffe and Jun 1999) ...... 31
Figure 13. A power analysis trace showing a RSA signature operation (Muir, 2001) 31
Figure 14. Secure Store Architecture .................................................................. 36
Figure 15. Secure Store Class Diagram .................................................................. 37
Figure 16. IStore Class Diagram .......................................................................... 38
Figure 17. ICryptoEngine Class Diagram .............................................................. 39
Figure 18. securestore.properties ........................................................................ 41
CHAPTER 1: INTRODUCTION

Information Security

Information security deals with several different "trust" aspects of information. Another common term is information assurance. Information security is not confined to computer systems, or to information in an electronic or machine-readable form. It applies to all aspects of safeguarding or protecting information or data, in whatever form.

The U.S. National Information Systems Security Glossary defines Information Systems Security (INFOSEC) as:

"The protection of information systems against unauthorized access to or modification of information, whether in storage, processing or transit, and against the denial of service to authorized users or the provision of service to unauthorized users, including those measures necessary to detect, document, and counter such threats." (National Security Agency, 2003)

Most definitions of information security tend to focus, sometimes exclusively, on specific usages and, or, particular media; e.g., "protect electronic data from unauthorized use". In fact it is a common misconception, or misunderstanding, that information security is synonymous with computer security—in any of its guises: computer and network security, information technology
IT) security, information systems security, information and communications technology (ICT) security. Each of these has a different emphasis, but the common concern is the security of information in some form (electronic in these cases): hence, all are subsets of information security. Conversely, information security covers not just information but all infrastructures that facilitate its use—processes, systems, services, technology, etc., including computers, voice and data networks, etc.

It is an important point that information security is, inherently and necessarily, neither watertight nor perfectible. No one can ever eradicate all risk of improper or capricious use of any information. The level of information security sought in any particular situation should be commensurate with the value of the information and the loss, financial or otherwise, that might accrue from improper use—disclosure, degradation, denial, or whatever. Information security is about risk management (Schneier, 2000).

Cryptographic Systems

Cryptographic systems can be implemented either by hardware or by software. In general, software implementations of cryptographic modules or security-related applications are known for being easier to develop and to maintain. However, software implementations are significantly less secure than their hardware equivalents. Because software solutions make use of shared memory space, they run on top of an operating system and are therefore more fluid in terms of ease of modification. Figure 1, depicts typical software based cryptosystem architecture.
Figure 1. Software Based Cryptosystem.

In all keyed systems, both symmetrical and asymmetrical, cryptographic implementation is critical. If the key is revealed, an interceptor can immediately decrypt all encrypted messages.

One way to achieve desired security is to store the key in a hardware security module (HSM). Such a device should preferably destroy its contents if ever opened, and be shielded against electromagnetic radiation attacks, a method of intercepting and copying emissions from an electronic device. The HSMs as shown in figures 2,3 are tiny computers complete with CPU and RAM.
Figure 2. Dallas Java iButton™.

Figure 3. Safenet Borderless Security iKey™ 2032 USB.

In addition, the use of hardware tokens mitigates the risks associated with developing software-only based cryptography solutions. Figure 4, depicts hardware based cryptosystem architecture.
Figure 4. Hardware Based Cryptosystem.

Purpose

The present need for security applications far exceed the number of individuals capable of designing secure systems. Of course, all applications have potential weak points and it is the responsibility of the organization using the software to assess the risks of using a particular design. Providing a framework for securing data is the primary motivation of this research. The design and cryptanalysis of cryptographic applications has been addressed by Clulow (2003) and Messerges, Dabbish, and Sloan (2002) with research in design and various security risks associated with using cryptographic hardware. While Clulow (2003) analyzes the various protocols, algorithms and cryptographic attacks in regards to cryptographic hardware, his detailed analysis is limited to the design of financial transaction system within the financial domain. Clulow (2003) does provide design solutions and criteria, but falls short of an implementation.
Because designs are abstract and open to interpretation of the software developer, most security failures are due to failures in implementation, not failure in algorithms or protocol. Thus, the importance of having a framework or architecture cannot be understated. Typically, it is the experience of many organizations that information security solutions are often designed, acquired, and installed on a tactical basis. A requirement is identified, a specification is developed and a solution is sought. In this process there is no opportunity to consider the strategic dimension, and the result is that the organization builds up a mixture of technical solutions on an ad hoc basis, each independently designed and specified and with no guarantee that they will be compatible and interoperable (Helmich, 2003). Helmich further states, the implementation of a secure framework provides the following:

- Ensures coherent, consistent security mechanisms, and standards to be followed in implementing security mechanisms.
- Provides a structure for the implementation or security mechanisms.
- Establishes the interfaces through which users and applications invoke security mechanisms.

The goal of this research is to provide a framework an Application Programming Interface (API) that can be used to develop secure applications using Hardware Security Modules (HSM) that meets the definition of a secure framework mentioned above by Helmich, but unlike Clulow (2003) it uses standardized, known technologies such as Java and PKCS#11 (RSA Laboratories, 2001). Providing a comprehensive API will enable application developers to implement HSM cryptography without extensive knowledge of the underlying cryptographic hardware.
CHAPTER 2: COMPUTER SECURITY

Security Goals

Computer security consists of maintaining three widely accepted goals of information security:

- Confidentiality
- Integrity
- Availability

Confidentiality means that the assets of a computing system are accessible only by authorized parties. The type of access is read-type access: reading, viewing, printing, or even just knowing the existence of an object (Pfleeger, 2002).

Integrity means that assets can be modified only by authorized parties or only in authorized ways. In this context, modification includes writing, changing, changing status, deleting, and creating (Pfleeger, 2002).

Availability means that assets are accessible to authorized parties. An authorized party should not be prevented from accessing objects to which he, she or it has legitimate access. For example, a security system could preserve perfect confidentiality by preventing everyone from

7
reading a particular object. However, this system does not meet the requirement of availability for proper access. Availability is sometimes known by its opposite, denial of service (Pfleeger, 2002).

These three goals make up security in computing. The three qualities can overlap, and they can even be mutually exclusive. For example, strong protection of confidentiality can severely restrict availability (Pfleeger, 2002). Computer security is often depicted as in Figure 5, which shows how these three properties are largely independent but sometimes overlapping.

![Figure 5. Relationship Between Confidentiality, Integrity, and Availability.](image)

The overall goal of this research is to provide a cryptosystem framework, which provides a balance of all three goals in computer security. Each component of the cryptosystem vulnerabilities (Hardware, Software, and Data) will be analyzed and the countermeasures will be applied to the framework.
**Vulnerabilities**

It is easier to consider vulnerabilities as they apply to the categories confidentiality, integrity, and availability as system resources (hardware, software, and data). These three assets and the interfaces between each, are all potential security weak points.

*Threats to Software.* Threats to software fall into three categories, deletion, and modification. Software deletion is surprisingly easy to do. Every programmer has at one time or another accidentally deleted files, current revisions of code, or overridden good code. Another way is someone without authorization to the computer simply and maliciously deletes the software. Software modification involves changing a working program to either fail in execution or perform unintended tasks. Some examples include:

- Trojan horses — a program that overtly does one thing while covertly doing another
- Viruses — a specific type of Trojan horse that can be used to spread infection from one computer to another.
- Trapdoors — a program that has a secret entry point
- Information leaks — a program, which makes information accessible to unintended people or programs.

