ARCHITECTURAL TECHNIQUES FOR ENABLING SECURE CRYPTOGRAPHIC PROCESSING

JOHN PATRICK MCGREGOR, JR.

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING

JUNE 2005

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Ruby B. Lee
(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Edward W. Felten

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Sun-Yuan Kung

Approved by the Princeton University Graduate School:

Dean of the Graduate School

Abstract

Cryptographic processing is a principal enabler of many secure computing systems. Using cryptographic techniques such as encryption and secure hashing, we can satisfy several essential security requirements for networks, computers, and data against a diverse set of threats.

This thesis proposes four architectural solutions to problems associated with enabling cryptographic processing in software and hardware. Two of the solutions involve protecting cryptographic keys, which are small secrets upon which cryptographic security critically depends. The other two solutions improve performance and reduce vulnerabilities in cryptographic software implementations.

First, since the security provided by cryptographic processing depends on the secrecy and integrity of cryptographic keys, we describe a flexible system for shielding a user's keys while in storage, transmission, and use on networked computing devices. Second, we present a new broadcast encryption system that enables the identification of users who contribute to piracy by divulging cryptographic keys that can be used to decode protected information. Third, since software rather than specialized hardware often supplies cryptographic functionality, we describe a method for alleviating performance problems suffered by cryptographic software implementations. In particular, we propose new processor instructions to improve the performance of bit-level mappings employed by several common cryptographic operations. Fourth, we present a processor-based method for mitigating certain software vulnerabilities in both cryptographic and general software. The method provides built-in and dynamic protection against buffer overflow attacks, which compose one of the most common classes of software exploits. By applying these four contributions individually or in concert, we can achieve improved cryptographic security in existing and future systems.

Acknowledgments

I wish to offer many thanks to colleagues, friends, and family. First, I thank Professor Ruby Lee, my research advisor. Her boundless energy, keen mind, and insistence on excellence were vital to the development of this dissertation and to the completion of my graduate studies. She was always very generous in sharing her time, and she was actively involved in every aspect of my research, from the overarching ideas to the finest system details.

I would like to thank Professors Ed Felten and S.-Y. Kung for reading my dissertation and for providing helpful and insightful feedback. Also, I learned much about building practical secure systems from Professor Felten's classes and research projects. I thank Yiqun Lisa Yin for the many hours that she spent working with me on the security analysis of the traitor tracing scheme. I found our collaboration to be highly enjoyable and enlightening. I would also like to thank Raj Kumar, who gave me the opportunity to work in his group at Hewlett Packard Laboratories.

It has been a pleasure to work with many talented engineering students at Princeton. My research has benefited from collaboration with Zhijie Shi, David Karig, Peter Kwan, Jeff Dwoskin, and Zhenghong Wang. Of particular note, David Karig and Ruby Lee were involved in the early development of the secure return address stack concept. I have also enjoyed working on projects with other members of PALMS and other Princeton EE students, including Murat Fiskiran, Xiao Yang, and Scott Craver.

I also owe thanks to Matt White. In addition to being a great friend through the ups and downs of my graduate school experience, he has been rather tolerant of my unusual business schedule resulting from my all-night research spurts.

Last but most important, I thank my family. I am forever grateful for the love, the inspiration, and the understanding that my wife Beth has given me since the first days of my journey at Princeton. And, I am very fortunate to have such caring parents who have provided unwavering support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As society, government, and enterprises become increasingly dependent on information technology, system security becomes essential.

However, despite growing awareness of threats and more proactive approaches to security, the number and virulence of security vulnerabilities are increasing rapidly. The Computer Emergency Response Team (CERT) states that security problems are growing exponentially: the number of reported information technology security incidents increased at a mean rate of 85% per year from 2000 to 2003 [31]. Participants in the 2004 CSI/FBI Computer Crime and Security Survey reported average annual losses of over $555,000 per company due to security breaches [33], which translates into collective corporate losses ranging in the tens of billions of U.S. dollars. Privacy and data secrecy issues are also significantly affecting the value of enterprises. Large public corporations that experienced a publicized privacy or data secrecy breach between the years 1995 and 2000 suffered an average short-term market capitalization reduction of over 5% [23].

Many reasons exist for the increase in security vulnerabilities. These reasons include the increasing complexity of computing systems, the reluctance to adopt security technologies because of cost or performance issues, and inadequate security designs. Complexity, in particular, can be computer security's worst enemy [140]. However, due to intense consumer demand for improved functionality and connectivity, there is no near-term indication that system capabilities and complexity will be reduced for the sake of security. This is evidenced by the continuing exponential growth of processor transistor counts [61] and the number of lines of code involved in operating systems [161].

For the past three decades, in systems based upon programmable microprocessors, software has performed the majority of security functions. The system hardware has provided few features for protecting resources and data. Furthermore, security responsibilities in software are sometimes relegated to secondary modules instead of core software components. That is, in some systems, security is interpreted and implemented as an application rather than a fundamental design goal.

Given the enormous resources that are becoming available in hardware, computer architects have an unprecedented opportunity to improve security by incorporating new protection features into the heart of computing systems. Moore’s Law continues to hold: the number of transistors per die is doubling every eighteen months [61]. By 2007, the number of transistors in advanced general-purpose processors is expected to exceed one billion [61]. Most of these transistors will be applied to construct enormous on-die memory caches and complex microarchitectural mechanisms that provide increasingly diminishing improvements to overall performance. Why squander millions of logic gates and memory cells to marginally accelerate the performance of general software? Some of these hardware resources can be better applied to address more critical problems. Issues that truly warrant the attention of hardware architects include enabling low-power operation, supporting system recoverability, and the *raison d’être* of this thesis: enhancing security.

Hardware can be applied to enable certain security mechanisms and acceleration features that are not realizable in software-only systems. In particular, processors and hardware platforms can be employed to store special secrets and perform special operations that are not available to any system software. This leads to numerous opportunities for enabling secure software execution by relying on trusted hardware. Furthermore, in order to facilitate the adoption of secure systems, specialized hardware-based techniques can and should be implemented to minimize undesirable side effects of security features. Such undesirable side effects may include performance degradation, increased monetary cost, and increased energy consumption.

By enhancing security with hardware, processors and platforms do not necessarily need to be fundamentally re-engineered. Rather, low-cost synergistic hardware and software techniques can provide immediate and potent results.

1.1 Thesis Contributions

This thesis investigates a set of topics in secure computing. Specifically, this thesis presents four architectural techniques for providing and enriching security. Each of the contributions provides a means for enabling secure and efficient cryptographic software, which is a core component of many security mechanisms and systems. The contributions are by no means intended to collectively serve as a security panacea; instead, this thesis demonstrates that hardware can be employed to mitigate crucial vulnerabilities and to assuage performance problems associated with existing approaches to security.

The first contribution of this thesis is a hardware and software system for defending users' core secrets in general-purpose platforms such as desktop computers and personal digital assistants (PDAs). Most conventional computing systems do not provide adequate protection for small quantities of secret information upon which security often depends, such as cryptographic keys. This thesis proposes a set of low-cost hardware architecture enhancements and a privileged software library that allow general-purpose processors to perform flexible, high-performance, and protected cryptographic computation. With this system, which we call Virtual Secure Coprocessing, users can employ their secret keys to safely and flexibly complete cryptographic operations in the presence of potentially vulnerable software applications and operating systems.

The second contribution of this thesis is an efficient scheme that enables the identification of users involved in the piracy of protected broadcasts. In this application scenario, sensitive information may be broadcast to many hardware devices, but only authorized hardware devices should contain the cryptographic keys needed to decode and make use of the information. However, some authorized entities, called traitors, may extract the cryptographic keys in order to construct and distribute devices that enable piracy. To identify such traitors, an information content provider may employ a *traceability scheme* for broadcast systems. Following the confiscation of a pirate decoding device in a traceability scheme, at least one user in a collusion of up to k traitors that contributed to the device can be identified. This thesis presents a new traceability scheme based on the RSA encryption algorithm. The proposed scheme significantly improves upon the decryption performance of past proposals while providing the same or better level of security. In the proposed solution, decryption only requires approximately one modular exponentiation, whereas past proposals require $O(k)$ or greater computationally expensive operations per decryption.

The third contribution of this thesis is a set of architectural enhancements for accelerating the performance of bit-level permutations and mappings, which are employed in bulk encryption and hash operations. Bulk encryption and secure hashing are often employed to provide confidentiality and integrity for data in secure communications and secure storage scenarios. In conventional architectures, however, bit-level permutations and mappings are not natively supported by the processor, and therefore performing the mappings in software can be highly time consuming. Thus, by improving the throughput of these operations, the potential negative performance impact of current and future cryptographic security procedures can be alleviated. This thesis proposes instruction set architecture (ISA) improvements, hardware enhancements, and software modifications for achieving high-performance permutations and mappings of 1-bit or larger subwords packed in an n-bit word. These enhancements significantly accelerate the mappings, and they also accelerate the popular Data Encryption Standard (DES) and the Triple Data Encryption Standard

(3DES) algorithms.

The fourth contribution is a processor-based mechanism that addresses one of the most common types of remote attacks on software: buffer overflows. This mechanism mitigates common security vulnerabilities in cryptographic security software as well as in general software by thwarting buffer overflow attacks involving procedure return address corruption. This thesis describes low-cost processor and operating system enhancements that effectively and transparently preclude such buffer overflow attacks and thus lessen the frequency of insertion and execution of malicious code. The proposed processor and operating system (OS) enhancements can be applied in tandem with other software countermeasures to provide robust protection.

1.2 Thesis Organization

The organization of this thesis is as follows.

Chapter 1 introduces the contributions of this thesis and the philosophy of applying hardware to improve the security offered by software-only techniques.

Chapter 2 introduces important security concepts and discusses the role of cryptography in secure systems. The chapter also identifies core issues and problems relating to cryptographic processing that are addressed by the contributions of this thesis.

Chapter 3 presents Virtual Secure Coprocessing. The chapter begins by characterizing the paradigm shifts that motivate the design of this system. Next, processor, platform, operating system, and software enhancements are presented that provide critical protection for keys in general-purpose platforms. Processor simulation results demonstrate that the proposed enhancements cause a negligible (i.e., less than 1%) performance impact on affected programs. This chapter is based in part on work presented by the author in [95, 108, 109].

Chapter 4 presents a new traitor tracing scheme based on RSA. Methods for user initialization, encryption, decryption, and traitor tracing are presented and evaluated. The strength of the scheme is evaluated via formal proofs of security. This chapter is based in part on work presented by the author in [110, 111, 112].

Chapter 5 presents architectural enhancements for accelerating bit-level permutations and mappings for general-purpose processors. Instruction set architecture changes, microarchitectural and circuit implementations, and corresponding software enhancements are investigated. Using a processor simulator, performance improvement is demonstrated for the permutations and the mappings in the popular DES and 3DES encryption algorithms. This chapter is based in part on work presented by the author in [104, 105].

Chapter 6 presents a processor-based mechanism for preventing procedure return address corruption resulting from buffer overflow. The chapter introduces a set of processor modifications and operating system enhancements needed to achieve the design goals and investigates the impact of varying values of design parameters. Processor simulation results for a set of representative programs show that the buffer overflow defense does not significantly affect the performance of programs on general-purpose processors. This chapter is based in part on work presented by the author in [93, 94, 103].

Chapter 7 concludes the thesis and provides directions and opportunities for future research.

Chapter 2

Cryptographic Processing

This chapter explores methods for and issues in implementing cryptographic primitives in secure systems. Cryptographic primitives effectively combat a diverse set of threats against networks, computers, and information. However, as this chapter demonstrates, many fundamental issues remain to be addressed relating to the secure and dependable implementation of cryptographic functionality. In subsequent chapters, solutions are proposed for several of the identified implementation problems.

This chapter is organized as follows. Section 2.1 establishes definitions of important security concepts, including secure systems, trust, threats, and security goals. Section 2.2 presents selected methods and standards for evaluating security functionality. Section 2.3 introduces and describes cryptographic primitives, such as encryption and hashing. Section 2.4 discusses how cryptography can be employed to satisfy common security goals in communications and storage systems. Section 2.5 identifies important issues in enabling cryptographic processing and proposes architectural approaches for addressing these issues. Section 2.6 summarizes this chapter.

2.1 Defining Security

To properly evaluate security, we must first establish a definition of a *secure system*. This thesis uses the following definition, which is based on statements articulated by Anderson in [4]:

> *A secure system is a system that remains dependable in the face of malice, error, or mischance.*

This definition is based on dependability as a general concept rather than a specific technical goal. In the context of the complex systems that we wish to secure, which include operating systems and software applications, dependability relates to many factors. These factors include — but are not limited to — the desired feature set, the threat model, and the system security goals.

2.1.1 Threats

A secure system is often defined and described in the context of its *threat model*. A threat model is a description of potential vulnerabilities and attacks of concern to a system. In this thesis, we focus on threats against which cryptographic processing can provide needed protection. A few examples of threats to networks and computers are listed below:

- **Network eavesdropping.** As computer networks continue to proliferate, unauthorized parties can obtain sensitive data using very simple equipment to eavesdrop inconspicuously on network communications. This is especially true for wireless networks that broadcast information using the public spectrum.

- **Physical data media theft.** Attackers can physically steal laptops, PDAs, etc., which may contain sensitive data or that may enable access to protected resources.

- **Entity impersonation.** In systems that use weak authentication mechanisms, such as short user passwords, attackers can undetectably impersonate authorized users and gain access to sensitive information or valuable resources.
- **Data and code piracy.** Users may acquire sensitive information in an authorized or unauthorized fashion and subsequently (and perhaps anonymously) divulge the content to unauthorized users.
- **Data and code tampering.** Attackers may modify data or code (ranging from 1 bit to many gigabytes in size) without being detected by relevant authorities.

Ideally, to protect against present and future threats more effectively, threats should be generalized, and defenses should be designed to thwart general classes of attacks.

2.1.2 Security Goals

Given a threat model, the *security goals* of the system define the security and protection required for information, access, and system resources against the identified threats. A few common security goals are described below:

- **Data confidentiality.** Data confidentiality, which is sometimes termed *data secrecy*, means that only authorized entities may access the target data. Though the term *privacy* has been used synonymously with confidentiality in the past, privacy should be reserved for the protection of personal information in modern contexts.
- **Data integrity.** Data integrity means that only authorized entities may modify data in authorized ways and implies that unauthorized entities should not be able to undetectably modify data in any way.
- **Data origin authentication.** Data origin authentication means that an authorized entity that supplies or modifies data can be identified. That is, it is not feasible for the origin of the data to be forged.

- **Entity authentication.** Entity authentication involves the identification of a communicating entity. Strong entity authentication systems prevent the impersonation of entities by adversaries, and therefore entity authentication is sometimes called host authentication or user authentication.

Other examples of system security goals include non-repudiation, access revocability, and traitor traceability, to name a few. The techniques chosen to realize security goals depend on the characteristics of the target system, the threat model, and the application domain. For instance, to ensure confidentiality against physical theft of a sensitive report that is written on paper, security engineers may choose to seek adequate confidentiality by simply storing the report in a fortified vault. However, such a vault would obviously not be appropriate to protect a report that is maintained digitally within hundreds of desktop computers that may be connected to the Internet. When seeking to protect information and computing resources, cryptography is often employed (in conjunction with other security mechanisms such as software-enforced access control).

It is important to distinguish the concept of a *secure system* from the concept of a *trusted system*. We use the NSA definition of a trusted system, which is a system that can defeat a security policy if it fails [4]. Furthermore, the *trusted boundary* of a system is a physical and/or virtual boundary that separates the untrusted environment from the resources and data that are designated as trusted. The physical exterior or software interface of a device is often defined to be the trusted boundary. Alternatively, a trusted boundary in an enterprise scenario might encompass all the information technology assets behind a firewall. The notions of a trusted system and a *trustworthy system* are also somewhat different: a trustworthy system is designed and expected to function correctly and to successfully realize a security policy.

2.2 Evaluating Security

Security evaluation is difficult due to innumerable ways that users and adversaries may interact with a system. This problem is exacerbated by growing system complexity. The transistor counts of high-end general-purpose processors currently range in the hundreds of millions and continue to grow exponentially. The complexity of defining and evaluating correct hardware operation grows at a similar or faster rate [61]. Furthermore, application software and modern operating systems support thousands of features via millions of lines of source code [161], and these features continue to grow in capability and number. As evidenced by the increasing number of publicized software vulnerabilities [31], threat models and security goals for software continue to be difficult to define and achieve, respectively.

Though the quantification and qualification of system security remains an unsolved problem, various methodologies for achieving security assurance do exist. The Trusted Computer System Evaluation Criteria (TCSEC) [42], which is commonly known as the "Orange Book", was published in 1985 by the National Computer Security Center, an arm of the U.S. National Security Agency. Though the Orange Book is becoming less relevant as computing systems evolve, the Orange Book was one of the first comprehensive sets of standards for structured security evaluation. Several other standards known as the "Rainbow Books" were published by the National Computer Security Center to address more specific systems such as networks and databases.

The Orange Book classifies a system into one of several security ratings, namely D, C1, C2, B1, B2, B3, and A1. The rating assigned to a system depends on several characteristics, which include the following:

- **Security Policy.** This set of criteria relates to mechanisms for discretionary and mandatory access control as well as for device and object labeling.

- **Accountability.** This set of criteria involves system features for identification, authentication, audit, and trusted paths.

- **Assurance.** These criteria evaluate the capabilities of the independent architectural, software, and configuration components designed to achieve the policy and accountability requirements.
- **Documentation.** These criteria relate to design documentation, testing documentation, and users' manuals.

A rating of D implies that a system essentially possesses no security mechanisms. A rating of A1 implies that a system is highly secure and that its security policy (but not necessarily the system itself) has been formally verified. Formal verification essentially involves the construction of a mathematical proof that a system always satisfies a particular security policy. Although known formal techniques can be effectively applied to simple, contained systems, such techniques are not practical for conventional complex systems such as desktop computers. Due to the continuing push for features and performance over security in software and hardware, developers do not allocate sufficient resources to apply comprehensive methods for ensuring that software is correct and devoid of vulnerabilities. Instead, security analysis performed on commercial software tends to be ad hoc.

A system, product, or device that is being evaluated using the Orange Book or related criteria is sometimes called the target of evaluation (TOE). Though the Orange Book is extensive in scope, the Orange Book criteria are based on a specific security model, which limits the types of TOE's that can be evaluated. Most commercial systems employ a different security model than that specified by the Orange Book, as the Orange Book was intended for government (especially military) computing systems. Thus, since the release of the Orange Book, other organizations and governments have published improved guidelines for evaluating system security. These publications include the Information Technology Security Evaluation Criteria (ITSEC) [70], which was composed by representatives from several European nations, and the Canadian Trusted Product Evaluation Criteria (CTPEC) [24]. In 1996, Version 1.0 of the Common Criteria (CC) was released [28], which integrates the approaches of the Orange Book, ITSEC, and CTPEC. Version 2.0 of the Common Criteria was standardized by the International Standards Organization (ISO) in 1999 [69], and this version is often used today to evaluate security for many systems.

The Common Criteria seek to provide companies and government agencies with flexibility in defining the security model to be evaluated. To this end, the CC employ the concepts of protection profiles and security targets. A *protection profile* is an abstract description of the functional needs and assurance needs for the TOE. A *security target* is a detailed and concrete description of how a protection profile is realized for the TOE. For every security component outlined in the protection profile, there is a corresponding component included in the security target. A Common Criteria evaluation for a TOE is based upon the statement of the security target.

Other standardized methods for evaluating smaller, more restricted secure systems also exist. One example of such methods is the Federal Information Processing Standard (FIPS) 140-2 [122]. This standard establishes security requirements for cryptographic modules. FIPS 140-2 defines four levels of security for these modules, 1 being the lowest, and 4 being the highest. The evaluation process is based upon implementation correctness, software security, and physical security. Levels 2 and higher involve some degree of physical tamper detection or resistance.

These federal and international standards are intended to evaluate restricted systems with static goals and pre-defined threat models. This approach is effective in some scenarios, such as in defense-related government agencies. However, in an increasingly dynamic and interconnected world, security evaluations based on these standards are often inadequate [4]. Conventional systems involve unpredictable users, frequent updates to components, and rapidly evolving threats, and these characteristics are not well accommodated by the Common Criteria and related standards.

Much work remains to be completed in the enormous task of analyzing security in modern computers and networks that comprise multiple complex subsystems. Given this fact, this thesis does *not* attempt to provide a complete framework for enabling robust system security. Rather, this thesis presents new results that address a select set of previously unsolved problems in cryptographic processing.

2.3 Cryptography

Cryptography is a mathematical toolbox for achieving certain important security goals. Examples of such security goals include confidentiality, integrity, and authentication. Cryptographic primitives differ in the degree of security provided as well as in the degree of flexibility. Examples of cryptographic primitives include encryption algorithms and pseudorandom number generators. The former can be employed to provide services such as confidentiality, integrity, and user authentication, but the latter may be applied only to generate pseudorandom bits for use by other operations.

Most conventional cryptographic primitives satisfy the design requirements articulated by the Flemish cryptographer Auguste Kerckhoffs in 1883 [77]. Specifically, Kerckhoffs stated that the security of a cryptographic method should not depend on the secrecy of the method. Rather, the cryptographic algorithm should be publicly known and available, and the security of the system should be based on small quantities of information known as *cryptographic keys*. The cryptographic algorithms that are employed by many non-government secure systems have been thoroughly and publicly analyzed by the security community and are resilient to known attacks. Thus, assuming the scrutinized algorithms are implemented correctly within properly constructed security protocols, their security depends on the measures taken to establish and protect key material rather than on the measures taken to ensure the secrecy of the algorithms. Presented below are selected classes of cryptographic primitives that are employed in this thesis and that align with Kerckhoffs' principles.

2.3.1 Symmetric-key Encryption

Encryption is the process of translating data into a disguised form. An encryption algorithm, which is also known as a *cipher*, accepts input data known as *plaintext* and outputs encoded plaintext known as *ciphertext*. Furthermore, the encryption algorithm encodes the plaintext into ciphertext using a quantity of information known as the *encryption key*. Many encryption algorithms disguise data by performing operations on the plaintext and the key material to achieve goals known as *confusion* and *diffusion* [143]. Confusion operations obfuscate the relationship between the ciphertext and the inputs. Diffusion operations spread the redundancy of the plaintext and key bits across the ciphertext. *Decryption* is the inverse of encryption; a decryption algorithm translates input ciphertext into plaintext using a *decryption key*.

In *symmetric-key encryption*, which is also known as *secret-key encryption*, a single key is used for both encryption and decryption. This key is kept secret by both the encrypting party and the decrypting party. Symmetric-key ciphers can be employed to provide confidentiality, integrity, and some forms of message and entity authentication.

Symmetric-key encryption involving two parties is illustrated in Figure 2.1. To follow the cryptographic community's tradition, these two parties are named Alice and Bob (based on the letters "A" and "B"). First, a secret key k must be distributed to both Alice and Bob via some secure means, e.g., an encrypted channel or a physically secure channel.[1] Then, Alice can encrypt a secret plaintext message p by inputting p and k into the encryption algorithm, E. E outputs a ciphertext message c, which can be safely transmitted over a public channel to Bob. Upon receipt of c, Bob can obtain p by inputting c and his copy of k into the decryption algorithm, D. E and D can be publicly known, but k must be kept secret by both parties.

If a secure symmetric-key encryption algorithm E is implemented correctly within an appropriate security protocol and k is kept secret, then eavesdropping adversaries on the

[1]This key distribution is a nontrivial problem which we discuss in further detail later in this chapter.

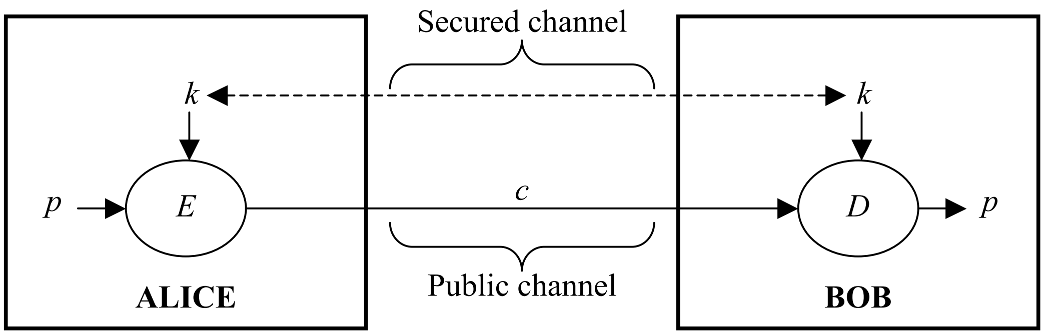


Figure 2.1: Symmetric-key encryption and decryption

unsecured channel cannot obtain p. That is, given only c and a reasonable amount of time, storage, and computation resources, the adversaries cannot deduce any bit of p with probability significantly greater than 1/2. This security for p against eavesdroppers with knowledge of c is also known as resilience against *ciphertext-only attacks*. Secure symmetric-key ciphers are also resilient against several other types of attacks. For example, if the cipher is secure against a *known-plaintext attack*, then an adversary cannot obtain the secret key k given plaintext p and corresponding ciphertext c. If the cipher is secure against a *chosen-plaintext attack*, then an adversary cannot obtain the secret key k given the ciphertext c corresponding to any plaintext p chosen by the adversary.

Two classes of symmetric-key encryption algorithms exist: *block ciphers* and *stream ciphers*. A block cipher performs encryption operations over fixed-length message blocks, using the same key for multiple block encryptions. The message to be encrypted can be of any length; it is divided into fixed-length blocks with padding of the last block, if necessary, before being encrypted. Typical block sizes range between 64 and 256 bits. A stream cipher essentially performs encryption operations on blocks of size one bit or one byte using a key stream rather than a fixed key. The key stream is provided using a *key stream generator*. The key stream generator output is simply a function of a secret input known as the seed. As the generator produces key bits, the value of the seed can be periodically updated by the generator.

Popular symmetric-key encryption algorithms include the Data Encryption Standard (DES) [119], Triple DES (3DES) [114], the Advanced Encryption Standard (AES) [120], RC5 [132], RC6 [134], IDEA [87], and Twofish [141]. The security of most symmetric-key algorithms is based on ad hoc techniques rather than formal proofs of security or reductions to number-theoretic problems. Popular symmetric-key algorithms that are regarded as being secure have been shown to be resilient against known cryptanalytic techniques (well-known examples of which include linear and differential cryptanalysis [114]). Common key sizes for implementations of popular symmetric-key ciphers range from 112 bits to 256 bits.

Symmetric-key encryption and other primitives are implemented using various cryptographic *modes of operation*. Such modes are often trivial extensions of the cryptographic primitive itself, but the extensions can lead to striking security differences. For example, two commonly employed modes of operation relating to symmetric-key block ciphers are Electronic Code Book (ECB) and Cipher Block Chaining (CBC) [114, pp. 228–233]. The encryption of n blocks using the ECB and CBC modes is displayed in Figure 2.2. In ECB, a plaintext block p_i is simply encrypted with the secret key k to generate a ciphertext block c_i. In CBC, each ciphertext block c_i for $i > 1$ is a function of both the plaintext block p_i and the previous ciphertext block $c_{(i-1)}$. The first plaintext block in CBC is XORed with a value known as the initialization vector (IV). When using ECB, an attacker may not be able to decrypt blocks, but an attacker can infer some information about the plaintext, e.g., whether or not a particular plaintext block is repeated. CBC, however, prevents such exposure. Furthermore, CBC is more resilient against attacks that seek to inconspicuously substitute a forged block for an encrypted block.

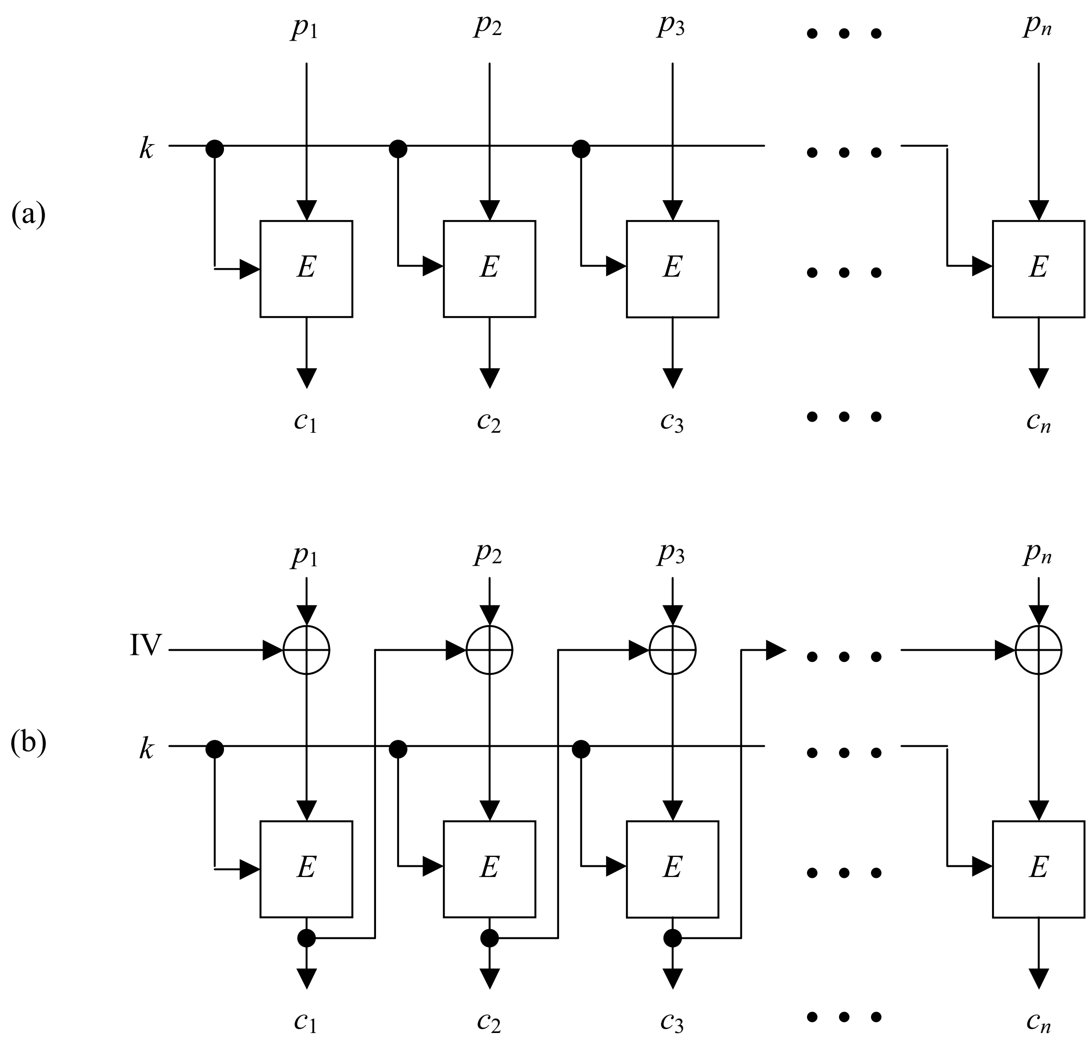


Figure 2.2: Encryption of n blocks in (a) ECB mode and (b) CBC mode

2.3.2 Asymmetric-key Encryption

Asymmetric-key encryption, which is also called *public-key encryption*, differs significantly from symmetric-key encryption in terms of operation, capability, and computation cost. This class of encryption algorithms enables data confidentiality and entity authentication over public channels. In addition, this class enables other security services such as digital signatures.

Figure 2.3 depicts asymmetric-key encryption involving two parties. Like symmetric key encryption, there exists an encryption key, a decryption key, an encryption algorithm, and a decryption algorithm. Unlike the symmetric-key case, however, the encryption key k_e is different from the decryption key k_d. Given the encryption key k_e, it should not be feasible to deduce the corresponding decryption key k_d. The encryption key, which can be exposed to the public, is called the *public key*. The decryption key, which must be kept secret by the decrypting party, is called the *private key*. Thus, the distribution of the public key can occur over unencrypted channels.[2] This provides significant advantages over symmetric-key cryptography in many scenarios. Also, though Bob's public key k_e may be generated by Bob, Bob does not need to assume full responsibility for distributing k_e to relevant parties. Instead, Bob can employ a trusted third party to distribute k_e over public channels to parties such as Alice.

As shown in Figure 2.3, upon obtaining Bob's public encryption key, Alice can encrypt a secret plaintext message p using the public key k_e and the encryption algorithm, E. E outputs a ciphertext message c, which can be transmitted over a public channel to Bob. Upon receipt of c, only Bob or other parties with knowledge of Bob's private key k_d can recover p by inputting c and k_d into the decryption algorithm, D. Parties that possess knowledge of k_e but do not know the value of k_d will not be able to recover p given c with a tractable or feasible amount of computation and storage. Furthermore, the asymmetric key

[2]Although the distribution channel does not need to be encrypted (to provide confidentiality), measures must be taken to guarantee the integrity, origin, and freshness of the public key.

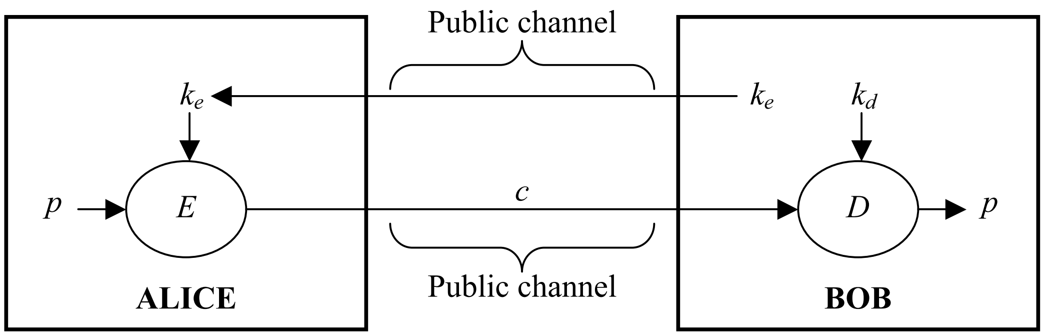


Figure 2.3: Asymmetric-key encryption and decryption

algorithm should be constructed such that parties that possess k_e and a reasonable number of (p, c) pairs should not be able to recover the corresponding k_d.

Examples of asymmetric-key encryption algorithms include RSA [133], Diffie-Hellman (DH) [45], elliptic curve cryptosystems (ECC) [113], McEliece [102], and ElGamal [48]. The security properties of these algorithms are reducible to the intractability of computational problems for which no efficient solution is known (i.e., no solution is known within the complexity class **P**).[3] For instance, the difficulty of recovering a private decryption key corresponding to a known public encryption key in RSA is equivalent to the difficulty of integer factorization. The integer factorization problem is known to be in **NP**[4], is believed to not be in **P**, and is believed to not be **NP-complete**[5]. In extreme cases and certain implementations, however, many attacks on asymmetric-key algorithms are known (e.g., [83, 149, 17]), and care must be taken to maintain security in software and hardware implementations. Common key lengths in secure implementations of asymmetric-key ciphers

[3]The complexity class **P** contains decision problems which can be solved by a polynomial-time deterministic Turing machine. In general, problems and in this class are considered to be "tractable".

[4]The complexity class **NP** contains decision problems that have solutions that can be verified using a polynomial-time deterministic Turing machine. It is commonly believed (though not proven) that **NP** includes problems that cannot be solved with any polynomial-time algorithm, i.e., $\mathbf{P} \subset \mathbf{NP}$.

[5]The complexity class **NP-complete** contains decision problems that, in terms of computation requirements, are the most difficult to solve in the class **NP**.

are 163 bits or higher for elliptic curve cryptography and 1024 or 2048 bits for RSA.

Though asymmetric-key encryption can be used to enable more flexible protocols than symmetric-key encryption, it costs more to implement than symmetric-key encryption. Software implementations of asymmetric-key algorithms encrypt data 100 to 1000 times as slowly as common symmetric-key ciphers [139]. Thus, in many systems, both symmetric-key and asymmetric-key encryption are used to achieve security goals. Asymmetric-key encryption is often employed to enable authentication and key exchange, and symmetric-key encryption is used to provide confidentiality and integrity for bulk data.

2.3.3 Cryptographic Hash Functions

Cryptographic hash functions, which are also called secure hash functions, accept arbitrarily-sized messages as inputs, and they output small, fixed-size *message digests*. A message digest, often called a *fingerprint*, may range from 128 bits to 512 bits in size. The purpose of a digest is to serve as an efficient "identifier" for an input message, for no two messages should yield the same digest value in practical scenarios. Two critical features that distinguish cryptographic hash functions from regular hash functions such as CRC checksums are *preimage resistance*, *weak collision resistance*, and *strong collision resistance*. The preimage resistance property ensures that given a random fingerprint, it is not feasible (with a tractable amount of computation and storage) to construct an input message that corresponds to that fingerprint. This property is therefore also called *one-wayness*. The weak collision resistance property ensures that given a message and the message's respective fingerprint, it is not feasible to obtain another message that yields the same fingerprint. The strong collision resistance property guarantees that it is not feasible to find two messages that yield the same fingerprint. Secure hash functions are both one-way and collision-resistant.

Hash functions that incorporate the notion of a secret cryptographic key are known as *keyed hash functions*. These functions can be used to provide data integrity, entity authentication, and components of digital signature generation and verification. One common manifestation of a keyed hash function is the Hash Message Authentication Code, or HMAC [85].

Figure 2.4 illustrates a simplified overview of HMAC computation and verification for ensuring data integrity. The process involves a message m, a publicly-known cryptographic hash function H, and a secret key k. In the figure, H' represents the HMAC function, which consists of a small number of invocations of H performed over m and k. Prior to the HMAC operation, the secret key k must be established and distributed to Alice and Bob via a secured channel or via a physically secure method (e.g., an armored truck). Then, Alice can perform a series of hash operations over the message m and the key k to produce the digest g. The digest g is then bundled with the message m, and the bundle can be transmitted to Bob over a public (unencrypted) channel. Upon receipt, Bob performs an identical sequence of operations using m, k, and H' to compute g'. By the properties of keyed hash functions, if g' equals g, then the message has not been modified by any adversaries that do not possess knowledge of k. If g' does not equal g, then unauthorized data modification or data corruption is detected.

Popular cryptographic hash functions include SHA-1 [121], SHA-256 [121], and MD5 [131]. Furthermore, primitives based upon block ciphers such as AES and 3DES may be employed to serve as keyed or unkeyed hash functions [114, pp. 338–343]. One such method of applying a cipher as a keyed hash function is CBC-MAC. In this construction, a message of any size is encrypted using a cipher in CBC mode with an IV and a secret key k. Then, the output of the last block encryption is used as the hash result.

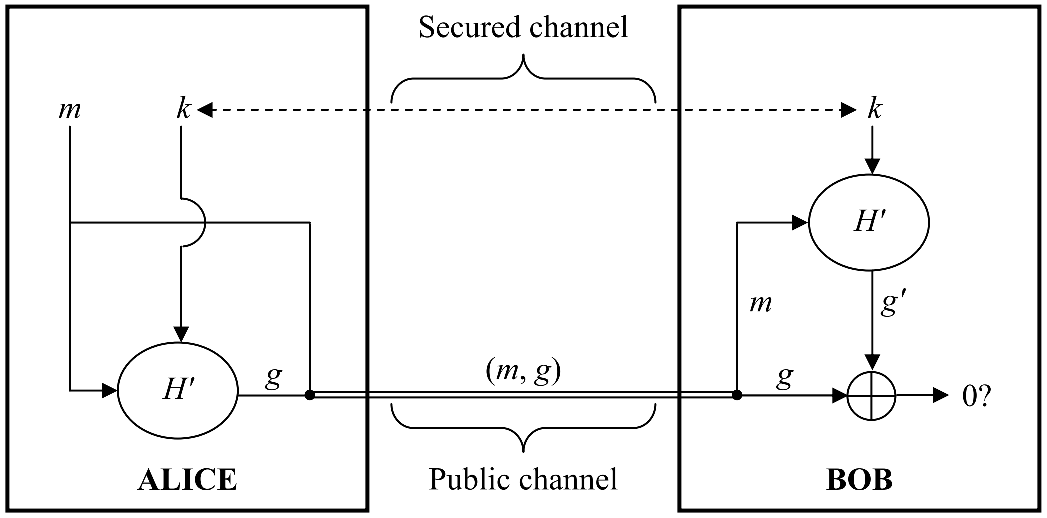


Figure 2.4: HMAC computation and verification

2.3.4 Other Primitives

In many secure systems, multiple cryptographic primitives are performed in order to implement a single useful primitive. One such primitive is a *pseudorandom number generator* (PRNG). These primitives are critical components of many authentication and key establishment protocols. PRNG’s can utilize a series of encryption operations to produce a pseudorandom integer and then update a secret PRNG seed employed by the generation process. The PRNG seed is an integer value that may range from 64 bits to kilobytes in size. A pseudorandom number can also be generated in other ways. For example, the seed or the generation function can be based on a truly physically random source.

A *digital signature* is a security primitive that, among other features, can flexibly enable the verification of the source and integrity of a message, file, or network transmission. Digital signatures are often realized by composing cryptographic hash functions with asymmetric-key encryption. If a sender wishes to digitally sign a message, an unkeyed cryptographic hash is computed over the entire message. Next, the hash of the message is encrypted using the private key of the sender's asymmetric encryption key pair. The output of the asymmetric key encryption is known as the digital signature and may be attached to the message. Any party that obtains the sender's public key can then verify the authenticity of the signature and the integrity of the message. This verification involves comparing the result of the hash of the received message to the result of decrypting the digital signature with the sender's public key.

2.4 Applying Cryptography

In many system scenarios, methods such as software-based access control are not sufficient to fulfill basic security requirements. If an attacker physically steals a laptop and removes a hard drive containing sensitive information, the password prompt traditionally required at boot up will not prevent the thief from extracting secrets from the device. Fortunately, we can apply cryptographic techniques to such problems to achieve necessary and powerful security goals.

2.4.1 Security Protocols

Secure systems often apply cryptographic primitives using *security protocols*. A security protocol is simply a sequence of steps required of one or more parties to achieve a desired security goal. The protocol steps may involve simple operations, such as incrementing an integer, as well as sophisticated cryptographic primitives, such as digital signatures. Care must be taken in properly designing and implementing protocols. Security threats and goals should be well articulated, each protocol step should be explicitly defined, and relevant underlying assumptions should be called out. Failure to adequately perform any of these design duties can lead to security vulnerabilities.

We now examine sample security protocols that employ cryptography to address two classic security problems: secure data transmission and secure data storage. For both problems, we define a set of relevant threats and security goals; we discuss typical cryptographic mechanisms used to address these goals; and we identify critical implementation problems that remain.

2.4.2 Secure Data Transmission Protocol Example

Protecting information in transmission between two or more points is perhaps the quintessential data security problem. Moreover, cryptographic systems that defend sensitive communications are certainly not new. Ciphers were used during the reigns of the Roman and Egyptian empires to protect messages [140]. Today, improved ciphers and security mechanisms provide flexible protection for information traveling over the Internet, local area networks, and wireless networks.

The threat models and security requirements of conventional communications systems vary widely. In this section, we discuss a cryptographically enabled security protocol for protecting network transmissions between a single authorized sender and a single authorized receiver. The threats of concern are unauthorized entities that may only perform the following actions on messages that are transmitted between the two authorized entities:

1. Observe the contents of any messages
2. Modify or inject information into the contents of any intercepted messages, without being detected
3. Send a forged message to the authorized receiver purporting to originate from an authorized sender, without being detected (which is known as *spoofing*)
4. Resend a message to the authorized receiver that was previously transmitted between the two authorized entities, without being detected (which is known as a *replay attack*)

The trusted boundaries in this scenario are defined to be the physical exteriors of the two computers belonging to the sender and receiver. Given this boundary definition and the threat model, we seek to achieve three security goals: data confidentiality, data origin authentication, and data integrity. To guarantee data confidentiality, only the sender and receiver should be able to interpret bits of plaintext messages. To guarantee data origin authentication, the receiver must be able to ensure that received messages were constructed only by an authorized user. Lastly, to guarantee data integrity, the receiver must be able to (i) validate that received messages have not been altered in any way by an unauthorized user and (ii) validate that received messages are fresh, i.e., ensure protection against replay attacks. Note that we do not attempt to defend against threats involving the destruction or redirection of messages in transit between the two authorized parties.

We now describe a cryptographic protocol for efficiently implementing these security requirements between the trusted boundaries. Note that the following description is not novel; the structure of this protocol is similar to many protocols in use today. Secure point-to-point communication protocols generally involve two steps: handshake and bulk data protection. The handshake is the initialization of the secure channel between the two trusted boundaries. During this step, the entities authenticate the identities of each other and securely exchange cryptographic keys that will be required to perform bulk data protection. The handshake may be performed with public key cryptographic techniques using one or more protocols such as Kerberos (initiated in [156]) or Internet Key Exchange (IKE) [63].

Upon completing the handshake, the two parties can securely transmit a large number of messages or data. We focus our attention on this bulk data protection step. Nearly all of the world's electronic communications travel over Internet Protocol (IP) networks [40, 127]. IP and associated transport protocols, such as the Transmission Control Protocol (TCP) [128], are the basis of the interoperability between the many heterogeneous networks that compose the Internet. We consider an implementation of the IP Security Protocol (IPsec) [76], which is used to achieve bulk data protection for messages that traverse IP networks. IPsec is an integral component of common secure networking systems such as virtual private networks. Within IPsec, the Encapsulating Security Payload (ESP) [75] and the Authentication Header (AH) [74] protocols enable the desired confidentiality and integrity features.

Figure 2.5 illustrates the operation of IPsec for the protection, transmission, and access of a single IP packet sent from Alice to Bob. Any digital message can be represented using one or more IP packets. In the figure, the dashed lines depict the trusted boundaries for Alice and Bob. P is the IP packet payload to be protected, and C is P in encrypted form. E_{sym} and D_{sym} are the symmetric-key encryption and decryption algorithms in CBC mode, respectively. The IV is the initialization vector used to enable CBC mode, and k_{sym} is the secret encryption/decryption key. H' is an HMAC function, k_{hash} is the secret key for use with H', and g is the hash result. Lastly, seq is a monotonically increasing packet sequence number that is used if protection against replay attacks is desired.

It is assumed that Alice and Bob securely established and exchanged the values of k_{sym}, k_{hash}, and seq during the handshake phase and prior to the IPsec operations. To protect an IP packet payload, Alice performs symmetric-key encryption over the payload for data confidentiality and then performs a keyed hash over the encrypted result for data integrity. The keyed hash is also performed over information such as the monotonically increasing packet sequence number, which prevents replay attacks. Data origin authentication is implicitly provided, as only Alice (or Bob) who possesses the secret keys will be able to construct a valid keyed hash of an encrypted packet and a sequence number.

Upon receipt of the protected packet, Bob employs his secret keys to decrypt the payload, validate the integrity of the packet, validate that the packet was generated by Alice, and validate the freshness of the packet. The characteristics of the cryptographic primitives enable the sender and receiver to ensure the security goals with overwhelming probability against an adversary who has access to reasonable amounts of computation and storage.

This cryptographic protocol effectively achieves the security goals against the defined

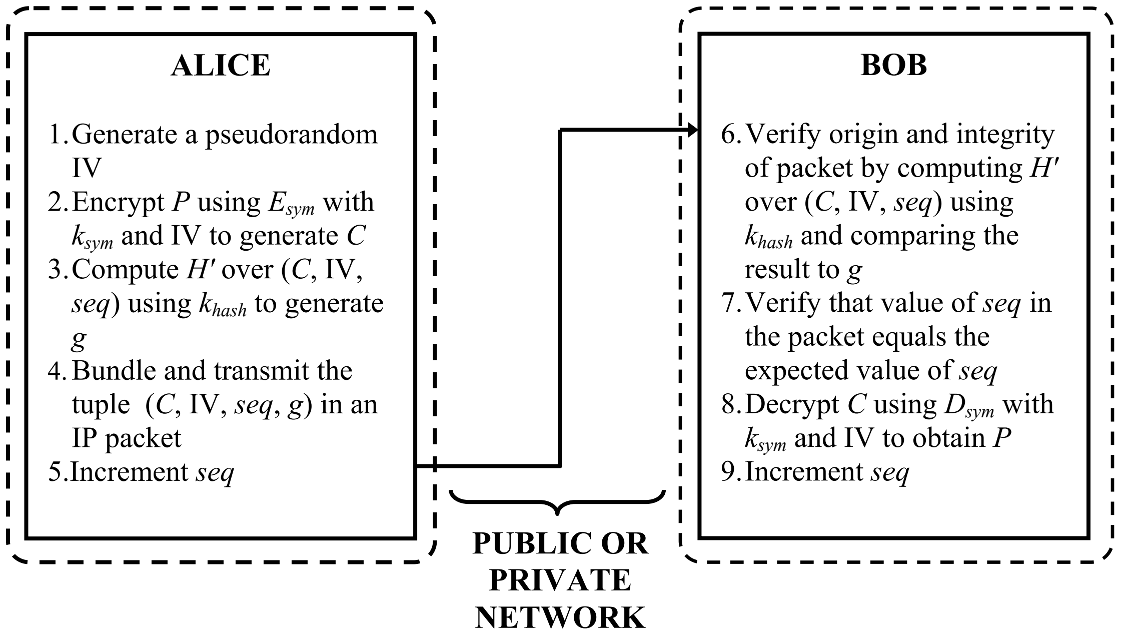

ALICE
1. Generate a pseudorandom IV
2. Encrypt P using E_{sym} with k_{sym} and IV to generate C
3. Compute H' over (C, IV, seq) using k_{hash} to generate g
4. Bundle and transmit the tuple (C, IV, seq, g) in an IP packet
5. Increment seq
BOB
6. Verify origin and integrity of packet by computing H' over (C, IV, seq) using k_{hash} and comparing the result to g
7. Verify that value of seq in the packet equals the expected value of seq
8. Decrypt C using D_{sym} with k_{sym} and IV to obtain P
9. Increment seq
PUBLIC OR PRIVATE NETWORK

Figure 2.5: Secure data transmission using IPsec

threat model, but two critical implementation problems remain. First, most systems fail to properly maintain security within trusted boundaries. Due to increasing number and seriousness of software vulnerabilities within computing systems, remote attackers can readily penetrate the boundaries defined above. Upon penetrating a trusted boundary, the attacker can expose or modify keys and data, thus circumventing the security features provided by IPsec. Second, encryption and hashing can be computationally intensive in general-purpose platforms. Potentially, each bit that enters a device from an external network may be encrypted and hashed, and therefore the cryptographic processing requirements increase as network bandwidth increases. As a result, the performance slowdown associated with the security mechanisms may inconvenience users or inhibit adoption. This thesis investigates approaches to enabling and securing cryptographic processing by addressing these two important problems.

2.4.3 Secure Data Storage Protocol Example

With the advent of networked storage and the increasing use of mobile computing devices, secure data storage systems are needed to combat new threats and attacks. We examine a simple approach to securing storage that operates similarly to the Pretty Good Privacy (PGP) file encryption protocol [8]. This protocol enables an individual entity to protect a unit of data storage, i.e., a file, and securely share the protected file with any number of other trusted entities.

We seek to defend against threats from unauthorized entities that may perform only the following actions on files while in storage or in transmission:

1. Observe the contents of a file
2. Modify or inject content into a file without being detected
3. Insert a forged file purporting to originate from an authorized user without being detected (i.e., spoofing)

As in the secure data transmission system, the trusted boundaries in this secure storage scenario are defined to be the physical exteriors of any authorized device or computer that seeks to access a file. Data that is stored outside of the computer — or even within the computer on the local hard drive or main memory in some cases — is subject to the threats elucidated above.

Given this boundary and threat model, we seek to achieve two security goals: data confidentiality and data integrity. Conventional storage systems are often exposed to many more threats and seek to enable many more security goals, but these two important goals will suffice for the purposes of this discussion. To guarantee data confidentiality, only authorized entities should be able to interpret bits of plaintext files. To guarantee data integrity, authorized entities must be able to (i) validate that a file has not been altered in any way by an unauthorized party and (ii) determine whether or not a file was created by an authorized party.

Note that we do not attempt to defend against attacks involving "file rollbacks," which are similar to replay attacks in the secure transmission scenario. In a file rollback attack, an attacker replaces the current version of a protected file with a different, "stale" version of the protected file. Furthermore, we do not attempt to defend against the destruction of files.

We now introduce a typical cryptographic protocol for achieving these security goals for files and other stored data. Like the secure transmission protocol, the secure storage protocol consists of two components: key distribution and bulk data protection. Prior to protecting a file that may be accessed by a set of authorized entities, the protecting entity must obtain the public encryption key associated with that set of entities. Also, before an entity can access a file to which the entity is privileged to view or modify, the entity must obtain the public digital signature verification key associated with the authorized entity that originally protected the file. This key generation and distribution is not trivial, but for the purposes of this discussion, we assume that all authorized users have previously obtained a public encryption key and a public signature verification key for each of the other authorized users in the set. Such pre-distribution could be performed using a variety of different public key infrastructures and techniques (e.g., [63]). Furthermore, for the sake of simplicity, we assume that the size and members of the set of authorized users are fixed and pre-determined. It is easy to extend this protocol to support variably sized dynamic sets of authorized users.

We now turn our attention to the bulk data protection of files. Figure 2.6 illustrates the protection of a file F followed by a subsequent access by an authorized user. In the figure, the trusted boundaries are depicted with dashed lines. Alice protects a file such that only authorized users Alice and Bob may access or properly modify the protected file in the future. It is assumed that, prior to the protection of the file, Alice and Bob securely establish and exchange public encryption keys and public signature verification keys. Also, Alice and Bob do not reveal their private decryption keys or their private signature keys to each other; Alice and Bob keep these keys secret.

To protect a file, Alice first computes a digital signature sig for the file F by computing a cryptographic hash g over F and then using an asymmetric-key encryption algorithm with her private (secret) signature key to encrypt the hash. The encrypted result is the signature, sig. This signature serves to ensure data integrity and to provide assurance that Alice, as opposed to any other user, generated the protected file. If a user other than Alice modifies the file in any way, that user cannot feasibly compute a new valid signature that appears to originate from Alice for the modified file. Next, Alice randomly generates a secret key k_{file} and an initialization vector IV that are used to encrypt the pair (F, sig) using a symmetric-key encryption algorithm in CBC mode. The key k_{file} is then encrypted twice, once with the public encryption key of Alice and once with the public encryption key of Bob, to obtain R_{Alice} and R_{Bob}, respectively. We encrypt the key k_{file} twice so that both Alice and Bob can decrypt and verify the file contents at any future time. These file and key encryption operations ensure the confidentiality of the file. That is, an unauthorized

F :	File to be protected
E_{sym} and D_{sym} :	Symmetric key encryption and decryption algorithms
E_{asym} and D_{asym} :	Public key (asymmetric) encryption and decryption algorithms
H :	Cryptographic hash function
g :	Output of cryptographic hash function
k_{file} :	Symmetric encryption/decryption key for the file and signature
IV :	Initialization vector for symmetric key encryption
ek_{Alice} and ek_{Bob} :	Alice's and Bob's public encryption keys
dk_{Alice} :	Alice's private decryption key (known only to Alice)
dk_{Bob} :	Bob's private decryption key (known only to Bob)
sk_{Alice} :	Alice's private signature key (known only to Alice)
vk_{Alice} :	Alice's public signature verification key
sig :	Digital signature produced by Alice
C :	The tuple (F, sig) encrypted using E_{sym} with k_{file}
R_{Alice} and R_{Bob} :	k_{file} encrypted using E_{asym} with ek_{Alice} and ek_{Bob}, respectively

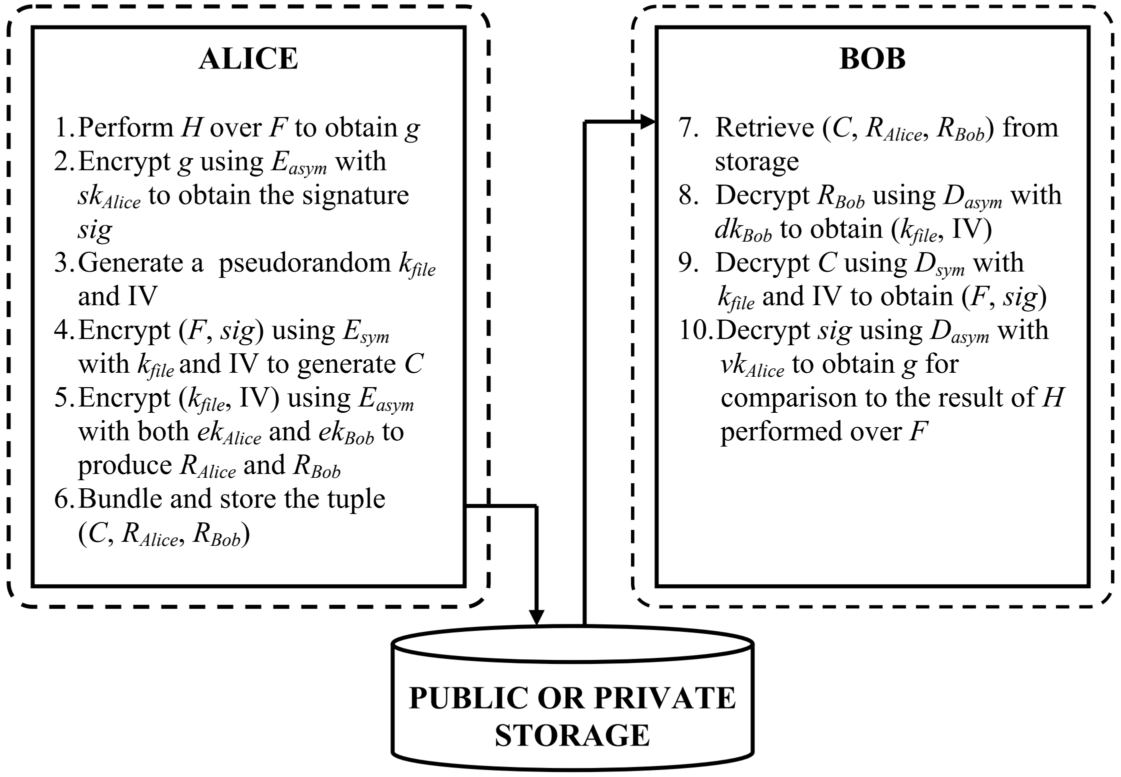

ALICE
1. Perform H over F to obtain g
2. Encrypt g using E_{asym} with sk_{Alice} to obtain the signature sig
3. Generate a pseudorandom k_{file} and IV
4. Encrypt (F, sig) using E_{sym} with k_{file} and IV to generate C
5. Encrypt (k_{file}, IV) using E_{asym} with both ek_{Alice} and ek_{Bob} to produce R_{Alice} and R_{Bob}
6. Bundle and store the tuple (C, R_{Alice}, R_{Bob})
BOB
7. Retrieve (C, R_{Alice}, R_{Bob}) from storage
8. Decrypt R_{Bob} using D_{asym} with dk_{Bob} to obtain (k_{file}, IV)
9. Decrypt C using D_{sym} with k_{file} and IV to obtain (F, sig)
10. Decrypt sig using D_{asym} with vk_{Alice} to obtain g for comparison to the result of H performed over F
PUBLIC OR PRIVATE STORAGE

Figure 2.6: Secure data storage and retrieval example

user will not be able to interpret the contents of the file or the digital signature without knowing either the private decryption key of Alice or the private decryption key of Bob. Confidentiality of the file therefore depends on the degree of secrecy of Alice's and Bob's private decryption keys. Finally, the two ciphertexts (R_{Alice} and R_{Bob}) are then stored with the encrypted result of (F, sig) as the protected file. The protected file may be stored inside or outside of the trust boundaries.

Upon retrieving the protected file, Bob decrypts the file encryption key k_{file} using his private encryption key, which is known only to Bob. Next, Bob decrypts the file F and Alice's digital signature sig using k_{file}. Bob then decrypts sig using Alice's public signature verification key to obtain the expected hash g of the file F. Lastly, Bob verifies the validity of the signature by computing a hash over the decrypted file F and comparing that hash result to g. If the signature is valid, by the properties of the cryptographic primitives involved, F is guaranteed to be authentic (i.e., created by an authorized user); uncorrupt (i.e., not modified by unauthorized users); and confidential (i.e., not accessible to unauthorized users). Many enhancements to this protocol may be applied to improve efficiency as well as to enable special security features.

With these cryptographic techniques, we can satisfy the security requirements against the threat model. However, this protocol suffers from the same two problems experienced by the secure transmission scenario. First, in most realistic systems, the trust boundary is not fortified. Attackers can breach the trust boundary via software vulnerabilities or physical attacks to obtain secret keys and sensitive plaintext data. Such breaches enable unauthorized parties to expose, inconspicuously modify, or spoof stored data. Second, in situations with intensive file I/O, the cryptographic procedures involved in the bulk data encryption may significantly negatively affect performance. Thus, the security and the efficiency of the secure storage protocol fundamentally depend on the measures taken to protect cryptographic keys and accelerate cryptographic primitives within the traditional trust boundary.

2.5 Architectural Opportunities for Cryptographic Software Security and Performance

In the past, software has assumed most of the responsibility for implementing security in general-purpose systems. Although specialized hardware modules may enable specific security functionality, general-purpose hardware provides few if any security features. However, software implementations of cryptographic protocols can suffer from vulnerabilities and performance issues that can prohibit or hinder the benefits offered by the protocols. Given the abundant hardware resources available in modern computing systems, many opportunities exist for using these resources to improve security. By applying low-cost enhancements to platforms and processors, we can enable more efficient and secure cryptographic processing than is currently available in software-only systems.

This section identifies a selected set of important challenges in enabling secure cryptographic processing that this thesis seeks to address. These challenges include protecting cryptographic keys in the presence of insecure software or hardware, accelerating bulk cryptographic primitives, and reducing security vulnerabilities in general and cryptographic software. This section presents overviews of the challenges and of the proposed architectural remedies. Subsequent chapters provide detailed analyses of past work and full descriptions of the proposed techniques for addressing the challenges.

2.5.1 Protecting Cryptographic Keys

The security provided by cryptographic primitives for many network and computer systems depends on the measures taken to ensure the secrecy and integrity of cryptographic keys. The exposure or corruption of such keys negates the benefits of secure transmission systems, secure storage systems, and other systems designed to satisfy security requirements for individuals, organizations, and enterprises. In many systems, vulnerabilities in software that resides within the trusted boundary threaten these critical keys. With enhancements to the processor and hardware platform, however, a higher degree of protection can be provided for keys and sensitive key computations. Specifically, special data and functions can be embedded in general-purpose hardware components that are not accessible by application and OS software. Using such hardware mechanisms in conjunction with special cryptographic software modules, we can provide strong and flexible protection for keys.

This thesis investigates architectural approaches to protecting keys in two generalized scenarios. The first scenario involves any type of keys that are utilized on a user's trusted local device. The second scenario involves special cryptographic keys that may be distributed to remote devices. Existing solutions to key protection problems suffer from a combination of disadvantages, including high cost, incomplete security, and poor throughput. By enlisting general-purpose hardware and new algorithmic techniques to defend and securely exercise cryptographic keys, systems can benefit from improved security and performance.

2.5.2 Accelerating Cryptography

Cryptographic techniques are integral to the protection of bulk data in communications and storage systems, but the computation requirements of cryptographic primitives can be highly expensive. It is well known that encryption and hashing can significantly decrease system performance by taxing memory bandwidth and processor resources (e.g., [22, 27, 73]). For example, the Secure Socket Layer (SSL) protocol [55], which provides cryptographically-enabled network security for HTTP transactions, may degrade system performance by a factor of 5 to 7 [73]. As secure systems become more pervasive, an increasing proportion of data and instructions that travel between the processor and the memory or I/O subsystems may be cryptographically protected by software. This trend will only exacerbate this performance problem.

In scenarios such as the secure storage and secure communications systems described above, the required cryptographic computation can be divided into two categories: (i) public-key encryption and (ii) symmetric-key encryption and cryptographic hashing. Although the public-key encryption is much slower than the symmetric-key and hashing operations, the public-key encryption routines are performed relatively rarely. That is, in the secure communication system, the handshake and public-key encryption may be performed once over a few kilobytes of data prior to performing the bulk symmetric-key encryption of gigabytes of data. As a result, in many instances, the total computation required to support bulk data protection exceeds the total computation required to perform key exchange and related public-key operations during the handshake [22, 27, 51, 52, 129]. Thus, to minimize the potential performance impact of the needed security operations, special attention should be paid to the acceleration of symmetric-key cryptography.

Several software-based techniques exist for accelerating symmetric-key cryptographic primitives. Examples of these techniques include general code optimization and new cryptographic algorithms for improved performance. For instance, recent symmetric-key encryption algorithms such as AES have been designed to execute very efficiently in software. Other software methods, such as compressing data prior to encryption [106, 107], can also lead to substantial performance gains.

There exist many opportunities to improve the performance of symmetric-key encryption by adding special features to the processor hardware. By enriching the instruction set architecture (ISA) with new instructions that incur low implementation costs, symmetric-key encryption routines can be greatly accelerated. In particular, the highly popular Data Encryption Standard (DES) and other encryption algorithms employ bit-level permutations and mappings to achieve the cryptographic property of diffusion [144]. Existing general-purpose processors cannot perform bit-level mappings efficiently, however.

This thesis proposes a set of general-purpose processor architecture enhancements for accelerating bit-level permutations and mappings. We can apply these enhancements to significantly improve the software performance of certain cryptographic primitives. Considering that stored data and network traffic is cryptographically protected with increasing frequency, such built-in changes to the general-purpose hardware are certainly prudent and desirable.

2.5.3 Mitigating Common Software Vulnerabilities

The general software vulnerabilities that plague operating systems and application software can also undermine the security offered by cryptographic software. If such a vulnerability is exploited within a cryptographic software module, the attacker may expose or corrupt critical cryptographic keys. Furthermore, if software weaknesses are employed to compromise non-cryptographic privileged software modules running in the system, those compromised modules may be employed to extract secret keys from cryptographic procedures that are running in other system threads. Thus, to ensure protection for cryptographic keys and computations, we must address common security vulnerabilities in general software.

Two common classes of low-level software vulnerabilities include *format string vulnerabilities* and *buffer overflow vulnerabilities*. Format string vulnerabilities enable attackers to exploit particular weaknesses in programs that employ C libraries. Specifically, an attacker can provide special inputs to applications that are subsequently passed to C functions such as `printf` and `scanf` and expose or corrupt data stored in memory. Recent proposals such as the FormatGuard seek to prevent format string exploits via software defenses [35].

We focus on buffer overflows, which have proven to be the most frequent vulnerability involved in system security breaches [4]. Buffer overflow attacks involve the input of a specially crafted string to a software module that exceeds the size of a memory buffer. As a result, particular contents of memory may be undesirably overwritten. Buffer overflows can occur in the stack (e.g., [1]) or in the heap (e.g., [34]); the overflows often involve the corruption of pointer values or corruptions of branch, jump, or return addresses. PointGuard is one example of a software defense that addresses pointer corruption [36].

This thesis considers the most common buffer overflow attack: overflows in the software stack that overwrite procedure return addresses. With this attack, adversaries can remotely inject and execute malicious code, which may enable the adversary to seize control of a victim system. This thesis proposes a hardware approach to preventing software buffer overflow attacks. Low-cost architectural enhancements can be employed to transparently prevent buffer overflow attacks involving procedure return address corruption for cryptographic software or any other software that is running on the system. Such built-in protection is an important step towards mitigating inevitable security vulnerabilities in complex software.

2.6 Summary

Cryptography is an essential tool for achieving system security goals. As computing devices become increasingly interconnected, cryptography will be increasingly employed to satisfy security requirements such as confidentiality, integrity, and authentication. Popular cryptographic primitives include symmetric-key encryption, secure hash functions, and asymmetric-key encryption. The security offered by these operations fundamentally depends on the secrecy and integrity of small pieces of information known as cryptographic keys.

Secure data communications and storage systems often employ multiple cryptographic primitives to provide security against threats that occur outside of a defined trusted boundary. Considering the increasing number of software vulnerabilities and the growing complexity of computing systems, however, new measures must be taken to protect cryptographic keys and cryptographic software inside and outside of the trusted boundary. Furthermore, as cryptography becomes more pervasive, steps must be taken to accelerate primitives to avoid significantly impacting system performance.

Traditionally, processors and platform hardware have supported few security features. As the value of information and computing resources continues to grow rapidly, however, hardware architects can and should take advantage of the plentiful hardware and processor transistor resources to improve security. Specifically, low cost hardware enhancements can be implemented to protect cryptographic keys, accelerate cryptographic primitives, and prevent common attacks on software.

The remainder of this thesis proposes and examines architectural techniques for improving cryptographic security and performance. Chapters 3 and 4 present new techniques for efficiently protecting users' and content providers' cryptographic keys, respectively. Chapter 5 proposes architectural enhancements for accelerating subword permutations and mappings that significantly improve the performance of popular encryption algorithms. Chapter 6 presents a hardware-based defense that mitigates certain vulnerabilities in software by preventing buffer overflow attacks. Chapter 7 concludes the thesis.

Chapter 3

Virtual Secure Coprocessing

Cryptographic processing is a principal enabler of secure network, computer, and data processing systems. However, the protection offered by cryptographic processing greatly depends on the methods employed to manage, store, and exercise a user's cryptographic keys. Existing mechanisms for cryptographic key protection suffer from combinations of user inconvenience, inflexibility, performance penalties, and high cost.

This chapter presents Virtual Secure Coprocessing, VSCoP (which is pronounced *vees-cop*), for the protection of users' cryptographic keys. We describe architectural and software enhancements for defending keys in general-purpose platforms against diverse physical and software attacks. The contributions of this chapter are based in part on the work previously published by the author in [109, 108, 95].

This chapter is organized as follows. Section 3.1 introduces threats to cryptographic keys. Section 3.2 discusses and compares past work in cryptographic key protection. Section 3.3 presents a new approach for protecting keys in general purpose platforms: VSCoP. Section 3.4 details the hardware and software enhancements involved in the implementation of the proposal. Section 3.5 explains how such an enhanced system can serve as a virtual secure coprocessor via user initialization, device initialization, and protected operation. Section 3.6 analyzes the security benefits of VSCoP. Section 3.7 investigates the performance impact of the proposal. Section 3.8 discusses possible extensions and design alternatives, and Section 3.9 summarizes the chapter.

3.1 Threats to Cryptographic Keys

For the purposes of this chapter, cryptographic keys consist of any secret information that is used directly or indirectly as the input key in a cryptographic operation. Examples of keys therefore include AES keys [120], decryption exponents, passphrases, PINs, biometric data, and even credit card numbers. We refer to a user's collection of cryptographic keys as the user's *key ring*. In common platforms such as personal computers, users often perform cryptographic operations "in the clear". Performing operations in the clear means that the users temporarily or permanently store their secret keys and associated sensitive information in unprotected system memory or other storage devices. When a user exercises secret keys in an unprotected manner, an unauthorized party may inspect the contents of memory to obtain the secret key material. As publicized by CERT [32] and the FBI [138], such system penetration can be realized by exploiting one of the numerous security vulnerabilities that occur in operating systems and application software. In addition, since the secret key is often a small quantity of information — perhaps only 16 bytes in size — an attacker may expose and make use of the secret key faster than the user can react to an intrusion.

Following the compromise of a key ring, the user must initiate the painful process of revoking certificates, resetting PINs, changing passwords, etc. If the user is unaware of such exposure or the user requires considerable time to complete the key revocation process, a malicious party can inflict significant damage. Such damage may include irreversible disclosure of medical records, theft of private correspondence, and unauthorized access to copyrighted audio and video. If cryptographic keys protect valuable assets such as online banking accounts, the results of key compromise can be truly devastating. The management and protection of cryptographic keys are therefore critical components of secure computing systems that utilize cryptographic operations.

We are concerned with the four classes of threats to cryptographic keys: threats to keys in storage, threats to keys in transport, threats to keys exercised by software, and threats to keys exercised by hardware. These classes, which we describe below, include several common and imminent threats against which we can construct practical defenses. In the descriptions of the classes, “authorized” means permitted or accepted within a given security policy, and “unauthorized” means not permitted or accepted within a given security policy.

3.1.1 Threats to Keys in Storage

This class includes threats to the confidentiality, integrity, and freshness of keys maintained in any electronic storage device. The set of possible storage devices includes (but is not limited to) tapes, disks, main memory, caches, and processor registers. With respect to these threats, key confidentiality implies that an unauthorized entity may not explicitly or indirectly obtain actual bits or other information about the key material. Key integrity implies that unauthorized entities may not inconspicuously spoof or transpose key material. Spoofing involves the substitution of key material with forged key material that purports to be valid. Transposition, which is also known as splicing, involves the rearrangement of bits within the key material. Key freshness implies that unauthorized parties cannot inconspicuously substitute valid key material with key material that was previously valid but now expired.

3.1.2 Threats to Keys in Transport

This class includes threats to the confidentiality, integrity, and freshness of keys during transport between two or more trusted boundaries (as defined in Chapter 2). Examples of transport vehicles for keys include the Internet, Ethernet networks, USB cables, and PCI

I/O buses. The definitions of the threats to confidentiality, integrity, and freshness of keys in this class are the same as those described for threats to keys in storage.

3.1.3 Threats to Keys Exercised by Software

This class includes threats stemming from software instructions executed on sensitive key material. Such code may threaten the confidentiality, integrity, and freshness of user keys. In addition to manifestations of these threats caused by malicious code, these threats can stem from incorrectly implemented trusted code or from compromised trusted code that employs keys to perform cryptographic operations.

3.1.4 Threats to Keys Exercised by Hardware

This class includes threats relating to any hardware device that performs operations on cryptographic keys or that executes software that exercises cryptographic keys. Examples of such hardware devices include general purpose or embedded processors. Like the code-related threats, this class includes threats to the secrecy, integrity, freshness, and authorized use of keys. Hardware involved in this class should be distinguished from storage and transport mechanisms that maintain or transmit key material but do not exercise the key material to perform any useful computation.

Note that the four threat classes are not mutually exclusive. Corruption of a thread's virtual memory space that contains both code and data pages, for instance, could be included in more than one of the classes. Furthermore, for each threat class, there exist both *physical* and *software* manifestations of the attacks. Physical attacks involve mechanisms such as bus probing or physical theft that lead to the undetectable modification or exposure of keys. Software attacks involve the locally or remotely launched execution of software, such as a virus that erases keys or spyware that reveals keys.

3.2 Past Work

Given these threats, many solutions have been proposed for the protection of keys. Due to the numerous security vulnerabilities that continue to plague software, however, local software-only key protection techniques are unsatisfactory. A software intrusion that exploits a common vulnerability may enable an attacker to remotely penetrate a network-connected device and expose keys that provide access to all of a user's sensitive information. Therefore, as we describe below, more robust key protection schemes involve a set of distributed hosts or a protected hardware device. However, these existing key protection mechanisms suffer from combinations of disadvantages, which may include high cost, poor performance, inconvenience to users, lack of adequate software compartmentalization, and protection that only covers limited types of keys.

We summarize prior work concerning distributed software-based and hardware-based key management schemes. Some techniques protect vendors and content providers from copyright violations and software piracy in untrusted hosts, and other techniques protect users from physical theft and attacks by malicious code.

3.2.1 Software-based Techniques

Several distributed software-only approaches protect certain types of cryptographic keys by forcing an adversary to compromise several hosts in a short time period in order to reveal those keys. Other distributed software-only approaches enable effective revocation mechanisms when key information is exposed. Some proposals allow a user to reconstruct cryptographic keys prior to use by engaging in a secure protocol that involves the participation of several servers (e.g., [53, 58]). In other approaches, users can perform certain cryptographic primitives (such as encryption) that employ secret keys with the aid of separate servers. Using particular mathematical features of certain ciphers, a key can be securely split into two or more shares that are distributed to two or more machines. By the construction of the system, a certain number or subset of these key shares and participating machines are required to complete a cryptographic operation. Thus, when a client device or a server is compromised, the keys can be permanently disabled by destroying or modifying some of the key shares (e.g., [101]).

These and other distributed schemes only effectively defend against specific attacks that involve limited types of keys, e.g., RSA decryption keys. This restriction results from the fact that most distributed schemes employ mathematical characteristics of certain ciphers that are not present in several other common ciphers (such as AES).

3.2.2 Cryptographic Coprocessors and Tokens

One of the first proposals to suggest using physically secure hardware processing devices to enable security features unattainable by software-only techniques was presented in [12]. Since that time, researchers have proposed a rich variety of applications and architectures for such hardware (e.g., [62, 164]). These physically secure devices perform cryptographic operations and other services using secret information that cannot be extracted from the hardware device. Examples of such devices include highly fortified cryptographic modules and cryptographic smart cards.

The IBM secure coprocessor boards are high-end tamper-resistant hardware modules that perform cryptographic operations (using secret keys), secure booting, and secure program loading for applications requiring a high level of security such as banking systems [47, 152, 153]. These products offer exceptional physical security for cryptographic keys, but they are too costly, inconvenient, and bulky for mobile and desktop computers. In addition, they are often shipped with factory-installed secrets, which may lead to several security and privacy problems resulting from threats to a factory's database.

Extremely low-cost, portable alternatives to cryptoprocessors and secure coprocessors are cryptographic tokens. These devices include smart cards, PDAs [9], and other small, physically tamper-resistant hardware components [4]. Some tokens simply protect user secrets by requiring a password to access the information stored within the token, and other tokens perform cryptographic primitives using the stored secrets without leaking key information to the untrusted environment [4, 14]. These devices cannot provide the same degree of security as powerful cryptoprocessors, but they cost much less and they facilitate increased user convenience. However, these devices have restricted capabilities: performance can be poor and the number of supported cryptographic primitives and protocols is often limited. Also, these devices are highly susceptible to loss and theft, and physical tamper resistance is difficult to implement at low costs [5, 84].

3.2.3 Trusted Computing Platforms

Existing, publicly known trusted computing platforms provide some degree of protection for users' cryptographic keys. The Trusted Computing Group (TCG) [163], which was formerly known as the Trusted Computing Platform Alliance; Intel's LaGrande Technology (LT) [68]; and ARM's TrustZone technology [6] seek to provide certain hardware-enabled security features for computing devices. These technologies support varying combinations of system attestation, protection of system inputs, secure booting, and process isolation. In these systems, secret information that is inaccessible to the end user is embedded in hardware modules such as on-board cryptographic coprocessors or general-purpose processors (in LaGrande systems). Microsoft's Next Generation Secure Computing Base (NGSCB) [115], formerly known as Palladium, seeks to provide resources for secure (i.e., validated and isolated) code execution via trusted hardware computing platforms. With such operating system support, a trusted device can complete operations such as verifying the integrity of installed software and preventing unauthorized access to copyrighted media and code.

A user can employ certain hardware resources provided by a trusted computing platform to encrypt sensitive cryptographic keys for storage on a single device, but keys must be exposed to software in the system to perform computations. Furthermore, the platform may ensure that the keys are only released to trusted software environments, but these trusted environments might be vulnerable to compromise. That is, trusted components are designed and engineered for high assurance, but the trusted software environment still may be vulnerable to software bugs that could lead to the exposure of sensitive cryptographic keys. As evidenced by software vulnerability reports, bugs in kernel and application software are commonplace and can enable the complete subversion of the trusted computing platform mechanisms that provide protection for user secrets.

In addition, the current TCG Trusted Platform Module and Microsoft's NGSCB do not defend against attacks on the device hardware. For instance, by physically monitoring and/or modifying data in the system buses and main memory, some security features of the trusted computing platform can be defeated.