*Threats to Hardware.* Threats to hardware are essentially deliberate attacks on any physical component (cables, motherboard, etc.) that would cause interruption service such as cutting power cables. The hardware can be stolen by removing it from the environment for future attacks such as unauthorized data extraction.
Threats to Data. Because of its visible nature, data attack is a more widespread and serious problem than either hardware or software attacks. Thus, data items have a greater public value than hardware and software, because more people know how to use or interpret data.

Data has essentially no intrinsic value. For this reason, it is hard to measure the value of data. However data has a cost, perhaps the cost of gathering or reconstruction lost data. Securing data varies according the value of the asset. Thus, the Principle of Adequate Protection (Pfleeger 2002) should be applied to ensure the security of data. This principle states computer items must be protected only until they lost their value. They must be protected to a degree consistent with their value.

Countermeasures

Physical Security. The most effective, and least expensive security measures are physical controls. Physical security includes locks on doors, guards at entry points, backup copies of important software and data, and physical site planning that reduces the risk of natural disasters. Often the simple physical controls are overlooked while more sophisticated approaches are sought (Pfleeger 2002).

Logical Security. Logical Security consists of software safeguards for an organization's systems, including user ID and password access, authentication, access rights and authority levels. These measures are to ensure that only authorized users are able to perform actions or access information in a network or a workstation.
CHAPTER 3: CRYPTOGRAPHY

History of Cryptography

For most of the history of cryptography, a key had to be kept absolutely secret and would be agreed upon beforehand using a secure; for example, a face-to-face meeting or a trusted courier. There are a number of significant practical difficulties in this approach to distributing symmetrical keys. For instance, the logistics of physical carrying a key is impractical and is likely to be handling insecurely. The Diffie-Hellman (D-H) key exchange addresses this problem. It is a cryptographic protocol, which allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher. Public key cryptography was invented to address these drawbacks with public key cryptography, users can communicate securely over an insecure channel without having to agree upon a shared key beforehand.

The first invention of an asymmetric key algorithm was by Clifford Cocks, then a recent mathematics graduate and a new staff member at GCHQ (Government Communications Headquarters) in the UK, early in the 1970s. This fact was kept secret until 1997.
The Cocks method was reinvented in 1977 by Rivest, Shamir and Adleman (Rivest, Shami, Adleman 1978) all then at MIT. The latter authors published their work in 1978, and the algorithm appropriately came to be known as RSA. RSA uses exponentiation modulo on a product of two large primes to encrypt and decrypt, performing both public key encryption and public key digital signature, and its security is based on the presumed difficulty of factoring large integers.

Since the 1970s, a large number and variety of encryption, digital signature, key agreement, and other techniques have been developed in the field of public key cryptography. The ElGamal cryptosystem (invented by Taher ElGamal then of Netscape) relies on the (similar, and related) difficulty of the discrete logarithm problem, as does the closely related DSA (Digital Signature Algorithm) developed by the NSA (National Security Agency) and NIST (National Institute of Standards and Technology). The Digital Signature Algorithm (DSA) is a United States Federal Government standard for digital signatures. The introduction of elliptic curve cryptography by Neal Koblitz in the mid '80s has yielded a new family of analogous public key algorithms. Although mathematically more complex, elliptic curves appear to provide a more efficient way to leverage the discrete logarithm problem, particularly with respect to key size.
Public Key Cryptography

With the growing need for information security in today's digital systems, cryptography has become one of its critical components. Public key cryptography is a form of cryptography, as previously explained, allows users to communicate securely without having prior access to a shared secret key. This is done by using a pair of cryptographic keys, designated as a public key and a private key, which are mathematically related. The term asymmetric key cryptography is a synonym for public key cryptography. In public key cryptography, the private key is generally kept secret, while the public key may be widely distributed. In a sense, one key "locks" a lock; while the other is required to unlock it. It should not be possible to deduce the private key of a pair given the public key. There are many forms of public key cryptography, including:

- Public key encryption - keeping a message secret from anyone that does not possess a specific private key.
- Public key digital signature - allowing anyone to verify that a message was created with a specific private key.
- Key agreement - generally, allowing two parties that may not initially share a secret key to agree on one.

Each of these forms of public key cryptography is further described in the following next sections.
Encryption

In general, encryption is the process of obscuring information to make it unreadable without special knowledge. While encryption has been used to protect communications for centuries, only organizations and individuals with an extraordinary need for secrecy had made use of it. In the mid-1970s, strong encryption emerged from the sole preserve of secretive government agencies into the public domain, and is now employed in protecting widely-used systems, such as Internet e-commerce, mobile telephone networks and bank automatic teller machines (ATMs).

Public Key Encryption

Public key encryption algorithms are based on the premise that each sender and recipient has a private key, known only to him/her and a public key, which can be known by anyone. Each encryption/decryption process requires at least one public key and one private key. Generally speaking, a key is a randomly generated set of numbers/characters that is used to encrypt/decrypt information. A public key encryption scheme has six major parts:

- Plaintext - this is the text message to which an algorithm is applied.
- Encryption Algorithm - it performs mathematical operations to conduct substitutions and transformations to the plaintext.
- Public and Private Keys - these are a pair of keys where one is used for encryption and the other for decryption.
- Cipher text - this is the encrypted or scrambled message produced by applying the algorithm to the plaintext message using key.
• Decryption Algorithm - This algorithm generates the cipher text and the matching key to produce the plaintext.

*Encryption Algorithms*

Different encryption algorithms use proprietary methods of generating these keys and are therefore useful for different applications. Here are details about some of these encryption algorithms. The key length used by the algorithm often discerns strong encryption.

**RSA**. In 1977, shortly after the idea of a public key system was proposed, three mathematicians, Ron Rivest, Adi Shamir and Len Adleman gave a concrete example of how such a method could be implemented. The method was referred to as the RSA Scheme. The system uses a private and a public key. To start two large prime numbers are selected and then multiplied together; \( n = p \times q \).

If we let \( f(n) = (p-1) \times (q-1) \), and \( e > 1 \) such that \( \text{GCD}(e, f(n)) = 1 \). Here \( e \) will have a fairly large probability of being co-prime to \( f(n) \), if \( n \) is large enough and \( e \) will be part of the encryption key. If we solve the Linear Diophantine equation; \( ed \) congruent \( 1 \) (mod \( f(n) \)), for \( d \). The pair of integers \( (e, n) \) are the public key and \( (d, n) \) form the private key with this, the encryption of \( M \) can be accomplished by the following expression; \( Me \equiv qn + C \) where \( 0 \leq C < n \). Decryption would be the inverse of the encryption and could be expressed as; \( Cd \) congruent \( R \) (mod \( n \)) where \( 0 \leq R < n \). RSA is the most popular method for public key encryption and digital signatures today.
To illustrate the RSA algorithm, look at the following example with small numbers. Choose public exponent $e = 3$. Then, let $p = 5$ and $q = 11$, which means $n = 55$ and $(p - 1)(q - 1) = 40$. This is a valid $p$ and $q$ combination since 3 is relatively prime with 40. The inverse of 3 modulo 40 is 27 (computed using the extended Euclidian algorithm $d = 27$) is the private exponent.