3.2.4 General-purpose Architecture for Secure Computation

Techniques for incorporating cryptographic functionality into general-purpose processor architecture have also been proposed. Recent work has addressed processor-based mechanisms for authenticating trusted software and verifying the integrity of physical memory [59, 81, 160]. In addition, by adding encryption and data authentication capabilities to general-purpose processors, it is possible to enable shielded program execution [60, 100, 160]. Such systems, e.g., eXecute Only Memory (XOM) [100] and AEGIS [160], prevent unauthorized modification and observation of software execution by untrusted components outside of the processor chip. This involves cryptographically authenticating instructions as well as shielding or obfuscating sensitive data via hardware compartmentalization or via encryption and secure hashing.

The primary objective of protected execution in these proposals is the prevention of software-based tampering or of proprietary code exposure. These proposals enable the protection of external parties' software when being employed on an end user's machine, but they do not provide strong mechanisms with which users can protect their secrets on their machines from external parties. For example, XOM can shield the local execution of digital rights management (DRM) software that is provided by a remote content provider. In this scenario, XOM can protect the provider's keys that are used by the DRM software to verify and/or enable user access to the valuable content.

XOM and AEGIS were not designed to protect users' keys, however, and therefore issues relating to user key security remain to be addressed. First, these two proposals do not support the mechanisms for securely transporting users' keys to protected storage within the processor. Second, although XOM and AEGIS can shield program execution, they are not designed to restrict the nature of the operations performed by software. These proposals allow any software to potentially access and employ all of a user's secrets. Thus, malicious code or buggy programs (such as the SSL module in a web browser) may reveal sensitive user information. Third, XOM requires public-key encryption techniques and key values to be incorporated into the processor at the factory. This may lead to performance and privacy issues.

3.3 A New Approach to Key Protection

Our goal is to efficiently improve security for users' secret keys while in storage, in transport, or in use on general-purpose platforms. Because of increasing network connectivity and the growing exploitation of software security vulnerabilities, remotely launched software attacks are our principal concern in this chapter. Although we can prevent some physical attacks, our efforts are focused on software-based attacks. We seek to defend keys against common threats by achieving the following security goals:

- Confidentiality (secrecy), integrity, and freshness for certain data that (i) represents secret keys or that (ii) can be used to infer useful information concerning secret keys

- Integrity and authorization of certain software that performs computations on secret keys

3.3.1 Virtual Secure Coprocessing

To realize the security goals described above, we introduce Virtual Secure Coprocessing (VSCoP). In VSCoP, the general-purpose processor effectively shields cryptographic keys and operations when needed. VSCoP enables secure and efficient key utilization, storage, and transport in the presence of potentially vulnerable networks, application software, and operating systems. The performance and implementation costs of VSCoP are modest; VSCoP does not require any auxiliary processors or additional hardware devices. From the point of view of the user and of application software, however, a virtual secure coprocessor functions as a separate secure coprocessor. By preventing unauthorized exposure or use of sensitive keys, VSCoP can enhance security for many applications.

VSCoP protects keys by effectively constricting the traditional trusted boundary and modifying the traditional access control paradigm in relation to users' cryptographic keys. The trusted boundary of a computing device is the boundary that separates the trusted domain from the untrusted environment. In many systems, disks, memory, and other peripherals are treated as trusted and are assumed to be safe from external threats. However, many relatively simple software and physical attacks can successfully obtain sensitive key information that may be stored in these devices. Such attacks include the inspection of sensitive swap files stored to disk, examination of other applications' virtual memory spaces, and network sniffing in distributed shared memory applications.

As illustrated in Figure 3.1, to thwart such attacks, VSCoP restricts the trusted boundary (indicated by the dashed lines) for cryptographic keys in the system to the physical boundary of the general-purpose processor chip. Memory that is off the processor chip, network interface cards, disks, buses, and any other peripherals will now be treated as untrusted.

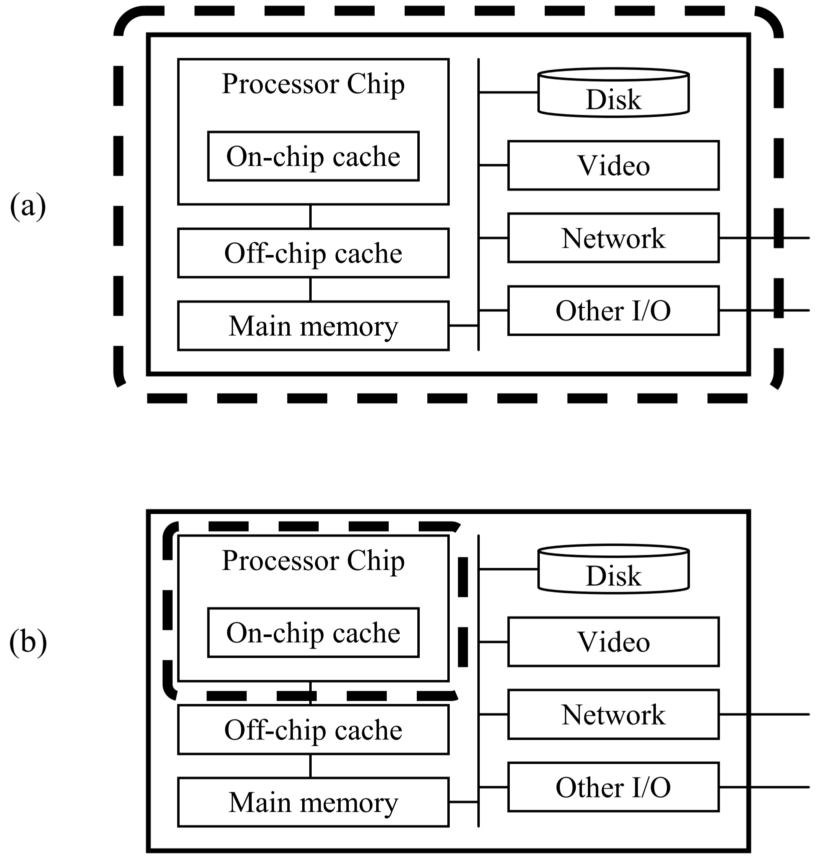


Figure 3.1: (a) Traditional and (b) proposed trusted boundaries for critical secrets

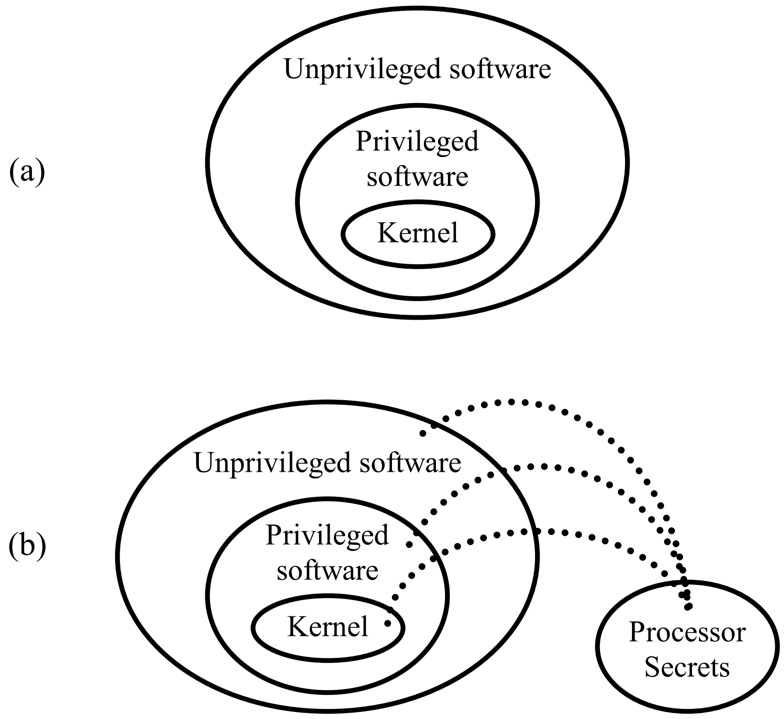


Figure 3.2: (a) Traditional and (b) proposed access control paradigms

In systems where the OS enjoys access to all system secrets, if the OS is penetrated, then all system secrets will be exposed. Thus, to avoid reliance on a potentially vulnerable operating system, we create a new disjoint region in the access control paradigm. This change to the access control paradigm is shown in Figure 3.2. The new region consists of processor-protected secrets that are inaccessible from the application software as well as from the OS kernel. The OS and other software can only perform operations using the secrets through a special hardware/software interface, which is illustrated by the dotted lines in Figure 3.2(b). The new region is not included within the kernel because operations that are permitted to execute within the new region do not require (and should not be allowed to) access all system secrets.

Although trusted computing platforms seek to achieve a isolated trusted domain (e.g., [163]) similar to that of VSCoP, the platforms do not ensure special protection for users' keys. That is, the compartmentalization features of existing trusted computing platforms can only provide the long-term protection of keys if the OS and certain application software prove to be perpetually impenetrable. The approach proposed in this chapter can provide protection for keys in certain scenarios where the OS or application software suffers a security breach.

We emphasize that the proposed new approach is *not* designed to replace trusted computing platform components. Trusted computing platform services, such as secure bootup and attestation, are essential to achieving robust system security. By enabling additional protection for highly sensitive pieces of information, e.g., cryptographic keys, the new approach complements rather than supplants the security services provided by these trusted computing platforms.

3.3.2 VSCoP Components

VSCoP consists of mechanisms that efficiently provide protection for keys while in storage, in transport, and in use. We provide a summary of the mechanisms here, and detailed architectural implementations are presented in Section 3.4.

Key Storage Protection

To protect keys in storage on devices outside of the processor's trusted boundary, VSCoP defines a special user key ring structure that is encrypted and cryptographically hashed. Since the ring is cryptographically protected for confidentiality and integrity, the ring can be deposited in any trusted or untrusted storage device. The key ring is only decrypted and exposed when being employed within the processor's trusted boundary.

Key Transport Protection

The encryption and hashing of key rings protects the rings while in transport between devices. Furthermore, to protect user authentication information while in transport from the user to the processor, VSCoP provides a protected I/O path. This protected path is implemented by the hardware platform and is designed to shield the sensitive information from any potentially vulnerable software (which includes the operating system kernel).

Key Exercise Protection

VSCoP protects key rings during use by shielding operations performed on users' secret keys on general-purpose processors against observation and tampering by attackers. In VSCoP, this goal is realized via the Concealed Execution Mode (CEM) and the Cryptographic Operations Library (COL). To securely exercise keys, software applications make calls to the COL (which runs in the CEM) as if the COL were a physically secure coprocessor.

The Concealed Execution Mode is a mode of operation in which the general purpose processor cryptographically protects and authenticates any data or code that enters or leaves the processor's trusted boundary. User key rings can only be accessed or modified by a thread running in the CEM. Invoking the Concealed Execution Mode does not require the suspension of ordinary threads, however. VSCoP enables secure context switching between CEM and non-CEM threads, and therefore multithreading functionality is not sacrificed.

The Cryptographic Operations Library (COL) is the only software module that is permitted to execute within the Concealed Execution Mode and exercise a user's cryptographic key ring in unencrypted (i.e., exposed) form.[1] The COL consists of cryptographic routines that perform operations on a user's secret keys. Most legacy application code does not need to be changed to implement VSCoP. Only applications that wish to invoke concealed execution would need to be modified to call the Cryptographic Operations Library.

[1]The operating system and other applications can readily obtain a user's key ring in protected (i.e., encrypted) form.

3.3.3 VSCoP Benefits

VSCoP provides many benefits for individual users, which include the following:

- **Improved Security.** Users can employ secret keys to perform computations on general-purpose platforms with improved protection against the potentially vulnerable software and hardware environment.
- **Protection for Many Key Types.** We seek to ensure the security of any cryptographic keys, whereas some key management schemes protect only limited classes of keys such as RSA keys.
- **Ubiquitous Key Access.** Users can conveniently and securely access their cryptographic key ring from *any* network-enabled device (that contains our enhancements). Furthermore, users do not have to carry and use a smart card or other protected, auxiliary hardware devices, which are susceptible to loss and theft. VSCoP-enabled devices also do not need to be pre-authorized in order to securely utilize secret keys, as may be required in existing systems.
- **Flexible Cryptographic Functionality.** Since the cryptographic primitives that we provide to applications are implemented in software instead of hardware, the system can support a wide range of evolving security functions.
- **Improved Performance.** Users also benefit from the high performance of general-purpose processors as opposed to the potentially low performance of constrained cryptographic processors found in some smart cards and other cryptographic tokens.

3.4 Architectural Implementation

The architecture of VSCoP is based upon two secrets that are stored and protected within the general-purpose processor: the *user secret* and the *device secret*. The user secret is the master key of the user's cryptographic key ring. This user master key is used to provide protection for the entire key ring via encryption and authentication, and it is maintained by the processor in protected volatile memory for limited periods of time. The device secret is used by the processor to perform a variety of security functions that enable protected storage and utilization of the user's secret keys. This device secret is maintained in the processor in protected non-volatile memory for extended periods of time.

This section describes the architectural enhancements that enable protection for keys and secrets while in storage, in transit, and in use by the processor. These enhancements begin with the definition and maintenance of a protected key ring structure for secure key storage. Next, we present the protected I/O path for securely transporting the user secret and the device secret to the processor. Lastly, we discuss the enhancements that support protected key utilization, which include a new software library, processor modifications, ISA additions, and minor OS modifications.

3.4.1 Key Ring Structure

We define a structure for a user's cryptographic key ring that facilitates ease of use and strong protection even when stored in untrusted devices. Figure 3.3 shows an example of the organization of a cryptographic key ring, which potentially can contain many more keys than depicted in the figure. A key ring includes a single user master key that is used to encrypt and authenticate the integrity of all of the other keys, and thus the security of the key ring fundamentally depends on the measures taken to protect the master key. In addition, since all the keys in a key ring are cryptographically protected by the user master key, a user can deposit his key ring (minus the master key) in a publicly accessible network or storage device without risking key exposure, undetectable forgery, or undetectable corruption.

In this paper, we define user master keys to be 128-bit keys for use in symmetric-key encryption or keyed hashing algorithms. Furthermore, we define this user master key to be

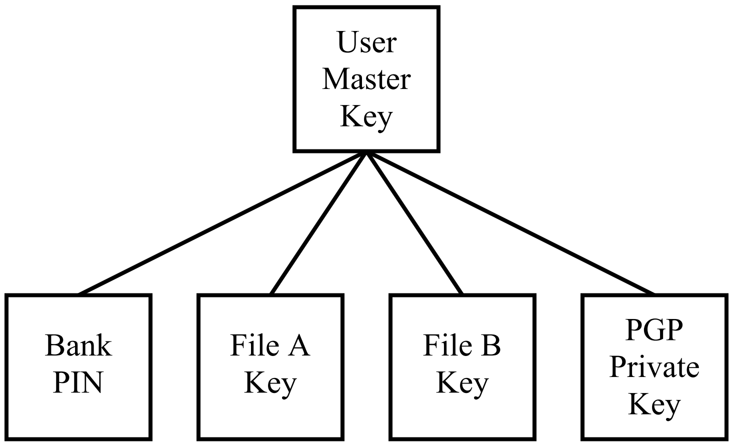


Figure 3.3: Example key ring structure

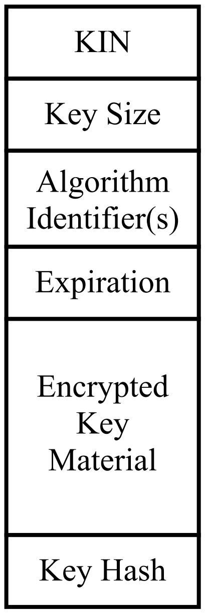


Figure 3.4: Structure of a single key

the output of a cryptographically-strong one-way hash of the user's passphrase (although this could certainly be supplemented with hardware token information in practice). Hence, users should carefully select passphrases with sufficient entropy to thwart off-line attacks [151, pp. 87–94].

Figure 3.4 depicts the organization of an individual key. Each key consists of a key identification number (KIN), the key size, an algorithm identifier, the key itself in ciphertext form (encrypted with the user master key), the key hash, and an expiration information field. The KIN is a non-secret 256-bit integer that uniquely identifies the key. To avoid possible attacks where an attacker may specify a KIN that would lead to a collision with a KIN of another key, VSCoP provides a pseudorandom number generation mechanism (described below) in the trusted processor that can be used to randomly generate KINs. Since the KIN is 256 bits in size and is randomly generated, the probability is negligible that a KIN would be used more than once. The key size is simply a 64-bit integer that represents the size in bytes of the individual key structure. The algorithm identifier specifies the algorithm (or set of algorithms) permitted to use the key. This field may also be used to specify other relevant information about the nature and proper use of the key.

The key is encrypted with the user master key. Furthermore, during key creation, the VSCoP pseudorandom number generator can be used to generate pseudorandom key bits. The key hash is the keyed cryptographic hash message authentication code (HMAC) based on the user master key for the entire key data structure (minus the key hash) that we can use to verify the integrity of the key. This hash guards against adversaries that seek to forge or inject bogus keys into a user's key ring. Examples of algorithms and modes that we can use to perform the encryption and hashing include AES-128-CBC and HMAC-SHA-256, respectively [114].

The expiration information field is a 64-bit integer that may be used to assist in a key expiration and revocation protocol. Such a protocol is not presented in detail here, but if a trusted clock is available and the expiration field is set to an appropriate value at key creation time, VSCoP can securely enforce the expiration of a key. However, if the user deletes a key from the key ring at an arbitrary time, the protocol may not fully prevent an attacker from successfully launching a “rollback attack”, which is the storage analogue of a transmission replay attack. In this attack, the latest version of a key ring may be replaced inconspicuously with a stale version (i.e., a formerly valid but not currently valid version) of the key ring.

We identify two methods for preventing such rollback attacks in VSCoP. In the first method, upon key revocation, the entire key ring is re-encrypted and re-hashed with a new master key that is based on a new passphrase. In the second method, the user must employ a trusted hardware device (e.g., a smart card or a USB token) during authentication or key ring retrieval. This device would store a small quantity of information, such as a key ring hash fingerprint, which could be employed to verify the freshness of the key ring retrieved from untrusted storage. Both of these methods achieve security against rollback attacks but may inconvenience users.

We encode keys and key rings for storage as searchable trees to facilitate efficient key ring traversal by software. Users can implement alternative secure key ring structures if other security and usability features are desired. For example, users can employ a hierarchical key ring based on a tree structure where parent keys (in addition the user master key) encrypt child keys [109, 108, 95].

3.4.2 Platform Enhancements and Protected Paths

In VSCoP, the hardware platform assumes responsibility for securely transporting the user secret to the processor and for securely resetting the device secret. That is, the hardware platform implements the protected I/O path for transporting and managing the secrets upon which the VSCoP architecture is built. The new platform features that enable this protected path include an “Authenticate” button, a “Device Reset” button, and a three-color VSCoP

Status Light, which are all located on the exterior of the device. The Authenticate button primarily relates to the user secret key, and the Device Reset button primarily relates to the device secret.

Upon receiving a used or new device, the new device owner should reset the device secret. The VSCoP device secret will be used to cryptographically protect the new user's keys, so it is critical to guarantee that neither the factory nor a previous owner will have knowledge of the device secret employed by the new user. Also, for similar reasons, before transferring the device to an untrusted party, it is necessary to reset the device secret to a random value or to zero. Thus, we must provide support for resetting (i.e., zeroizing) the device secret stored within non-volatile memory in the processor.

Furthermore, this feature needs to be tied to a physical action in order to prevent a software attacker from replacing the device secret with one that the attacker may use to expose key bits. We can prevent such an attack by implementing a physical "Device Reset" button (similar to that of many PDAs) that must be physically pressed while the device is turned on in order to reset the device secret registers in the processor to zeroes. The platform can confirm a successful reset by illuminating a new VSCoP Status Light (or LED) on the exterior of the device to "red" when the device secret equals zero. Upon writing new values to the device secret registers, which the processor will permit to occur only after the device secrets have been physically reset, the VSCoP Status Light is set to "blue". Note that only the platform hardware (and not any software) can respond to the Device Reset button or influence the VSCoP status light. This is enforced simply by not providing an interface with which OS or application software could observe or interact with the Device Reset button and VSCoP status light hardware.

The hardware platform (rather than the potentially vulnerable OS) also assumes responsibility for gathering and hashing the user authentication information to generate the user secret. During user authentication, the platform temporarily prevents keyboard or similar input from reaching OS I/O buffers. Instead, the platform sends these user inputs (e.g., a passphrase) directly to the processor chip. The processor then hashes the information to obtain the user's master key (which is the user secret). A user initiates this procedure by pressing a special "Authenticate" button on the device. While the user authentication information is being inputted (via a keyboard or another input device), the VSCoP Status Light blinks "green". After the operation is complete, which occurs after the "Enter" key is pressed, the platform turns the Status Light to a solid green to indicate that user authentication information is loaded into the processor.

Although the platform hardware can inform the OS that the user is entering authentication information, the hardware does not allow any software to intercept this authentication data. Hence, we avoid "man in the middle" attacks from malicious or corrupted kernels. However, we do not prevent more complex physical attacks in which an adversary steals a device, installs a "sniffer" that can intercept user authentication information at the hardware level, and then returns the device to the oblivious user.

After a user has used his keys to complete a particular task (such as a remote electronic vote), the user may wish to wipe all traces of his key ring from the device. By performing such a wipe, no future system software or hardware security breaches can reveal or employ any of the user's keys. To achieve this goal, the user can press the "Authenticate" button to inform the processor to clear the user secret and any sensitive state information from all relevant locations in the processor. Following the successful wipe, the platform turns the VSCoP Status Light from "green" to "blue" to indicate that the user secret is no longer contained in the device.

3.4.3 Cryptographic Operations Library

We now turn our attention to support for protected key use. The Cryptographic Operations Library (COL) is a trusted code module that applications can employ to securely perform cryptographic procedures with a user's secret keys. This library is the only software that is authenticated using the device secret and permitted to employ the Concealed Execution Mode. We envision the COL as being an operating system component, but application software developers could certainly develop and distribute this library as well.

Calls to the COL by applications can occur using one of two methods, as shown in Figure 3.5. In the first method, as illustrated in Figure 3.5(a), applications make calls directly to the COL interface. The arrows represent function calls, and the rectangles along the interface represent software function entry points. In the second method, as illustrated in Figure 3.5(b), applications make calls to a COL wrapper that is presented by the underlying OS. Then, the OS may or may not propagate the call by making a subsequent call to the COL on behalf of the application. The second method allows the OS to restrict access to the COL to privileged or highly trusted software applications. In the remainder of this chapter, we assume that we implement VSCoP using the second (i.e., OS-mediated) COL calling method.

The COL does not run as an independent application or thread; instead, the COL is a shared library that can be dynamically loaded (much like a DLL in Windows operating systems). Thus, the COL will share the same virtual memory space as the calling application, which facilitates ease of use for application programmers. When using the OS-mediated COL calling model, employing the COL as a shared, dynamically loaded library is natural.

Figure 3.6 lists a few functions from the COL application programming interface (API). The COL API is structured similarly to PKCS # 11, the Cryptographic Token Interface Standard [135]. A software application interprets the COL API like entry points to procedures implemented by a hardware device. The COL also contains functions that allow an application to generate and add keys to the user key ring.

Consider the high-level operation of the COL function `Encrypt`. This function accepts virtual address pointers to input and output buffers, the buffer sizes, the desired mode of operation of the encryption algorithm (e.g., CBC), a virtual address pointer to the user's cryptographic key ring, the KIN of the key that should be used to perform the encryption

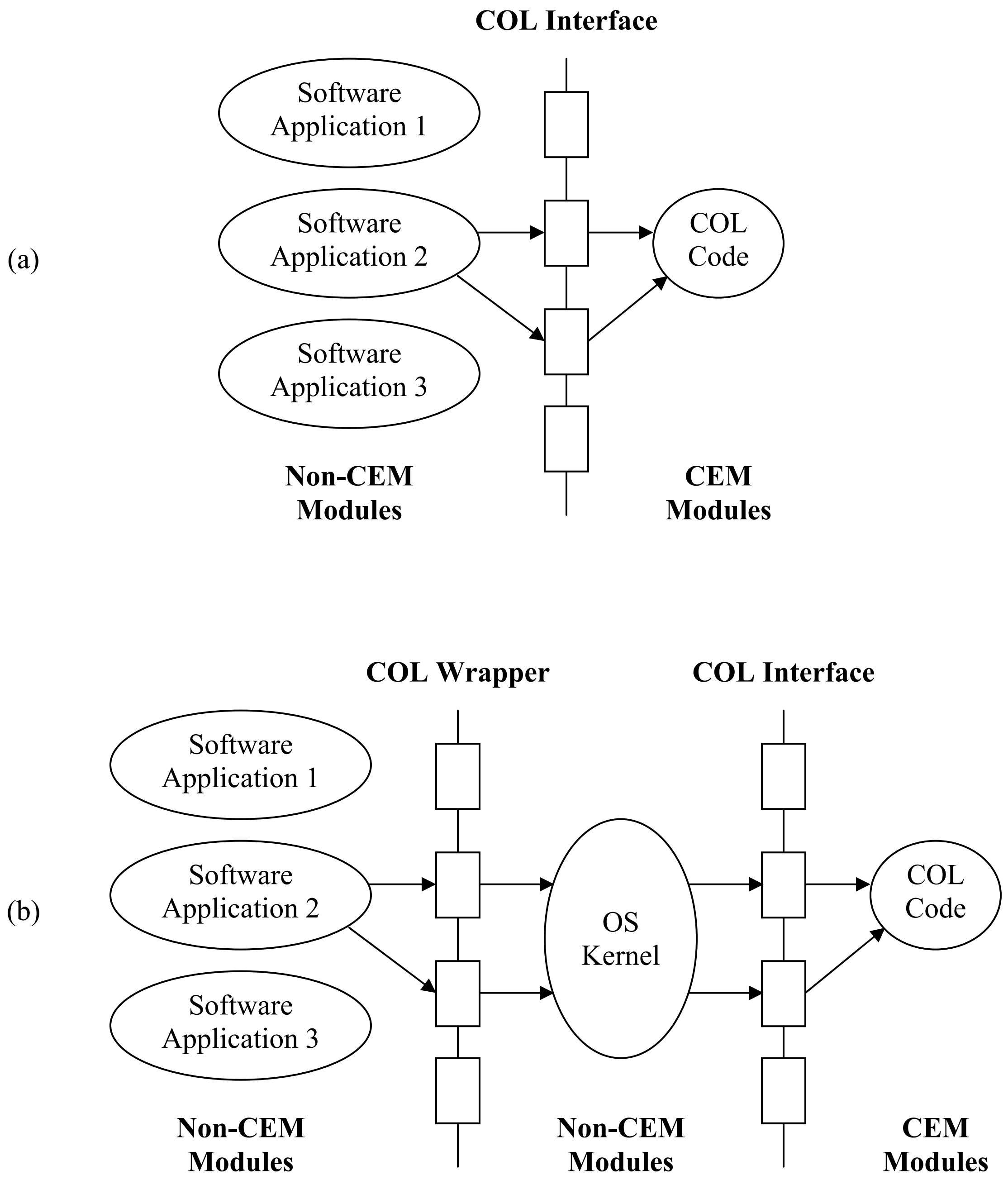


Figure 3.5: Application and COL interfacing via (a) direct calls and (b) OS-mediated calls

```
int Encrypt(input, output, isize, osize, mode,
        keyring, KIN, algorithm, initial_info)
int KeyedHash(input, output, isize, osize, KIN,
        keyring, mode, algorithm, initial_info)
int AddKeyToRing(algorithm, parent, KIN,
        keyring, initial_info, output, osize)
```

Figure 3.6: Example functions in the COL API

operation, an encryption algorithm identifier, and other data such as initialization vectors.

Following a call to `Encrypt`, the application program jumps to the appropriate function and enters the Concealed Execution Mode. The COL then traverses the key ring to obtain the appropriate user key or keys. Next, with the user's master key stored in the processor, the COL decrypts and verifies the integrity of the user's specified key. The COL then applies the decrypted key to perform the desired encryption operation on the input data, and the result is copied to the memory range specified by the output data pointer. Upon completion, the COL will terminate concealed execution, and control will be returned to the calling application. If a failure occurs at any point in this process, the COL will gracefully exit the called COL function by returning an error code. Examples of possible failures and errors include a key integrity check failure, the attempted use of an expired key, and the attempted use of invalid algorithm identifiers.

The COL functions must be implemented carefully to avoid leaking any keys or sensitive intermediate information [15]. The function will fail gracefully if, for example, a buffer address points to unallocated memory, the key is not authorized for use in the algorithm specified in the function call, or the key integrity check fails. By "fail gracefully", we mean that the COL will return an error condition without crashing or revealing secret information. To simplify the necessary architectural support and eliminate certain security vulnerabilities, we require that the COL be entirely self-contained. That is, the COL cannot call a function in external library, and the COL cannot make any system calls to the kernel.

This means that all necessary libraries must be statically linked into the COL at compile-time. In addition, the COL must statically allocate any memory that may ultimately be required to securely store intermediate data variables.

The structure of the key ring prevents software applications from using the COL to obtain decrypted key values. The COL defines the KIN of the user master key to be zero, and no function will allow a KIN of zero to be specified as an argument (meaning the function will promptly return an error code upon receiving an input KIN of zero). That is, COL functions such as `Decrypt` will not allow an application to employ the user master key; only internal COL operations involving key ring traversal can exercise the user master key. Thus, since each key is encrypted with the user master key, and since the application cannot directly use the master key, the plaintext key values are inaccessible to calling applications.

While a user master key is loaded into the processor, it is conceivable that an attacker could compromise the operating system and then attempt to instruct the COL to perform cryptographic computations with secret keys (although an attacker will not be able to obtain the plaintext key values). To provide added protection against such malicious code execution that may occur between the loading and the clearing of the user master key, VSCoP can be integrated with the attestation, secure booting, and general code verification techniques provided by proposed trusted computing platforms and related technologies (e.g., [81, 115, 163]). Attestation involves mechanisms for verifying that a system is using a known hardware and software configuration. Secure boot techniques seek to guarantee that a system boots to a trusted state. Mechanisms for providing secure bootup include boot sequence verification and the enforcement of the integrity of the BIOS, the boot sector, and other booting code. Code verification techniques include any static or dynamic mechanisms for verifying the integrity, origin, or correctness of application software. This chapter does not include a detailed discussion of mechanisms for validating and authenticating software applications to the COL and vice versa, but this is an important issue to be addressed in future research.

3.4.4 Processor Enhancements

The VSCoP enhancements to the processor enable concealed execution and support the operation of COL functions. We choose to enable concealed execution via dynamic memory protection rather than on-chip protected storage in order to avoid constraining the CEM to a limited memory space. Thus, the processor architecture support required to implement the virtual secure coprocessor includes a few new registers in the processor chip, cryptographic engines at the cache-memory interface, new cache line flag bits, and a pseudorandom number generator (PRNG). Figure 3.7 illustrates a typical processor with the new components shown in bold. We assume that the processor die contains split first level (L1) data and instruction caches and a unified second level (L2) cache. However, we could easily modify the system to support other configurations.

First, we create special storage within the processor for the user secret, the device secret, and state information for the Concealed Execution Mode. We implement a minimum of four new registers: the 128-bit Device Master Key, the 128-bit User Master Key, the 256-bit PRNG seed, and the 2-bit CEM Status register. The system does not permit the contents of the first three of these four registers to ever exit the processor. Also, none of these register values are set at the factory; the user defines the register contents in the field. The user secret comprises the contents of the User Master Key register, and the device secret comprises the contents of the Device Master Key and the PRNG seed.

The master key of a user's cryptographic key ring is stored in the User Master Key register. The 2-bit CEM Status register consists of two 1-bit flags that indicate whether any thread on the system is currently employing the CEM and whether the CEM is in use for the current instruction stream. We do not need to nor want to preserve these two registers in the device when power is turned off, so we implement these registers using

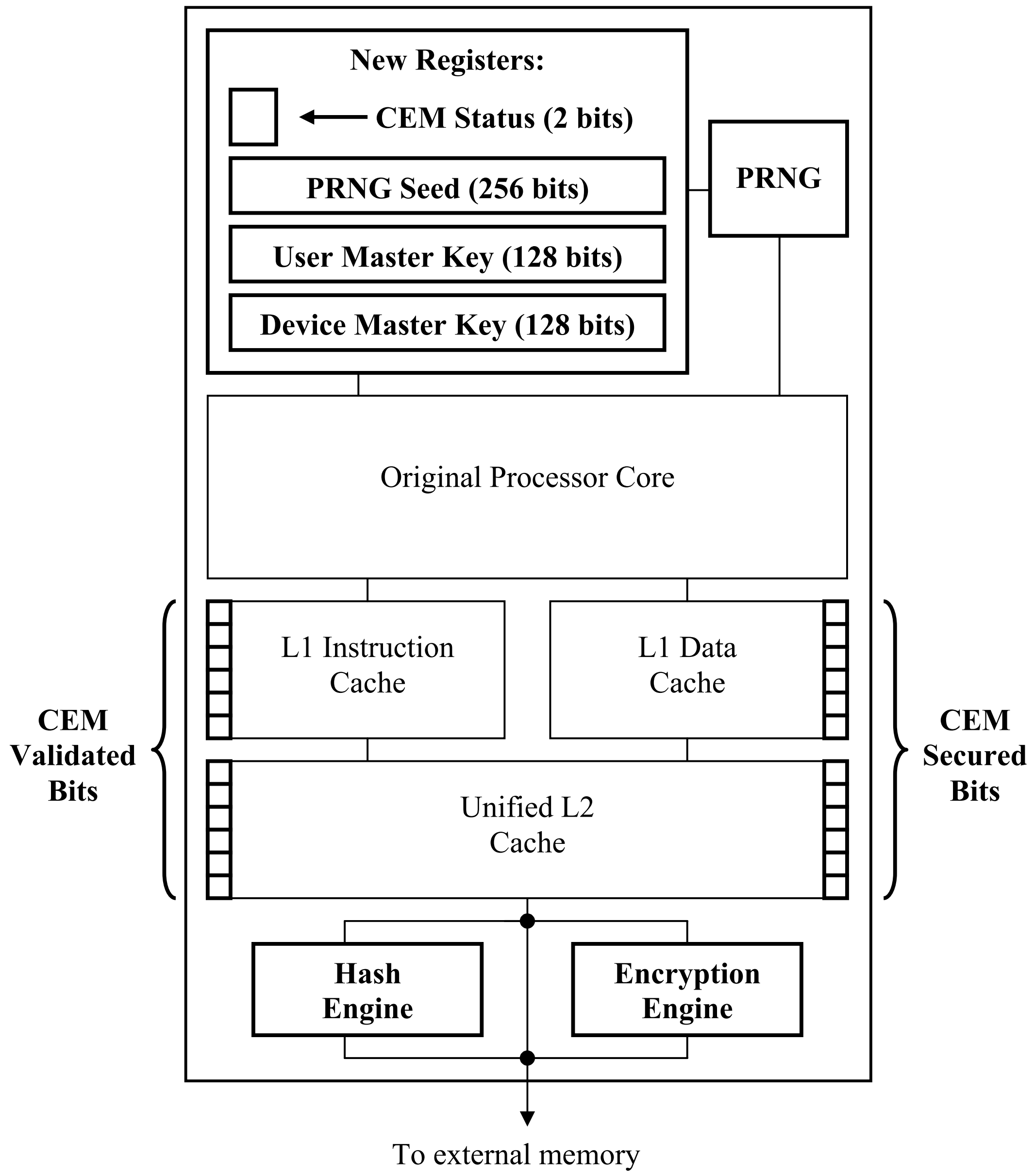


Figure 3.7: New processor features

volatile SRAM. When power is removed, the contents of these registers will be drained (i.e., cleared to zeroes).

The Device Master Key, which is used to authenticate software and to protect memory, must be maintained in the processor when power is turned off. The seed register for the pseudorandom number generator must also be preserved when power is removed, for the processor does not have an existing mechanism for securely generating a random seed value for the PRNG that an attacker could not predict. Thus, we use one of many possible non-volatile memory technologies to implement these two registers. The PRNG is used to provide pseudorandom bits to the COL and to enable secure context switching. Many pseudorandom number generators exist; we suggest applying AES encryption to generate a pseudorandom number using the 256-bit PRNG seed register similarly to the method described in ANSI X9.17 [2].

The remaining processor enhancements support concealed execution of trusted COL software. This involves verifying the authenticity of COL code as well as ensuring the secrecy and integrity of sensitive data. The processor performs the hash verification of COL code and protected data using a hardware-based hash engine as instructions enter the on-chip L2 cache from external caches or main memory. We append to each cache line a keyed MAC. We compute this MAC over the memory address of the first word in the cache line, the secret Device Master Key, and the contents of the (data or instruction) cache line itself. This keyed MAC can be a 16-byte AES-CBC-MAC [114, 120], which is an acronym for the Advanced Encryption Standard employed in cipher block chaining mode to produce a message authentication code. The three inputs to the hash function serve to prevent unauthorized code or data transpositions within protected memory, to preclude hash forgeries, and to prevent the unauthorized modification of the code or data cache line, respectively. If the cache line hash read from memory does not correspond to the hash calculated by the processor, the processor triggers an interrupt.[2]

[2]This hashing approach works only if the COL code and data are *always* loaded into same contiguous

To reduce the overhead associated with embedding hash results in code and data, we compute hashes for entire cache lines rather than for individual bytes or words. Hence, for a processor with 64-byte cache lines, the hash message authentication code information increases code size by 25%. In addition, we can implement an optional address translator in hardware that converts hashed code and data addresses to and from regular code addresses. With such a translator, CEM programs are not required to accommodate the awkward 16-byte hash values.

Upon verifying the instructions or data, the hash values are discarded rather than stored in the L1 or L2 caches. Assuming that we do not allow self-modifying code to execute in the Concealed Execution Mode, there is no need to maintain hash values within the processor chip or to re-verify code and data prior to use. Hence, we discard hash values following verification, but we add a CEM Verified flag bit to each cache line that indicates whether the hash for that line has been validated. During concealed execution, if fetched code or data does not possess a valid MAC, the processor can either throw an exception or simply exit the Concealed Execution Mode with an error condition.

Sensitive data that leaves the processor chip during concealed execution is encrypted via the AES cipher [120] or some other symmetric-key encryption algorithm in cipher block chaining (CBC) mode [114]. Cache line encryption and decryption is performed at the processor boundary outside of the L2 cache using the Device Master Key. We require another extra bit for each cache line, the CEM Secured bit, which indicates whether any of the current contents of the cache line contain sensitive information generated during concealed execution. The processor sets a cache line's CEM Secured bit when trusted software executes a write to secured memory or executes a load that fetches (and validates) a secured cache line from external memory. If a cache line's CEM Secured bit is set, the

memory pages or frames for all software processes on a particular device. If the COL can be loaded into multiple possible pages or frames in memory, the addresses of COL code and data may vary, and the hashes of COL instruction cache lines created at COL installation time may not be consistent with the hashes computed by the processor at execution time. To solve this problem, the OS can simply dedicate a fixed physical memory address range on the device to the COL.

processor will prohibit non-CEM threads from accessing that cache line. These two new bits per cache line, CEM Secured and CEM Verified, allow us to implement compartmentalized, secure memory in a simple and low-cost manner. With these bits, we can partition the on-chip cache memory space into secured and non-secured memory very flexibly and inexpensively on a cache line basis.

There exist attacks on external memory that remain to be addressed: secured data replay attacks. In some situations, an adversary may replace encrypted data and its associated hash value (that has been evicted from the processor) in external memory with legitimate but stale encrypted data and an associated stale hash from previous concealed execution operations. When the encrypted data is pulled back into the processor, the processor as it is currently defined cannot differentiate the stale hash from the fresh hash.

We identify two solutions to this problem that experience varying degrees of security and implementation cost. First, we could construct a processor-based session key using the pseudorandom number generator during concealed execution. This fresh session key would be used instead of (or in addition to) the Device Master Key to perform the encryption and hashing operations needed to secure external memory. Since the session key would change for every invocation of the CEM, this approach would defend against replay attacks involving encrypted data and hashes from a previous invocation of the Concealed Execution Mode. However, the COL would not be able to maintain state between function calls. Furthermore, replay attacks are still possible within the same invocation of the CEM. As a second option, the memory authentication system presented in [59], which is based upon Merkle hash trees, could be integrated cleanly with our proposal to provide complete protection against such replay attacks. Though the system of [59] incurs a higher implementation cost than that of the first option, we advocate the second option because of its flexibility and superior level of security.