Apply the RSA algorithm with these parameters to the “plaintext message” of 14.

$$c = m^e = 14^3 = 2744$$

$$2744 - (49 \times 55) = 2744 - 2695 = 49 \mod 55$$

Now compute the inverse.

$$c^d = (2744^{27}) \mod 55 = 14$$

**DES/3DES.** The Data Encryption Standard (DES) was developed and endorsed by the U.S. government in 1977 as an official standard and forms the basis not only for Automatic Teller Machines (ATM) PIN authentication but a variant is also utilized in UNIX password encryption. DES is a block cipher with a 64-bit block size that uses 56-bit keys. Due to recent advances in computer technology, some experts no longer consider DES secure against all attacks; since then Triple-DES (3DES) has emerged as a stronger method. Using standard DES encryption, Triple-DES encrypts data three times and uses a different key for at least one of the three passes giving it a cumulative key size of 112-168 bits.

**IDEA.** The International Data Encryption Algorithm (IDEA) is an algorithm that was developed by Dr. X. Lai and Prof. J. Massey in Switzerland in the early 1990s to replace the DES standard. It uses the same key for encryption and decryption, like DES operating on 8
bytes at a time. Unlike DES though it uses a 128 bit key. This key length makes it impossible to break by simply trying every key, and no other means of attack is known. It is a fast algorithm, and has also been implemented in hardware chipsets, making it even faster.

**BLOWFISH.** Blowfish is a symmetric block cipher just like DES or IDEA. It takes a variable-length key, from 32 to 448 bits, making it ideal for both domestic and exportable use. Bruce Schneier designed Blowfish in 1993 as a fast, free alternative to the then existing encryption algorithms. Since then Blowfish has been analyzed considerably, and is gaining acceptance as a strong encryption algorithm.

**SEAL.** Rogaway and Coppersmith designed the Software-optimized Encryption Algorithm (SEAL) in 1993. It is a stream-cipher, i.e., data to be encrypted is continuously encrypted. Stream Ciphers are much faster than block ciphers (Blowfish, IDEA, and DES) but has a longer initialization phase during which large sets of tables are generated using the Secure Hash Algorithm. SEAL uses a 160-bit key for encryption and is considered very safe.

**RC4.** RC4 is a cipher invented by Ron Rivest, co-inventor of the RSA Scheme. It is used in a number of commercial systems like Lotus Notes and Netscape Navigator. It is a cipher with a key size of up to 2048 bits (256 bytes), which on the brief examination over the past year or so seems to be a relatively fast and strong cipher. It creates a stream of random bytes and 'XOR' those bytes with the plain text. It is useful in situations in which a new key can be chosen for each message.
Hash Algorithms

In cryptography, a cryptographic hash function is a hash function with certain additional security properties to make it suitable for use as a primitive in various information security applications, such as authentication and message integrity. A hash function takes a long string (or message) of any length as input and produces a fixed length string as output, sometimes termed a message digest or a digital fingerprint.

In various standards and applications, the two most-commonly used hash functions are MD5 and SHA-1; however, as of 2005, security flaws have been identified in both algorithms.

MD5. MD5 (Message-Digest algorithm 5) is a widely-used cryptographic hash function with a 128-bit hash value. As an Internet standard (RFC 1321), MD5 has been employed in a wide variety of security applications, and is also commonly used to check the integrity of files.

MD5 was designed to replace an earlier hash function, MD4. In August 2004, researchers reported generating collisions in MD4 using "hand calculation" Wang, Feng, Lai and Yu (2004). In 1996, similar flaws were found with the design of MD5; while it was not a clearly fatal weakness, cryptographers began to recommend using other algorithms, such as SHA-1 (recent claims suggest that SHA-1 has been broken, however). In 2004, more serious flaws were discovered making further use of the algorithm for security purposes questionable. Figure 6, depicts a hash operation on plaintext.
SHA-1. The SHA (Secure Hash Algorithm) family is a set of related cryptographic hash functions. The most commonly used function in the family, SHA-1, is employed in a large variety of popular security applications and protocols, including TLS, SSL, PGP, SSH, S/MIME, and IPSec. SHA-1 is considered to be the successor to MD5, an earlier, widely-used hash function. The SHA algorithms were designed by the National Security Agency (NSA) and published as a US government standard.

The first member of the family, published in 1993, is officially called SHA; however, it is often called SHA-0 to avoid confusion with its successors. Two years later, SHA-1, the first successor to SHA, was published. Four more variants have since been issued with increased output ranges and a slightly different design: SHA-224 bits, SHA-256 bits, SHA-384 bits, and SHA-512 bits — sometimes collectively referred to as SHA-2.

Attacks have been found for both SHA-0 and SHA-1. No attacks have yet been reported on the SHA-2 variants, but since they are similar to SHA-1, researchers are worried, and are developing candidates for a new, better hashing standard.
Message Authentication Code

A message authentication code (MAC) is an authentication tag (also called a checksum) derived by applying an authentication scheme, together with a secret key, to a message. Unlike digital signatures, Macs' are computed and verified with the same key, so that they can only be verified by the intended recipient. Figure 7, depicts a MAC lifecycle.

Figure 7. Message Authentication Code (MAC) lifecycle.

Digital Signatures

A digital signature (or public-key digital signature) is a type of method for authenticating digital information analogous to ordinary physical signatures on paper, but implemented using techniques from the field of public-key cryptography. A digital signature method generally defines two complementary algorithms, one for signing and the other for verification. The output of the signing process is also called a digital signature. There are three common reasons for applying a digital signature to communications:
• Authenticity - Public-key cryptosystems allow anybody to send a message using the public key. A signature allows the recipient of a message to be confident that the sender is indeed who she/he claims to be. Of course the recipient cannot be 100% sure that the sender is indeed who she/he claims to be - the recipient can only be confident - since the cryptosystem may have been broken. The importance of authenticity is especially obvious in a financial context. For example, suppose a bank sends instructions from its branch offices to the central office in the form \((a, b)\) where \(a\) is the account number and \(b\) is the amount to be credited to the account. A devious customer may deposit $100, observe the resulting transmission and repeatedly retransmit \((a, b)\). This is known as a replay attack. Including a timestamp in the document and digitally signing it can prevent replay attacks. If the timestamp is not within a threshold of the receiving system it will be rejected.

• Integrity - Both parties will always wish to be confident that a message has not been altered during transmission. The encryption makes it difficult for a third party to read a message, but that third party may still be able to alter it in a useful way. A popular example to illustrate this is the homomorphism attack: consider the same bank as above which sends instructions from its branch offices to the central office in the form \((a, b)\) where \(a\) is the account number and \(b\) is the amount to be credited to the account. A devious customer may deposit $100, intercept the resulting transmission and then transmit \((a, b^3)\) to become an instant millionaire. The integrity of a document can be verified by decrypting the signature with the public key resulting in the initial hash \((h_1),\)
hashing the document with the same hashing algorithm (h2) and comparing them. This verification operation is illustrated in Figure 10.

- Non-repudiation - In a cryptographic context, the word repudiation refers to the act of denying association with a message (i.e. claiming it was sent by a third party). The recipient of a message may insist that the sender attach a digital signature in order to prevent any later repudiation, since the recipient may show the message to a third party to prove its origin.

_Digital Signature Scheme_

Digital signature schemes rely on public-key cryptography. In public-key cryptography, each user has a pair of keys: one public and one private. The public key is distributed freely, but the private key is kept secret and confidential; another requirement is that it should be infeasible to derive the private key from the public key. A general digital signature scheme consists of three algorithms:

- A key generation algorithm
- A signing algorithm
- A verification algorithm

For example, Bob has been given two keys. One of Bob's keys is called a Public Key, the other is called a Private Key. With his private key, Bob can put digital signatures on documents and other data. A digital signature is a "stamp" Bob places on the data which is unique to Bob, and is very difficult to forge. In addition, the signature assures that any changes made to the data that
has been signed cannot go undetected. To sign a document, Bob's software will crunch down the
data into just a few lines by a process called "hashing" (Figure 8). These few lines are called a
message digest. (It is not possible to change a message digest back into the original data from
which it was created.)

![Diagram showing hashing process]

*Figure 8. Hashing.*

Bob's software then encrypts the message digest with his private key. The result is the
digital signature (Figure 9).
Finally, Bob's software appends the digital signature to the document. All of the data that was hashed has been signed. Bob now passes the document on to Pat. First, Pat's software decrypts the signature (using Bob's public key) changing it back into a message digest. If this worked, then it proves that Bob signed the document, because only Bob has his private key. Pat's software then hashes the document data into a message digest. If the message digest is the same as the message digest created when the signature was decrypted, then Pat knows that the signed data has not been changed. In other words, it is verified (Figure 10).

**Figure 9.** Signing.

**Figure 10.** Appending a signature.
CHAPTER 4: HARDWARE SECURITY MODULES

A Hardware Security Module (HSM) is defined as a piece of hardware and associated software/firmware that usually attaches to the inside of a PC or server and provides at least the minimum of cryptographic functions. These functions include (but are not limited to) encryption, decryption, key generation, and hashing. The physical device offers some level of physical tamper-resistance and has a user interface and a programmable interface.

Other names for an HSM include Personal Computer Security Module (PCSM), Secure Application Module (SAM), Hardware Cryptographic Device or Cryptographic Module. A simple design for an HSM consists of the following basic components:

- Communications interface
- Central processing unit (CPU)
- Cryptographic accelerators
- General purpose memory
- Secure memory

Figure 11, models these components mentioned above.
Figure 11. Basic design for a hardware security module.

**HSM Functionality**

An HSM can perform a number of important security-related functions. It provides accelerated cryptographic operations such as encryption, digital signatures, hashing, and Message Authentication Codes. A Message Authentication Code (MAC) is an algorithm that mathematically combines a key with a hash to provide a “code” that can be appended with a given piece of data to ensure its integrity.

For example, suppose a database contains a list of account balances. It is very desirable from a security perspective to be able to prevent an unauthorized person from manually changing these values. Therefore, when an authorized entry is made, the HSM would provide an interface to MAC the input value that would be contained within the record itself. Because the HSM maintains the key that formulates the MAC, nobody else can theoretically reproduce a valid...
MAC for a given account balance. So when an authorized program retrieves the database value, the data provider would automatically ask the HSM to verify that the MAC for the value is correct. If the MAC verification fails, the program would know that the data has been tampered with and can perform the appropriate action such as auditing, logging, generating alarms, etc.

Another important function of an HSM is key management. With any type of system that uses cryptographic keys, it is imperative that the tools that generate, backup and hold these keys do so in a secure manner. To be optimally secure, the HSM should store all of the keys on the physical device itself. The key backups should be done using a secure connection to another HSM or to one or more smart cards (preferably more than one). The card reader should attach directly to the HSM to prevent the data from being intercepted.

*Common Implementations of an HSM*

An HSM has a number of different uses. The functionality and security vary with price. Generally HSMs are implemented for the following uses:

- The key generator and safe key storage facility for a certificate authority.
- A tool to aid in authentication by verifying digital signatures.
- An accelerator for SSL connections.
- A tool for securely encrypting sensitive data for storage in a relatively non-secure location such as a database.
- A tool for verifying the integrity of data stored in a database.
- A secure key generator for smartcard production.
Typically, an HSM is installed inside a server box or within an Ethernet cluster within your architecture. The HSM is "wrapped" by your company’s software, the vendor’s software, a third party’s software, or a combination of the three. It is this software that provides access to the cryptographic functionality provided within the HSM. Ideally, the HSM will conform to PKCS #11, a standard that outlines the programmatic interface that the HSM supports. This standard is available online from RSA’s web site at

http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/index.html (RSA 2001) and will be discussed in detail in the following section.

Positive Attributes of an HSM

The following attributes are actually desirable from a security perspective:

Federal Information Processing Standards (FIPS). FIPS 140-1 or 140-2 is a widely known standard provides four well-defined levels for validating HSMs (National Institute of Standards and Technology, December 2001). This validation does not mean that the product is perfect. However, if it is validated then there is at least a reasonable baseline of security tests performed on the HSM by qualified professionals at FIPS accredited testing facilities. Therefore, an important aspect of choosing an HSM comes down to understanding the FIPS certification levels and weighing the costs of these levels versus the value of what is being secured. It is important to distinguish between vendors that claim FIPS 140 “compliance” versus “validation” since any vendor can claim that their product is compliant (National Institute of Standards and Technology, May 2001).
Secure cryptographic algorithms. The use of widely accepted and open secure cryptographic algorithms is preferred. Many vendors’ products will offer secret proprietary algorithms. It is preferable to stay away from these (Schneier, 1999). For digital signatures, look for RSA or DSA based algorithms. For encryption, look for 3-DES or another well-known and secure algorithm. For hashing, look for SHA-1 or MD5 (Reed, 2002). Some offer proprietary algorithms in addition to these open standard ones. It is important that the HSM is properly configured not to use the proprietary algorithms in this case.

Strong random number generation. The random number generation (RNG) or pseudo-random number generation is critical to many cryptographic functions including key generation. If the RNG is weak, the entire product is cryptographically unsound (Anderson, 1998).

Standardized interface. A company looking to purchase an HSM needs to consider the complexity of their cryptographic needs. If they have more than basic needs, then products that conform to PKCS #11 will offer a good industry-accepted standard. This is the “Cryptographic Token Interface Standard” (RSA Laboratories 2001), which defines how software will interface with the cryptographic functions specified by the device.
Drawbacks to Using HSMs

Vendors typically withhold a lot of information about how their security products work. HSM device are no exception to this. For example, while there are guidelines available for implementing and testing random number generators (RNGs), most vendors simply specify their RNG capability as “true”, “strong”, or “hardware-based”. Part of the problem is that there is currently no sufficient standard for randomness. The following describes attacks that could be done on HSM devices.

Leaking Secrets. As early as the 50’s and 60’s, unintentional emissions were being used to compromise cryptographic equipment. Former MI5 agent Peter Wright details an attempt to monitor French communication (Wright, 1987). The initial attempts to break the French diplomatic cipher failed. However, it was noticed that a faint secondary signal was carried along with the normal enciphered traffic. It turned out that this was the plaintext that had somehow leaked through the cipher machine! The prevention of ‘compromising emissions’ became an important (and classified) subject within the defense industry, leading to the creation of the TEMPEST program and requirements for TEMPEST shielded devices within the government and defense industries.

Power Analysis. Power analysis was introduced by (Kocher, Jaffe and Jun 1999) as a side channel attack, which exploited the power consumption of a cryptographic device. It received widespread attention largely due to the very real threat it posed to HSM devices. Power analysis is effectively a visual analysis of a plot of the power consumption of the device during operation. Consider the graph in Figure 12 below, which (Kocher, Jaffe and Jun 1999) shows
the power consumption during a DES operation. Key features of the algorithm such as the 16 rounds are clearly evident.

Figure 12. Power consumption of DES operation (Kocher, Jaffe and Jun, 1999).

Consider the partial trace of an RSA signature operation shown in Figure 13.