An attacker can benefit from knowledge of the sequence of instructions fetched during concealed execution. Hence, while in the CEM, we shield the value of the program counter from external observation. The shielding mechanism varies between different processor architectures, but in general, we achieve this PC shielding goal by never allowing such sensitive information to reach the processor package's pins while in the Concealed Execution Mode. For example, in a processor that stores the program counter in one of the general purpose registers, the shielding mechanism is provided by the existing CEM register protection (i.e., encryption) features.[3] In addition, we must disable testing scan chains and other processor hardware debugging features that may dump secret information from the processor during CEM execution. There are many inexpensive ways to realize this goal, including blowing fuses in the processor directly following factory testing.

The hash engine, encryption engine, and the PRNG can all be implemented using a single AES module, which requires as few as 25,000 gates [3]. The four new processor registers consume only 514 bits of register storage with read and write control. Also, the additional cache line flag bits do not significantly increase the size of the cache memories. In a processor with 64-byte cache lines, the new cache line flag bits increase storage requirements by less than 1%. Hence, with the possible exception of the non-volatile memory required for two of the registers, the implementation cost is small.

3.4.5 New Instructions

We extend the Instruction Set Architecture (ISA) with new instructions to exercise the processor features that enable the Concealed Execution Mode. These instructions include `device_key_mv` and `user_key_mv` for handling the processor keys, `begin_cem` and `end_cem` for entering and exiting the CEM mode, `cem_store` and `cem_load` for accessing protected memory, and `prn_gen` for generating pseudorandom numbers. Some of these instructions operate similarly to ISA additions described in [100]. We summarize the

[3]Note that, as currently defined, the VSCoP processor enhancements do not prevent all types of physical attacks that may indirectly obtain sensitive information relating to instruction flow. For example, by measuring processor energy consumption or by observing main memory access patterns resulting from instruction cache misses, it may be possible to infer which instructions are being executed.

Table 3.1: New instructions

Instruction	Function
`begin_cem`	Enters the CEM. CEM Status register bits are set to 1's. All subsequently fetched instructions are cryptographically validated before execution.
`end_cem`	Exits the CEM. CEM-secured cache lines invalidated; general-purpose registers are reset to zeroes. CEM Status bits are reset.
`cem_store`	Stores a 64-bit datum to secured memory. The CEM Secured cache line bits are set for every cache line touched by this instruction.
`cem_load`	Loads a 64-bit datum from secured memory. The CEM Secured cache line bits are checked to guarantee the integrity and secrecy of the data.
`device_key_mv`	Transfers information from a register to individually addressable 64-bit chunks of the Device Master Key and the PRNG seed.
`user_key_mv`	Transfers 64-bit blocks of information to a register from individually addressable 64-bit chunks of the User Master Key.
`prn_gen`	Employs the processor pseudorandom number generator to write a pseudorandom value to a register, but only when running in the CEM.

functionality of our proposed instructions in Table 3.1.

At device initialization time, `device_key_mv` is used to write values to the PRNG seed and Device Master Key registers. The `device_key_mv` instruction is "one-way", however, for it cannot be employed by software to read the contents of those two special registers. All operations that require reading the Device Master Key and PRNG seed registers are implemented in processor hardware.

Only software running in CEM can obtain contents of the User Master Key register via the `user_key_mv` instruction. However, `user_key_mv` cannot be used by software to write values to the User Master Key; only the processor hardware can write values to that register.

When an application wishes to enter the Concealed Execution Mode by calling a function in a privileged library, the `begin_cem` instruction is executed. This instruction checks bit 0 of the CEM Status register to ensure that the CEM is not currently in use by any thread, as only one process may employ the CEM at any given time. This allows the system to avoid complexities caused by sharing secured memory. If bit 0 of the CEM Status equals 1, then another thread in the system is employing the CEM, so the CEM request will be denied.[4] Otherwise, if bit 0 of the CEM Status equals 0, then no other thread is currently using the CEM. In this case, the processor sets both CEM Status bits to 1, and all instructions that enter the processor following the execution of `begin_cem` are cryptographically validated using the Device Master Key or a fresh session key.

During concealed execution, privileged software can securely transfer data to and from memory using the `cem_load` and `cem_store` instructions. These instructions prevent spoofing and exposure of data using the processor's hash engine, encryption engine, and the new cache line flag bits. Note that programs running in the Concealed Execution Mode can also complete unsecured memory loads and stores, which are essential for transferring the inputs and the results of the cryptographic function from and to the relevant software application. For example, an encryption function running in the CEM must possess the ability to access unencrypted source data from the unsecured data memory space of the calling application in order to complete the encryption operation.

In addition, while in the CEM, the privileged software can obtain pseudorandom numbers from a trusted source (the processor) by using the `prn_gen` instruction. When this instruction is executed, the processor writes a 64-bit pseudorandom integer to a specified

[4]In Section 3.4.6, we discuss OS mechanisms for ensuring that processes cannot misuse this feature to block access to the CEM by other processes.

general purpose register, and then the processor writes a new value to the PRNG seed register that is a nonlinear function of the original seed. The `prn_gen` instruction can be used for many cryptographic applications (and for KIN generation described in Section 3.4.1).

Upon completion of a COL function, the COL executes the `end_cem` instruction to exit the CEM. At this time, all of the general-purpose register values associated with the CEM instruction stream are reset to zeroes, and both bits of the CEM Status register are reset to 0. Cache lines that contain secured CEM data are invalidated using existing cache line flags to prohibit reuse of results in future CEM invocations. Alternatively, cache line contents could be cleared to zeroes for extra security.

3.4.6 Operating System Enhancements

To fully enable virtual secure coprocessing, we must implement minor changes to the operating system. We do not wish to suspend the execution of other processes while a CEM function is executing, so we must provide support for secure OS context switching. We handle preemptive context switches that occur during concealed execution as follows:

- The PRNG is employed to generate a session key that is used to encrypt and compute a keyed hash over the sensitive context before evicting it to memory. Note that if we were to employ the same session key to encrypt the registers for every CEM context switch, the system would be vulnerable to data replay attacks. When a new key is requested from the PRNG, the processor writes a new value to the PRNG seed register that is a nonlinear function of the original seed. Then, the new seed can be used to generate a fresh session key. The processor can either calculate the session key on the fly (when needed) using the current value of the PRNG seed, or the processor can maintain the value of the session key by implementing and using a new 128-bit volatile session key register. In either case, the processor protects the session key by never allowing the session key to leave the processor chip (in the same manner that the processor protects the device master key).

- Bit 1 of the CEM Status register is then reset to 0 for the incoming non-CEM process. Bit 0 of the CEM Status register remains 1 to indicate that some thread in the system (i.e., the outgoing thread) still owns a “lock” on the Concealed Execution Mode.

Until the CEM thread begins executing again, the processor will prohibit any attempt by a non-CEM process to access sensitive data in the caches or the User and Device Master Key registers. If a non-CEM process requires CEM-secured data to be evicted to the cache, the encryption and hash engines protect the information as described above without invoking the OS or the CEM thread.

We handle a context switch that restores control to the CEM thread as follows:

- The PRNG-generated session key is used to decrypt the incoming protected context and to verify the integrity and freshness of that context.

- If the context is decrypted and verified successfully, bit 1 of the CEM Status register is restored to 1, and CEM thread may begin to access CEM-secured data and execute again.

In addition, since we only allow one process to employ the CEM at a given time, we must implement an OS mechanism for queuing CEM requests in order to avoid possible CEM contention (or blocking) between processes. The Cryptographic Operations Library, which is the only library that is permitted to use the CEM, does not include routines that consume unbounded processing time. Hence, deadlock will not occur in processes that are waiting for another process to relinquish the CEM. Note that related proposals and devices, such as the IBM secure coprocessors [152], also require that secure execution requests be performed serially.

Also, the OS should enable users to access their encrypted key rings from remote storage, i.e., provide a mechanism for fetching an encrypted key ring over a network and delivering that encrypted data to the virtual secure coprocessor.

3.5 Applying VSCoP

We now provide a summary of the VSCoP processes involved in applying the new architectural enhancements to protect users' keys. We define three primary processes: device initialization, user initialization, and protected operation.

3.5.1 Device Initialization

Device initialization occurs when a user first obtains a computing device containing our proposed security features. In this step, the user installs the Cryptographic Operations Library, the only software module that will be permitted to directly access users' keys. First, if the device secrets have not already been reset to zero by the device producer (i.e., the factory) or a previous user, the user presses the Device Reset button to wipe the device. Note that the new user can employ a previously used device to securely store and utilize his cryptographic keys without the risk of exposing his secrets or previous users' secrets. Second, the installation procedure writes new random values to the Device Master Key and PRNG seed registers.

Third, the user verifies the authenticity of the COL by checking its digital signature using software-based Public Key Infrastructure (PKI) techniques.[5] Then, the COL is signed

[5]Using cryptographic primitives such as digital signatures, PKI techniques enable users to verify the origin and integrity of a software module. The verification process, which involves a trusted third party, a software creator, and a user, may proceed as follows. First, the trusted third party issues a unique certificate to the software creator. A certificate is a publicly known data block that associates the creator's name with his public signature key. Second, the creator digitally signs the software module using his private signature key that corresponds to the public key in the certificate. Third, upon obtaining the signed module, the user can employ the creator's public key obtained from the creator's certificate to verify the integrity and origin of the software module. The creator's certificate also must be verified, however. The user can verify the

using the Device Master Key via a keyed MAC. Note that PKI and asymmetric encryption techniques are not implemented in hardware and are not required by the Concealed Execution Mode; public-key operations are only employed in software at installation time. During COL installation, a malicious OS kernel can interfere with the MAC generation process to facilitate the installation of a corrupted and dangerous COL. To prevent such attacks, the user should install the COL only when the OS kernel is guaranteed to be uncompromised. This condition is difficult to satisfy at arbitrary times, so it is most prudent to install the COL immediately following or during the installation of a validated OS kernel or secure BIOS.

3.5.2 User Initialization

User initialization occurs when a user creates a new cryptographic key ring with an initialized device. This operation simply involves selecting a master key for the key ring, which is the output of a cryptographically strong one-way hash of a user-supplied passphrase. As keys are added to the ring, a user can store his encrypted key ring locally or remotely. By depositing the key ring in on-line accessible storage, the user can access his secret keys and perform protected computations on any VSCoP-enabled networked device.

3.5.3 Protected Operation

Protected operation is the process in which an initialized user securely employs a secret key in an initialized device. This process begins with a user securely inputting his passphrase into the device via the protected path. The system hardware then computes the user's master key and stores the result as the user secret.

validity and integrity of the software creator's certificate by using a separate certificate that is associated with the trusted third party. Trusted third party certificates are often embedded in operating systems and software applications.

Next, when a software application needs to perform a cryptographic primitive that involves one of the user's secret keys, the application makes an appropriate call to a function in the Cryptographic Operations Library as if it were an interface to a secure coprocessor. Thereupon, the processor verifies the integrity of the COL using the device secret. We note that we do not need to ensure the secrecy of individual library instructions, as the library routines are not confidential. If verification is successful, the processor enters the Concealed Execution Mode and begins executing instructions in the called COL routine. In order to prevent a potential attacker from exposing any user secrets during the CEM, the processor maintains the secrecy and integrity of all sensitive data that is available to other processes or is released from the processor chip. After completing an operation that requires the use of a key ring, a user can clear the master key of his key ring from the device by pressing the Authenticate button. As long as the OS or other trusted software is not compromised before or while the master key is loaded in the processor, a software attack on a trusted component cannot reveal or access a user's master key.

3.5.4 Application Example

One application example for VSCoP involves secure email. Upon receiving an encrypted email, an initialized user may wish to employ his key ring to decrypt email on an initialized device. To begin this process, the email reader may prompt the user to input his master key into the device. As described above, the user would press the appropriate physical buttons and input his passphrase. Then, following either OS or user-initiated notification that the master key has been received, the email reader or the OS would fetch the user's encrypted key ring from a local or remote storage device. The email application would then make a call to the COL by passing the user's encrypted key ring and the encrypted email as inputs. The COL then enters the CEM, uses the master key to decrypt the key ring and the email, returns the decrypted email result to the unsecured memory space of the email application, and exits the CEM. At that point, the user may press the Authenticate button to clear the master key from the device, and the email can be read by the user.

3.6 Security Analysis

By implementing VSCoP, we can achieve practical and improved security for keys against threats from each of the threat classes identified in Section 3.1. Below we discuss the threats prevented by VSCoP as well as the threats that remain to be addressed.

3.6.1 Protection for Keys in Storage

The protected key ring structure and the VSCoP processor enhancements provide strong protection for key material in storage against software and physical attacks. When the key ring is in storage in a local or a network-accessible device, the entire ring is cryptographically protected against any physical or software attacks involving exposure, spoofing, and transposition attacks that threaten the confidentiality and integrity of the keys. Furthermore, when a key ring is decrypted in the Concealed Execution Mode by the cryptographically authenticated COL, sensitive data that directly or indirectly represents keys is cryptographically protected by the processor before being stored to processor-external caches, main memory, or swap devices. These processor enhancements defend against physical and software attacks outside of the processor that involve the exposure, spoofing, transposition, and replay attacks on key material.

VSCoP does not ensure availability of keys, however: denial of service attacks and deletion attacks are not prevented.

3.6.2 Protection for Keys in Transit

VSCoP provides strong protection for keys in the protected key ring during transport against physical and software attacks. Because the protected key ring is not decrypted until the processor accesses the ring in the Concealed Execution Mode, any transmissions of the key ring structure over networks and system buses are protected via the same cryptographic mechanisms that protect the ring while in storage.

VSCoP also protects the user master key during transport against software attacks. The user master key that is required to access or modify the key ring is only transferred to the trusted processor via the platform-enabled protected I/O path. However, the VSCoP protected I/O path does not defend against any physical attacks (such as active bus probing) that may occur during master key transport between the user and the trusted processor.

3.6.3 Protection for Keys Exercised by Software

VSCoP defends keys against certain unauthorized use, exposure, or corruption of potentially vulnerable application or OS software. The protection is provided by only allowing the authenticated COL to directly exercise user keys (in plaintext form), and the key ring can only be used by the COL when the user master key is loaded into the processor. During COL computations, the processor cryptographically protects any sensitive data that exits the processor, which prevents other software from obtaining and directly exercising plaintext key material. If the most secure design options are chosen, an enhanced key ring structure can preclude any keys from being passed in decrypted form to software applications (as described in Section 3.4.3). Furthermore, the OS can restrict the set of applications that are privileged to make calls to the COL interface (as described in Section 3.4.3).

If a user clears the device of his master key following use, an attacker *cannot* expose or corrupt the user's secret key ring during any subsequent penetration of the OS. If the OS is compromised before or while the user master key is loaded into the processor, however, an attacker *can* use the compromised OS to exercise secret keys. This is because the COL does not authenticate software that makes calls to its interface. Thus, to maximize security of VSCoP, the user master key should be loaded in the processor for the minimum period of time needed to complete a security task. Such an approach is quite similar to methods that are sometimes used to apply cryptographic tokens such as smart cards. In those methods, the token functionality is only provided to applications or the OS for a limited period of time to minimize risk of token abuse or compromise. In future work, this security issue may be further addressed via integration with features of trusted computing platforms.

3.6.4 Protection for Keys Exercised by Hardware

The VSCoP enhancements ensure that only the protected I/O path and the hardware on the processor chip may directly exercise cryptographic keys. Other hardware in the system that can be accessed or controlled by software cannot perform operations using decrypted key rings. After a user master key has been loaded into the processor, the VSCoP security mechanisms ensure that all sensitive key data that enters and exits the processor is cryptographically protected. Thus, any hardware outside of the processor (following master key loading) that seek to access or tamper with keys will be thwarted and detected, respectively.

Though VSCoP successfully prevents software attacks that employ hardware outside of the processor to exercise keys, VSCoP is not designed to defend against several physical attacks on hardware. First, VSCoP does not prevent active physical attacks on the protected I/O path during master key transport. Second, the processor is not assumed to be tamper-proof, and therefore VSCoP does not prevent physical attacks that involve probing the internals of the processor chip while the master key is loaded in the processor. We argue that such attacks are significantly more difficult to realize than disk probing, network sniffing, or bus probing, and therefore physical attacks on the processor are of lesser concern. Third, VSCoP does not defend against hardware attacks involving physical theft followed by hardware tampering and the inconspicuous replacement of the altered device. Fourth, VSCoP is not designed to prevent all side-channel attacks on the hardware, such as timing attacks, power analysis, and cache miss pattern observations, which can lead to the leakage of sensitive key material.

3.7 Performance Analysis

The performance impact of our proposal is negligible for software packages that do not employ the Cryptographic Operations Library. However, performance changes may be experienced by programs (such as SSL and secure storage software) that employ user key rings with the COL. In such software, performance degradation may occur due to the increased quantity and costs of memory accesses during COL operations. By hashing and possibly encrypting/decrypting some information at the processor boundary, we add latency to external memory accesses.

It is important to note that since the COL contains only cryptographic functions, we only need to evaluate performance degradation associated with those cryptographic functions. Thus, we obtain performance statistics by simulating the execution of common cryptographic routines in the Concealed Execution Mode: the RSA encryption algorithm [133], the AES encryption algorithm [120], and the MD5 one-way hash function [131].

To obtain the results, we use a modified version of the SimpleScalar cycle-accurate superscalar processor simulator [21]. The processor model is based upon the enhanced processor and memory system described in Section 3.4. We implement the benchmarks in C and compile for the Alpha instruction set architecture using `gcc` with the `-O2` optimization flag. During execution, we provide the benchmarks with 1 megabyte of input data to be encrypted or hashed. We conduct simulations for a 4-way superscalar processor, and we present the processor simulation parameters in Table 3.2.

Table 3.2: Processor model parameters

Parameters	**Characteristics**
Instruction window	64-entry RUU
Fetch/decode/issue width	4 instructions
Commit Width	8 instructions
Functional Units	4 integer ALUs, 1 integer mult. 4 FP ALUs, 1 FP multiplier
BTB	4K-entry, 2-way set associative
Branch Predictor	Hybrid: 4K 2-bit selector 4K 2-bit bimodal predictor 1K 2-bit local w/ 10-bit history
L1 data cache	64 KB 2-way set-associative 64 byte blocks, 2 cycle latency, 2 ports
L1 instruction cache	64 KB 2-way set-associative 64 byte blocks, 1 cycle latency
L2 unified cache	2 MB 4-way set-associative 64 byte blocks, 12 cycle latency
Main memory	100 cycle latency
Load/store queue	64 entries
I-TLB and D-TLB	128-entry fully associative

We model our proposed enhancements to the interface between the L2 cache and external memory as follows. We use 128-bit AES-CBC to enable data encryption/decryption and 128-bit AES-CBC-MAC to provide code and data authentication [114, 120]. The keys involved in the AES operation are based upon either the Device Master Key or a session key generated by the PRNG. The AES-CBC encryption and decryption of 64-byte cache lines can be completed with 4 serial AES operations and 4 parallel AES operations, respectively. The initialization vector (IV) is equivalent to the address of the cache line. MAC computation for authenticating both 64-byte instruction and data cache lines requires a latency of 5 AES operations. We use 5 rather than 4 AES operations to compute the MAC in order to properly hash all four 16-byte blocks of the cache line as well as the 8-byte address of the cache line. The AES encryption of a 16-byte datum requires 10 rounds of work, and the critical path in each AES round simply involves a lookup into 256-entry ROM table, a hardwired byte-level permutation, and 3 XOR gates. We conservatively estimate that one AES round can be completed in at most two processor cycles, but it is likely that one round could be completed in a single cycle. Assuming one AES round can be completed in 1 or 2 cycles, the total latencies involved in encryption/decryption and MAC computation are at most 80 and 100 cycles, respectively.

We can parallelize the processing of the encryption/decryption and MAC calculation to improve performance. Figures 3.8, 3.9, and 3.10 depict secure data loads, secure data stores, and authenticated instruction loads, respectively. In the figures, the AES blocks represent AES encryption using the Device Master Key or a session key, and the AES^{-1} blocks represent AES decryption using the same key as the AES encryption blocks. As shown in Figure 3.8, for secure data cache line loads, the decryption can be performed in parallel with the MAC computation without incurring any additional latency. Secure data cache line stores operate similarly to data cache line loads, as Figure 3.9 illustrates. However, in a secure data cache line store, the first 16-byte AES encryption operation must be completed before the MAC computation begins. This results from the fact that

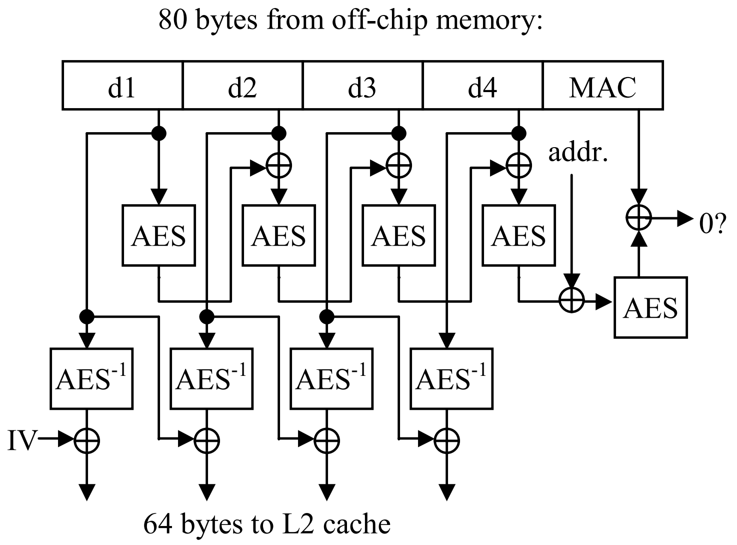


Figure 3.8: Secure data cache line load

the MAC is performed on encrypted data instead of plaintext data, so encrypted data is needed for the MAC computation to commence. The remaining encryption operations can be completed in parallel with the MAC operations. The processing time of secure loads and secure stores is therefore equivalent to 5 and 6 serial AES operations, respectively. As displayed in Figure 3.10, authenticated instruction cache line loads require only a complete MAC computation, so the added latency is 5 serial AES operations. Hence, the maximum external memory access penalties (per 64-byte cache line) for secure data loads, secure data stores, and authenticated instruction loads are 100, 120, and 100 cycles, respectively.

Despite the increase in external memory access latencies, our simulations show that the performance impact of the proposed enhancements for the benchmark programs is negligible (i.e., less than 1%) when using the parameters described above. This results from the fact that secured data employed by the benchmarks is rarely evicted to external memory; most external memory activity involves unsecured data. Also, the number of static instructions employed by the benchmarks is modest, so the number of instruction fetches (and

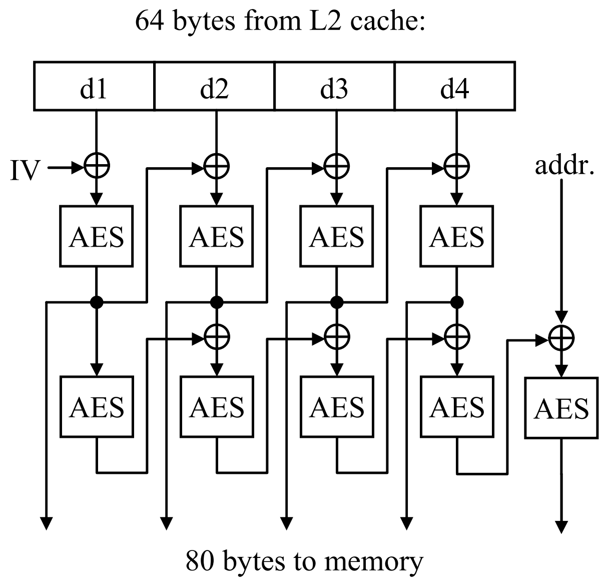


Figure 3.9: Secure data cache line store

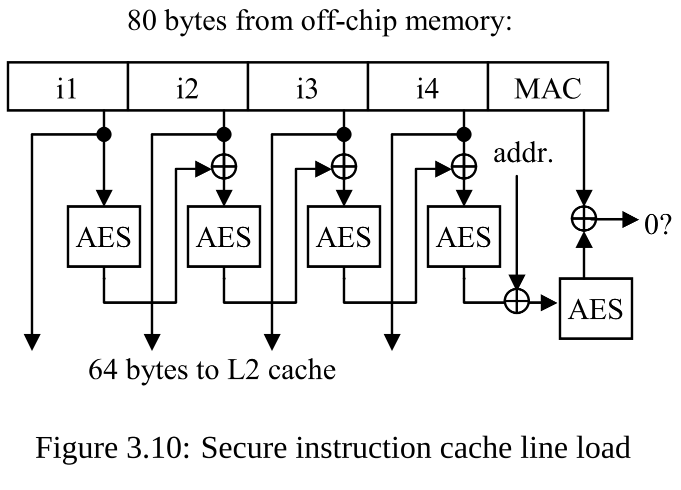


Figure 3.10: Secure instruction cache line load

subsequent authentications) from external memory is relatively low.

3.8 Extensions and Alternatives

Architectural alternatives exist for some components in our proposal. For example, we could use one master key rather than two at the cost of some flexibility. We can employ the User Master Key to enable all code and data security functions, thus eliminating the Device Master Key. However, this would require users to "sign" the COL once for each device on which they expect to employ their secret keys. Also, when multiple users share a device, maintaining several authenticated COLs for different users could become taxing.

Another design option is to employ the Device Master Key in addition to the User Master Key to encrypt and authenticate the user's cryptographic key ring. This scheme would provide added security by disallowing access to a user's keys on devices that have not been previously authorized. This approach would also limit flexibility, however, for users would be required to engage in the inconvenient process of pre-authorizing individual devices and re-authorizing devices following COL updates.

An additional possible extension is to augment cryptographic key rings to support multiple privilege levels; the processor could prevent certain untrusted applications from calling a function in the COL that employs certain highly sensitive keys. Furthermore, we could implement a processor-based mechanism that enables User Master Key register expiration. That is, the processor forces user logout by periodically zeroizing the User Master Key register. This would increase security by potentially preventing unauthorized parties from accessing a user's key ring if the user neglects to logout and the user's device is subsequently penetrated or stolen. Such functionality would only require minor additions to existing on-chip cycle counters.

Instead of supporting dynamic encryption and authentication of data memory, we could implement on-chip storage for intermediate data values generated during the CEM. Hence, there would be no need for encrypting CEM-secured data because all sensitive information would be maintained in on-chip protected storage. The processor would ensure that non-CEM processes could not access or expel data from this protected storage. This design alternative trades chip area for reduced CEM performance degradation. We choose to implement dynamic memory encryption rather than on-chip protected storage to avoid constraining the COL to a limited protected memory space.

Lastly, CEM modes could be added to enable the protected execution of software other than the COL. For example, a user could permit certain software to employ CEM services such as data memory encryption but disallow access to secrets such as the User Master Key register.

3.9 Summary

This chapter presents a new approach to protecting the storage, transmission, and use of cryptographic keys in general-purpose and embedded platforms through virtual secure coprocessing. Most secure systems depend on cryptographic primitives to achieve security goals. Thus, the protection of cryptographic keys is essential for defending networks, computers, and data. However, most existing key protection solutions suffer from a combination of poor performance, inconvenience, high cost, and insufficient security.

We describe architectural and software enhancements that provide flexible, efficient, and built-in protection of users' cryptographic keys. These enhancements effectively constrict the traditional trusted boundary for cryptographic keys to the boundary of the general-purpose processor. Also, a new access control paradigm is implemented to protect keys from potentially vulnerable hardware and software. With these changes, users can store, transport, and employ their secret keys to safely complete cryptographic primitives and prevent many physical and software attacks.

The VSCoP architecture is composed of two sets of components: components that provide protection for users' keys in storage and in transit, and components that protect keys while being exercised by software and hardware. To protect keys in storage and in transit, we present a protected key ring structure and a hardware-enabled protected path for transmitting master secrets to the processor. To defend keys while in use, we propose processor modifications and special software support that effectively transform the general-purpose processor into a virtual secure coprocessor when needed. These modifications include a special Cryptographic Operations Library and a Concealed Execution Mode. The Cryptographic Operations Library (COL) is the only software that is privileged to directly employ keys in the system. Applications and operating systems can only access keys through a special interface to the COL. The Cryptographic Operations Library protects keys during execution by running in the processor's Concealed Execution Mode. The Concealed Execution Mode shields keys by authenticating software that exercises keys and by protecting data that leaves the processor chip that could otherwise reveal information about the values of keys.

VSCoP provides a foundation with which users can more securely access their secret keys on any Internet-connected computing device (that supports VSCoP) without requiring auxiliary hardware such as smart cards. Unlike VSCoP, no past work facilities the high-performance and secure utilization of key rings from any Internet-connected device; enables a wide array of cryptographic techniques; avoids the use of potentially expensive, auxiliary devices such as coprocessors, smart cards, or sets of servers; and provides strong protection for keys while in storage and use.

We now turn our attention to a different class of key protection mechanisms. While this chapter focused on architectural techniques for enabling users to protect their own keys, the next chapter addresses issues relating to the distribution of sensitive cryptographic keys to potentially untrusted users.

Chapter 4

A Traceability Scheme for Broadcast Encryption

In broadcast encryption and related systems, a content provider seeks to securely transmit sensitive information to a set of authorized users. Authorized users can defeat the security of such systems by providing unauthorized users with valid decryption keys that enable access to sensitive content. To deter this threat, researchers have proposed traitor tracing schemes. These schemes enable the identification of authorized users that engage in unauthorized decryption key distribution.

This chapter proposes Traitor Tracing using RSA (TTR), which is a new key traceability scheme for broadcast encryption systems. If k or fewer authorized users collude to create a pirate decryption device, at least one of the contributing traitors can be identified with certainty, and innocent users cannot be framed. Thus, TTR protects broadcast decryption keys by enabling the detection of key misuse. Furthermore, the proposed scheme significantly improves on the decryption performance of past schemes. The contributions of this chapter are based in part on the work previously published by the author in [110, 111, 112].

This chapter is organized as follows. Section 4.1 introduces broadcast encryption schemes, threats to these schemes, and traitor tracing. Section 4.2 discusses past proposals for broadcast encryption and traitor tracing. Section 4.3 introduces a new traitor tracing scheme based on the RSA encryption algorithm that improves upon the decryption performance of past proposals. Section 4.4 analyzes the security properties of the new scheme. Section 4.5 presents tracing algorithms that identify traitors upon confiscation of unauthorized broadcast decoders. Section 4.6 explores the implementation and performance impact of the new scheme. Section 4.7 investigates a key issuance and revocation protocol that exercises this traitor tracing scheme. Section 4.8 summarizes the chapter.

4.1 Broadcast Encryption

Broadcast encryption is beneficial in scenarios where a content provider wishes to securely distribute the same sensitive information to all or a subset of authorized users. The confidentiality of the broadcast content is ensured with encryption, and only the subset of authorized users should possess the information (e.g., a decryption key) necessary to properly access the transmitted information.[1] Subscription-based satellite television is an example of a broadcast encryption system. The information is transmitted worldwide, but only paying users who receive a decoder box can properly access the information. Furthermore, in some cases, only a subset of those paying users should be able to receive a particular broadcast (such as a premium channel).

4.1.1 Broadcast Encryption Model

The broadcast encryption model used in this chapter involves the following entities:

- **Universe of Users.** Broadcast messages are transmitted to the universe of all authorized and unauthorized users, U.

- **Authorized Users.** Only the members of the set of authorized users, T, are provided

[1]Broadcast encryption schemes can also support security features such as data integrity using other cryptographic primitives, but this chapter focuses on data confidentiality.

with the information needed to decode broadcast messages. The maximum number of authorized users is n, so $T = \{t_1, t_2, \ldots, t_n\}$, and $T \subseteq U$.

- **Content Provider.** The content provider encrypts and broadcasts information to the universe of possible users, U.

- **Privileged Authorized Users.** For a particular broadcast, only a subset T' of the set of authorized users T should be able to decode the broadcast messages. The members of this subset are defined by the content provider, and the size of this subset at a given time is n', where $n' \leq n$. Thus, $T' = \{t'_1, t'_2, \ldots, t'_{n'}\}$, and $T' \subseteq T$. The content provider can change the members of this subset T' without having to send new keys to the authorized users. Instead, the content provider can effect such changes to T' by using different encryption keys or different methods to prepare broadcast messages.

Given these sets of entities, the following five components are typically involved in a secret-key broadcast encryption scheme:

- **Provider Initialization.** A content provider generates initial values required to produce the broadcast encryption keys and the user decryption keys.

- **User Initialization.** An authorized user t_i is added to the set of authorized users, T, by requesting that the content provider generate and securely distribute a user decryption key to t_i.

- **Encryption.** The content provider encrypts a message one or more times using one or more secret encryption keys.

- **Transmission.** The content provider transmits the encrypted message to all users.

- **Decryption.** Upon receipt of an encrypted message from the content provider, each authorized user decrypts the message using his respective decryption key.

4.1.2 Example Operation

Figures 4.1 and 4.2 illustrate the operation of a broadcast encryption system.

Provider and User Initialization

Prior to transmitting secure broadcast messages, the content provider generates a set of broadcast encryption keys, E. Also, the provider creates and issues a personal decryption key DK_i to each authorized user t_i. These personal decryption keys can be distributed via a secure channel or a physically secure courier service. Furthermore, these keys can be embedded in software or in tamper-resistant hardware devices such as smart cards. In both figures, trusted boundaries are depicted with dashed lines. Since the content provider and the authorized users have knowledge of secret keys, those entities and their respective computing devices are treated as part of the trusted domain.

Encryption and Transmission

Figure 4.1 depicts the preparation and transmission of broadcast messages to authorized and unauthorized users. Each broadcast message may consist of two components: a *cryptographic header* and a *payload*. In some applications, broadcast messages consist only of a payload containing code and/or data that has been encrypted multiple times. In the following description, however, we define broadcast messages to contain both a cryptographic header and a payload. The header contains cryptographic key information needed for decoding, and the payload consists of the protected content (in encrypted form).

The content provider prepares and transmits broadcast messages as follows.

1. The provider first randomly generates a per-message encryption key k_e and a corresponding per-message decryption k_d.

2. The provider uses k_e to encrypt the sensitive content using any publicly-known encryption algorithm. The output of this encryption step is the encrypted payload.

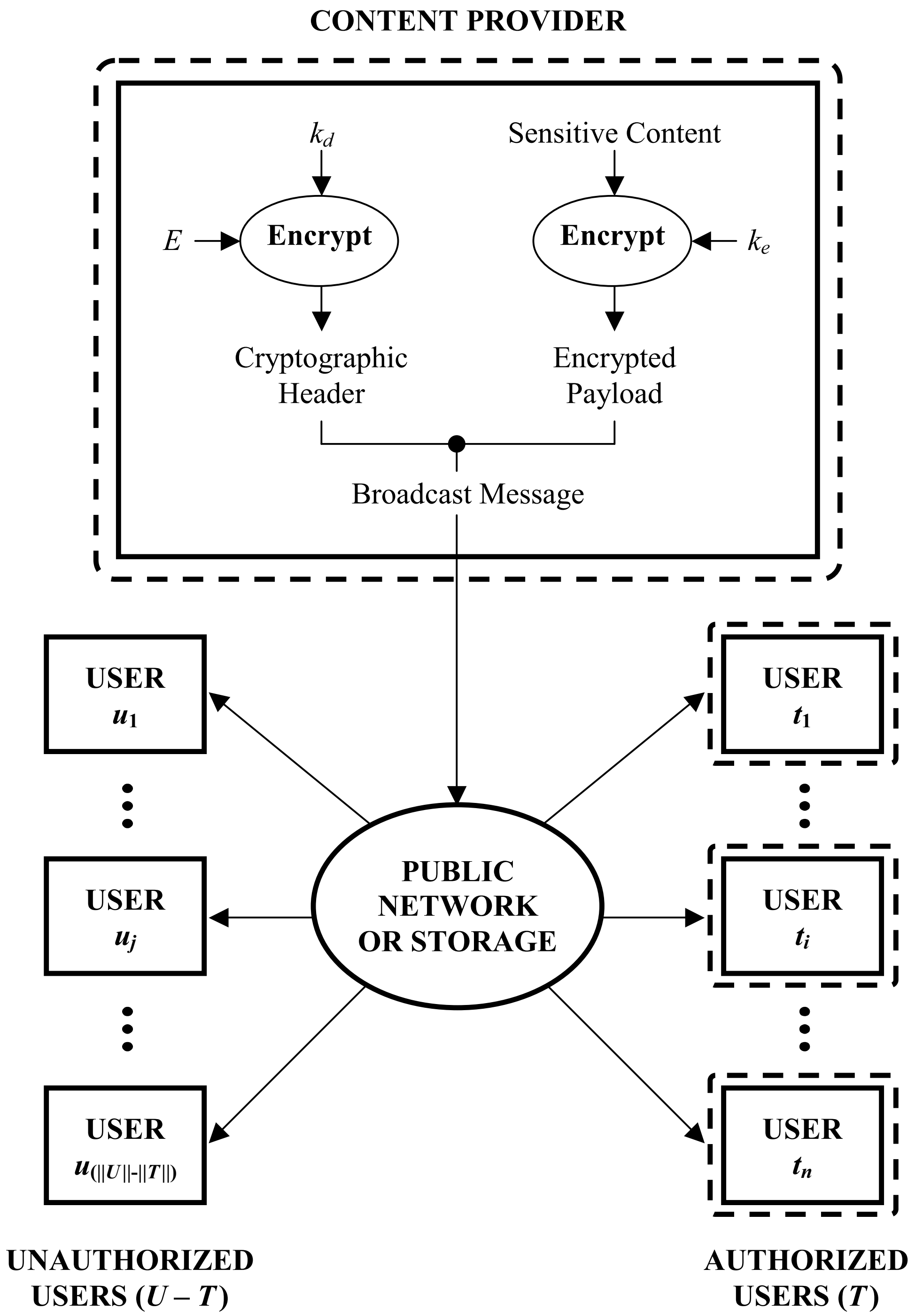


Figure 4.1: Encryption and transmission of messages

3. The provider encrypts k_d multiple times using some or all of the encryption keys in E. The cryptographic header comprises these multiple encrypted results. The exact encryption method used to obtain the cryptographic header depends on the characteristics of the specific broadcast encryption scheme. Furthermore, the encryption is performed such that only members of the privileged subset T' can decrypt the sensitive content. In this example, we assume that $T' = T$; therefore all authorized users will be able to decrypt the broadcast message.

4. The provider transmits a broadcast message that contains the cryptographic header and the encrypted payload to all of the users in U. The transmission can occur over a network or via a set of storage devices (e.g., CDs or DVDs).

Decryption

Figure 4.2 illustrates the operations conducted by the authorized users to obtain the sensitive content upon receiving a broadcast message. To decrypt the message, an authorized user $t_i \in T$ performs the following steps.

1. The authorized user employs his decryption key DK_i and the cryptographic header to obtain k_d. The exact decryption method used to obtain k_d depends on the characteristics of the specific broadcast encryption scheme.

2. The authorized user t_i decrypts the payload using k_d to obtain the sensitive content.

Note that in secure broadcast encryption schemes where $T' \neq T$, an authorized user t_j belonging to T but not belonging to T' should not be able to apply his key DK_j to decipher the payload.

This is only one of the many possible methods of preparing messages for secure broadcasts.