Figure 13. A power analysis trace showing a RSA signature operation (Muir, 2001).

Clearly identifiable are the power spikes during either a squaring or multiplication operation. In fact the multiplication operations are distinguishable from the squaring operations owing to a wider spike as a result of extra register loads (a multiplication operation has 2 operands and a modulus compared to the squaring operation’s single operand and modulus).
Thus we can identify the following sequence of operations – squaring, squaring, squaring, multiplication, squaring, multiplication, squaring, multiplication and finally squaring. Realizing that this is obviously an implementation of the square-and-multiply algorithm where the multiplication operation is conditional on the value of the associated exponent bit. As a result, one can identify the exponent as containing the bit sequence \{1, 1, 1, 0, 0\} (Clulow, 2003).

The nature of these attacks is very complex and requires sophisticated equipment. They also require the attacker be in the vicinity of the HSM. These attacks can easily be counter-measured by implementing an environmental security policy.

*Public Key Cryptography Standards #11 (PKCS#11)*

As cryptography begins to see wide application and acceptance, one thing is increasingly clear: if it is going to be as effective as the underlying technology allows it to be, there must be interoperable standards. Even though vendors may agree on the basic cryptographic techniques, compatibility between implementations is by no means guaranteed. Interoperability requires strict adherence to agreed-upon standards.

Towards that goal, RSA Laboratories has developed, in cooperation with representatives of industry, academia and government, a family of standards called Public-Key Cryptography Standards, or PKCS for short. PKCS is offered by RSA Laboratories to developers of computer systems employing public-key and related technology. It is RSA Laboratories' intention to improve and refine the standards in conjunction with computer system developers, with the goal of producing standards that most if not all developers adopt.
The PKCS#11 standard specifies an API, called Cryptoki. Cryptoki, pronounced “crypto-key” and short for cryptographic token interface, follows a simple object-based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the device called a cryptographic token.

Cryptoki isolates an application from the details of the cryptographic device. The application does not have to change to interface to a different type of device or to run in a different environment; thus, the application is portable.
CHAPTER 5: THE CRYPTOGRAPHIC SYSTEM FRAMEWORK

At this point, it is worth reiterating the goal of this research. This research is to provide a framework; an Application Programming Interface (API) that can be used to develop secure applications using Hardware Security Modules (HSM) that uses standardized, known technologies such as Java and PKCS#11. Providing a comprehensive API will enable application developers to implement HSM cryptography without extensive knowledge of the underlying cryptographic hardware. The following section describes the software architecture used to meet this goal.

*Design Overview*

The implementation of a secure storage application referred to as Secure Store has been developed. The Secure Store provides, as its name implies, a location to which data can be stored with protection from both viewing and tampering. While it is primarily intended for the storage of passwords, keys, or other sensitive security related items, it is not limited to such. Data items stored in the Secure Store are identified by an alias specified by the client application. The protection of the data is supplied through the use of standard cryptographic practices, which includes digitally signing data and then encrypting it prior to storage in a database.
As shown in Figure 14, the Secure Store is comprised of four primary components, the Secure Store API (Client Interface), the Storage Manager (Controller), the Storage API (Data Store), and the Crypto API (Hardware Interface). Each of the components publishes client interfaces. In the case of the Storage Manager, the interface is truly meant for client applications to use, while the intention of the interfaces published by the Secure Store and Crypto Engine are for the Storage Manager. It is important to note that this separation has been designed into the system to allow both the Storage component and the Crypto Engine to be specified at runtime. Furthermore, this allows replacement of either without modification to the Storage Manager or client application.
Figure 14. Secure Store Architecture.
Secure Store Class Diagram

Figure 15. Secure Store Class Diagram.

Public Interfaces

Secure Store API. The purpose of the Secure Store API is to abstract the implement of the persistence layer. The storage mechanism used is not known to the client and is loaded at runtime. The types of storage mechanisms that can be used are RDBMS, LDAP, XML, and file based. Each of these storage mechanisms would have to implement the IStore interface class to compile with the design specification.
IStore Interface Class

<table>
<thead>
<tr>
<th>IStore</th>
</tr>
</thead>
<tbody>
<tr>
<td>+setData(in key : string, in value : string) : void</td>
</tr>
<tr>
<td>+getData(in key : string) : string</td>
</tr>
<tr>
<td>+getDesc() : string</td>
</tr>
<tr>
<td>#setDesc(in desc : string) : void</td>
</tr>
</tbody>
</table>

*Figure 16. IStore Class Diagram.*

- public abstract void setData(String key, String value)
  This method stores data using a key, value pair.
- public abstract String getData(String key)
  This method returns data using a key.
- public String getDesc()
  This method returns a description of the persistence layer used.
- protected abstract String getDesc()
  This method returns a description of the persistence layer used
ICryptoEngine Interface Class

<table>
<thead>
<tr>
<th>ICryptoEngine</th>
</tr>
</thead>
<tbody>
<tr>
<td>+verify(in b : byte, in signature : byte) : bool</td>
</tr>
<tr>
<td>+sign(in b : byte) : byte</td>
</tr>
<tr>
<td>+encrypt(in b : byte) : byte</td>
</tr>
<tr>
<td>+decrypt(in b : byte) : byte</td>
</tr>
<tr>
<td>+getDesc() : string</td>
</tr>
<tr>
<td>+hash(in b : byte, in algorithm : string) : byte</td>
</tr>
<tr>
<td>+generateKeyPair(in algorithm : string, in keysize : int) : void</td>
</tr>
<tr>
<td>+printMechanisms(in p) : void</td>
</tr>
<tr>
<td>#setDesc(in desc : string) : void</td>
</tr>
</tbody>
</table>

Figure 17. ICryptoEngine Class Diagram.

- public abstract boolean verify(byte[] b, byte[] signature)

  Performs a verify operation

- public abstract byte[] sign(byte[] b)

  Performs a signing operation.

- public abstract byte[] encrypt(byte[] b)

  Performs an encryption operation

- public abstract byte[] decrypt(byte[] b)

  Performs a decryption operation

- public String getDesc()

  Returns a description of the implementation

- public abstract byte[] hash(byte[] b, String algorithm)

  Performs a hash operation given an algorithm

- public abstract void generateKeyPair(String algorithm, int keysize)

  Generates a key pair given an algorithm and key size
• public abstract void printMechanisms(Provider p)
  Lists all the mechanism available given a provider. This is helpful in discovering the
  underlying availability of operations given a Provider.

• protected void setDesc(String desc)
  This method returns a description of the persistence layer used

_Configuration_

The Secure Store uses a property files to dynamic load IStore and ICryptoEngine classes
at runtime.

_Engine Configuration._ The ‘engine’ property specifies which ICryptoEngine class to be
used. For example, _engine=HardwareToken_

_Store Configuration._ The ‘store’ property specifies which IStore class to be used.
For example, _store=textfile!C:/eclipse/workspace/Thesis/mydata.txt_

_HSM Configuration._ The ‘hsmpin’ property specifies the pin to be used by the HSM
during login. For example, _hsmpin=PASSWORD_

_Java Cryptographic Provider Configuration._ The ‘alias’ property is used by a software
version of ICryptoEngine and it defines what alias to use in the Java Keystore. For example,
(alias=mykey). Figure 18, depicts a typical securestore.properties file.
engine=HardwareToken
store=textfile!C:/eclipse/workspace/Thesis/mydata.txt
hsmpin=PASSWORD
alias=mykey

Figure 18. securestore.properties.