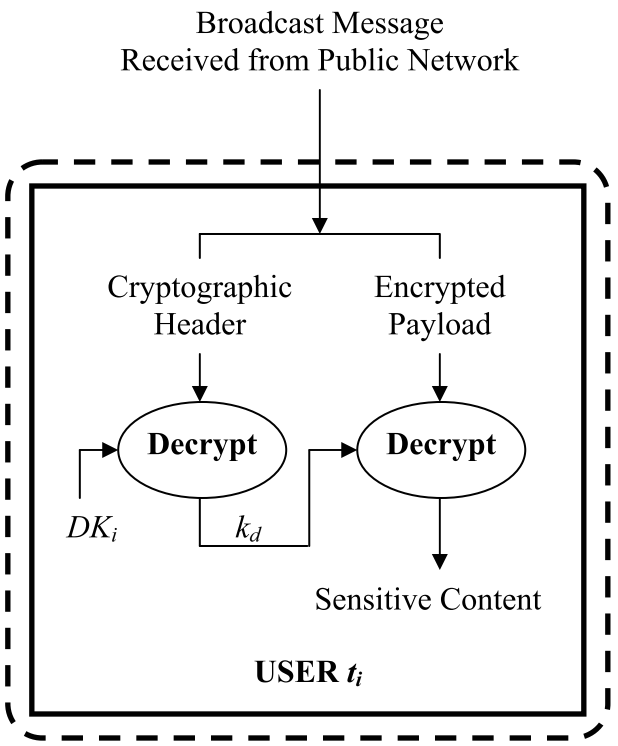


Figure 4.2: Decryption of broadcast messages

4.1.3 Attacks and Defenses

Though the authorized users in T and their devices are within the trusted domain, they are not necessarily trustworthy. That is, systems like VSCoP can protect a user's keys from other users, but such systems cannot fully protect a user's keys from intentional misuse by the user himself. It is well known that current tamper-resistant hardware and software systems are vulnerable to a variety of attacks [4]. Thus, authorized, trusted users can extract decryption keys from a legitimate software or hardware decoder. The users can then circumvent the security of the broadcast system by simply distributing the compromised decryption keys to unauthorized users. Alternatively, the users can possibly employ the compromised keys to generate and issue new decryption keys. Authorized users who illegally extract and distribute decryption keys are called *traitors*, and the unauthorized users who unfairly obtain the keys are *pirates*. The illegal decoder software or hardware devices created by the traitors are *pirate decoders*.

Traitor tracing schemes, which are also called *traceability schemes*, protect keys by enabling the detection of the misuse of broadcast decryption keys. In systems that incorporate a traitor tracing scheme, it is possible to identify at least one of the contributing traitors upon confiscation of a pirate decoder using a *traitor tracing algorithm*. For a traitor tracing algorithm to be valuable, the traceability scheme must be *frameproof*. The frameproof property ensures that a collusion of traitors cannot create a pirate decoder that would implicate an innocent user as being a traitor when the tracing algorithm is executed.

We define two types of traitor tracing algorithms: “clear-box” algorithms and “black-box” algorithms. In “clear-box” algorithms, we assume that it is possible to explicitly extract all keys embedded in a pirate decoder. In “black box” algorithms, we cannot explicitly obtain the keys in the pirate decoder, but we can infer the values of those keys by applying special inputs to the decoder and observing the corresponding outputs.

In this chapter, we focus on traitor tracing schemes where the set of privileged authorized users, T', is always equivalent to the set of authorized users T. This means that all authorized users can decode any encrypted broadcast message that is sent to any authorized user. Thus, a key revocation event or a user removal event may require a global re-keying of all users in T. We note, however, that protocols can be wrapped around such $T' = T$ traceability schemes that would enable the selective transmission of protected messages to small subsets of T [26].

We can construct a naïve traitor tracing scheme as follows. The content provider simply creates a distinct encryption/decryption key pair for each of the n users. The content provider then securely distributes the decryption keys to the appropriate user and keeps the encryption keys secret. To send a message, the content provider encrypts and transmits the message (or the message payload decryption key) n times with the n encryption keys, once for each user. Upon receipt, an authorized user employs his decryption key to decrypt one of the n transmissions. Since each user has a unique key, pirate decoders may be easy to trace via simple comparisons of keys in the decoder to keys belonging to authorized users. However, the encryption and communication steps of this scheme are rather inefficient (i.e., $O(n)$). This inefficiency can be problematic for practical usage scenarios with values of n ranging in the millions. Recent proposals for broadcast encryption and traceability schemes address these performance issues by applying more sophisticated cryptographic techniques.

4.2 Past Work

Fiat and Naor introduced broadcast encryption in [49]. In their scheme, there exists a set of n authorized users, and a content provider can dynamically specify a privileged subset (of size $\leq n$) of authorized users that can decrypt certain encrypted messages. A message can be securely broadcast to such a privileged subset unless a group of $k + 1$ or more authorized users not belonging to the privileged subset collude to construct a pirate decoder and recover the message. The communication overhead, i.e., the factor increase in message size, is $O(k^2 \log^2 k \log n)$. Also, each user must store $O(k \log k \log n)$ decryption keys. Many improvements to this scheme have been presented, but few enable the identification of traitors that collude to distribute pirate decryption keys to unauthorized users.

To combat such piracy of decryption keys, Chor, Fiat and Naor introduced traitor tracing schemes [25, 26]. Table 4.1 compares the performance of the most efficient and relevant past work to the traceability scheme proposed in this chapter (TTR). In the table, n is the maximum number of authorized users, k is the maximum tolerable collusion size, and M is a typical value for an RSA modulus (e.g., $\sim 2^{1024}$). "Sym. decryption" means a symmetric-key decryption operation. Each of the schemes cited in the table are *fully k-resilient*, which means no traitor collusion of k or fewer authorized users can successfully create an untraceable pirate decoder. If k or fewer traitors contribute to the construction of the pirate decoder, at least one of those traitors can be identified. We do not compare the new scheme to certain proposals such as the probabilistic schemes described in [26], which do not guarantee traitor traceability upon confiscation of a pirate decoder.

The Encryption Complexity & Communication Overhead column lists the number of encryption operations performed by the content provider as well as the number of ciphertexts that must be transmitted by the content provider. The Decryption Complexity column lists the number of decryption operations that must be performed by a decoding device per user. Note that in a broadcast encryption system such as the one described in Section 4.1, these two columns only relate to the cryptographic header of the broadcast message. In such a system, the broadcast message payload is always encrypted/decrypted only once, and therefore the encryption complexity, decryption complexity, and communication overhead only pertain to the cryptographic header. In many other broadcast encryption systems,

Table 4.1: Traceability scheme comparison

Traitor Tracing Scheme	**Encryption Complexity & Communication Overhead**	**Decryption Complexity per User**	**Number of Decryption Keys per User**
Naïve	$O(n)$	1 sym. decryption	1
1-level symmetric [25, 26]	$O(k^4 \log n)$	$O(k^2 \log n)$ sym. decryptions	$O(k^2 \log n)$
2-level symmetric [25, 26]	$O(k^3 \log^4 k \log(n/k))$	$O(k^2 \log^2 k \log n)$ sym. decryptions	$O(k^2 \log^2 k \log(n/k))$
Public-key [18, 78, 86, 169]	$O(k)$	$O(k)$ exponentiations	$O(1)$
TTR	$O(\max(k \log n,$ $k \log\log M/ \log k))$	~ 1 exponentiation	1

however, these two columns would reflect the encryption complexity, decryption complexity, and communication overhead for both the header *and* the payload. Lastly, the Decryption Keys column lists the number of keys that must be persistently stored by a decoder device per user.

The first row of Table 4.1 lists the costs of the naïve scheme described at the end of Section 4.1.3. The next two rows of the table list the symmetric-key one-level and two-level schemes of [25, 26]. In these two schemes, members of a traitor collusion (containing k or fewer traitors) can be identified with certainty. In the deterministic symmetric-key one-level scheme, the computation and communication costs depend on the total number of users, n, and on the largest tolerable collusion size, k. To securely broadcast secret content to the set of authorized users, each user must store $O(k^2 \log n)$ decryption keys, and each user must perform $O(k^2 \log n)$ operations to decrypt the content upon receipt of the broadcast transmission. The one-level scheme also increases the communication cost of broadcasting secret content by a factor of $O(k^4 \log n)$. The deterministic symmetric-key two-level scheme reduces the encryption complexity and communication overhead relative to the one-level scheme at the cost of increasing the decryption complexity and the number of decryption keys per user.

Pfitzmann introduced *asymmetric* traitor tracing in [125]. In the context of traitor tracing schemes, the concepts of symmetry and asymmetry differ from those in the context of encryption. In *symmetric* traceability schemes, the authorized users share all of their key information with the content provider. Thus, in a symmetric traceability scheme, a dishonest content provider can frame an innocent authorized user as being a traitor by building an "unauthorized" decoder that contains a particular user's decryption key. In the asymmetric traceability schemes, such as the scheme presented in [125], users do not share all key information with the content provider. This property can enable the implementation of a traceability scheme that allows a content provider to unambiguously convince a third party of a traitor's guilt.

The fourth row of Table 4.1 summarizes the performance of *public-key* k-resilient traitor tracing schemes (e.g., [18, 78, 86, 169]). In a public-key traceability scheme, publicly known encryption keys can be used to encrypt and subsequently transmit a secret to the entire set of authorized users. The authorized users then employ their respective private decryption keys to decode the transmission. The public-key scheme presented in [18] is symmetric, and the one described in [86] is asymmetric but requires a trusted third party. Asymmetric public-key traitor tracing schemes that do not require a trusted third party are described in [78, 169]. As listed in Table 4.1, the most efficient schemes of [18, 78, 86, 125, 169] incur a communication overhead of $O(k)$, require each user to store one decryption key, and require each user to perform $O(k)$ exponentiations per broadcast secret.

In some situations, a traitor may decrypt the broadcast information and then transmit the plaintext result to pirates rather than distribute a pirate decoder that contains valid decryption keys. Researchers have suggested combining digital fingerprinting and traitor tracing to prevent such piracy [50, 126, 136]. The systems discussed in [126] employ provably secure, robust digital watermark constructions presented in [19, 38]. More efficient and effective integrated fingerprinting and traceability schemes are described in [50, 136]. The security of all of these systems depends on the assumption that a traitor cannot reliably remove or distort digital fingerprints without greatly sacrificing content quality. However, as illustrated by the attacks on the Secure Digital Music Initiative (SDMI) technologies, it is extremely difficult to design a practical digital fingerprinting scheme that a savvy attacker cannot thwart [39, 168]. In this chapter, we do not consider scenarios in which traitors command the resources necessary to efficiently distribute decrypted content to pirates. Instead, we consider scenarios where authorized users contribute to large-scale piracy by distributing small keys that enable decryption of broadcast content.

Researchers have presented many other broadcast encryption and traceability schemes that employ a rich variety of mathematical tools (e.g., [26, 46, 57, 79, 80, 86, 117, 118, 148, 158, 159]). For instance, Kiayias and Yung propose a public key traitor tracing scheme with "constant transmission overhead" [80]. In that scheme, however, the minimum size of the broadcast message may be impractical if protection against large collusions is desired. Furthermore, unlike the proposal of this chapter, some of these previous schemes integrate $T' \subseteq T$ broadcast encryption with traitor tracing (e.g., [57, 117, 158, 159]). Though these schemes offer several different valuable security services, the decryption performance of these schemes does not significantly exceed that of the work summarized in Table 4.1.

In summary, recent traitor tracing proposals have focused on reducing the encryption and network communication requirements while providing an extensive suite of security services. Decryption in sophisticated traitor tracing schemes can be slow, however. Instead of requiring a single decryption per user as in the naïve scheme, existing optimized schemes may require dozens of modular exponentiations or thousands of symmetric-key decryptions per user per broadcast secret. As network bandwidth is growing exponentially relative to software performance, the speed of decryption is an increasingly important issue. In realistic scenarios, the content provider may possess large encryption resources (e.g., a room full of servers) and abundant network transmission bandwidth. The computation resources of the authorized user, however, are often limited to a smartcard or a single processor in a desktop computer. Thus, to improve the usability of traitor tracing schemes, decryption performance must be addressed.

4.3 A New Traitor Tracing Scheme

We now present a new fully k-resilient traceability scheme based on the RSA encryption algorithm that improves upon the decryption performance of past proposals. Naturally, we call this new scheme Traitor Tracing using RSA (TTR). Though we utilize RSA, TTR is not a public-key traitor tracing scheme; we implement RSA as a secret-key cryptosystem rather than as a public-key cryptosystem. In TTR, only the content provider has knowledge of the broadcast encryption keys; no user in U or T can directly access or utilize the encryption keys. We sacrifice public-key operation to avoid RSA common modulus attacks (which are described in Section 4.4).

The security of TTR is based upon the assumed intractability of the RSA problem and of the integer factorization problem. These computation problems are defined in Table 4.2. As we will prove in Section 4.4, the TTR encryption scheme prevents unauthorized users from successfully decrypting broadcast messages if the RSA problem is hard. Also, we will prove that TTR enables traceability of at least one traitor from a collusion of k or fewer traitors without implicating innocent users if the factoring problem is hard. We will present a clear-box tracing algorithm for TTR that can identify traitors for any type of pirate decoder, and we will present a black-box tracing algorithm for TTR that can identify traitors for special (limited) types of pirate decoders.

Essentially, TTR improves decryption performance relative to past proposals at the cost

Table 4.2: Hard computation problem definitions

Problem	**Definition**
Integer factorization problem	Given an integer M, which is the product of one or more prime integers, determine the prime factorization for M.
RSA problem	Given (i) an integer M that is the product of two distinct primes p and q, (ii) a positive integer e that is relatively prime to $\phi(M)$, and (iii) an integer b, compute an integer a such that $a^e \equiv b \bmod M$.

of increasing the computation and transmission requirements of the content provider relative to past proposals. As shown in Table 4.1, TTR requires only a single modular exponentiation operation and a relatively insignificant number of modular multiplication operations to conduct decryption upon receipt of a broadcast secret. Though modular exponentiations are computationally more expensive than symmetric key encryptions, TTR still exhibits the highest decryption performance for realistic numbers of users and traitors. Furthermore, TTR only requires each authorized user to store a single decryption key, and the communication overhead and encryption complexity are $O(\max(k \log n, k \log\log M / \log k))$. We present detailed comparative performance analysis in Section 4.6.

4.3.1 RSA Preliminaries

Since TTR is based upon the RSA encryption algorithm, we will first briefly describe the operation of standard RSA. In RSA, security is related to the computational difficulty of factoring large integers [133]. Let M be the product of two large prime integers, p and q, where p and q are roughly the same size. We call M the RSA modulus. Now, find two integers e and d such that $ed \equiv 1$ mod $\phi(M)$, where ϕ is Euler's totient function: $\phi(M) = (p-1)(q-1)$. The integers e and d are called the encryption exponent and the decryption exponent, respectively. The pair $\langle M, e\rangle$ is the encryption key, and the pair $\langle M, d\rangle$ is the decryption key. Given a plaintext block $a \in \mathbf{Z}_M^*$, where $\mathbf{Z}_M^*$ is the multiplicative group of $\mathbf{Z}_M$, a sender can encrypt a to produce a ciphertext c using the public key by computing $c = a^e \bmod M$. The receiver can decrypt c using the private key by computing $a = c^d \bmod M$. There are many minor implementation details required to fortify protocols using RSA against potential attacks. Boneh summarizes several attacks on RSA implementations in [17].

Consider a simple traitor tracing scheme based on RSA: the content provider generates a common modulus M and a key pair $\langle e_i, d_i\rangle$ for each user t_i. The provider keeps the encryption exponent e_i secret (i.e., unknown to t_i) and passes the decryption key $\langle M, d_i\rangle$ to t_i. To broadcast information to authorized users, the provider simply encrypts messages individually for each user. However, this scheme is inefficient, since the number of encryption keys and the communication overhead are the same as the number of users, n.

We can improve upon the performance of this simple scheme. Using the multiplicative properties of RSA, we can generate ciphertexts with a few encryption keys that can be decrypted using many decryption keys. The general method, which employs techniques also used in RSA-based threshold cryptosystems (initiated in [43]), operates as follows. Given a plaintext block a and a modulus M, we generate L ciphertexts $C = \{c_1, c_2, \ldots, c_L\}$ encrypted using L different non-zero positive encryption exponents $e_1, e_2, \ldots, e_L$:

$$c_j = a^{e_j} \bmod M \tag{4.1}$$

Now, a user can multiply (modulo M) all L ciphertexts to obtain a "product ciphertext", c_{PROD}, that is equivalent to encrypting a one time using a single encryption exponent that is the sum of the L encryption exponents:

$$c_{PROD} = \prod_{i=1}^{L} c_i \bmod M = \left(\prod_{i=1}^{L} a^{e_i} \bmod M\right) \bmod M = a^{e_{SUM}} \bmod M, \tag{4.2}$$

where $e_{SUM} = \sum_{i=1}^{L} e_i$. A user could subsequently decrypt the c_{PROD} using a decryption key that consists of an exponent d_{SUM} of the following form:

$$d_{SUM} = \left(\sum_{i=1}^{L} e_i\right)^{-1} \bmod \phi(M) \tag{4.3}$$

In general, upon obtaining the L ciphertext blocks, a user could multiply (modulo M) a subset of C to obtain a different product ciphertext. This new product ciphertext could be decrypted using a new decryption exponent. Since there exist $2^L - 1$ nonempty subsets of C, there exist $2^L - 1$ product ciphertexts that can be decrypted with $2^L - 1$ decryption keys. Hence, we can generate ciphertexts that can be deciphered using up to $2^L - 1$ decryption keys by performing only L encryptions with L encryption keys.

4.3.2 Components and Parameters

TTR takes advantage of the multiplicative properties of RSA to generate and support many decryption keys using relatively few encryption keys. TTR incorporates the five broadcast encryption components described in Section 4.1: Provider Initialization, User Initialization, Encryption, Transmission, and Decryption. In addition, TTR incorporates one additional component, Traitor Tracing Algorithms. The five broadcast encryption components of TTR are described below, and the tracing algorithms are described in Section 4.5. In TTR, the methods employed to perform encryption and decryption are public, but the keys used to perform these operations are secret. The content provider does not reveal secret encryption keys to the users, and authorized users (who are not traitors) do not reveal their personal decryption keys to other users.

TTR is parameterized by M, n, k, s, L, and α. These parameters are summarized in Table 4.3. M is an RSA modulus, and n is the number of authorized users in T. The parameter k represents the maximum tolerable traitor collusion size, and s is the security parameter of the scheme. For example, in a scenario with $k = 10$ and $s = 20$, any collusion of size at most 10 can produce a non-traceable key with probability at most 2^{-20}. Also, we use the parameters L and α, which are based on the values of M, n, k, and s. L represents

Table 4.3: TTR parameters

Parameter	**Expected Value(s)**	**Description**
M	$\sim 2^{1024}$	RSA modulus
n	2 to 10 billion	Number of authorized users in T
k	2 to n	Maximum tolerable collusion size
s	10 to 80	Security parameter
L	$O(\max(k \log n, k \log\log M / \log k))$	Number of encryption exponents
α	$1/k$	Probability that a Boolean decryption vector element equals 1

the number of encryption exponents in the scheme, and α relates to the construction of user decryption keys. We present the precise values and the rationale for the values of L and α in Section 4.4.4.

4.3.3 Provider Initialization

During provider initialization, the content provider creates the secret encryption keys and the information required to generate future user decryption keys. First, the content provider generates a RSA modulus $M = pq$, where p and q are both prime. For reasons that we describe below, we require that p and q be safe primes, i.e., the integers $(p-1)/2$ and $(q-1)/2$ are also prime. Second, the content provider randomly generates a vector E of L unique encryption exponents for M. For each $e_j \in E$, $e_j \sim M$. The content provider keeps all of the encryption exponents and the values of p, q, and $\phi(M)$ secret.

We assume that the content provider's secrets are contained in a single device, but we note that the content provider is not required to be centralized. We could improve security by using RSA threshold techniques (initiated in [43]) to securely store the content provider's secrets and to securely perform key generation operations across multiple devices. Furthermore, the encryption exponents and operations can be distributed across multiple content provider devices using known RSA threshold techniques. Applying such techniques would require an attacker to compromise most or all of the devices (rather than compromise only a single device) in order to successfully expose E or $\phi(M)$.

4.3.4 User Initialization

When an authorized user t_i joins the system, the content provider generates and securely distributes a user decryption key DK_i to t_i as follows:

1. Repeat the following steps until a d_i is obtained that is probabilistically prime.

   (a) Randomly generate an L-dimensional Boolean vector $v^{(i)}$. Each element in the vector is set to 1 with probability α, and each element in the vector is set to 0 with probability $1-\alpha$. In addition, $v^{(i)}$ must not consist of all zeroes, and no two $v^{(i)}$'s are the same. Repeat until $v^{(i)}$ is found where $\sum_{j=1}^{L} v_j^{(i)} e_j$ is relatively prime to $\phi(M)$.

   (b) Using the extended Euclidean algorithm, calculate a decryption exponent d_i such that:

   $$d_i = \left(\sum_{j=1}^{L} v_j^{(i)} e_j \right)^{-1} \bmod \phi(M) \tag{4.4}$$

   Then, perform a probabilistic primality test on d_i.

2. Distribute $DK_i = \langle v^{(i)}, d_i, M \rangle$ to t_i via a secure channel.

We note that the system may be implemented *without* requiring decryption exponents to be prime. We choose to require all d_i's to be prime in order to simplify the security analysis. However, as we will detail in Section 4.4.4, the encryption, decryption, and transmission performance effects of requiring the decryption exponents to be prime on encryption are insignificant in realistic scenarios.

4.3.5 Encryption, Transmission, and Decryption

To encrypt a plaintext message P using this scheme, the content provider performs L RSA encryptions on P using each of the encryption exponents in E. The resulting ciphertext C is as follows:

$$C = \langle P^{e_1} \bmod M, ..., P^{e_L} \bmod M \rangle. \tag{4.5}$$

To ensure that the ciphertext does not leak any information about the plaintext message, all plaintext messages P should be encoded using Optimal Asymmetric Encryption Padding (OAEP) [11, 16, 56, 147] or a similar method. OAEP is a provably secure mechanism for padding and encoding plaintext messages prior to RSA encryption. OAEP ensures that, regardless of the contents of the plaintext message, an adversary must be able to solve the RSA problem in order to decrypt the message.

The resulting ciphertext C is then broadcast to all users in U.

Upon receiving the ciphertext C, an authorized user t_i in T can decrypt C using his decryption key $DK_i = \langle v^{(i)}, d_i, M \rangle$, as follows:

$$P = \left(\prod_{j=1}^{L} (c_j)^{v_j^{(i)}} \right)^{d_i} \bmod M, \text{ where } c_j = P^{e_j} \bmod M. \tag{4.6}$$

In this decryption operation, the user first obtains a product ciphertext by multiplying (modulo M) the ciphertexts c_j that correspond to vector elements in $v^{(i)}$ that equal 1. Then, the user performs an exponentiation operation modulo M on the product ciphertext using the user's unique decryption exponent d_i. It is easy to see that decryption works by the definition of the decryption keys.

4.4 Security Analysis

We now evaluate the security of the TTR scheme. We consider two sets of threats: threats from unauthorized users, and threats from traitor collusions of authorized users. Rather than attempt to identify and enumerate all possible attacks that may be launched against the system, we take a different approach. We formally demonstrate security against general classes of attacks via provable polynomial-time[2] reductions involving intractable computation problems. That is, we show that if the security of TTR can be defeated with a method that uses a reasonable amount of time and resources, then that method can also be applied to efficiently solve a hard instance of an intractable computation problem. Specifically, we construct reductions using the integer factorization problem and the RSA problem, which are defined above in Table 4.2.

This section includes several theorems that formalize our security claims.

4.4.1 Security against Unauthorized Users

We first consider threats from unauthorized users u_i that exist outside of the trusted domain. Recall that the focus of this chapter is the secrecy of the broadcast messages (instead of integrity, freshness, etc.). Thus, the only attacks of concern from unauthorized users are threats to the secrecy of broadcast messages.

We model the adversary, i.e., a set of unauthorized users, as follows. Unauthorized users do not possess any encryption or decryption keys, but an unauthorized user may have access to some or all of the following information:

- A public RSA modulus M used by the content provider and the authorized users
- A polynomial number of previously transmitted plaintext-ciphertext pairs

Given this information, upon receiving a new encrypted broadcast message, the unauthorized users may attempt to decrypt that message. We do not attempt to identify all possible methods that the unauthorized users could use to decode encrypted messages, however.

[2]Throughout this section, the terms "polynomial number", "polynomial size", and "polynomial time" imply that a number or running time is bounded by a function that is polynomial in a reasonable security parameter, such as s or the number of bits in the modulus M.

Instead, we prove that if any such method exists, then that same method could be used to solve the RSA problem. This is summarized by the following theorem.

Theorem 4.1 *Given an RSA modulus M and a polynomial number of known plaintext-ciphertext pairs, a passive adversary cannot decrypt a new ciphertext with non-negligible probability assuming the intractability of the RSA problem.*

Proof. Assume that there exists a polynomial-time algorithm A such that a passive adversary can use algorithm A to decrypt a new ciphertext with non-negligible probability given a polynomial number of plaintext-ciphertext pairs. We show how to construct a polynomial time algorithm B that finds a solution to the RSA problem by using A as a subroutine.

The inputs to algorithm B are an RSA modulus M, a random encryption exponent e, and a ciphertext $c = m^e \bmod M$. The goal of B is to output the plaintext message m with non-negligible probability. Given the inputs, algorithm B works as follows.

1. Set the first encryption exponent $e_1 = e$, which is random. For $i = 2, ..., L$, generate a random value x_i. Define e_i to be $e_i = e_1 x_i \bmod \phi(M)$, but the values of e_i are not calculated explicitly because $\phi(M)$ is not known to B. Since e_1 is random and all values x_i are random, all of the encryption exponents e_i will be random.

2. For $r = 1, ..., R$, where R is polynomial in the security parameter, generate a random plaintext P_r and compute the following ciphertext:

$$C_r = \langle P_r^{e_1}, P_r^{e_2}, ..., P_r^{e_L} \rangle. \tag{4.7}$$

   For $i = 2, ..., L$, $P_r^{e_i}$ can be computed without knowing the explicit value of e_i as follows:

$$P_r^{e_i} \bmod M = P_r^{e_1 x_i \bmod \phi(M)} \bmod M = (P_r^{e_1} \bmod M)^{x_i} \bmod M. \tag{4.8}$$

3. Using the input ciphertext c in the RSA problem, create a new ciphertext C_{R+1} in the proposed traitor tracing scheme as follows:

$$C_{R+1} = \langle c, c^{x_2}, ..., c^{x_L} \rangle \tag{4.9}$$

4. Input all the R plaintext-ciphertext pairs, the new ciphertext, and the modulus M to A. When A outputs P_{R+1} as the corresponding plaintext for C_{R+1}, algorithm B outputs $m = P_{R+1}$ as the plaintext in the RSA problem.

Since $c = m^e = m^{e_1}$ and for $i \geq 2$, $c^{x_i} = m^{e_i}$, the new ciphertext C_{R+1} has the proper form. Upon decryption, the plaintext $P_{R+1} = m \bmod M$. □

Thus, in TTR, unauthorized users cannot decode encrypted broadcast messages.

4.4.2 Security against Traitor Collusions

We now consider threats from collusions of up to k authorized users that exist within the trusted domain. Authorized users already have legitimate access to all TTR broadcast messages (unlike the unauthorized users), so attacks against the secrecy of messages are not of concern. Instead, we seek to prevent two classes of attacks from traitor collusions involving the creation of new decryption keys. The first class of attacks involves the creation of a new *untraceable* decryption key. That is, we wish to prevent collusions of k or fewer authorized users from being able to successfully construct a new broadcast decryption key from which we could not efficiently identify at least one of the contributing traitors. The second class of attacks involves the creation of a new decryption key that implicates (or "frames") an innocent authorized user as being a traitor.

We model the adversary, i.e., a collusion of up to k authorized users, as follows. Collusions of authorized users may have access to some or all of the following information:

- A public RSA modulus M used by the content provider and the authorized users

- A polynomial number of previously transmitted plaintext-ciphertext pairs
- k or fewer distinct and valid decryption keys (containing k or fewer linearly independent decryption vectors and k or fewer distinct decryption exponents)

If the k or fewer decryption vectors of the traitor collusion's decryption keys are not linearly independent, then there exists a feasible attack that will allow the collusion to factor M and defeat the scheme [110]. Thus, the vectors must be constructed such that any k vectors chosen at random will be linearly independent with overwhelming probability. As we will detail in Section 4.4.4, the minimum length L that we require for the decryption vectors ensures that the collusion's decryption vectors are always linearly independent.

We do not attempt to identify all possible methods that the adversary could use to create an untraceable decryption key or to create a decryption key that frames an innocent user. Instead, we prove that if any such method exists, then that same method could be used to solve the integer factorization problem. We show that if the number of encryption exponents, L, exceeds a particular value, then a collusion of up to k traitors cannot produce a decryption key that does not implicate at least one member of the collusion as being a traitor. Furthermore, we show that, for sufficient values of L, a collusion of up to k traitors cannot produce any key that implicates an innocent user as being a traitor.

We consider the following two classes of keys that may be produced by a traitor collusion:

- The first class includes new valid decryption keys of the standard form defined by the scheme.
- The second class includes new decryption keys that are *not* of the form as defined by the scheme but can be used to successfully decrypt ciphertext and obtain correct plaintext.

We will demonstrate security for TTR against new keys from both of these classes. In particular, the pirate decryption vector generated by the collusion is not required to be a

Boolean vector, and the decryption exponents generated by the collusion are not required to be large prime numbers. Instead, we demonstrate security against pirate decryption keys in which (i) any vector element can be any integer and (ii) any decryption exponent can be any integer.

Given this generalized set of possible keys, the decryption process still follows the two basic steps (for the purposes of the security proofs):

1. Obtaining a product ciphertext c_{PROD} using a decryption vector v, i.e., $c_{PROD} = \prod_{j=1}^{L}(c_j)^{v_j} \bmod M$.

2. Computing the plaintext message P via modular exponentiation with a decryption exponent d, i.e., $P = c_{PROD}^{d} \bmod M$.

We define a "traceable key" as follows. Recall that each authorized user is issued a distinct key with a distinct decryption exponent. We say that a pirate key is traceable to an authorized user t_i if the decryption exponent associated with t_i divides one of the integer components of the pirate key. More formally stated, given a user t_i with a key $\langle v^{(i)}, d_i, M\rangle$, a pirate decryption key of the form $\langle v^*, d^*, M\rangle$ is traceable to the user t_i if d_i divides v_j^* for any $1 \leq j \leq L$ or d_i divides d^*. Thus, an "untraceable key" produced by a traitor collusion is a new key in which neither the decryption exponent nor any of the vector elements are divisible by any of the decryption exponents of the traitors' original keys.

The security of TTR against traitor collusions is summarized by the following theorems. First, Theorem 4.2 demonstrates that a collusion of k or fewer traitors cannot generate an untraceable key (as defined above) that does not implicate at least one traitor in the collusion. We prove security against the most powerful adversary, which is a collusion that possesses knowledge of M, k decryption keys, and a polynomial number of past plaintext-ciphertext pairs.

Before we present Theorem 4.2 , we state a lemma that is used by the proof of the theorem. The lemma demonstrates that given a set of k or fewer decryption vectors, each of a certain minimum length L, then there is at least one element in each vector that equals 1 where that same element equals 0 in all the other $k-1$ or fewer vectors. This property is important for ensuring that a collusion of users cannot generate an untraceable key. For the lemma, the maximum number of disjoint traitor collusions in a realistic scenario is defined to be n, which is the case where n collusions exist that each contain a single traitor. If this maximum number of collusions were exponential rather than polynomial in size, the property described by the lemma could be guaranteed by simply increasing the size of L by a polynomial factor.

Lemma 4.1 *If $L \geq (k-1)(s+\log k+\log n)/(e\log_2 e)$ and $\alpha = 1/k$, then given a maximum of n groups, each consisting of a maximum of k decryption vectors constructed as defined in the scheme, the probability is at least $1-2^{-s}$ that, for any vector v in any group, there exists an element λ such that $v_\lambda = 1$ and the λth element equals 0 in all of the other vectors in the group.*

Proof. Recall that α is the probability that a vector element equals 1. Given h decryption vectors, where $2 \leq h \leq k$, the probability S that the λth element equals 1 for a specific vector $v^{(1)}$ but equals 0 for the other $h-1$ vectors from $v^{(2)}$ to $v^{(h+1)}$ is:

$$S = (\alpha^1)((1-\alpha)^{h-1}). \tag{4.10}$$

Since $\alpha = 1/k$, it is easy to show that S is minimized when $h = k$. In this case, $S = 1/((k-1)e)$. We define the event A_1 as the event where no such index λ exists for a specific vector $v^{(1)}$ in a group of k vectors. We define P_1 to be the probability that event A_1 occurs for a group of k vectors constructed as defined by the scheme. Given S, the value of P_1 is therefore at most $(1-(1/((k-1)e)))^L$. Furthermore, the probability P_i, which corresponds to the event A_i in which no such index λ exists for a specific vector $v^{(i)}$ where $2 \leq i \leq k$, is also at most $(1-(1/((k-1)e)))^L$.

We now calculate the probability P_{group} corresponding to the event A_{group}, which is the event where, for at least one vector v in a single group, there does not exist an element λ

such that $v_\lambda = 1$ and the λth element equals 0 for all of the other vectors in the group. The value of P_{group} therefore equals the probability $\Pr\left(\bigvee_{i=1}^{k} A_i\right)$. Even though the A_i's are not independent events, we use the loose bound $P_{group} \leq \sum_{i=1}^{k} P_i$ for simplicity of analysis.

Given P_{group}, we now establish an expression for the total probability P_{total} corresponding to the event A_{total}, which is the event where, for some vector v in *any* group, there does not exist an element λ such that $v_\lambda = 1$ and the λth element equals 0 for all of the other vectors in that group. Since the maximum number of disjoint collusions within the set of n authorized users is n, $P_{total} \leq nP_{group}$, so $P_{total} \leq n \sum_{i=1}^{k} P_i$.

We now compute an expression for L such that P_{total} is exponentially small in the security parameter, i.e., $2^{-s} \geq P_{total}$. By the values of the P_i's:

$$2^{-s} \geq nk\left(1 - \frac{1}{(k-1)\,e}\right)^{L}. \tag{4.11}$$

Thus,

$$L \geq \left(\frac{k-1}{e\log_2 e}\right)(s + \log_2 k + \log_2 n)\,. \tag{4.12}$$

□

We can use the lemma to prove the following Theorem. A sketch of the proof of this theorem is presented below, and a detailed proof is presented in [111].

Theorem 4.2 *If $L \geq (k-1)(s + \log_2 k + \log_2 n) / (e \log_2 e)$ and $\alpha = 1/k$, then no collusion of k authorized users can create an untraceable decryption key with probability greater than 2^{-s} assuming the difficulty of factoring.*

Proof Sketch. Assume that there exists a polynomial-time memoryless algorithm A such that a collusion of k authorized users can employ algorithm A to create a new, untraceable decryption key with non-negligible probability. We show how to construct a polynomial time algorithm B that factors a given modulus M by using A as a subroutine.

At a high level, on input $M(= pq)$, algorithm B operates as follows. B begins by randomly generating k valid and unique decryption keys $DK_i = \langle v^{(i)}, d_i, M\rangle$ of the same form described in the scheme. These keys represent the keys of the traitor collusion. Next, the algorithm B generates an additional $(L-k)$ decryption keys of a special form, where the decryption exponents are very large and the decryption vectors are very sparse. Using all L keys, B generates a polynomial number of plaintext-ciphertext pairs that are consistent with the encryption exponents corresponding to the L keys. B then applies the first k keys and the plaintext-ciphertext pairs as inputs to A to obtain a new decryption key, DK_{NEW}. Using Lemma 4.1, we can show that B can efficiently apply the first k valid keys, the key DK_{NEW}, and the additional $(L-k)$ decryption keys to obtain a multiple of $\phi(M)$. If the multiple is non-zero, B can efficiently factor M [41]. If the multiple is zero, we can show that given an L of the required size, the key DK_{NEW} is traceable. That is, we can show that at least one of the first k d_i's must divide one of the vector elements or the decryption exponent of DK_{NEW}. □

Next, Theorem 4.3 demonstrates that a collusion of k or fewer traitors cannot generate a new key that implicates an innocent user as being a traitor. Given the definition of a traceable key described above, the following theorem shows that a collusion cannot construct a new key with a decryption exponent or vector element that is an integer multiple of the decryption exponent associated with an innocent user (that does not belong to the collusion). A sketch of the proof of this theorem is presented below, and a detailed proof is presented in [111]. As in the proof of Theorem 4.2, the proof of Theorem 4.3 demonstrates security against the most powerful adversary (i.e., a collusion with knowledge of M, k decryption keys, and a polynomial number of past plaintext-ciphertext pairs).

Theorem 4.3 *If the number of possible valid decryption keys exceeds 2^s, then the probability is exponentially small in s that a collusion of k authorized users can create a decryption key of size that is polynomial in s and that implicates an innocent user as a traitor.*

Proof Sketch. Assume that a collusion of up to k authorized users can create a new decryption key. In the proposed scheme, the selection of the $(n-k)$ innocent users' decryption vectors and exponents can be performed entirely independently of the selection of the k traitor keys. That is, the keys that represent the innocent authorized users may be any $(n-k)$-subset chosen uniformly at random from the set of $(2^s/\log M - k)$ possible decryption keys that do not belong to the traitor collusion. Given these facts, we can show that the probability is exponentially small in s that the new polynomial-sized decryption key will implicate one of the $(n-k)$ innocent authorized users. □

Thus, collusions of size k or fewer traitors cannot create untraceable decryption keys or traceable decryption keys that frame innocent users.

4.4.3 Security against Attacks on RSA

The theorems presented above demonstrate the security of the scheme against unauthorized users and against traitor collusions. In this section, we provide insight as to why the scheme is resilient against the RSA common modulus attacks described in [41, 149], even though all authorized users employ the same RSA modulus M.

We classify common modulus attacks into two types. In the first type, if an adversary has knowledge of two (or more) RSA encryption exponents used to encrypt the same message, the adversary can recover the message using the two (or more) ciphertexts without requiring knowledge of $\phi(M)$ or any decryption exponents [149]. TTR defends against this attack by treating RSA as a *secret-key cryptosystem* rather than as a *public-key cryptosystem*. The encryption keys employed by the content provider are not revealed to the authorized users, and therefore neither a collusion of users nor a passive adversary can implement this attack.

The second type of common modulus attack operates as follows. If an adversary has knowledge of an encryption key and the corresponding decryption key for a given RSA modulus M, the adversary can factor the modulus using a probabilistic algorithm or can calculate the decryption key corresponding to any encryption key [41]. Neither unauthorized nor authorized users can realize such attacks in TTR, however, for they do not possess knowledge of both the encryption and decryption keys for the modulus M. Even given a valid decryption exponent, it is not feasible for an authorized user to guess the corresponding encryption exponent: given a random decryption exponent in TTR, the corresponding encryption exponent is one of an exponential number of equally likely possibilities.

4.4.4 Choosing the Parameters

In this section, we discuss the requirements for the values of the parameters α and L. As required by Lemma 4.1, α equals $1/k$, where α is the probability that a decryption vector element will be set to 1, and where k is the maximum collusion size. It may possible for the value of α to be increased or decreased; determining the optimal value is a subject for future work. We note, however, that some values of α will cause the scheme to be insecure. For example, setting α to $1/2$ prevents traceability in some scenarios [111]. This is because traitors can combine their keys in a particular way when $\alpha = 1/2$ to effectively hide the identifying values of their decryption exponents.