Secure Store Utilization

The system utilization section describes in detail the processes that comprise the Secure Store system.

Secure Store Initialization. The initialization process begins with the ‘application’ or client creating an instance of CryptoFactory and StoreFactory. The application then requests an instance of a ICryptoEngine and IStore from both CryptoFactory and StoreFactory (factories). The factories are responsible for dynamically loading the appropriate implementations of IStore and ICryptoEngine. The advantage of this design is that the application or client is not aware of any concrete classes or how to initialize them. The client is unaware of the type of storage or crypto engine. This works because all concrete implementations of a store or crypto engine must implement ICryptoEngine and IStore respectively. Figure 19, depicts a UML sequence diagram of the initialization process.
Figure 19. Secure Store Initialization.

Message Digest / Hashing. Performing a message digest operation requires a method call to ICryptoEngine.hash(byte[] plaintext, String algorithm). The first argument is the plaintext in bytes and the second argument is your digest algorithm of choice. Figure 20, depicts a UML sequence diagram of a message digest/hashing process.
Figure 20. Secure Store Message Digest/Hashing.

**Digital Signatures.** Performing a digital signature operation requires a method call to ICryptoEngine.sign(byte[] plaintext, String algorithm). The first argument is the plaintext in bytes and the second argument is your signature algorithm of choice. Figure 21, depicts a UML sequence diagram of a digital signature process.
Figure 21. Secure Store Digital Signatures.

Verifying Signatures. Performing verification of a signature operation requires a method call to ICryptoEngine.verify(byte[] plaintext, byte[] signature, String algorithm). The first argument is the plaintext in bytes, the second is the signature in bytes, and finally the algorithm to be used. Figure 22, depicts a UML sequence diagram of a verifying signature process.
Figure 22. Secure Store Verifying Signatures.

**Encrypting Data.** Performing an encrypting data operation requires a method call to ICryptoEngine.encrypt(byte[] plaintext). The first argument is the plaintext in bytes. Figure 23, depicts a UML sequence diagram of an encrypting data process.

Figure 23. Secure Store Encrypting Data.
Decrypting Data. Performing a decrypting data operation requires a method call to `ICryptoEngine.decrypt(byte[] ciphertext)`. The first argument is the cipher text in bytes. Figure 24, depicts a UML sequence diagram of a decrypting data process.

![UML Sequence Diagram](image)

*Figure 24. Secure Store Decrypting Data.*

Generating a Key Pair. Generating key pair operation requires a method call to `ICryptoEngine. generateKeyPair(String algorithm, int keysize)`. The first argument is the algorithm and the second is the key size. Figure 25, depicts a UML sequence diagram of a generating key pair process.
Figure 25. Secure Store Generate Key Pair.

**Saving Data.** Performing a set data operation requires a method call to IStore.setData(String key, String value). The first argument is the key and the second is the value. Figure 26, depicts a UML sequence diagram of a saving data process.

Figure 26. Secure Store Saving Data.
**Getting Data.** Performing a set data operation requires a method call to 
IStore.getData(String key). The first argument is the key. Figure 27, depicts a UML sequence 
diagram of a getting data process.

![UML sequence diagram](image)

**Figure 27.** Secure Store Getting Data.

**Removing Data.** Performing a remove data operation requires a method call to 
IStore.removeData(String key). The first argument is the key. Figure 28, depicts a UML 
sequence diagram of a removing data process.
Figure 28. Secure Store Removing Data.

Modify Data. Performing a modify data operation requires a method call to
IStore.setData(String key, String value). The first argument is the key and the second is the
value. Figure 29, depicts a UML sequence diagram of a modify data process.
Figure 29. Secure Store Modify Data.
CHAPTER 6: CONCLUSION

The main advantages of using the Secure Store framework are:

- The reduction of programming complexity resulting from a well designed and
documented abstract interface.
- The increased readability of source code by using known design patterns.
- The ease of extending the interface with additional properties and methods.
- The simplicity of HSM initialization.
- The simplicity of data store initialization.
- The usage of the Singleton design pattern to create an instance of both the
ICryptoEngine and IStore to enhance performance.

Decreased Programming Complexity Across Implementations

Table 1, shows the number of lines of code needed to perform the same
operations across implementations. It clearly illustrates the decrease of complexity when using
the Secure Store framework.
Table 1

*Lines of Code Across Implementations*

<table>
<thead>
<tr>
<th>Operation</th>
<th>Lines of code – Secure Store Framework</th>
<th>Lines of code – Native Java Support using PKCS#11</th>
</tr>
</thead>
<tbody>
<tr>
<td>verify</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>sign</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>encrypt</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>decrypt</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>hash</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>generateKeyPair</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>printMechanisms</td>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>