The value of L, which is the number of elements in a decryption vector, depends on several factors. First, to satisfy the conditions of Lemma 4.1 and Theorem 4.2, we require the following lower bound for the value of L (where e is Euler's constant):

$$L \geq (k-1)\left(s + \log_2 k + \log_2 n\right) / \left(e \log_2 e\right) \tag{4.13}$$

Second, we must ensure that there are enough possible distinct decryption keys to accommodate all of the users in the system. Also, to satisfy the Theorem 4.3, we must ensure that the number of possible distinct decryption keys exceeds 2^s. Since the expected number of 1's in a vector of length L is $L\alpha$, the number of possible vectors is roughly $\binom{L}{L\alpha} = \binom{L}{L/k}$. However, only a subset of these vectors will correspond to a prime (and therefore valid) decryption exponent. Considering that the probability that a large exponent is prime is approximately $1/\log M$, we have $\binom{L}{L/k} \geq n \log M$, and $\binom{L}{L/k} \geq 2^s \log M$, where $\log$ is the natural logarithm. Using Stirling's approximation, a simple calculation shows that $L \geq (k(s + \log_2 n + \log_2 \log M))/\log_2 ek$ is sufficient.

Third, as explained in 4.4.2, we must ensure that any k vectors produced by the scheme are linearly independent. Otherwise, a traitor collusion of size k or fewer may be able to factor M. Hence, to maintain security, L should be large enough such that, with overwhelming probability, a set of k randomly generated Boolean vectors of length L are linearly independent. In [72], it is shown that the probability of linear independence is at least $1 - O((1-\epsilon)^L)$ for some $\epsilon > 0$ if $k \leq L$. Thus, the lower bound for L cited above is sufficient; the bound above requires $L > s$, and therefore the probability of linear dependence will be exponentially small in the security parameter s.

Hence, we have the following expression for L:

$$L \geq \max\left(\frac{(k-1)\,(s + \log_2 k + \log_2 n)}{e \log_2 e}, \frac{ks + k\log_2 n + k \log_2 \log M}{\log_2 ek}\right) \tag{4.14}$$

If the security parameter s is treated as a constant, the size of L and the communication overhead of the scheme is $O(\max(k \log n, k \log\log M/\log k))$. In realistic scenarios, $\log n > \log\log M$, so the communication overhead would be $O(k \log n)$.

4.5 Identifying Traitors

Upon confiscation of a pirate decoder device, the content provider invokes traitor tracing algorithms to identify at least one of the authorized users that contributed to the construction of the device. We now describe tracing algorithms that we can use to identify contributing traitors in a collusion of size k or fewer.

First, we explore the "clear-box" case, where it is possible to explicitly extract the representations of all the keys embedded in the pirate decoder. Using the clear-box tracing algorithm, we can always efficiently identify at least one of the traitors that conspired to construct an arbitrary pirate decoder. Second, we present a limited “black-box” tracing algorithm. In this case, we cannot extract keys from the pirate decoder, but we can apply inputs to the decoder and observe the resulting outputs. Unlike the clear-box algorithm, the black-box algorithm only enables the tracing of keys in special cases.

4.5.1 A Clear-box Tracing Algorithm

We assume that a pirate decoder contains easily recognizable representations of one or more valid decryption keys; these keys are employed by the decoder to perform all message decryptions. As shown in Section 4.4, a traitor collusion can only generate new decryption keys of a certain form. That is, traitors cannot create untraceable keys or traceable keys that implicate innocent users, and therefore we can use the keys in a pirate decoder to identify contributing traitors.

The clear-box tracing algorithm simply compares components of the decryption keys within the pirate device to all existing user decryption keys. The algorithm proceeds as follows:

1. Let $\langle v^*, d^*, M \rangle$ be a pirate key extracted from a pirate decoder, where $v^* = \{v_1^*, ..., v_L^*\}$. For $1 \leq i \leq n$, repeat the following for each authorized user t_i (whose decryption exponent is d_i):

   (a) If d_i divides v_j^* for any $1 \leq j \leq L$ or d_i divides d^*, then user t_i is a traitor.

We now present a theorem stating that, without framing innocent users, this clear-box algorithm can identify at least one of the traitors that colluded to build the pirate decoder.

Theorem 4.4 *Given a pirate decryption key generated by a collusion of at most k traitors using their respective decryption keys, the clear-box traitor tracing algorithm can identify* *at least one traitor in the collusion with probability exceeding* $1 - 2^{-s}$ *without implicating any innocent users.*

Proof. As shown by Theorem 4.2, no group of traitors of size fewer than $k + 1$ can generate a new decryption key with non-negligible probability that does not implicate at least one of the colluding traitors using the clear-box traitor tracing algorithm. Furthermore, as shown by Theorem 4.3, no collusion of traitors of size fewer than $k + 1$ can generate a new decryption key with non-negligible probability that implicates an innocent user. □

Upon discovering the existence of one or more traitors in the proposed scheme, the content provider must re-issue decryption keys to the set of authorized users (who are not identified traitors). We can address this issue by constructing a protocol that distributes new decryption keys to individual, legitimate users at fixed intervals. An example of such a protocol is described in Section 4.7.

4.5.2 A Limited Black-box Tracing Algorithm

For the black-box algorithm, we wish to achieve the same goals as desired for the clear-box tracing algorithm, i.e., identification of at least one contributing traitor and no false implications of guilt. We achieve these goals for a limited set of pirate decoders: *limited-ability single-key decoders* and *limited-ability multiple-key decoders*. We define a limited-ability single-key pirate decoder to be a device that employs a single decryption key that is identical to a valid decryption key issued by the content provider. Furthermore, the device only uses this single decryption key to perform a single decryption operation per broadcast message.

We define a limited-ability multiple-key pirate decoder to be a device that contains k or fewer decryption keys that have been issued by the content provider; any one and only one of these keys can be used to perform a single decryption for a given broadcast message. However, a limited-ability decoder cannot utilize multiple keys to perform multiple decryptions per broadcast message and compare the results. If this behavior were possible, the pirate decoder could employ a technique (such as the one described in [79]) to recognize invalid ciphertexts and defeat the black-box tracing algorithm presented below. Also, a limited-ability decoder cannot include any valid keys that are not of the form explicitly issued by the content provider.

Restricting the pirate device model to limited-ability single-key and multiple-key decoders is reasonable in many practical situations. In a smart card-based decoder or a mass-produced ASIC decoder, storage space may only be available for a single decryption key from a single traitor. Also, recall that decryption involves the computationally expensive modular exponentiation operation. Therefore, it may not be feasible for a pirate decoder to perform multiple decryptions per broadcast message and maintain adequate throughput.

In the black-box algorithm, we identify traitors by applying random data as ciphertext input to the pirate decoder. The decryption of the random data using a decryption key (issued by the content provider) will yield a different and predictable plaintext result for each distinct decryption key. Thus, we can infer which keys are stored in a limited-ability pirate decoder without performing explicit inspection of the pirate device's contents. The decryption key for authorized user t_i is $DK_i = \langle v^{(i)}, d_i, M \rangle$, and the black-box algorithm operates as follows:

1. Randomly generate a set C of L $\lceil \log_2 M \rceil$-bit values, $C = \{c_1, c_2, ..., c_L\}$.

2. Repeat for all i such that $1 \leq i \leq n$, where n is the number of authorized users:

   (a) Randomly select an integer z such that $v_z^{(i)} = 1$.

   (b) Construct $C' = \{c'_1, c'_2, ..., c'_L\}$ such that $c'_j = c_j$ for $j = z$, and $c'_j = 1$ for $j \neq z$.

   (c) Apply C' as the input to the pirate decoder.

   (d) Obtain the decrypted result, P, which is the output of the pirate decoder.

(e) Compute $P_{TEST} = \left(\prod_{j=1}^{L}(c_j')^{v_j^{(i)}}\right)^{d_i} \bmod M$.

(f) If P_{TEST} equals P, user t_i is a traitor.

In Step 1 of the black-box algorithm, we generate a random set C of L values that we will use to construct ciphertext for input to the pirate decoder. For an authorized user t_i that may be a traitor, Step 2 of the algorithm operates as follows. In Step 2(a), we identify an element z in user t_i's decryption element that equals 1. Then, in Step 2(b), we construct a special ciphertext C' such that exactly one of the elements in C' equal one of the elements in C generated in Step 1; the other elements in C' are set to 1. When this value of C' is applied to a decoder that is using only t_i's key to perform decryption, the decoder will construct a product ciphertext that equals exactly one of the elements of C. Then, the decoder will create a value of P that equals the product ciphertext raised to the power of d_i modulo M. This value of P will be unique to a user t_i, and therefore a user t_i can be identified by comparing P to the expected value P_{TEST} for that t_i.

For example, suppose t_i's decryption vector is $\{1, 1, 0, 1, 0\}$. Given a random $C = \{c_1, c_2, c_3, c_4, c_5\}$, one of many possible values for C' would be $\{1, 1, 1, c_4, 1\}$. Now, when C' is inputted to the pirate decoder that employs only user t_i's key to decrypt messages, then the P outputted by the decoder will equal $(c_4)^{d_i} \bmod M$, which is unique to t_i.

When ciphertext input is applied to a limited-ability multiple-key decoder, the device chooses one of its keys and employs that key to perform the decryption operation. As a result, the black-box tracing algorithm may only identify one of the many keys stored in the device. To find all traitors with high probability, one can simply repeat the black-box tracing algorithm a number of times that is a multiple of k, e.g., $10k$ times, assuming the decryption keys are chosen at random by the pirate decoder.

We now present a theorem stating that, without framing innocent users, the black-box algorithm can identify at least one of the traitors that colluded to build a limited-ability pirate decoder.

Theorem 4.5 *Given a limited-ability pirate decoder constructed by a collusion of at most k traitors using their respective decryption keys, then with overwhelming probability, the black-box traitor tracing algorithm can identify at least one traitor in the collusion without implicating any innocent users.*

Proof. Recall that a limited-ability pirate decoder only contains exact copies of one or more authorized decryption keys associated with one or more of the contributing traitors. Also, the limited-ability decoder uses only one of the keys at a time to perform decryption using the decryption method defined by the scheme.

The black-box tracing algorithm simply involves the application of large integers (chosen uniformly at random from $\mathbf{Z}_M^*$) as inputs to the pirate decoder. These integers are not the properly encrypted results of a plaintext P that the decoder would normally receive. As a result, the limited-ability pirate decoder performs the decryption operation on the randomly generated integers using a valid decryption key, and the the decoder outputs a garbage result. Since the number of authorized users is negligible relative to the size of the RSA modulus in practical scenarios, the probability is negligible that two different user decryption keys will generate the same garbage result. Hence, the black-box algorithm can identify a traitor by computing the expected garbage result for all possible authorized decryption keys and subsequently comparing those results to the output of the pirate decoder device. If the decoder contains multiple traitor decryption keys, the black-box algorithm will identify all the traitors with high probability after repeating the algorithm for a number of times that is a multiple of the maximum number of traitors (assuming the decoder employs each traitor decryption key with equal probability). $\square$

4.6 Performance Analysis

This section investigates the computation and storage costs of TTR.

4.6.1 Provider Initialization Costs

The computation and storage costs of the provider initialization procedure described in Section 4.3.3 are as follows. First, the provider must generate two safe primes to produce M. The probability that an $(\log M)$-bit random number is a safe prime is $1/O(\log^2 M)$, so the computation required to generate the modulus M is dominated by $O(\log^2 M)$ $(\log M)$-bit random number generations and $O(\log^2 M)$ $(\log M)$-bit probabilistic primality tests such as Miller-Rabin (a summary of which can be found in [114]). Second, the generation of the encryption exponents E simply requires L $(\log M)$-bit random number generations. Note that since the provider initialization is performed only once, the amortized computation cost of the provider initialization is not significant.

For convenience of user initialization and encryption of broadcast messages, it is prudent for the content provider to store E, M, and $\phi(M)$. Thus, the expected storage requirement for the content provider is at minimum $L\lceil\log_2 M\rceil + 2\lceil\log_2 M\rceil$ bits.

4.6.2 User Initialization Costs

The computation and storage costs of the three-step user initialization procedure (described in Section 4.3.4) are as follows.

Since E and values of $v^{(i)}$ in Step 1(a) (of the initialization process described in Section 4.3.4) are chosen uniformly at random, the number of possible summations modulo $\phi(M)$ that are relatively prime to $\phi(M)$ is $\phi(\phi(M))$. Thus, the probability β that the summation in Step 1(a) is relatively prime to $\phi(M)$ is $\phi(\phi(M))/\phi(M)$. Since both p and q are safe primes, simple calculation shows that $\beta \approx 1/2$. This means that we can consider 1 out of every 2 vectors as being possible user decryption keys in Step 1(b). If we do not ensure that p and q are safe primes, the lower bound for the value of β is $1/(6 \log\log(M - p - q + 1))$ [114]. In that case, the number of possible user decryption keys is much smaller, and we would have to increase the number of encryption exponents, L, to guarantee the existence of a sufficient number of unique user decryption keys. By performing extra computation (with negligible amortized cost) in the provider initialization phase, we can guarantee that p and q are safe primes, and thus we can significantly decrease the costs of preparing and distributing broadcast messages by reducing L.

In Step 1(a), the computation includes the generation of a random L-bit value and the computation of the sum (modulo $\phi(M)$) of at most L $(\log M)$-bit integers. The extended Euclidean algorithm does not need to be explicitly performed to determine if the sum is relatively prime to $\phi(M)$, as any sum that is odd will be relatively prime to $\phi(M)$ with overwhelming probability. Since the probability of the sum not being relatively prime to $\phi(M)$ is $1/2$, Step 1(a) will be executed fewer than 2 times per each iteration of Step 1.

In Step 1(b), the computation is dominated by a single probabilistic primality test such as Miller-Rabin. Since the probability of d_i being prime is approximately $1/\log M$, Steps 1(a) and 1(b) will be repeated fewer than $\log M$ times on average. Hence, the total computation time of all $\log M$ iterations of Step 1 is dominated by $O(\log M)$ $(\log M)$-bit probabilistic primality testing operations.

Following key generation, the costs required in Step 2 to securely distribute the user decryption key highly depend on the method that is chosen to secure the channel.

A user decryption key, which consists of a Boolean vector v, a prime decryption exponent d, and a modulus M, requires at most $L + 2\lceil\log_2 M\rceil$ bits of storage. This equates to a decryption key size of approximately 256 bytes in realistic scenarios when using a 1024-bit modulus. The content provider also needs to store a copy of each issued decryption key to avoid issuing the same decryption key to two different users.

4.6.3 Encryption and Transmission Costs

To encrypt a broadcast message as described in Section 4.3.5, the content provider must perform L multiple-precision modular exponentiations using L different encryption exponents. We note that we can significantly reduce the bit length of the encryption exponents to improve the speed of the encryption operations without compromising security.

Since one encrypted block is transferred for each encryption exponent, the communication overhead is $O(L)$. Though the size L may range in the hundreds, for the broadcast encryption model described in Section 4.1, the increased transmission costs apply only to the cryptographic header of the broadcast messages. The message payload does not increase in size.

4.6.4 Decryption Costs

To decrypt a broadcast message header as described in Section 4.3.5, an authorized user must perform αL multiple-precision modular multiplications and a single multiple-precision modular exponentiation. For reasonable values of M, n, and k, the computation associated with the $O(\alpha L) = O(\max(\log n, \log\log M/\log k))$ modular multiplications is much less than the cost of the single modular exponentiation.

Tables 4.4 and 4.5 list the computation required to perform decryption for various sizes of n and k. The values in the table are normalized to a single random 1024-bit modular exponentiation. For example, a value of 1.015 indicates that the decryption operation requires 1.5% more computation than a 1024-bit modular exponentiation. Table 4.4 displays the computation for the case where $2^{-s} = 2^{-20}$, and Table 4.5 lists the computation for the case where $2^{-s} = 2^{-80}$. Some table cells do not have entries because the maximum collusion size k cannot exceed the total number of users. As shown by the tables, the cost of generating the product ciphertext never exceeds 2% of the overall decryption computation.

For example, if $k = 10$, $2^{-s} = 2^{-80}$, and n equals one million users, then the number

Table 4.4: Decryption computation cost for $2^{-s} = 2^{-20}$

Number of Users (n)	**Number of exponentiations**			
	$\boldsymbol{k = 2}$	$\boldsymbol{k = 10}$	$\boldsymbol{k = 100}$	$\boldsymbol{k = 1000}$
2	1.002	-	-	-
10	1.003	1.003	-	-
100	1.004	1.004	1.005	-
1,000	1.005	1.004	1.005	1.006
10,000	1.005	1.005	1.006	1.007
100,000	1.006	1.005	1.006	1.007
1 million	1.007	1.006	1.007	1.008
1 billion	1.010	1.007	1.009	1.009

of encryption exponents L is 237. The number of modular multiplications required to obtain the product ciphertext is therefore $237\alpha - 1 = 237/10 - 1 \approx 23$. If the size of the RSA modulus is 1024 bits, the exponentiation requires 1535 modular multiplications on average [114]. Hence, in this case, generating product ciphertext requires only 1.48% of the computation involved in decryption.

In practice, a 1024-bit modular exponentiation can be 1000 times slower per decrypted bit than a 128-bit symmetric key decryption operation [114]. However, for realistic numbers of authorized users and traitors, the new scheme still exhibits the highest decryption performance among the past proposals listed in Table 4.1. Assuming a modular exponentiation is 1000 times as slow as a symmetric key decryption operation, Table 4.6 summarizes the decryption speedups provided by TTR over the one-level symmetric-key-based scheme of [26]. That is, the speedup values in the table represent the decryption throughput of TTR (when $2^{-s} = 2^{-80}$) divided by that of the one-level scheme in [26]. The table entries that denote positive speedups are highlighted in bold, and some table cells do not have entries because the maximum collusion size k cannot exceed the total number of users. As illustrated Table 4.6, when k is 10 or greater, TTR outperforms the schemes of [26]. In

Table 4.5: Decryption computation cost for $2^{-s} = 2^{-80}$

Number of Users (n)	**Number of exponentiations**			
	$\boldsymbol{k=2}$	$\boldsymbol{k=10}$	$\boldsymbol{k=100}$	$\boldsymbol{k=1000}$
2	1.006	-	-	-
10	1.006	1.012	-	-
100	1.007	1.013	1.015	-
1,000	1.007	1.013	1.015	1.016
10,000	1.007	1.014	1.016	1.016
100,000	1.007	1.014	1.016	1.017
1 million	1.008	1.015	1.017	1.018
1 billion	1.010	1.016	1.019	1.019

a realistic implementation — such as a satellite video broadcast — where n may be one million and k may be 10, the decryption speedup exceeds 7.86.

4.6.5 Tracing Algorithm Costs

The computation costs of the tracing algorithms described in Section 4.5 are as follows.

Clear-box Traitor Tracing

The clear-box tracing algorithm runs in polynomial time. The algorithm requires at most $O(nL)$ integer division operations to identify a traitor. We note that the actual computational complexity of the algorithm is a function of n_{CUR}, which is the current number of authorized users, rather than n, which is the maximum number of authorized users. This is an important distinction, as the values of n_{CUR} and n can differ by orders of magnitude in practice.

Table 4.6: Decryption speedup of TTR

Number of Users (n)	**Speedup for maximum collusion size k**			
	$k = 2$	$k = 10$	$k = 100$	$k = 1000$
2	0.02	-	-	-
10	0.05	**1.31**	-	-
100	0.11	**2.62**	**261.91**	-
1,000	0.16	**3.93**	**392.65**	**39,237.97**
10,000	0.21	**5.24**	**523.25**	**52,288.93**
100,000	0.26	**6.55**	**653.71**	**65,325.73**
1 million	0.32	**7.86**	**784.03**	**78,348.40**
1 billion	0.47	**11.77**	**1,174.15**	**117,331.91**

Limited Black-box Traitor Tracing

The limited black-box tracing algorithm runs in polynomial time. Including the operations performed by the pirate decoder, the algorithm requires at most $O(n)$ modular exponentiations and $O(n\alpha L) = O(\max(n \log n, n \log\log M/\log k))$ modular multiplications to identify a traitor. Also, as in the clear-box case, we can substitute n with n_{CUR}, the current number of users, to obtain tighter bounds on the performance. We note that the values of P_{TEST} for each user can be precomputed (at user initialization time) and can be stored in a hash table. Using this precomputed hash table, we can reduce the expected computation required by the black-box tracing algorithm for single-key decoders to 1 modular exponentiation and an insignificant number (i.e., $O(\alpha L) = O(\max(\log n, \log\log M/\log k))$) of modular multiplications.

In the multiple-key case, the computational complexity will increase by a factor of the number of times that we have to repeat the algorithm. Since we repeat the algorithm a small multiple of k times, when using the hash table described above, the computation for the multiple-key case is $O(k)$ modular exponentiations and $O(k\alpha L)$ modular multiplications.

4.7 A Key Renewal and Revocation Protocol

The traceability scheme presented in this chapter can identify traitors upon confiscation of a pirate decoder, but the scheme does not include a method for re-issuing keys to the remaining authorized users. In this section, we describe a simple protocol that incorporates the proposed traceability scheme and enables renewal and revocation of user decryption keys. Since every user possesses a unique decryption key, the content provider can broadcast special messages that can only be decrypted by individual users. This fact allows us to construct a simple and efficient key management protocol.

Issuing new decryption keys to the set of authorized users upon every group modification event in a broadcast encryption scheme can seriously degrade performance [142]. Such group modification events include users joining the group and users exiting the group due to key expiration, key theft or key piracy. By re-issuing decryption keys to authorized users at fixed time intervals rather than at every group modification event, the performance impact of key updates is significantly reduced [142]. We employ this concept in the proposed protocol, which consists of five operations: INITIALIZE, USERJOIN, USERRENEW, USERLEAVE, and SWITCHKEYS. These operations employ the components of the proposed TTR scheme described in Section 4.3 with minor amendments. The parties involved are the content provider and the authorized users.

- **INITIALIZE.** The content provider initializes the system by generating two RSA moduli, M_a and M_b. The content provider subsequently generates two sets of encryption exponents, E_a and E_b, using the same procedures described in Section 4.3.3. To encrypt broadcast messages as per Section 4.3.5, the content provider uses M_a and E_a.

- **USERJOIN.** When a user t_i joins the set of authorized users, the content provider securely issues a decryption key $DK_i = \left\langle v^{(i)}, d_i, M_a \right\rangle$ to that user. DK_i is generated using the modulus M_a and the set E_a with the technique described in Section 4.3.4.

- **USERRENEW.** After a period of time, the content provider issues a new decryption key to user t_i. This new decryption key, $DK'_i = \left\langle v'^{(i)}, d'_i, M_b \right\rangle$, is generated using the modulus M_b and the set E_b using the same technique presented in Section 4.3.4. The content provider then encodes DK'_i as a message P, and P is encrypted as follows:

$$C = P^x \bmod M_a, \text{where } x = \left(\sum_{j=1}^{L} v_j^{(i)} e_j \right), \text{and } e_j \in E_a \tag{4.15}$$

  The encrypted result C is then broadcast to all users, but it can be shown that only user t_i can decrypt C and obtain DK'_i using the decryption key DK_i. User t_i then stores DK'_i but continues to use DK_i to perform decryption operations.

- **USERLEAVE.** Upon theft, loss, revocation, or piracy of a key DK_i associated with user t_i, DK_i is no longer considered valid. No explicit action is taken in this operation, but the user will no longer receive USERRENEW messages for DK_i that would enable decryption following the next SWITCHKEYS operation. In addition, if the user's key was stolen or lost (as opposed to pirated by the user or revoked), the content provider will assign a new identifier j to the user, where $j \neq i$, and the content provider will issue a new decryption key DK_j to the user via USERJOIN.

- **SWITCHKEYS.** After all authorized users have received new decryption keys corresponding to the modulus M_b, the content provider issues a special message instructing users to begin using their new keys DK'_i. Following this message, $DK_i \leftarrow DK'_i$ for all users t_i; the decryption keys DK_i are discarded. In addition, the content provider sets $E_a \leftarrow E_b$ and $M_a \leftarrow M_b$. Lastly, the content provider generates new values for E_b and M_b in anticipation of future USERRENEW and SWITCHKEYS events.

USERRENEW and SWITCHKEYS operations should be executed as often as possible without significantly impacting performance as discussed in [142]. This periodic key renewal system achieves high performance through imperfect security: the latency between

SWITCHKEYS events may allow unauthorized parties to access encrypted messages for a short period of time following a USERLEAVE operation.

4.8 Summary

In broadcast encryption systems, a content provider seeks to securely transmit information to n subscribing users over public channels. Authorized users can subvert the security of such systems by replicating or generating valid decryption keys and distributing decoding devices that include these keys. To deter such piracy, researchers have proposed traitor tracing schemes. Using these schemes, one or more of users who contribute to a pirate decoder can be identified based upon the behavior or contents of the confiscated decoder. Most past traitor tracing proposals guarantee k-resilience. This means that tracing is possible for traitor collusions of size k or fewer.

This chapter introduces a new traitor tracing scheme for broadcast encryption. The new scheme applies RSA as a secret-key cryptosystem rather than as a public-key cryptosystem. A single RSA modulus is shared by all the authorized users, but the system is not vulnerable to known RSA common modulus attacks. If k or fewer authorized users collude to create a pirate decryption device, at least one contributing traitor can be identified. Furthermore, the scheme prevents traitors from framing innocent users. The scheme also supports limited black-box tracing. In addition, unlike other traitor tracing proposals that require the implementation of uncommon cryptographic primitives, the new scheme uses the widely deployed RSA algorithm. A key issuance and revocation protocol is also described that can facilitate the incorporation of the new scheme into many different secure computing architectures.

The scheme significantly improves upon the decryption efficiency of past traitor tracing proposals. To decrypt a broadcast message, an authorized user essentially performs a single modular exponentiation. This exceeds the decryption performance of previous schemes, which require $O(k)$ exponentiations or thousands of symmetric key operations per message decryption. Also, in the new scheme, each authorized user needs to store only a single decryption key (or two decryption keys if we employ the key renewal protocol described in Section 4.7).

Chapter 5

Processor Support for Fast Subword Mappings

Several popular cryptographic primitives, such as symmetric-key encryption algorithms, employ bit-level permutations and mappings to rapidly achieve a desired level of security. However, general-purpose processors cannot complete these mapping operations efficiently. As a result, cryptographic primitives can cause significant system performance problems that inhibit the adoption of security technologies.

This chapter describes hardware architectural enhancements for the acceleration of subword permutations and mappings. More specifically, we propose two new processor instructions, their hardware implementations, and their associated software usage, with which we can accelerate subword mappings in cryptographic software. This acceleration can lead to significant improvement in the throughput of current and future ciphers. The contributions of this chapter are based in part on the work previously published by the author in [77, 105].

This chapter is organized as follows. Section 5.1 discusses subword arithmetic and the mathematics of bit-level and subword mappings. Section 5.2 describes and compares past work. Section 5.3 presents two new instructions for fast subword permutations and mappings. Section 5.4 demonstrates how to apply these instructions to perform mappings for variably sized subwords packed in 64-bit words or packed in words that are multiples of 64 bits in size. Section 5.5 presents the supporting hardware required to implement the two new instructions. Section 5.6 analyzes the performance impact of the enhancements on subword permutations and mappings. This section also evaluates the performance improvement effected by the instructions for a popular symmetric-key encryption algorithm. Section 5.7 summarizes the chapter.

5.1 Subword Processing, Permutations, and Mappings

A data word can be interpreted as a ordered set of data subwords. For example, a 32-bit word may consist of eight 4-bit subwords. Many multimedia and cryptographic applications perform operations involving 1-bit or multiple-bit subwords. Due to the commonality of such operations, several microprocessor instruction set architectures have been extended to include subword-parallel integer arithmetic instructions that improve performance by executing several operations on low-precision data in parallel. Some of these extensions include MAX [88] and MAX-2 [91] for HP PA-RISC, VIS [162] for Sun SPARC, AltiVec [44] for PowerPC, 3DNow! [123] by AMD, MMX [124] for Intel IA-32, and IA-64 multimedia instructions [66, 92].

In many application scenarios, it is necessary to rearrange the subwords within a single register or between multiple registers using permutation and mapping techniques. A *permutation* is an invertible rearrangement of the elements in an ordered set. That is, a permutation is a *bijective mapping* from an ordered set S to itself [7]; each element in S is mapped to one and only one element in S. Conversely, a *non-bijective mapping* from a source ordered set S to a destination ordered set D can map an element in S to zero, one, or multiple elements in D.

For example, if S is the ordered set $(a,\ b)$, there exist 2 possible permutations of S, $(a,\ b)$ and $(b,\ a)$, but there exist 4 possible non-bijective mappings of S: $(a,\ b)$, $(b,\ a)$,

$(a,\ a)$, and $(b,\ b)$. In the remainder of this thesis, we use the term “mappings” to include both bijective mappings (i.e., permutations) and non-bijective mappings.

We can employ subword permutations to efficiently perform transformations such as matrix transposition in multimedia applications [89]. Furthermore, several cryptographic algorithms use subword mappings in conjunction with other operations (such as table lookups) to achieve diffusion [157] (which is described in Section 2.3.1). Cryptographic operations that employ such mappings include symmetric-key encryption algorithms such as DES [119], Twofish [141] and Serpent [13]. Some of the mappings in these cryptographic algorithms are bijections, but others are not. For instance, the “expansion permutation” in DES maps some bits in the source datum to multiple destinations in the result datum. If no information is lost in a mapping, the mapping is invertible and therefore can be used in any cryptographic algorithm. Even if information is lost, cryptographic hash functions and encryption algorithms based upon Feistel networks can still employ non-bijective mappings [139].

5.2 Past Work

Several methods exist for performing mappings in software. In one method, individual bits of the source datum are selected and shifted to their destination locations using a series of bitwise AND, bitwise OR, and shift instructions [97]. For an arbitrary mapping of the bits in an n-bit word, this procedure requires as many as $4n$ instructions. If the architecture includes instructions such as `extract` and `deposit` [90][1], one can reduce the instruction count of this procedure to $2n$, yet this method is still unacceptably slow.

[1]The `extract` and `deposit` instructions essentially combine a logical shift instruction and a bitwise AND instruction into a single instruction. The `extract` instruction selects a variable-sized contiguous bit field from anywhere in a source register and places the field right-aligned in another register. A `deposit` instruction places a right-aligned field of bits from a source register into any location in another register. These instructions can improve the performance of table lookups by reducing the number of steps needed to prepare the memory address associated with an index to a table with multi-byte entries.

Alternatively, we can employ lookup tables to perform mappings in software [97]. First, we divide the n-bit source datum into x groups of bits; each group of bits is then applied as an index to a unique lookup table corresponding to that group. The output of a lookup table represents the input group of bits permuted per the desired mapping. The bits of the table output that do not represent any of the input bits are set to zeroes. Therefore, we can combine the outputs of the x lookup tables using $(x - 1)$ bitwise OR or bitwise XOR operations to generate the desired mapped n-bit result.

In general, assuming the `extract` instruction is available, we require $(3x-1)$ instructions to complete an n-bit mapping using x lookup tables. These $(3x - 1)$ instructions are composed of the following. First, each of the x table lookups requires a single `extract` instruction to generate the table index and a single `load` instruction to obtain the desired table entry. Thus, $2x$ instructions are needed to perform the actual lookups. Second, an additional $(x - 1)$ XOR instructions are needed to combine the x results of the lookups.

Each of the x lookup tables consists of $2^{(n/x)}$ entries, and each table entry is n bits in size. Hence, the total size of the tables is $(nx) \cdot 2^{(n/x)}$ bits. This technique is commonly used but is unattractive because the mappings must be statically encoded in the tables at compile-time. Furthermore, the space required to store the lookup tables can be large and expensive. For example, we need 2 megabytes of storage to map a 64-bit datum in 11 instructions using 4 lookup tables. With 8 lookup tables, we require 16 kilobytes of storage and 23 instructions to map a 64-bit value.

A detailed treatment of existing and proposed permutation support in general-purpose processors appears in [144]. Multiple instruction set architectures have been amended to include instructions for mappings of 8-bit or larger subwords. The `permute` instruction in the MAX-2 extension to PA-RISC supports permutations and non-bijective mappings of 16-bit subwords in a 64-bit word by statically encoding the mapping function in the instruction [91]. In IA-64, the `permute` instruction supports a small set of mappings of 8-bit subwords in a 64-bit word and supports all mappings of 16-bit subwords in a 64-bit word

[66]. Similar to `permute` in MAX-2, the mapping function is statically encoded in the `permute` instruction at compile-time. The `vperm` instruction in the AltiVec extension to the PowerPC instruction set architecture maps the 8-bit subwords of a 128-bit vector register [44]. This instruction requires three 128-bit register reads and one 128-bit register write, and the mapping function is encoded in one of the vector source registers. None of the mapping instructions in popular ISAs efficiently support arbitrary mappings of 4-bit or smaller subwords.

Recently, researchers have proposed several instructions for performing arbitrary, dynamically specified permutations and mappings of 1-bit or larger subwords. Using the `pperm` instruction, we can complete an arbitrary mapping of n bits in $O(\log n)$ instructions [97, 145]. The `xbox` instruction performs n-bit permutations in a similar fashion [22]. We can conduct a 64-bit mapping by executing 8 `pperm` or `xbox` or instructions followed by 7 bitwise XOR or OR instructions to combine the results. Essentially, the `xbox` and `pperm` instructions dynamically configure and invoke an n-by-n crossbar without requiring the processor to maintain any additional state information. Amending an ISA by requiring additional state variables would be undesirable: such changes require explicit OS support and increase the complexity of context switches and interrupts. Also, the number of `pperm` or `xbox` instructions that we need to complete an arbitrary mapping does not decrease as subword size increases (and the total number of subwords to map decreases).

Using the `grp` instruction, we can complete an arbitrary (bijective) permutation of b-bit subwords packed in an n-bit word in $\log_2(n/b)$ instructions [97, 145]. The hardware needed to support the `grp` instruction can be expensive, however. The `cross` instruction employs a Benes network to complete an arbitrary permutation using $\log_2(n/b)$ instructions [97, 174]. The `omflip` instruction improves upon the `cross` instruction by using more efficient hardware to complete arbitrary permutations in the same number of instructions [97, 173]. The `bfly` and `ibfly` instructions [146] and the techniques presented in [98, 99] reduce the number of cycles required to complete an arbitrary permutation to

$O(1)$. These instructions and techniques achieve this high level of performance via various combinations of storing additional state in the processor, treating multiple independent instructions as atomic instruction bundles, and using VLIW architectural extensions. Although `grp`, `cross`, `omflip`, and `bfly/ibfly` can perform an arbitrary permutation of 1-bit subwords quickly, these instructions cannot efficiently perform non-bijective mappings.

5.3 New Instructions for Subword Mappings

We propose two new instructions to efficiently support permutations and non-bijective mappings of 1-bit or multiple-bit subwords: `swperm` and `sieve`. Using these instructions, we can dynamically specify permutations and mappings during program execution rather than force the mappings to be statically encoded at compile-time.

5.3.1 Preliminaries

We can encode a mapping by specifying an element in the source set S that is written to a particular element in the destination set D for all of the elements in D. If the mapping is arbitrary, the following expression describes the minimum number of bits needed to encode a mapping:

$$\sum_{i=1}^{||D||} \log_2 ||S|| \tag{5.1}$$

We examine mappings of b-bit subwords packed in an n-bit source register that we write to b-bit subwords of an n-bit destination register. Hence, $||S||$ is equivalent to the number of bits in the source register, n, divided by the subword size, b, and $||D||$ is the number of bits in the destination register, n, divided by the subword size, b. We can rewrite the expression as follows:

$$\sum_{i=1}^{||D||} \log_2 ||S|| = \sum_{i=1}^{n/b} \log_2(n/b) = \frac{n}{b} \log_2 \left(\frac{n}{b}\right) \tag{5.2}$$

Table 5.1: Minimum number of bits needed to specify an arbitrary mapping

Subword size	**Number of subwords per 64-bit register**	**Number of bits to encode a 64-bit mapping**
32 bits	2 subwords	2 bits
16 bits	4 subwords	8 bits
8 bits	8 subwords	24 bits
4 bits	16 subwords	64 bits
2 bits	32 subwords	160 bits
1 bit	64 subwords	384 bits

We assume that all registers are 64 bits wide; therefore n equals 64. Table 5.1 summarizes the minimum number of bits needed to specify an arbitrary 64-bit mapping when using subword sizes ranging from 1 bit to 32 bits.

RISC instructions typically allow two register reads and one register write per instruction. We wish to design instructions that allow us to dynamically specify mappings at run-time, so we use one of the 64-bit source registers, `rs`, to store the information to be permuted, and we use the other 64-bit source register, `rp`, to store information concerning the mapping function. As shown in Table 5.1, for subwords of size greater than or equal to 4 bits, we require at most 64 bits of information to specify the entire mapping. Hence, we can describe the entire mapping in a single instruction. Since we can specify 64 more configuration bits with each additional mapping instruction, mappings of 32 2-bit subwords require at least 3 RISC instructions, and mappings of 64 1-bit subwords require at least 6 RISC instructions.

5.3.2 The `swperm` Instruction

The `swperm` instruction maps 4-bit subwords of a 64-bit source register `rs` according to information stored in a 64-bit source register `rp` with a single instruction. This instruction

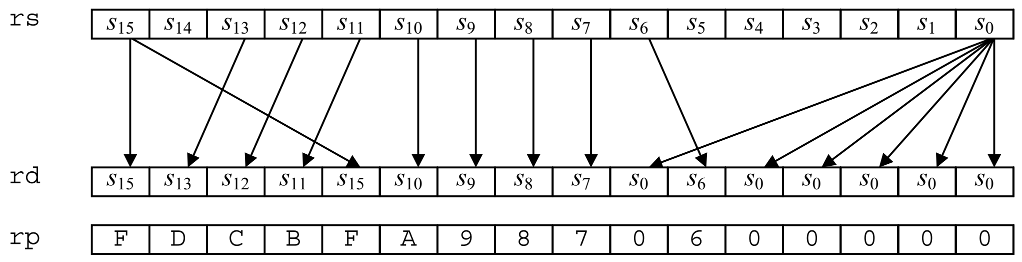

rs
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
rd
s_{15} s_{13} s_{12} s_{11} s_{15} s_{10} s_9 s_8 s_7 s_0 s_6 s_0 s_0 s_0 s_0 s_0
rp
F D C B F A 9 8 7 0 6 0 0 0 0 0

Figure 5.1: Example operation of the `swperm` instruction

writes the mapped result to the 64-bit destination register `rd`. The information stored in `rp` fully describes the desired mapping, so one can specify the mapping function dynamically. The instruction format of `swperm` is:

```
swperm rd,rs,rp
```

We designed this instruction to map subwords of size 4 bits or greater in a single cycle and to expedite mappings of 1-bit and 2-bit subwords.

Figure 5.1 illustrates an example operation of `swperm`. In the figure, s_i is the ith 4-bit aligned subword of the source register `rs`. We express the contents of `rp` necessary to complete the example mapping in hexadecimal. The value of the ith 4-bit subword in `rp` indicates which aligned 4-bit subword in the source register should be mapped to the ith 4-bit subword in the destination register.

5.3.3 The `sieve` Instruction

We use the `sieve` instruction to “filter” bits from `rs` and then direct the resulting bits into particular destinations in `rd`. While the `swperm` instruction operates *globally* over the 4-bit subwords of `rs`, the `sieve` instruction operates *locally* within the 4-bit subwords of `rs`. `sieve` directs 1 (or 2 bits) from each 4-bit subword of `rs` to 4 (or 2) possible locations in the corresponding 4-bit subword of `rd`. The `sieve` instruction utilizes a third register, rp, to configure the bit filter. In combination with the swperm instruction, we can employ the sieve instruction to implement arbitrary mappings of 1-bit or 2-bit subwords. The instruction format for sieve is:

```
sieve,h,f rd,rs,rp
```

The 4-bit function code of sieve consists of a 1-bit value, h (denoted as h in this discussion), and a 3-bit value, f (denoted as $f_2f_1f_0$ in this discussion).

Figure 5.2 illustrates two example operations of the sieve instruction on the ith 4-bit subword of rs. In the figure, $s_{i,j}$ represents the jth bit of the ith subword of rs, and $d_{i,j}$ represents the jth bit of the ith 4-bit subword of rd. The sieve instruction operates in one of two modes: "1-bit mode" enables mappings of 1-bit subwords, and "2-bit mode" facilitates mappings of 2-bit subwords. In 1-bit mode, the instruction directs one of the four bits in the ith subword of rs to one of the four bits of the ith subword of rd; the instruction sets the remaining three bits in the ith subword of rd to 0. Similarly, in 2-bit mode, sieve directs either the leftmost (i.e., most significant) two bits or the rightmost (i.e., least significant) two bits of the ith 4-bit subword of rs to either the left half or the right half of the ith 4-bit subword of rd. The instruction sets the remaining two bits of the ith subword of rd to 0.

Bits from rp and the function code bit h specify which 1-bit or 2-bit subword the instruction selects from the ith subword of rs. In 1-bit mode, one bit from every 4-bit subword of rs is selected and passed to rd. Hence, there exist four possible selection operations per rs subword, so we need two bits to encode the selection operation for each subword. Since a 64-bit register consists of sixteen 4-bit subwords, we need a total of 32 bits to encode the selection operations for all sixteen subwords. We store these 32 bits in the register rp. To minimize the number of memory access instructions that we potentially need to load the bit selection information into registers, we use one 64-bit register to store the 32 bits of selection information for two sieve instructions. The function code bit h

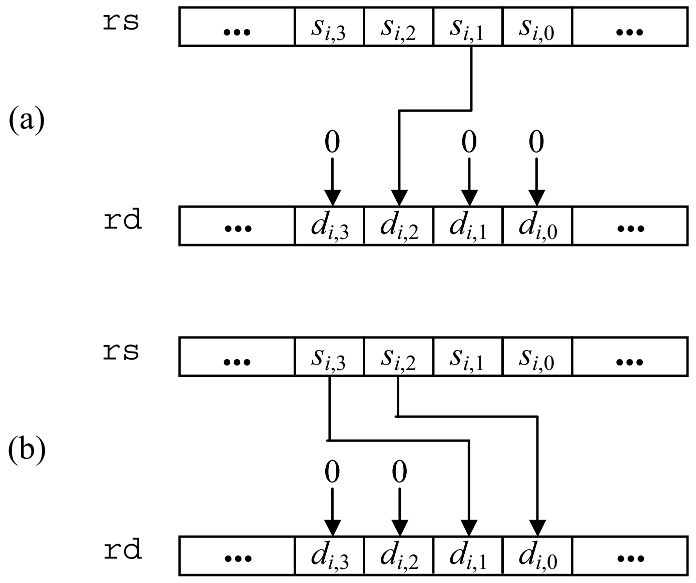


Figure 5.2: Example operation of the `sieve` instruction (a) in 1-bit mode and (b) in 2-bit mode

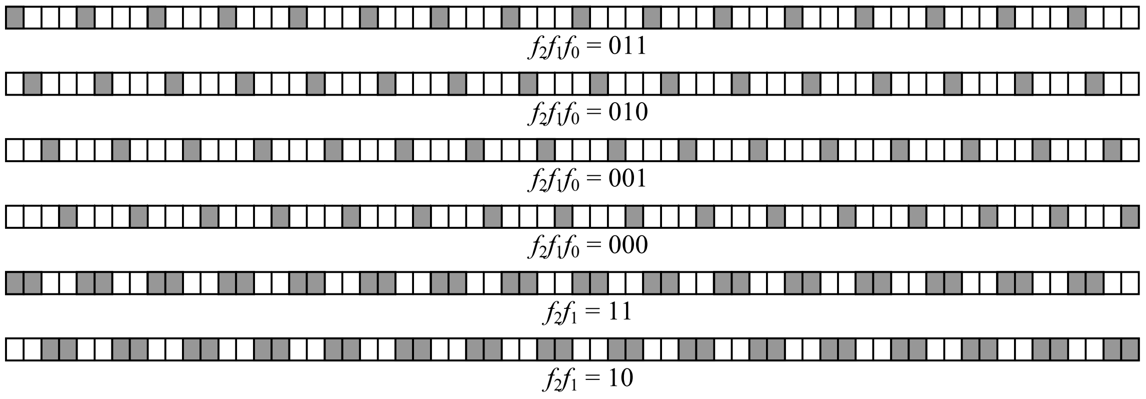


Figure 5.3: Effect of `sieve` function code bits on `rd`

indicates whether to use the most significant or least significant 2-bit half of each 4-bit `rp` subword to perform the `rs` bit selection. In 2-bit mode, we only require one bit (rather than two bits) of selection information per 4-bit `rs` subword. Hence, we encode `rp` and h as described above, but the even bits of `rp` are ignored.

Figure 5.3 illustrates which bits of `rd` receive bits of `rs` given different values of the three function code bits $f_2f_1f_0$. In the figure, the boxes containing 64 blocks represent the 64-bit register `rd`. The gray blocks represent bits that receive bits from `rs`; the white blocks represent the bits of `rd` that we set to zeroes. f_2 indicates whether to use 1-bit or 2-bit mode. Bits f_1f_0 of the function code indicate which bit of each 4-bit `rd` subword receives a selected bit from `rs` in 1-bit mode. For example, when $f_1f_0 = 00$, only the zeroth bit of each 4-bit `rd` subword receives a bit from `rs`. In 2-bit mode, f_0 is ignored, and f_1 indicates which 2-bit half of each 4-bit `rd` subword receives selected bits from `rs`.

To summarize, the `sieve` instruction selects a single bit or an aligned pair of bits from each of the sixteen 4-bit subwords of the source register `rs`, but `sieve` only maps these selected bits to the destination register `rd` in one of six possible ways, as shown in Figure 5.3. Figure 5.4 illustrates a complete example operation of the `sieve` instruction in 1-bit

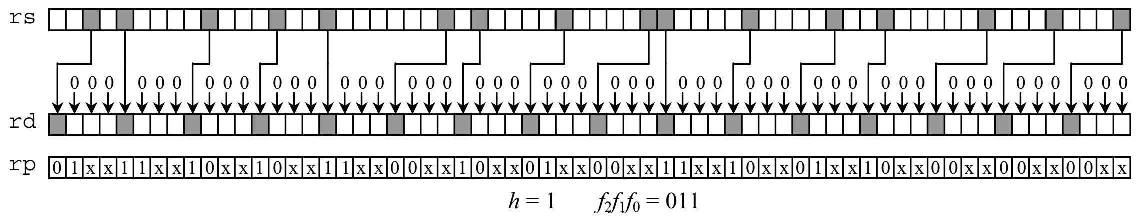

rs
rd
rp
0 1 x x 1 1 x x 1 0 x x 1 0 x x 1 1 x x 0 0 x x 1 0 x x 0 1 x x 0 0 x x 1 1 x x 1 0 x x 0 1 x x 1 0 x x 0 0 x x 0 0 x x 0 0 x x
0 0 0
$h = 1$ $f_2 f_1 f_0 = 011$

Figure 5.4: Complete example operation `sieve`

mode. For each of the registers, the least significant bit is located on the right end of the box representing the register. The gray blocks in the `rs` and `rd` boxes indicate which bits are selected and the locations where the selected bits are placed, respectively. The 64 1-bit values in the `rp` box specify the contents of the configuration register `rp` required to complete the example `sieve` operation. The right 2-bit halves of each 4-bit subword of `rp` possess values of xx, i.e., "don't care", because h equals 1.

5.4 Applying the Instructions

This section describes how to apply `swperm` and `sieve` to perform permutations and mappings of variable-sized subwords packed in words that are multiples of 64 bits in size.

5.4.1 Mapping 1-bit and 2-bit Subwords

Using `swperm` and `sieve`, we can complete an arbitrary permutation or mapping of 64 1-bit subwords with 11 instructions as shown in Figure 5.5(a). We can perform an arbitrary mapping of 32 2-bit subwords with 5 instructions as shown in Figure 5.5(b). In both cases, we initially store the 64-bit value to be mapped in `r1`; upon completion, `r1` will contain the desired mapped result. For 1-bit subwords, `r5` through `r10` store configuration information for the `swperm` and `sieve` instructions, and we use `r1` through `r4` to

```
swperm        r2,r1,r5
swperm        r3,r1,r6
swperm        r4,r1,r7
swperm        r1,r1,r8
sieve,0,000   r1,r1,r9
sieve,1,001   r2,r2,r9
sieve,0,010   r3,r3,r10
sieve,1,011   r4,r4,r10
xor           r1,r1,r2
xor           r3,r3,r4
xor           r1,r1,r3
```

(a)

```
swperm        r2,r1,r3
swperm        r1,r1,r4
sieve,0,100   r1,r1,r5
sieve,1,110   r2,r2,r5
```

(b)

Figure 5.5: Assembly code for performing 64-bit mappings (a) for 1-bit subwords and (b) for 2-bit subwords

store intermediate values. For 2-bit subwords, `r1` and `r2` store intermediate values, and `r3` through `r5` store configuration information.

To complete a permutation or mapping of 1-bit subwords, we first perform 4 mappings of 4-bit subwords using `swperm`. Upon completion of these 4 instructions, the 4-bit subwords in registers `r1`, `r2`, `r3`, and `r4` will contain the zeroth, first, second, and third bits of the corresponding subwords of the desired mapped result, respectively. For example, after execution of the first `swperm` instruction, one of the four bits contained in the ith subword of `r2` will ultimately be placed in bit position 1 of the ith subword of the desired mapped result. Likewise, following the execution of the second `swperm` instruction, one of the four bits stored in the ith subword of `r3` will eventually be placed in bit position 2 of the ith subword of the desired mapped result.

The four `sieve` instructions (in 1-bit mode) move 1 bit from every 4-bit subword of `r1` through `r4` to either the zeroth, first, second or third bit positions of the corresponding subwords in the destination registers. Upon completion of the `sieve` instructions, the desired mapped result is distributed across four 64-bit registers. The 16 bits in the zeroth position of each 4-bit subword in `r1` are the bits that belong in the zeroth position of each subword in the desired result. We set the remaining 48 bits of `r1` to zeroes with the first `sieve` instruction. Similarly, the bits located in the first positions of the 4-bit `r2` subwords, the second positions of the 4-bit `r3` subwords, and the third positions of the 4-bit `r4` subwords belong in the first, second, and third positions of the corresponding subwords of the desired mapped result, respectively. The `sieve` instructions set the bits in `r1`, `r2`, `r3` , and `r4` that do not correspond to bits of the desired result to zeroes. We collect the results of the 4 `sieve` instructions into a single register by performing 3 bitwise XOR (or bitwise OR) operations. Following the completion of the `xor` instructions, `r1` will contain the 64-bit mapped result.

To permute or map 32 2-bit subwords packed into a 64-bit register, we use the same method but fewer instructions, as shown in Figure 5.5(b). The last two rows in Figure 5.3 show how the 64-bits of the desired mapped result are distributed over the two registers `r2` and `r1` after the `sieve` instructions complete. We can combine these two registers into the final 64-bit mapped result by performing a single `xor` (or a single `or`) instruction.

We assume that the registers used to store configuration information are loaded with the appropriate data prior to the execution of these code segments. This pre-loading may require 6 or 3 memory load instructions for mappings of 1-bit or 2-bit subwords, respectively. Cryptographic algorithms often employ the same fixed mapping in every encryption or hash round, however. One can usually perform a round without spilling any registers to memory, so one could load the 6 or 3 mapping values into general-purpose registers once before the execution of thousands of rounds required to encipher or hash kilobytes of data. As a result, the amortized cost of the loads would be negligible. Alternatively, these configuration registers may be intermediate encryption or hash results; therefore, no memory loads would be required.

5.4.2 Mapping 4-bit or Larger Subwords

We can perform a permutation or mapping of 4-bit or larger subwords using a single `swperm` instruction. An example of a 64-bit mapping of 4-bit subwords is illustrated in Figure 5.1. Given a register `r1` that stores a 64-bit value to be mapped and a 64-bit register `r2` that contains the configuration information necessary to conduct the mapping, the following instruction completes a mapping of 4-bit subwords in a single cycle:

```
swperm r1,r1,r2
```

The `swperm` instruction stores the desired mapped result in `r1`. One can also complete 64-bit permutations or mappings of 8-bit, 16-bit, and 32-bit subwords by executing a single `swperm` instruction. We can divide 8-bit or larger subwords into 4-bit subwords, and it is easy to translate a mapping encoding for 8-bit or larger subwords into a mapping encoding usable by `swperm` for 4-bit subwords.

5.4.3 Generating the Configuration Information

We describe an efficient and simple algorithm that runs in $O(n)$ time, where n is the number of bits in a register, which produces the configuration information necessary to complete an arbitrary 64-bit mapping. Choosing the appropriate instructions to use, as described above, is a trivial operation that only depends on the subword size. Generating the configuration registers for these instructions is a more complicated process, however. We present source code that produces the mapping configuration information when provided with a simple description of the desired mapping.

The C function `GenMapInfo`, displayed in Figure 5.6, generates the `rp` values for the `sieve` and `swperm` instructions involved in a 64-bit mapping. In the figure, `i64` is a type declaration for a 64-bit unsigned integer (i.e., `unsigned long long`). The function accepts three inputs: `sigma`, `sigma_size`, and `inverse`. The input `sigma` is an array of integers with `sigma_size` elements; `sigma_size` must be a power of

2 and less than or equal to 64, as the function computes the configuration information necessary to complete a permutation of at most 64 elements. The contents of the array, which are specified by the programmer, represent the desired mapping of a 64-bit register. The array element `sigma[i]` indicates which subword from the source register should be directed to the `i`th subword in the mapped result. The number of subwords therefore equals `sigma_size`, and the size of each subword (in bits) equals 64 divided by `sigma_size`.

In some situations, it may be desirable to generate configuration information required by `swperm` and `sieve` to perform the inverse of a given mapping. If the value of `inverse` is 1, and the mapping specified by `sigma[]` is a bijection (and therefore is a permutation and is invertible), then `GenMapInfo` produces the configuration information required to conduct the inverse of the permutation specified by `sigma[]`. The algorithm generates this information by first quickly computing the inverse of the provided permutation. Then, `GenMapInfo` produces configuration information for the inverse permutation by performing the same procedure used to generate configuration information for a regular (i.e., non-inverted) mapping.

`GenMapInfo` outputs two integer arrays: `swperm_rp[]` and `sieve_rp[]`. Upon completion of the `GenMapInfo` routine, these two arrays contain the appropriate values of the `rp` registers required by the `sieve` and `swperm` instruction(s) to complete the desired mapping. The algorithm operates by simply extracting bits from the elements of `sigma[]` and placing them in pre-specified destination locations in `swperm_rp[]` and `sieve_rp[]`. For instance, suppose we wish to generate configuration information for a mapping of 64 1-bit subwords. In this case, `sigma_size` equals 64. Each element of `sigma[]` is an integer between 0 and 63, inclusive, so we require six bits to encode each element. `GenMapInfo` extracts the two least significant bits from all of the elements of `sigma[]` and writes those bits to appropriate locations in `sieve_rp[0]`and `sieve_rp[1]`. The algorithm also extracts the four most significant bits from each 6-bit element of `sigma[]` and places those bits in certain locations in the elements of

```
void GenPermInfo (i64 sigma[], i64 sigma_size,
                  i64 swperm_rp[], i64 sieve_rp[],
                  i64 inverse) {

  i64 j,k,limit,subword_size,sigma2[64];
  subword_size = 64/sigma_size;

  /* Initialize configuration register values */
  for (j=0;j<4;j++) swperm_rp[j]=sieve_rp[j>>1]=0;

  /* Produce inverse permutation */
  if (inverse==1) {
    for (j=0;j<sigma_size;j++) sigma2[sigma[j]]=j;
    sigma = sigma2; }

  /* Now generate configuration register values
     for up to 4 swperm and 4 sieve instructions */

  /* For mappings of 64 1-bit subwords */
  if (subword_size == 1) {
      for (j=0;j<64;j++) {
      swperm_rp[j&0x3]      |= (sigma[j]>>2)
        << (j&0x3C);
      sieve_rp[(j>>1)&0x1] |= (sigma[j]&0x3)
        << ((j&0x3C)+((j&0x1)<<1)); } }

  /* For mappings of 32 2-bit subwords */
  else if (subword_size == 2) {
    for (j=0;j<32;j++) {
      swperm_rp[j&0x1] |= (sigma[j]>>1)
        << ((j&0x1E)<<1);
      sieve_rp[0]      |= (sigma[j]&1)
        << (1+(j<<1)); } }

  /* For mappings of sixteen 4-bit, eight 8-bit,
     four 16-bit, or two 32-bit subwords */
  else /* (subword_size >= 4) */ {
    limit = subword_size/4;
    for (j=0;j<sigma_size;j++)
      for (k=0;k<limit;k++)
        swperm_rp[0] |= (sigma[j]*limit+k)
          << ((j*limit+k)<<2); } }
```

Figure 5.6: C source code for configuration data generation

`swperm_rp[]`. Observe that `sigma_size` and `inverse` are the only input variables upon which the destination locations of the bits extracted from `sigma[]` depend.

Performing a (bijective) permutation using `swperm` and `sieve` requires the same number of instructions as completing its inverse using these instructions. Therefore, the size of the `GenMapInfo` output is independent of the value of `inverse`. For 64-bit mappings of 1-bit subwords, `GenMapInfo` outputs six 64-bit `rp` values for 4 `swperm` and 4 `sieve` instructions. For 2-bit subwords, `GenMapInfo` outputs three 64-bit `rp` values for 2 `swperm` and 2 `sieve` instructions. In addition, if the subword size is 4 bits or greater, the function generates a single 64-bit `rp` value for a single `swperm` instruction.

Inspection of the function reveals that the maximum number of steps is a constant multiplied by the number of bits in a register, n. Hence, the running time of the algorithm is $O(n)$. For the `sieve` and `swperm` instructions presented in this chapter, $n = 64$.

5.4.4 Mappings in Large Values

The techniques presented above involve arbitrary permutations and non-bijective mappings of a 64-bit word with 1-bit or larger subwords. It may be desirable, however, to complete an arbitrary mapping of 128-bit, 256-bit, or larger words that are distributed across multiple 64-bit registers. We describe a method of applying the `swperm` and `sieve` instructions to complete such arbitrary mappings of large words. Let m be the size of the large word in bits. Let m be a multiple of 64, and $x = m/64$. Therefore, we have x 64-bit blocks in the initial large word and x 64-bit blocks in the destination (mapped) large word. Each of the x blocks of the initial word can contribute 0 to 64 bits to each of the x blocks of the destination word.

We perform a large word mapping as follows. For each block in the initial word, we perform x 64-bit mappings, which is one mapping for each block in the destination word. After each 64-bit mapping, we perform a bitwise AND operation on the 64-bit mapped result and a 64-bit mask. The mask corresponding to a particular 64-bit initial block and 64-bit destination block pair contains a 1 in bit position i if and only if a bit from the initial block should be mapped to bit position i in the destination block. Since we require one mask for each 64-bit initial block and 64-bit destination block pair, at most x^2 unique masks will be required to conduct the mapping. Upon completing all of the 64-bit mappings and masking operations for a single 64-bit destination block, the x 64-bit results are collected into a 64-bit destination block by performing $(x - 1)$ bitwise XOR operations. By making only minor changes to the configuration information generation code presented in Figure 5.6, it is possible to efficiently and dynamically generate both the configuration registers and the masks necessary to conduct a mapping of a large word. Such changes to the code would increase the computational complexity of the configuration generation by at most a factor of $O(x^2) = O(m^2/4096)$.

We present a block diagram that conceptually illustrates the operations needed to complete an arbitrary mapping of a 128-bit word in Figure 5.7. In the figure, the 64-bit Map objects include the instructions required to complete an arbitrary mapping of a 64-bit word. Depending on the size of the subwords, this code sequence may consist of 11, 5, or 1 instruction(s), as described above. We assume that the initial large word, the masks, and the configuration information for the `sieve` and `swperm` instructions have been previously loaded into registers. Hence, if y is the number of instructions needed to perform an arbitrary mapping of a 64-bit word, the total number of instructions required to complete a mapping of a 128-bit word is $4y + 6$. In general, for a large word of size $m \geq 128$, the maximum number of instructions required to complete an arbitrary mapping is $x^2(y+1)+x(x-1) = yx^2+2x^2-x$. When using 4-bit subwords, mappings of 128-bit and 256-bit words require at most 10 instructions and 44 instructions, respectively. To map 128 1-bit subwords stored in two 64-bit registers, we require 50 instructions.

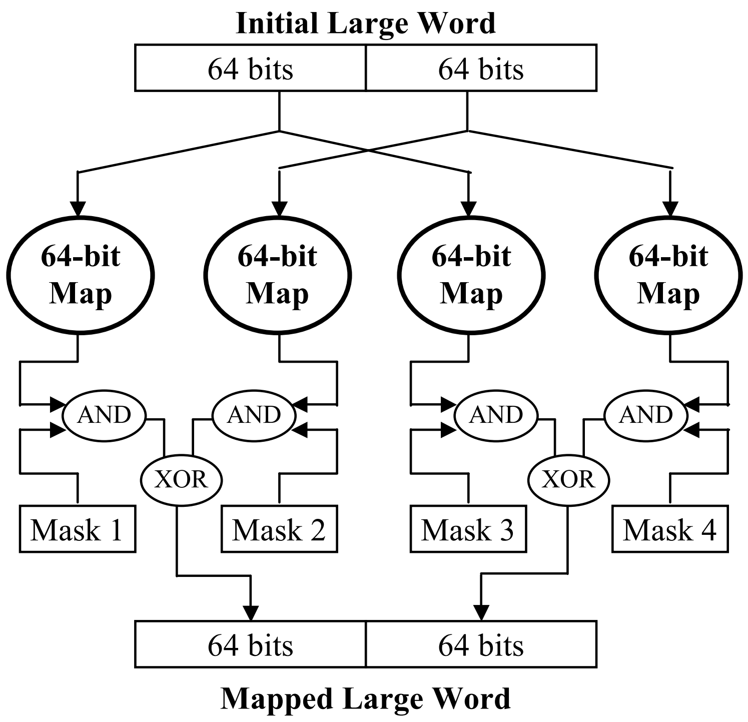


Figure 5.7: Mapping of a 128-bit word

5.5 Hardware Implementation

We now describe the CMOS hardware implementation for the `swperm` and `sieve` instructions. First, we present the Selection Unit, which enables the execution of the `swperm` instruction. We can implement the Selection Unit by building a 4-bit 16-to-1 multiplexer for every 4-bit subword in `rd`. Such a design is extremely expensive in hardware, however. Using a reduced crossbar, we can greatly decrease the transistor and wire cost. The reduced crossbar only requires one decoder for every 16 intersections between `rs` and `rd` tracks as opposed to one decoder for each intersection in a full crossbar.

A high-level representation of the reduced crossbar is illustrated in Figure 5.8(a). s_i is the ith 4-bit subword of `rs`, d_j is the jth 4-bit subword of `rd`, and p_j is the jth 4-bit subword of `rp`. A rectangle represents a single cell, and we present an example cell in Figure 5.8(b). Each cell consists of a 4-input AND gate, 4 n-type transistors, and 0, 1, 2, 3 or 4 inverters. Recall that the `swperm` instruction directs the s_i to d_j if and only if p_j equals i. In the example cell, the leftmost and bottommost wires are the most significant bits of the subwords. From inspecting the negation bubbles on the inputs to the AND gate, we know that i = 5 in Figure 5.8(b). Hence, if p_j = 5, only the fifth 4-bit subword, s_5, is enabled onto d_j. The other fifteen 4-bit subwords from `rs` similarly connected to d_j are not enabled onto d_j when p_j equals 5.

We now discuss the hardware cost of this implementation in terms of transistor and track counts. Since we need 16 cells for each of the sixteen 4-bit subwords of `rd`, the total number of cells in the reduced crossbar is 16 · 16 = 256. On average, there are 2 negation bubbles on the inputs to the AND gate per cell, so the average number of transistors per cell is 16. These 16 transistors include 8 transistors to implement a 4-input AND gate, 4 transistors to implement 2 inverters, and 4 n-type transistors controlled by the output of the AND gate. The reduced crossbar consists of 256 cells, so the total transistor count is 4096. Note that this count does not include any buffers that we may potentially need to drive the

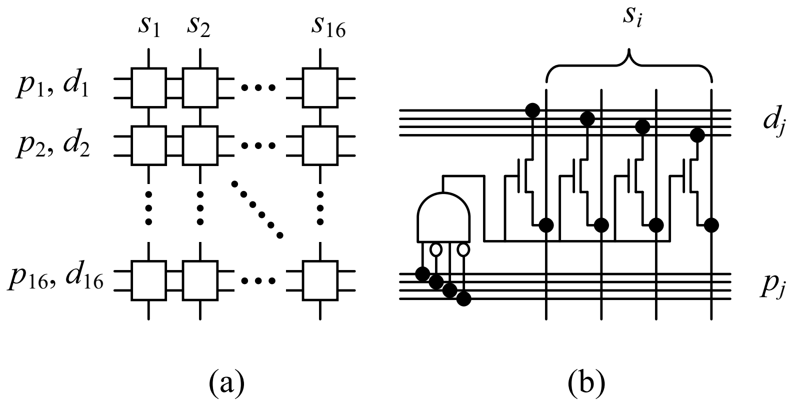
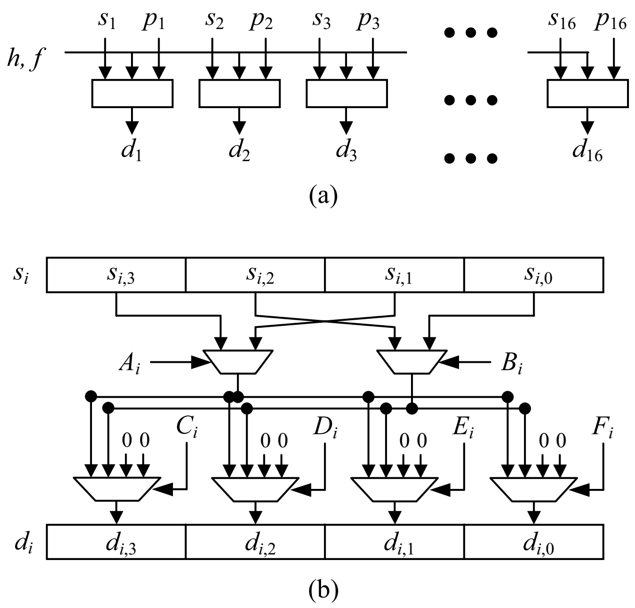


Figure 5.8: The hardware implementation of the Selection Unit: (a) the high-level organization and (b) an example of a cell

long wires.

We define a track to be a wire routing lane that is reserved for connections between different cells. The number of vertical tracks is roughly the number of bits in `rs`, 64, and the number of horizontal tracks is the number of bits in `rd` plus the number of bits in `rp`, 128. The critical path latency of this circuit is the time needed for a signal to traverse two long wires (that each span the width of sixteen Selection Unit cells) plus the logic delay through a single Selection Unit cell. This is at most the sum of the propagation delays of two long wires, a 4-input AND gate, an inverter, and an n-type transistor. Assuming the delays through the wires are not extremely high, the Selection Unit can complete a `swperm` instruction in a single cycle. In a deeply pipelined processor, however, the propagation delays through wires could force multiple-cycle execution of `swperm` instructions.

We present a block diagram of the Filter Unit, which supports the `sieve` instruction, in Figure 5.9(a). Each rectangle represents a single 4-bit slice, and we can implement a 4-bit slice with four 1-bit 5-to-1 multiplexers. Each of these multiplexers simply select the bit value “0” or one of four input bits from a 4-bit subword of `rs`; the multiplexer output



s_1 p_1 s_2 p_2 s_3 p_3 s_{16} p_{16}
h, f
d_1 d_2 d_3 d_{16}
(a)
s_i $s_{i,3}$ $s_{i,2}$ $s_{i,1}$ $s_{i,0}$
A_i B_i
C_i D_i E_i F_i
0 0
d_i $d_{i,3}$ $d_{i,2}$ $d_{i,1}$ $d_{i,0}$
(b)

Figure 5.9: Hardware implementation of the Filter Unit: (a) High-level organization and (b) Structure of a 4-bit slice

is directed to a single bit in the corresponding 4-bit subword of `rd`. Using the 4-bit slice structure illustrated in Figure 5.9(b), however, we can reduce the transistor count without increasing the critical path latency by eliminating redundant logic operations. We replicate the slice shown in Figure 5.9(b) sixteen times, once for each 4-bit subword in `rd`. The variable $s_{i,j}$ represents the jth bit of the ith subword of `rs`; the variable $d_{i,j}$ represents the jth bit of the ith subword of `rd`. Each 4-bit slice requires two 1-bit 2-to-1 multiplexers and four 1-bit 4-to-1 multiplexers. In addition, the ith subword slice includes a set of signals to control these multiplexers: A_i, B_i, C_i, D_i, E_i, and F_i. We define these signals in Figure 5.10, where p_k is the kth bit of `rp`, and h, f_2, f_1, and f_0 are function code bits.

We can implement a 2-to-1 multiplexer using 4 transistors, and we can implement a 4-to-1 multiplexer using only 7 transistors each since the two lowest bit inputs are always zeroes. Using buffers to reduce the fan-out of the function code bits and logic optimization

$$\boldsymbol{A_i} = \boldsymbol{B_i} = (h \cdot p_{4i+3}) + (\neg h \cdot p_{4i+1})$$
$$\boldsymbol{C_{i,1}} = f_1 \cdot (f_2 + f_0) \qquad \boldsymbol{D_{i,1}} = f_1 \cdot (f_2 + \neg f_0)$$
$$\boldsymbol{E_{i,1}} = \neg f_1 \cdot (f_2 + f_0) \qquad \boldsymbol{F_{i,1}} = \neg f_1 \cdot (f_2 + \neg f_0)$$
$$\boldsymbol{C_{i,0}} = \boldsymbol{E_{i,0}} = (\neg f_2 \cdot ((h \cdot p_{4i+2}) + (\neg h \cdot p_{4i}))) + f_2$$
$$\boldsymbol{D_{i,0}} = \boldsymbol{F_{i,0}} = (\neg f_2 \cdot ((h \cdot p_{4i+2}) + (\neg h \cdot p_{4i})))$$

Figure 5.10: Control signals in the Filter Unit

techniques to reduce the transistor count, each 4-bit subword slice requires 116 transistors. The total number of transistors required for the sixteen 4-bit subword slices of the Filter Unit is 1856. Nearly all of the data and control for each 4-bit subword slice in the Filter Unit is local, so we do not require many long vertical or horizontal tracks. We only need 4 horizontal tracks for the 4 `sieve` function code bits. The critical path latency in the Filter Unit is at most the sum of the propagation delays through a horizontal wire (that spans the width of sixteen 4-bit slices), a 2-to-1 multiplexer, a 4-to-1 multiplexer, and the logic required to compute A_i. Therefore, it is highly likely that the Filter Unit can complete the execution of a `sieve` instruction in a single cycle.

The total number of transistors needed to implement a Mapping Unit, which consists of a Selection Unit and a Filter Unit, is 5952. This transistor count is of the same order of magnitude as that required to construct a simple 64-bit CMOS ripple-carry adder [171]. We compare the hardware cost of the Mapping Unit to past work in Table 5.2. Due to the imprecision of the track metric, we compare numbers of tracks using *O*-notation in terms of the number of bits in a register, n. When considering both transistor count and wire area, it appears that the Mapping Unit is nearly as efficient as a VLSI implementation of the `omflip` instruction [173]. The Mapping Unit requires nearly twice as many transistors as an `omflip` implementation, but it potentially consumes only half as much wire area due to constants hidden by the *O*-notation. The Mapping Unit also requires significantly fewer transistors and tracks than a crossbar network [173].

Table 5.2: Hardware cost comparison

Implementation	**Horizontal tracks**	**Vertical tracks**	**Transistor count**
Mapping Unit (`swperm/sieve`)	$O(n)$	$O(n)$	$\sim$6000
Omega flip network (`omflip`)	$O(n)$	$O(n)$	$\sim$3100
Crossbar network	$O(n)$	$O(n \log n)$	$>$ 73,728

5.6 Performance Analysis

This section investigates the performance improvement provided by `sieve` and `swperm` for general subword mappings and for the popular DES encryption algorithm.

5.6.1 Impact on 64-bit Permutations and Mappings

Table 5.3 summarizes the number of instructions, cycles and registers required to complete arbitrary mappings of different-sized subwords packed into a 64-bit register. For subword sizes of four bits or larger, we only need 1 `swperm` instruction and 2 registers to complete an arbitrary 64-bit mapping. Using both `sieve` and `swperm`, arbitrary 64-bit mappings of 2-bit and 1-bit subwords require 5 and 11 instructions, respectively. In past work, Yang and Lee demonstrated that the `omflip` instruction could be used to complete 64-bit bijective mappings using 5 and 6 instructions, respectively [173]. These `omflip` instruction sequences must be executed serially, however. Therefore, even on an ultra-wide superscalar processor, a 64-bit mapping of 1-bit subwords requires 6 cycles using `omflip` instructions.

Superscalar execution can accelerate mappings that employ `sieve` and `swperm`, however. True data dependencies do not exist between any of the `swperm` instructions or between any of the `sieve` instructions listed in Figure 5.5. Hence, a multiple-issue processor

Table 5.3: Performance of 64-bit mappings using `sieve` and `swperm`

Subword size	**Max. # of instructions**	**Minimum # of cycles**			**Max. # of registers**
		1-issue	**2-issue**	**4-issue**	
32 bits	1	1	1	1	2
16 bits	1	1	1	1	2
8 bits	1	1	1	1	2
4 bits	1	1	1	1	2
2 bits	5	5	3	3	5
1 bit	11	11	6	4	10

can improve the performance of an arbitrary 64-bit mapping that employs the proposed instructions by executing certain instructions in parallel. On a 4-way superscalar processor, assuming there are four Mapping Units available, we can complete mappings of 1-bit and 2-bit subwords in as few as 4 and 3 cycles, respectively. For 1-bit subwords, the 4 `swperm` instructions can be executed in parallel in a single cycle, and the 4 `sieve` instructions can be executed in parallel in the following cycle. The 3 `xor` instructions must be executed in 2 cycles following the completion of the `sieve` instructions due to data dependencies.

We compare the performance of `sieve` and `swperm` to past work in Tables 5.4 and 5.5. We consider only past work that relies on the same design assumptions as those of `sieve` and `swperm`. That is, the tables only consider past work that (i) assumes a single instruction may only specify a single destination register and two source registers, (ii) assumes that the processor cannot store any additional state (that could be subject to OS context switching) to support the implementation of the new instructions, and (iii) assumes that the new independent instructions cannot be bundled into a single instruction during execution by the microarchitecture. We note that if we were to relax or change any of these assumptions, `sieve` and `swperm` could certainly be enhanced to provide higher performance than is listed in the tables below.

Table 5.4 and Table 5.5 list the number of instructions and cycles, respectively, required

Table 5.4: Instruction count comparison for subword mappings

Instruction(s) used to perform a 64-bit mapping	**Non-bijective support**	**Max. # of instructions needed to map subwords of bit size:**					
		32	**16**	**8**	**4**	**2**	**1**
`sieve`/`swperm`	Yes	1	1	1	1	5	11
`pperm` and `xbox`	Yes	15	15	15	15	15	15
`omflip`, `cross` and `grp`	No	1	2	3	4	5	6
Existing ISAs	Yes	1	1	≥ 1	23	23	23

Table 5.5: Cycle count comparison for subword mappings

Instruction(s) used to perform a 64-bit mapping	**Non-bijective support**	**Max. # of instructions needed to map subwords of bit size:**					
		32	**16**	**8**	**4**	**2**	**1**
`sieve`/`swperm`	Yes	1	1	1	1	3	4
`pperm` and `xbox`	Yes	5	5	5	5	5	5
`omflip`, `cross` and `grp`	No	1	2	3	4	5	6
Existing ISAs	Yes	1	1	≥ 1	10	10	10

by the different methods to complete a 64-bit mapping and to write the mapped result to a single 64-bit register. The bit values in the heading of the table indicate the size of the subwords to be permuted and/or mapped within a 64-bit word. We determine the cycle counts using a simulation of a 4-way superscalar processor with four integer execution units and a single load/store unit.

The Existing ISAs row indicates the minimum number of instructions in conventional ISAs required to perform a 64-bit permutation or non-bijective mapping using eight lookup tables or existing subword permutation and mapping instructions for multimedia acceleration (see Section 5.2). Note that instructions in existing ISAs that map 8-bit subwords are generally limited to performing a small set of predefined mappings. Also, the instruction counts listed for mappings of 4-bit and smaller subwords using existing ISAs are only achievable if the mapping is statically encoded in lookup tables. The cycle counts listed in Table 5.5 were obtained using a perfect data cache model with a single-cycle access latency. If the data cache were small or initially cold, however, table lookup operations could require many additional cycles to complete due to cache misses. Hence, the cycle counts in the Existing ISAs row could be much larger in certain scenarios.

Other than `sieve/swperm`, only the `pperm` instruction can efficiently complete both bit-level bijective and non-bijective mappings. The `cross`, `omflip`, and `grp` instructions only perform (bijective) permutations. However, `cross`, `omflip`, and `grp` can be applied to any register size n that is a power of 2, but `swperm` and `sieve` are only defined for n = 64.

For 64-bit permutations or non-bijective mappings, we observe that `sieve` and `swperm` perform as well as or better than all previously proposed instructions and existing ISAs with the exception of the number of instructions required to complete a 64-bit mapping using 1-bit subwords. Note that the performance improvement provided by `sieve` and `swperm` over existing methods on 2-way and 4-way superscalar processors requires two or four Mapping Units, respectively. Methods that employ `cross`, `grp`, and `omflip` only require one permutation functional unit to achieve the cycle counts listed in Table 5.5.

5.6.2 Impact on the Data Encryption Standard

We now demonstrate the degree to which the proposed mapping instructions can improve the performance of a highly popular symmetric-key block cipher, the Data Encryption Standard (DES) [119]. A large number of secure communications, banking, and storage protocols employ DES (and its more secure variant, Triple DES) to provide services such as data confidentiality and data integrity. We begin with an optimized C implementation of the

DES algorithm that is based upon Eric Young's `libdes` [175]. We compile the implementation for the 64-bit Alpha ISA (augmented with the proposed mapping instructions) using `gcc` with the `-O2` optimization flag. To improve the performance of the block cipher, we apply the proposed mapping instructions to four mapping operations within DES: the initial permutation (IP), the final permutation (FP), the P-box permutation (PP), and the compression permutation (CP). The CP is actually a non-bijective mapping, although it is called a permutation by the DES standard. Most software implementations of DES complete these mappings using a series of table lookup operations. We seek to increase performance by replacing these table lookup operations with the proposed mapping instructions.

For processors with small and simple caches, we can achieve a significant speedup for the P-box permutation. In the baseline software implementation, the P-box permutation is built into the lookup tables used to complete operations known as S-box substitutions. Performing the P-box permutation using the proposed instructions allows us to decrease the size of the S-box lookup tables and consequently reduce cache misses. Also, the compression permutation in the round key computation function can consume a large percentage of the total clock cycles involved in a DES operation. By accelerating the compression permutation using the new mapping instructions, we can greatly improve performance in some scenarios, which we describe below. We can also accelerate the performance of the IP and the FP, although these mappings only account for a small percentage of the computation required per DES operation.

We use the SimpleScalar superscalar processor simulator [21] to obtain cycle-accurate performance statistics concerning the execution of DES. We perform simulations for four different processor configurations, which range from a typical embedded processor found in low-power wireless information appliances to a wide superscalar processor used in high-end servers. The four microarchitectural configurations consist of a single-issue processor core with small cache, a 2-issue superscalar processor, a 4-issue superscalar processor, and an 8-issue superscalar processor. For each model, the fetch, decode, and commit widths

Table 5.6: Simulation memory parameters

Parameter	**Processor models**			
	1-issue	**2-issue**	**4-issue**	**8-issue**
L1 data cache	1-way, 2KB	1-way, 8 KB	2-way, 16 KB	2-way, 32 KB
L1 instruction cache	1-way, 2KB	1-way, 8 KB	2-way, 16 KB	2-way, 32 KB
L2 cache	none	2-way, 128 KB	4-way, 256 KB	4-way, 256 KB
L1/L2 latency	1/-	1/20	1/5	1/5
Memory latency	50	50	100	100
Memory ports	1	1	2	2
L/S queue size	4	16	32	64

equal the issue width. Also, the number of ALUs equals the issue width, and we assume that each ALU contains a Mapping Unit. In Table 5.6, we summarize the memory system parameters used in the SimpleScalar simulations. The L2 latency for the 2-way processor is larger than those of the 4-way and 8-way processors because we model the 2-way superscalar's L2 cache as being off-die (similar to the off-die L2 caches of 2-way superscalar processors such as the Pentium II). Also, the main memory latencies for the 1-way and 2-way processors are smaller than those of the 4-way and 8-way processors because more aggressive microarchitectures are often clocked faster relative to main memory than less aggressive microarchitectures.