Reduction of complexity has the additional benefit of increased code readability. Figure 30 illustrates this point.
```java
public static void main(String[] args) {
    try {
        String plaintext = "Some plain text";

        //Create a Crypto Factory
        CryptoFactory cf = new CryptoFactory();
        //Create a Store Factory
        StoreFactory sf = new StoreFactory();

        //Initialize Crypto Engine
        ICryptoEngine c = cf.getCryptoEngine();
        //Initialize Store
        IStore s = sf.getStore();

        //Perform hash
        byte[] hashedText = c.hash(plaintext.getBytes(), "SHA1");

        //Perform signature
        byte[] signedHashedText = c.sign("SHA1withRSA", hashedText);

        //Save data
        s.setData("1", new String(signedHashedText));

        //Get data key 1 and decrypt
        c.decrypt(s.getData("1").getBytes());

        //Verify the signature
        boolean isVerified = c.verify(hashedText, signedHashedText, "SHA1withRSA");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure 30. Code Snippet.

The Ease of Extending the Interface

All good object oriented designs allow for extensibility and the Secure Store framework is no different. Adding additional methods to the interfaces is simple. All implementation classes provide this functionality.

The Simplicity of HSM Initialization

The Secure Store initialization of the HSM is done more securely by managing the secret PIN on behalf of the client. The initialization of the HSM is costly; therefore a single instance is used. This is a common design pattern known as a ‘Singleton’. 

53
The Simplicity of Data Stores Initialization

The Secure Store initialization of the data store is also abstracted from the client API. The client can then easily make storage calls to initialize the storage component.

The Secure Store framework API provides basic functionality for developing secure applications. As previously mentioned, one of the major benefits of using such an API is its extensibility. This API can be easily extended to allow for future enhancements such as bootstrapping, PIN management, and security policy specification. Refer to Appendix – Source Code for complete code implementation.

Bootstrapping is a common problem with applications that require credentials prior to initialization. In the case of the Secure Store, a PIN is required to start the ICryptoEngine. The problem resides in the management of the PIN. By design, initializing the application requires an administrator to provide credentials. If the user mismanages the PIN, the security of the system can be compromised. Ultimately, a security policy procedure with strong physical security will address this problem.

Currently, there are no provisions for managing the PIN programatically. Future API enhancements could include secure PIN management.
It would be advantageous to provide capabilities to change cryptographic algorithms, key size and cipher suites at runtime. This, if used properly, would allow for more flexibility and/or increased security of the encryption.
APPENDIX – SOURCE CODE

ICryptoEngine Class

/* Created on Mar 10, 2005
 * Vico Minnoci
 */
package net.vicominnocci.security.crypto;

import java.security.Provider;

public abstract class ICryptoEngine {
    public String desc;
    public abstract boolean verify(byte[] b, byte[] signature, String algorithm);
    public abstract byte[] sign(String algorithm, byte[] b);
    public abstract byte[] encrypt(byte[] b);
    public abstract byte[] decrypt(byte[] b);
    public String getDesc(){return desc;}
    public abstract byte[] hash(byte[] b, String algorithm);
    public abstract void generateKeyPair(String algorithm, int keysize);
    public abstract void printMechanisms(Provider p);
    protected void setDesc(String desc){this.desc=desc;}
}
**IStore Class**

/* Created on Sep 22, 2005  
* Vico Minnocci  
*/

package net.vicominnocci.securestore.storage;


public abstract class IStore {
    public String desc;  
    public abstract void setData(String key, String value);  
    public abstract String getData(String key);  
    public String getDesc(){return desc;}  
    public abstract void removeData(String key);  
    protected void setDesc(String desc){this.desc=desc;}
}


StoreFactory Class

/* Created on Sep 22, 2005
 * Vico Minnocci
 */
package net.vicominnocci.securestore.storage;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class StoreFactory {
    static final String _TEXT_FILE = "textfile";
    static final String _DB = "database";
    String [] store;
    IStore cStore;
    /* (non-Javadoc)
     * @see net.vicominnocci.securestore.AbstractCryptoFactory#getCryptoFactory()
     */
    public StoreFactory(){
        Properties properties = new Properties();
        try {
            properties.load(new FileInputStream("C:/eclipse/workspace/Thesis/securestore.properties");
            String temp = properties.getProperty("store");
            store = temp.split("!");
            setIStore(store);
        } catch (IOException e) {
            System.out.println(e);
            System.exit(0);
        }
    }
    private void setIStore(String[] store){
        try {
            cStore = stringMappingImpl(store);
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    public IStore getStore() throws Exception {
        // TODO Auto-generated method stub
    }
}
return stringMappingImpl(store);
}
private IStore stringMappingImpl(String[] sArgs) throws
Exception{
    // examine the config to extract the db identifier
    if (sArgs[0].equalsIgnoreCase(_TEXT_FILE)) {
        return new TextFile(store[1]);
    }
    else {
        throw new IllegalArgumentException("Unknown crypto engine
        identifier. " + sArgs[0]);
    }
}
/* (non-Javadoc)
 * @see
net.vicominnocci.securestore.AbstractCryptoFactory#getCryptoFact
ory()
 */
/**
 * @param string
 * @return
 */
private IStore TextFile(String string) {
    // TODO Auto-generated method stub
    return null;
}
CryptoFactory Class

/* Created on Mar 13, 2005
 * Vico Minnoci
 */
package net.vicominnoci.securestore.crypto;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class CryptoFactory{
    static final String JAVA_CRYPTO = "SoftwareBased";
    static final String SMART_CARD = "HardwareToken";

    private static String engine, filename, pin, alias;

    ICryptoEngine cEngine;

    public CryptoFactory(){
        Properties properties = new Properties();
        try {
            properties.load(new FileInputStream("C:/eclipse/workspace/Thesis/secu" +
                          "restore.properties"));

            engine = properties.getProperty("engine");
            pin = properties.getProperty("hsmpin");
            alias = properties.getProperty("alias");
            try {
                setCryptoEngine(engine);
            } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        } catch (IOException e) {
            System.out.println(e);
            System.exit(0);
        }
    }

    /* (non-Javadoc)
     * @see net.vicominnoci.securestore.AbstractCryptoFactory#getCryptoFacto" +
    }
public ICryptoEngine getCryptoEngine() throws Exception {
    // TODO Auto-generated method stub
    return cEngine;
}

private ICryptoEngine stringMappingImpl(String sCryptoFactory) throws Exception{
    //examine the config to extract the db identifier
    if (sCryptoFactory.equalsIgnoreCase(_JAVA_CRYPTO)) {
        return JavaCrypto.getInstance();
    } else if (sCryptoFactory.equalsIgnoreCase(_SMART_CARD)) {
        return HSM.getInstance(pin, alias);
    } else {
        throw new IllegalArgumentException("Unknown crypto engine identifier. "+sCryptoFactory);
    }
}

private void setCryptoEngine(String engine){
    try {
        cEngine = stringMappingImpl(engine);
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

/* (non-Javadoc)
 * @see net.vicominnocci.securestore.AbstractCryptoFactory#getCryptoFactory()
 */
}
DesEncrypter Class

/* Created on Mar 13, 2005
 * Vico Minnocci
 */

package net.vicominnocci.securestore.crypto;

import java.io.UnsupportedEncodingException;

import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.SecretKey;

public class DesEncrypter {
    Cipher ecipher;
    Cipher dcipher;

    DesEncrypter(SecretKey key) {
        try {
            ecipher = Cipher.getInstance("DES");
            dcipher = Cipher.getInstance("DES");
            ecipher.init(Cipher.ENCRYPT_MODE, key);
            dcipher.init(Cipher.DECRYPT_MODE, key);
        } catch (javax.crypto.NoSuchPaddingException e) {
        } catch (java.security.NoSuchAlgorithmException e) {
        } catch (java.security.InvalidKeyException e) {
        }
    }

    public String encrypt(String str) {
        try {
            // Encode the string into bytes using utf-8
            byte[] utf8 = str.getBytes("UTF8");

            // Encrypt
            byte[] enc = ecipher.doFinal(utf8);

            // Encode bytes to base64 to get a string
            return new sun.misc.BASE64Encoder().encode(enc);
        } catch (javax.crypto.BadPaddingException e) {
        } catch (IllegalBlockSizeException e) {
        } catch (UnsupportedEncodingException e) {
        }
    }
}
public String decrypt(String str) {
    try {
        // Decode base64 to get bytes
        byte[] dec = new sun.misc.BASE64Decoder().decodeBuffer(str);

        // Decrypt
        byte[] utf8 = dcipher.doFinal(dec);

        // Decode using utf-8
        return new String(utf8, "UTF8");
    } catch (javax.crypto.BadPaddingException e) {
    } catch (IllegalBlockSizeException e) {
    } catch (UnsupportedEncodingException e) {
    } catch (java.io.IOException e) {
    }
    return null;
}
HSM Class

/*
 * Created on Mar 13, 2005 Vico Minnocci
 */
package net.vicominnoccisecurestore.crypto;

import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.math.BigInteger;
import java.security.InvalidKeyException;
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.Provider;
import java.security.PublicKey;
import java.security.Security;
import java.security.Signature;
import java.security.SignatureException;
import java.security.UnrecoverableKeyException;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.util.Iterator;
import java.util.Set;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;

public class HSM extends ICryptoEngine {
    private static HSM uniqueInstance;
    private static KeyStore ks;
    private static Provider p;
private String alias;

/*
 * (non-Javadoc)
 * @see
 * net.vicominnocci.securestore.ICryptoEngine#encrypt(java.lang.String)
 */
private HSM(String pin, String alias) {
    setDesc("Hardware Token.");
    String configName = "pkcs11.cfg";
    p = new sun.security.pkcs11.SunPKCS11(configName);
    Security.addProvider(p);
    printMechanisms(p);
    try {
        char[] hsmpin = pin.toCharArray();
        ks = KeyStore.getInstance("pkcs11", p);
        ks.load(null, hsmpin);
        this.alias = alias;
    } catch (Exception e) {
        System.out.println(e);
    }
}

public static synchronized HSM getInstance(String pin, String alias) {
    if (uniqueInstance == null) {
        uniqueInstance = new HSM(pin, alias);
    }
    return uniqueInstance;
}

public byte[] encrypt(byte[] b) {
    try {
        // Get private key
        Key key = ks.getKey(alias, null);
        if (key instanceof PrivateKey) {
            // Get certificate of public key
            java.security.cert.Certificate cert =
                    ks.getCertificate(alias);

            // Get public key
            PublicKey publicKey = cert.getPublicKey();
            // Return a key pair

        }
    }
}
KeyPair kp = new KeyPair(publicKey, (PrivateKey) key);

PublicKey pubk = kp.getPublic();
Cipher rsaCipher = Cipher
            .getInstance("RSA/ECB/PKCS1Padding", p);
rsaCipher.init(Cipher.ENCRYPT_MODE, publicKey);
byte[] cipherText = rsaCipher.doFinal(b);
return cipherText;

    /*
     * rsaCipher.init(Cipher.DECRYPT_MODE, prik);
     * byte[] plainText =
     * rsaCipher.doFinal(cipherText);
     * System.out.println(new
     * String(plainText));
     */
}
} catch (UnrecoverableKeyException e) {
    System.out.println(e);
} catch (NoSuchAlgorithmException e) {
    System.out.println(e);
} catch (KeyStoreException e) {
    System.out.println(e);
} catch (NoSuchPaddingException e) {
    
    e.printStackTrace();
} catch (InvalidKeyException e) {
    
    e.printStackTrace();
} catch (IllegalBlockSizeException e) {
    
    e.printStackTrace();
} catch (BadPaddingException e) {
    
    e.printStackTrace();
}
return null;

/*
* (non-Javadoc)
*
* @see
net.vicominnocci.securestore.ICryptoEngine#decrypt(java.lang.String)
*/
public byte[] decrypt(byte[] b) {

66
try {
    // Get private key
    Key key = ks.getKey(alias, null);
    if (key instanceof PrivateKey) {
        // Get certificate of public key
        java.security.cert.Certificate cert =
            ks.getCertificate(alias);

        // Get public key
        PublicKey publicKey = cert.getPublicKey();
        // Return a key pair
        KeyPair kp = new KeyPair(publicKey, (PrivateKey) key);
        PublicKey pubk = kp.getPublic();
        PrivateKey prik = kp.getPrivate();
        Cipher rsaCipher = Cipher
            .getInstance("RSA/ECB/PKCS1Padding", p);

        rsaCipher.init(Cipher.DECRYPT_MODE, prik);
        byte[] plainText = rsaCipher.doFinal(b);
        return plainText;
    }
} catch (UnrecoverableKeyException e) {
    System.out.println(e);
} catch (NoSuchAlgorithmException e) {
    System.out.println(e);
} catch (KeyStoreException e) {
    System.out.println(e);
} catch (NoSuchPaddingException e) {

    e.printStackTrace();
} catch (InvalidKeyException e) {

    e.printStackTrace();
} catch (IllegalBlockSizeException e) {

    e.printStackTrace();
} catch (BadPaddingException e) {

    e.printStackTrace();
} }
return null;

/**
 * (non-Javadoc)
 * @see
*/

67
* @see net.vicominnocci.securestore.crypto.ICryptoEngine#verify(byte[])
*/
public boolean verify(byte[] b, byte[] signature, String algorithm) {
    try {
        Key key = ks.getKey(alias, null);
        if (key instanceof PrivateKey) {
            // Get certificate of public key
            java.security.cert.Certificate cert = ks.getCertificate(alias);
            // Get public key
            PublicKey publicKey = cert.getPublicKey();
            // Return a key pair
            KeyPair kp = new KeyPair(publicKey, (PrivateKey) key);
            PublicKey pubk = kp.getPublic();

            Signature sig = Signature.getInstance(algorithm, p);
            sig.initVerify(publicKey);
            sig.update(b, 0, b.length);
            return sig.verify(signature);
        } catch (SignatureException e) {
        } catch (InvalidKeyException e) {
        } catch (NoSuchAlgorithmException e) {
        } catch (KeyStoreException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (UnrecoverableKeyException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return false;
    }
}

/*
 * (non-Javadoc)
 * @see net.vicominnocci.securestore.crypto.ICryptoEngine#sign(byte[])
 */
public byte[] sign(String algorithm, byte[] b) {
    try {
        // Get a reference private key
        Key key = ks.getKey(alias, null);

        return null;
    }
}
if (key instanceof PrivateKey) {
    // Get certificate of public key
    java.security.cert.Certificate cert =
        ks.getCertificate(alias);
    PublicKey publicKey = cert.getPublicKey();
    // Return a key pair
    KeyPair kp = new KeyPair(publicKey, (PrivateKey) key);

    PrivateKey prik = kp.getPrivate();
    Signature sig = Signature.getInstance(algorithm, p);
    sig.initSign(prik);
    sig.update(b, 0, b.length);
    return sig.sign();
}
} catch (Exception e) {
    System.out.println(e);
}
return null;
/*
 * (non-Javadoc)
 * @see net.vicominnocci.securestore.crypto.ICryptoEngine#hash(byte[])
 */
public byte[] hash(byte[] b, String alg) {
    try {
        MessageDigest md5 = MessageDigest.getInstance(alg);
        md5.update(b);
        return md5.digest();
    } catch (NoSuchAlgorithmException e) {
    }
    return null;
}

public void printMechanisms(Provider p) {
    if (p == null)
        p = this.p;
    Set set = p.getServices();
    Iterator it = set.iterator();
while (it.hasNext()) {
    // Get element
    System.out.println(it.next());
}

/**
 * (non-Javadoc)
 * @see net.vicominnocci.securestore.crypto.ICryptoEngine#generateKeyPair()
 */
public void generateKeyPair(String algorithm, int keysize) {
    KeyPairGenerator kpg;
    KeyPair kp;
    try {
        if(algorithm!=null&&keysize>1024){
            kpg = KeyPairGenerator.getInstance(algorithm,p);
            kpg.initialize(keysize);
            kp = kpg.generateKeyPair();
        }
    } else{
            kpg = KeyPairGenerator.getInstance(algorithm,p);
            kp = kpg.generateKeyPair();
            PublicKey key = kp.getPublic();
            System.out.println(new BigInteger(key.getEncoded()).toString(16));
    }
    } catch (Exception e) {
            System.out.println(e.getMessage());
    }
}

// Returns the contents of the file in a byte array.
private static byte[] getBytesFromFile(File file) throws IOException {
    InputStream is = new FileInputStream(file);
    // Get the size of the file
    long length = file.length();
// You cannot create an array using a long type.
// It needs to be an int type.
// Before converting to an int type, check
// to ensure that file is not larger than
Integer.MAX_VALUE.
if (length > Integer.MAX_VALUE) {
    // File is too large
}

// Create the byte array to hold the data
byte[] bytes = new byte[(int)length];

// Read in the bytes
int offset = 0;
int numRead = 0;
while (offset < bytes.length
    && (numRead=is.read(bytes, offset, bytes.length-
    offset)) >= 0) {
    offset += numRead;
}

// Ensure all the bytes have been read in
if (offset < bytes.length) {
    throw new IOException("Could not completely read
    file "+file.getName());
}

// Close the input stream and return bytes
is.close();
return bytes;
}
REFERENCES


http://www.cl.cam.ac.uk/~rja14/wcf.html.


72