Rather than modify the C compiler to identify and utilize the mapping instructions, we strategically insert standard RISC integer ALU instructions that represent the mapping instructions into the DES source code. The DES implementation that uses these special integer ALU instructions maintains the same instruction-level control dependence and data dependence structure as that of a DES implementation that employs the proposed mapping instructions. We carefully choose the special ALU instructions such that the compiler does not eliminate or combine any of those instructions during code optimization. In addition, we modify the SimpleScalar simulator to recognize the special integer ALU instructions and treat them as mapping instructions.

We obtain performance data by simulating the execution of DES for 8-kilobyte input data blocks after allowing the caches sufficient time to warm up. The input size is not a critical simulation parameter, however; we find that once the caches are warm, the performance speedup results are independent of the input data size. The speedups achieved by the proposed mapping instructions are presented in Figure 5.11. The graph illustrates the speedups associated with each processor configuration for various frequencies of re-keying events that would require computation of new round keys. We express the re-keying frequency as the number of input bytes per re-keying event, i.e., the number of input bytes per round key computation. For example, data points associated with the number 32 on the horizontal axis of the figure corresponds to 8-kilobyte inputs in which the round key computation is performed one time for each 32-byte block of the input. We use the variable Z to represent the number of input bytes per round key computation.

Although the simulation results are independent of the total input size, the results are heavily dependent on the value of Z. The round key computation must be performed at least once for each unique key used to complete DES operations. When DES is used for encryption, a single key is often employed to encrypt all input data. As a result, we only need to perform the round key computation once during the encryption of an entire input block. Z is therefore equal to the total input size in this case. However, when DES (or any other block cipher) is used to implement a cryptographic hash function for digital signature and data integrity operations, a different key is often employed for each 8-byte input block, for the key is a function of the 8-byte input block [130, 172]. Hence, we must perform the round key computation once for every 8 bytes of input, so $Z = 8$.

Figure 5.11 displays speedup results when we complete the IP, FP and CP (but not the PP) using the new mapping instructions. The speedup results for this case are also

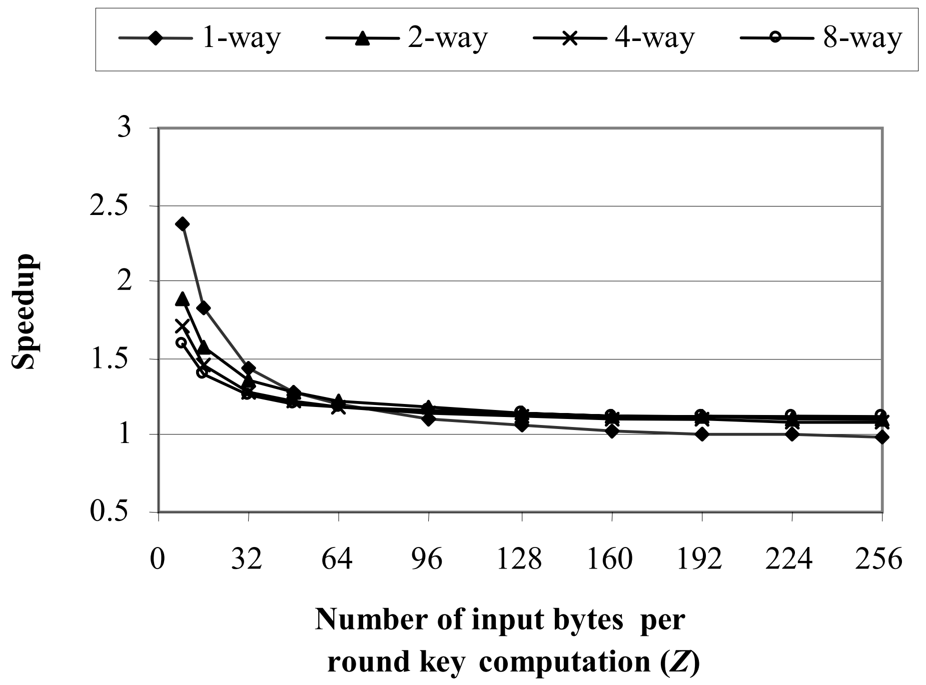


Figure 5.11: DES speedups for IP/FP/CP optimization

Table 5.7: DES speedups for IP/FP/CP optimization

Z	**DES Speedup**			
	1-way superscalar	**2-way superscalar**	**4-way superscalar**	**8-way superscalar**
8	2.370	1.885	1.711	1.587
16	1.827	1.581	1.455	1.401
32	1.433	1.363	1.280	1.264
48	1.283	1.275	1.213	1.211
64	1.196	1.226	1.176	1.183
96	1.108	1.175	1.137	1.155
128	1.057	1.148	1.117	1.140
160	1.036	1.131	1.104	1.131
192	1.014	1.120	1.096	1.125
224	1.004	1.112	1.090	1.121
256	0.994	1.105	1.085	1.117

summarized in Table 5.7. We attain speedups of 2.37 and 1.71 when Z = 8 on a single-issue processor and a 4-way superscalar processor, respectively. As the number of input bytes per round key computation increases, the speedups decrease to 1.11 or less, however. This occurs because the relative computational cost of the CP decreases as the number of input bytes per round key computation increases. Consequently, the CP performance acceleration caused by the mapping instructions becomes less significant.

We obtain different results when we complete all DES permutations of interest, i.e., the IP, FP, CP, and PP, using `sieve` and `swperm`. The speedup results for this case are summarized in Table 5.8. For a single-issue processor with a small cache, we achieve a speedup of 3.71 when Z = 8. As the number of input bytes per round key computation increases, the speedup falls to 1.57. The single-issue processor experiences a much larger performance improvement for all values of Z in the IP/FP/CP/PP optimization case than in the IP/FP/CP case due to memory system behavior. When the PP is built into the S-box lookup tables, the amount of memory required to store the tables and intermediate values

Table 5.8: DES speedups for IP/FP/CP/PP optimization

Z	**DES Speedup**			
	1-way superscalar	**2-way superscalar**	**4-way superscalar**	**8-way superscalar**
8	3.707	1.669	1.527	1.402
16	2.854	1.357	1.266	1.190
32	2.227	1.144	1.093	1.055
48	1.982	1.061	1.026	1.003
64	1.830	1.015	0.990	0.977
96	1.722	0.967	0.955	0.950
128	1.664	0.943	0.937	0.937
160	1.628	0.927	0.925	0.928
192	1.604	0.917	0.917	0.923
224	1.586	0.910	0.912	0.919
256	1.573	0.904	0.908	0.916

exceeds the size of the single-issue processor's data cache. As a result, performance suffers due to frequent cache misses. By implementing the PP using the proposed mapping instructions, the number of cache misses experienced by the single-issue processor is greatly reduced, and therefore performance is significantly enhanced.

The wider processors do not suffer many cache misses because their caches easily accommodate the S-box lookup tables. Consequently, reducing the size of the lookup tables by implementing the PP with mapping instructions actually degrades performance for any value of Z greater than 64 on the superscalar processors. This results from the fact that the PP optimization increases the dynamic instruction count, for the S-box table lookups must be performed regardless of whether we incorporate the PP into the S-box tables or we implement the PP using the new mapping instructions. Thus, we achieve the highest performance for 2-way and wider processors when we only use the permutation instructions to implement the IP, FP, and CP.

We conclude that we should always employ the proposed mapping instructions to perform the IP, FP and CP in software implementations of DES. When using DES as a cryptographic hash function, the performance impact of the proposed mapping instructions is substantial: we obtain speedups ranging from 1.59 to 2.37. Software implementations of DES should only use 1-bit mapping instructions to perform the PP if the target processor contains an extremely small or non-existent cache, however. This is often true for processors found in smart cards and wireless information appliances. Such processors containing a Mapping Unit could achieve large speedups for DES encryption without incurring the cost and power consumption associated with the extra memory required by table lookup schemes.

5.7 Summary

This chapter examines architectural techniques for improving the performance of cryptographic software. As the popularity of cryptographically-enabled secure systems grows, cryptographic processing consumes an increasingly larger percentage of processor workloads. Hence, in order to avoid significant system performance degradation, cryptographic algorithms such as bulk encryption should be accelerated. Many popular cryptographic procedures such as the DES encryption algorithm perform permutations and non-bijective mappings of subwords packed in registers to achieve desirable security properties. Existing general-purpose processors do not efficiently support such operations, however.

This chapter proposes two 64-bit processor instructions for accelerating the performance of subword permutations and non-bijective mappings: `swperm` and `sieve`. In addition to defining the operation of the instructions, the chapter describes associated software routines and efficient hardware implementations for the instructions. Using the enhancements, we can complete 64-bit mappings of 4-bit or larger subwords in 1 instruction.

In addition, we can perform 64-bit mappings of 1-bit and 2-bit subwords using 11 instructions and 5 instructions, respectively. These instructions are highly parallelizable, and a 4-way superscalar processor can execute these two instruction sequences in 4 cycles and 3 cycles, respectively. This improves upon previous results by requiring fewer instructions to map or permute 4-bit or larger subwords packed in a 64-bit register and fewer execution cycles for 1-bit subwords on wide superscalar processors.

We also demonstrate that we can accelerate the performance of the popular DES block cipher using the proposed instructions. We obtain significant DES performance improvements in constrained embedded environments, and we achieve significant acceleration when applying DES as a cryptographic hash function on superscalar processors. Furthermore, using these bit-level mapping instructions, cryptographers can design future ciphers and hash algorithms that obtain a desirable level of diffusion more rapidly. As a result, fewer encryption rounds may be required to achieve adequate security, and the throughput of encryption algorithms can be significantly improved [96].

Chapter 6

A Hardware Defense against Buffer Overflows

This chapter presents a hardware-based solution that prevents a common class of buffer overflow attacks in software. This mechanism enables built-in security for software that supplements the cryptographic security and performance enhancement techniques provided by previous chapters.

Buffer overflow vulnerabilities in the memory stack continue to pose serious threats to system security. By exploiting these vulnerabilities, a malicious party can strategically overwrite the return address of a procedure call and obtain control of a system. Following a successful buffer overflow intrusion, the attacker can use the system to infect other systems or expose sensitive information such as cryptographic keys.

We add a Secure Return Address Stack (SRAS) to the processor to provide built-in protection against buffer overflow attacks involving procedure return address corruption. The protection provided by this processor-based defense is transparent and dynamic, and it does not require any effort by users or application programmers. Thus, this defense can mitigate common vulnerabilities in cryptographic, application, and operating system software. The contributions of this chapter are based in part on the work previously published by the author in [93, 94, 103].

This chapter is organized as follows. Section 6.1 defines buffer overflow vulnerabilities and attacks. Section 6.2 discusses previous approaches to solving the problem. Section 6.3 describes a Secure Return Address Stack. This section discusses hardware return address stacks in existing processors and presents architectural and OS changes that achieve strong protection. Section 6.4 considers methods for handling abnormal procedure control flow. Section 6.5 analyzes the security improvement provided by the proposed enhancements. Section 6.6 investigates the performance impact of this proposal. Section 6.7 compares the benefits of the SRAS to the features of previously proposed software-based solutions. Section 6.8 summarizes this chapter.

6.1 Buffer Overflows and Return Address Corruption

Buffer overflows have caused security problems for decades. In 1988, the Morris Worm infected a significant percentage of Internet-connected computers using a buffer overflow attack as a method of intrusion. The Code Red worm and its variants, which peaked during the summer of 2001, exemplify the severity of problems that buffer overflow vulnerabilities continue to cause. Code Red spread by taking advantage of a buffer overflow problem in Microsoft IIS. The total economic cost of Code Red is estimated to be $2.6 billion [116]. In addition, various intrusion tools that establish distributed denial of service (DDoS) networks often exploit buffer overflow vulnerabilities to compromise oblivious hosts [65].

Despite existing countermeasures, buffer overflow vulnerabilities continue to plague computer systems and networks. Table 6.1 shows the percentages of CERT Advisories from 1996 to 2003 relating to buffer overflows [32].[1] In 2003, more than 60 percent of CERT advisories involved buffer overflow. Furthermore, buffer overflow weaknesses play a very significant role in the 20 most critical Internet security vulnerabilities identified by

[1]Since 2003, CERT has been using a new system for informing the public of security alerts and vulnerabilities. CERT advisories are now a core component of CERT's Technical Cyber Security Alerts.

Table 6.1: CERT buffer overflow advisories

Year	**Total advisories**	**Advisories involving buffer overflow**	**Percent buffer overflow**
1996	27	5	18.52%
1997	28	15	53.57%
1998	13	7	53.85%
1999	17	8	47.06%
2000	22	2	9.09%
2001	37	19	51.35%
2002	37	22	59.46%
2003	28	17	60.71%

the SANS Institute and the FBI [138].

The majority of buffer overflow attacks involve corruption of procedure return addresses in the memory stack. During the execution of a procedure call instruction, the processor transfers control to code that implements the target procedure. Upon completing the procedure, control is returned to the instruction following the call instruction. This transfer of control occurs in a LIFO (i.e., Last In First Out) fashion, which is also termed stack or properly nested fashion. Thus, a procedure call stack, which is a LIFO data structure, is used to save the state between procedure calls and returns. We describe stack behavior and buffer overflow attacks for the IA-32 architecture [67], but the general procedures apply to all conventional instruction set architectures (ISAs).

Figure 6.1 illustrates the operation of the memory stack for the example program shown in Figure 6.2. The memory stack consists of a set of stack frames; a single frame is allocated for each procedure (e.g., `g`) that has yet to return control to an ancestor procedure (e.g., `f`). The stack pointer (SP) points to the top of the stack frame of the procedure that is currently executing, and the frame pointer (FP) points to the base of the stack frame for the currently executing procedure.

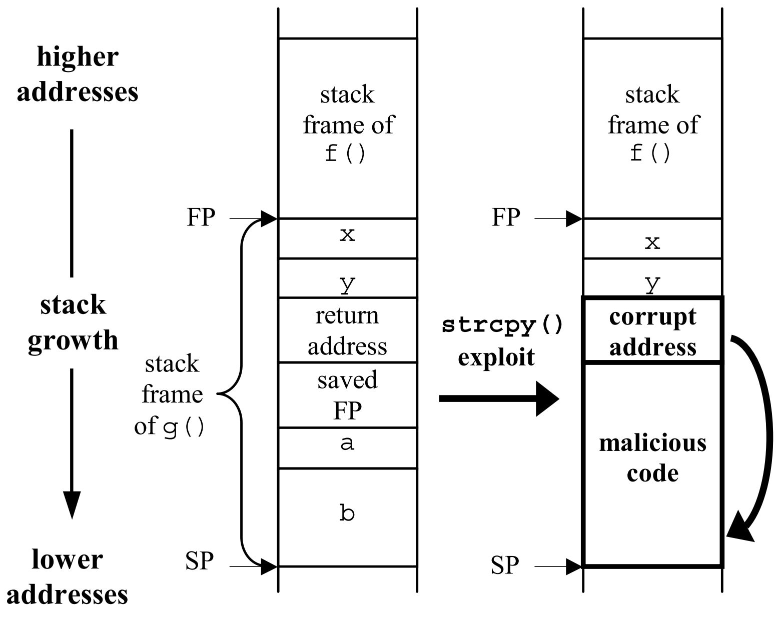


Figure 6.1: Buffer overflow attack

```
int f()
{
    …
    g(x, y);
    …
}

int g(char * x,
      char * y)
{
    int a;
    char b[64];
    …
    strcpy(b, x);
    …
    return;
}
```

Figure 6.2: Vulnerable code example

When function `f()` calls `g()`, a new stack frame is pushed onto the stack. The stack on the left of Figure 6.1 shows the state of the memory stack immediately following the call to `g()`. The new frame includes the input pointers `x` and `y`, the procedure return address, the frame pointer, and the local variables `a` and `b`. Upon completing `g()`, the program should return to the return address stored in `g`'s stack frame; this address should equal the location of the instruction immediately following the call to `g()` in the function `f()`. The SP and the FP should also be restored to their former values, and the stack frame belonging to `g()` should be effectively popped from the stack.

Because the function `strcpy()` does not perform bounds checking, a security vulnerability exists in the code shown in Figure 6.2 if the source of `x` is not guaranteed to limit the buffer size to which `x` points. In the function `g()`, if the string to which `x` points exceeds the size of `b`, `strcpy()` will overwrite data located adjacent to `b` in the memory stack. A malicious party can exploit this situation by strategically constructing a string that contains malicious code and a corrupt return address. If `x` points to such a string, `strcpy()` will copy malicious code into the stack, and the return address in `g()`'s stack frame will be set to the initial instruction of the malicious exploit code. This is illustrated in Figure 6.1. Consequently, once `g()` completes, the program will jump to and execute the exploit code instead of returning control to `f()`. In addition, after the exploit code is executed, the exploit code may return control to `f()` and therefore a user may not become aware that an attack actually occurred. There are many variations of this form of attack [82], but the majority relies on the ability to modify the return address. For example, rather than the attacker injecting his own exploit code, the return address may be modified to point to legitimate, preexisting code that can be used for malicious purposes.

6.2 Past Work

Researchers have proposed many software-based countermeasures for buffer overflow exploits. These methods differ in the strength of protection provided, the effects on performance, and the ease with which they can be effectively employed. One solution is to store the stack in non-executable pages. This can prevent an attacker from executing code injected into the memory stack. However, the return address may instead be redirected to preexisting code in memory that the attacker wishes to run for malevolent reasons. In addition, it is difficult to preserve compatibility with existing applications, compilers, and operating systems that employ executable stacks. For instance, Linux depends on executable stacks for signal handling [37].

StackGuard is a compiler-based solution involving a patch to the C compiler `gcc` that defends against buffer overflow attacks that corrupt procedure return addresses [37]. In the procedure prologue of a called function, a “canary” value is placed on the stack next to the return address, and a copy of the canary is stored in a general-purpose register. In the epilogue, the canary value in memory is compared to the canary register to determine whether a buffer overflow has occurred. The application randomly generates the 32-bit or 64-bit canary values, so the application can detect improper modification of a canary value resulting from a buffer overflow with high probability. However, there exist attacks that can circumvent StackGuard’s canaries to successfully corrupt return addresses and defeat the security of the system [20].

StackGhost employs the SPARC architecture’s register windows to defend against buffer overflow exploits [54]. Return addresses that have stack space allocated in register windows are partially protected from corruption. The OS has the responsibility of spilling and filling register windows to and from memory, and once a return address is stored back in memory, the return address is potentially vulnerable. Various methods of protecting such spilled stacks are defined. Buffer overflow protection without requiring re-compilation of application source code is a benefit of StackGhost, but the technique is only applicable to SPARC systems [154].

Researchers have also proposed using more secure (or safe) dialects of C and C++, for a high percentage of buffer overflow vulnerabilities can be attributed to features of the C programming language. Cyclone is a dialect of C that focuses on general program safety, including the prevention of stack smashing attacks [64]. Safe programming languages have proven to be very effective in practice. While programs written in Cyclone may require less scrupulous checking for certain types of vulnerabilities, the downside is that programmers have to learn the numerous distinctions from C, and legacy application source code must be rewritten and recompiled. In addition, safe programming dialects can cause significant performance degradation and executable code bloat.

Methods for the static, automated detection of buffer overflow vulnerabilities in code have also been developed [165, 166, 167]. Using such techniques, complex application source code can be scanned prior to compilation in order to discover potential buffer overflow weaknesses. The detection mechanisms are not perfect: many false positives and false negatives can occur. Also, as true with Cyclone, these techniques ultimately require the programmer to inspect and often rewrite sections of application source code.

Transparent run-time defenses have also been proposed. The dynamically loaded libraries `libsafe` and `libverify` provide a run-time defense against stack smashing attacks and do not require programs to be re-compiled [10]. `libsafe` intercepts unsafe C library functions and performs bounds-checking to protect frame pointers and return addresses. `libverify` protects programs by saving a copy of every function and every return address in the heap. The first instruction of the original function is overwritten to execute code that stores the return address and jumps to the copied function code. The return instruction in the copied function is replaced with a jump to code that verifies the return address before returning.

The downside to `libsafe` is that it only defends against buffer overflow intrusions resulting from certain C library functions. In addition, static linking of these C library functions in a particular executable precludes `libsafe` from protecting the program. Implementations of `libverify` can double the code space required for each process, which is taxing for embedded devices with limited memory. Also, `libverify` can degrade performance by as much as 15% for some applications.

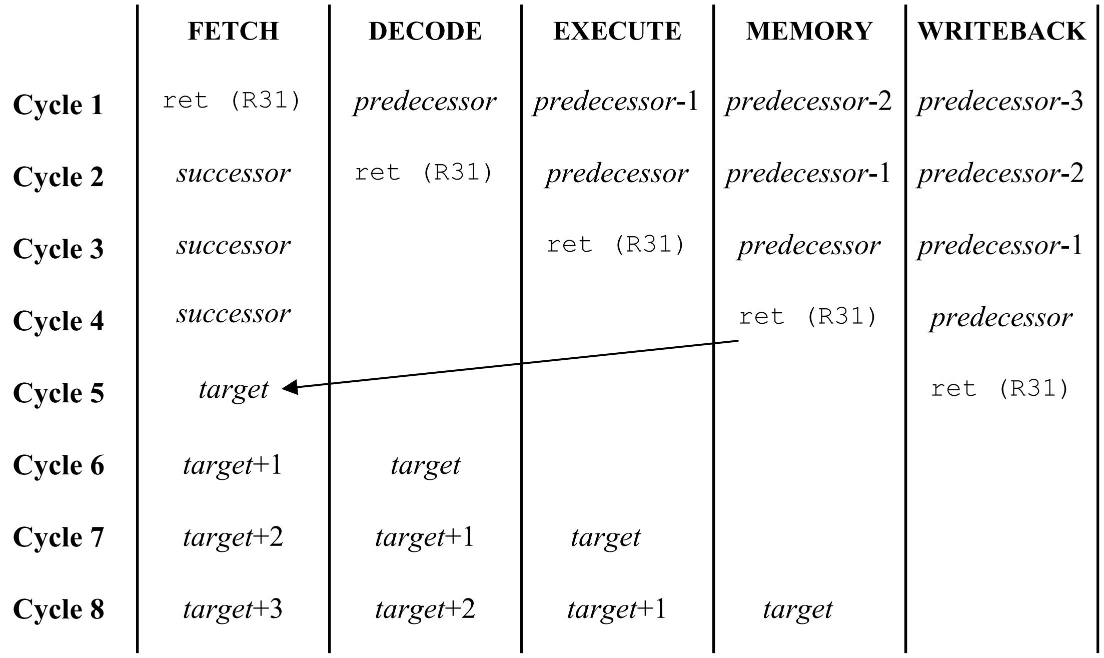
6.3 A Secure Return Address Stack

We now describe low-cost enhancements to the core hardware and software of future programmable machines that enable the detection and prevention of return address corruption. The examination of potential solutions at the hardware architecture level is justified by the frequency of this type of attack, the number of years it has been causing problems, and the continuing emergence of such problems despite existing software solutions.

We modify the implementation of procedure call and return instructions, employ a special hardware return address stack, and present a secure method for swapping the contents of the hardware stack to and from memory. Since we do not require changes to programming languages or application source code, both legacy and future software applications can benefit from the security provided by these enhancements.

6.3.1 Hardware Return Address Stacks

The branch target of a procedure return instruction is often calculated using the contents of one or more registers and/or memory words. Therefore, the target address cannot be resolved until the return instruction has passed through several stages of the processor pipeline. This address resolution delay can lead to performance degradation due to pipeline stalls. Figure 6.3 illustrates a simple 5-page processor pipeline that experiences a multi-cycle stall due to delayed return address resolution. In the figure, `ret (R31)` represents a



	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
Cycle 1	`ret (R31)`	*predecessor*	*predecessor*-1	*predecessor*-2	*predecessor*-3
Cycle 2	*successor*	`ret (R31)`	*predecessor*	*predecessor*-1	*predecessor*-2
Cycle 3	*successor*		`ret (R31)`	*predecessor*	*predecessor*-1
Cycle 4	*successor*			`ret (R31)`	*predecessor*
Cycle 5	*target*				`ret (R31)`
Cycle 6	*target*+1	*target*			
Cycle 7	*target*+2	*target*+1	*target*		
Cycle 8	*target*+3	*target*+2	*target*+1	*target*	

Figure 6.3: Pipeline stall caused by a return instruction

return instruction that does not resolve the return address until the memory access (MEMORY) stage of the pipeline, and *target* represents the instruction pointed to by the correct return address. The *predecessor* and *successor* instructions are the instructions that immediately precede and immediately follow the return instruction in the static program, respectively. Upon decoding the return instruction in the DECODE stage, the processor stalls the FETCH stage until the return address is obtained in the MEMORY stage. In the following cycle, the successor instruction is discarded, and the return target instruction is fetched. Thus, the arrow in the figure represents a control dependence, and the total performance penalty caused by the return instruction is 3 cycles.

Due to the LIFO nature of procedure calls, a simple stack structure that stores return addresses can alleviate such performance penalties by facilitating highly accurate prediction of the return instruction targets [71, 170]. Many processors, such as the Alpha 21164

[29] and the Alpha 21264 [30], employ such return address stacks (RAS) to improve performance by predicting procedure return addresses early in the pipeline.

Processor branch predictors can employ a hardware return address stack in conjunction with a branch target buffer (BTB) as illustrated in Figure 6.4. Branch target buffers are on-chip memory structures that store expected instruction address targets for non-return branch instructions. Upon executing a procedure call instruction, an entry from the BTB that is indexed by the program counter (PC) is used as the predicted target of the call instruction. The address of the following instruction (i.e., the return address) is pushed onto the RAS. During the execution of a return instruction, the topmost entry of the RAS is popped and used as the predicted target (instead of using an entry from the BTB). The RAS is unaffected during the target prediction of other branch and jump instructions. RAS structures are often implemented as circular buffers. When overflow of the RAS occurs, the least recently pushed address is overwritten with the value of the most recent return address. We henceforth refer to the hardware return address stack as the *RAS*, and we refer to the call stack for storing local variables and return addresses in memory as the *memory stack*.

Unfortunately, the RAS provides no protection against corruption of the return addresses in the memory stack. This is because the processor treats the RAS contents as branch prediction “hints” that are expected to be correct most but not all of the time. When a call instruction is executed, a valid return address is pushed on to the hardware RAS, and depending on the ISA, the return address may also be stored in the memory stack. Suppose the return address is subsequently corrupted by a buffer overflow in memory. At the moment when a valid return instruction is fetched, the corrupt address is located in a register or a memory location specified by the return instruction. Upon full execution of this instruction, the processor learns that the value popped from the hardware RAS does not equal the return address associated with the instruction. However, rather than jump to the correct return address popped from the hardware RAS, the processor flushes the pipe and

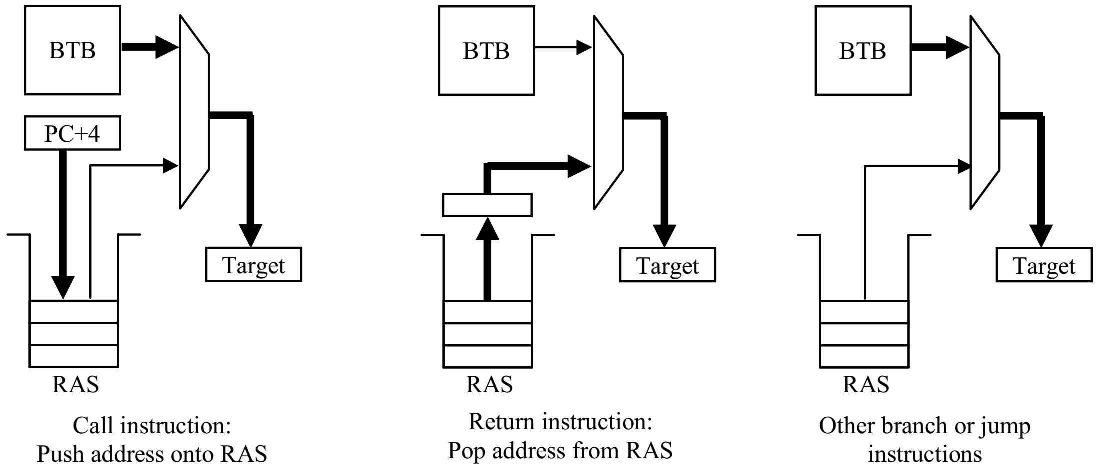


Figure 6.4: Branch target prediction with the BTB and the RAS

starts executing instructions beginning at the corrupt return address. The instructions based upon the correct return address from the RAS, which the processor issued and executed speculatively, are all nullified.

6.3.2 Architectural Modifications

By using special protected hardware and memory structures, we can defend against return address corruption in the memory stack. We define a special hardware RAS, the Secure Return Address Stack (SRAS), which always provides correct, uncorrupt return addresses. To properly employ the SRAS, we first modify the operation of procedure call and return instructions. We require that these call and return instructions are clearly recognizable. For instance, in a RISC ISA, a `branch_and_link` instruction is identified as a procedure call, and a `branch` to the link register (such as `R31`) is identified as a procedure return instruction [90].

We maintain the ISA definitions and visible behavior of call and return instructions, but we alter the manner in which the processor executes these instructions. Upon executing a procedure call instruction, the processor always pushes the return address onto the top of SRAS. The program counter is set to the target of the call instruction. When a processor executes a return instruction, the return address popped from the top of the hardware SRAS — not the target specified by a register or memory value — is assigned to the processor's program counter. The processor then determines whether the return address from the memory stack equals the return address popped from the SRAS. If these addresses differ, corruption of the memory stack's return addresses has occurred, and the processor may terminate the current process, may inform the OS of the corruption by issuing a new invalid return address trap, or may continue execution of the program based upon the correct address popped from the secure RAS.

6.3.3 Swapping Contents of the SRAS

For the SRAS to satisfy the security goals articulated above, the SRAS must maintain all of the return addresses represented in a program's memory stack. Though a program may employ an essentially unbounded degree of nesting, a hardware stack (such as the SRAS) can only contain a finite number of entries. Thus, to avoid overwriting valid addresses in the SRAS, we must define an efficient method for the processor to securely swap the contents of the SRAS to memory when the SRAS becomes completely full. We define the event in which the SRAS becomes full following a call instruction as *SRAS overflow*; the event where the SRAS becomes empty following a return is defined as *SRAS underflow*. We discuss the typical stack depth sizes observed for common programs in Section 6.6

The SRAS core consists of an n-address stack implemented as a circular buffer. Upon overflow, the processor will store the $n/2$ least recently pushed addresses to memory. Upon underflow, the processor will retrieve up to $n/2$ most recently pushed addresses from memory. The processor stores and retrieves $n/2$ addresses to and from memory rather than all n

addresses to prevent an SRAS thrashing scenario. In some programs, a policy of transferring the entire contents of the SRAS could lead to frequent storage of all SRAS addresses to memory immediately followed by the retrieval of n SRAS addresses from memory. We now investigate two different approaches to handling SRAS swapping: operating system-managed and processor-managed swapping.

OS-managed SRAS Swapping. In the first approach, the operating system assumes complete responsibility for swapping SRAS entries. In the events of SRAS overflow or underflow, the processor issues an OS interrupt. The OS then executes code that transfers contents of the SRAS to or from memory; the application does not observe or participate in SRAS content transfers. The kernel is responsible for managing the memory structures required to store the spilled SRAS entries for all threads running on the system. This is achieved by simply maintaining one stack of spilled SRAS return addresses for each process. In addition, the virtual memory regions that store the SRAS contents are mapped to physical pages that can only be accessed by the kernel. Hence, user-level application threads cannot corrupt the contents of their respective spilled stacks.

Processor-managed SRAS Swapping. In this scheme, we implement hardware enhancements to reduce the number of OS invocations associated with SRAS overflow and underflow. Figure 6.5 illustrates the hardware components required to support this SRAS swapping approach. We store information in the processor concerning the physical memory locations of the OS-managed data structures that contain spilled SRAS addresses. Upon SRAS underflow or overflow, the processor can employ this information to directly transfer SRAS contents to and from physical memory rather than invoking the OS. To support this functionality, the processor maintains two pointers to two physical pages that store spilled SRAS contents for the active process. Also, the processor maintains a counter that indicates how much space is available in the two physical pages. Although the two pages may be virtually or physically separated in memory, the two pages are treated as being adjacent to form a single “superpage”. When the superpage underflows or overflows, the OS is invoked

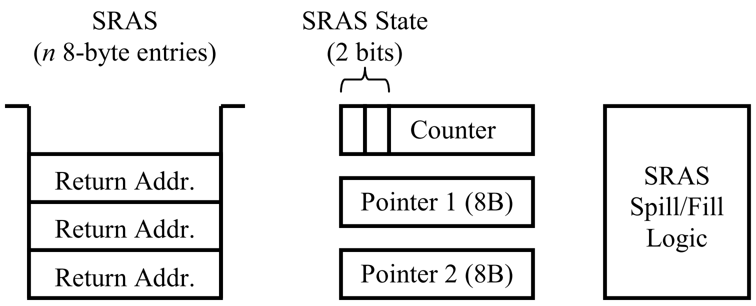


Figure 6.5: Hardware enhancements for processor-managed swapping

to allocate a new physical page or deallocate one of the two physical pages.

The processor maintains two pointers to two physical pages rather than one pointer for one physical page in order to avoid a thrashing scenario in which a page is repeatedly allocated and immediately deallocated. In such a thrashing situation, the OS could be invoked for every SRAS spill and fill, and performance would decay. By providing the processor with access to two physical frames at once, we can avoid calls to the OS caused by jumping back and forth over a page boundary. The processor logic required to manage the two pointers and the counter is based upon a simple 4-state machine.

Figure 6.6 illustrates this state machine, which consists of the states A, B, C, and D. The two bits that represent the state are stored by the processor in the high order bits of the counter. State A is the starting state of the machine, in which the SRAS has never overflowed and the two page pointers are empty. B is the state in which the two page pointers point to the only two pages that have been allocated by the OS for SRAS overflows in the current process. We call these two pages the “base superpage”. C and D represent the states in which three or more pages have been allocated by the OS for SRAS overflows in the current process. State C indicates that Pointer 1 points to the high-order page of the superpage, and State D indicates that Pointer 2 points to the high-order page of the superpage. States C and D could be consolidated, but such a consolidated state may require

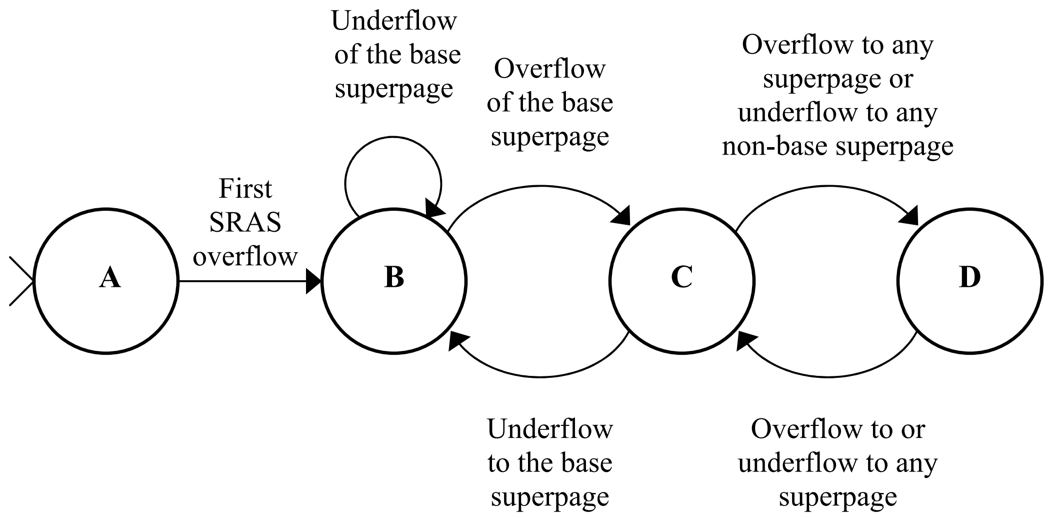

Underflow of the base superpage
Overflow of the base superpage
Overflow to any superpage or underflow to any non-base superpage
First SRAS overflow
A
B
C
D
Underflow to the base superpage
Overflow to or underflow to any superpage

Figure 6.6: State machine for processor-based SRAS swapping

the processor to swap the values of Pointers 1 and 2 upon superpage overflow or underflow. Note that the state machine presented in Figure 6.6 is only one of many possible machines for supporting the processor-managed swapping scheme.

The OS is invoked much less often in this scheme than in the OS-managed swapping scheme. For example, if the SRAS consists of 64 8-byte return address entries and the page size is 8 KB, the OS is invoked only once after $8192/((64/2) \times 8) = 32$ consecutive SRAS overflows. In the OS-managed scheme, the OS would be invoked for each of those 32 SRAS overflows. In Section 6.6, we compare the performance impact of these two schemes on a set of benchmark programs.

We also note that since the values popped from the SRAS must always be valid to preserve correct execution, all of the SRAS contents and any associated configuration state bits must be transferred to and from memory on context switches.

6.4 Non-LIFO Procedure Control Flow

If software always exhibited LIFO procedure control flow behavior, the SRAS would transparently provide hardware-based protection of return addresses for all programs. No compiler changes or recompilation of existing source code would be necessary; the system would provide protection for all legacy and future binary executables. Unfortunately, however, some existing executables use non-LIFO procedure control flow. For example, some compilers seek to improve performance by allowing certain procedures to return to an address located deep within the stack. The memory stack pointer is then set to an address of a frame buried within the stack; the frames located in between the former top of the stack and the reassigned stack pointer are effectively popped and discarded. Exception handling in C++ is one technique that can lead to such non-LIFO behavior.

Other common causes of non-LIFO control flow are the C `setjmp` and `longjmp` library functions. These functions are employed to support software signal handling. The `longjmp` function may cause a program to return to an address that is located deep within the memory stack or to an address that is no longer located in the memory stack. More specifically, a particular return address may be explicitly pushed onto the stack only once, but procedures may return to that address more than once. Note that tail call optimizations, which seem to involve non-LIFO procedure control flow, do not cause problems for the SRAS. Compilers typically maintain proper pairing of procedure call and return instructions when implementing tail call optimizations.

The security of this proposal depends on the correctness of the address popped from the top of the hardware SRAS. However, the SRAS mechanism described so far does not accommodate non-LIFO procedure control flow. We can address this issue in at least four ways. These four options, which we summarize in Table 6.2, trade varying degrees of security and non-LIFO support for implementation cost and complexity.

The first two options enable zero or complete support for non-LIFO behavior while

Table 6.2: Non-LIFO procedure control flow handling options

Non-LIFO control flow handling method	**Permits non-LIFO control flow**	**Provides complete protection**	**ISA changes**	**Compiler changes**
None	No	Yes	No	No
`sras_off` option	Yes	No	Yes	No
Static handling	Yes	Yes	Yes	Yes
Dynamic handling	Some	Yes	Yes	No

facilitating high or low security against procedure return address corruption, respectively. The first option is to implement the SRAS as described above and completely prohibit code and compiler practices that employ non-LIFO procedure control flow. This provides the highest degree of security against return address corruption. Legacy executables that exhibit non-LIFO procedure calling behavior will terminate with an error (if not recompiled). The second option is to allow users to disable the SRAS with a new privileged `sras_off` instruction. This enables the execution of potentially insecure code that exhibits any non-LIFO behavior as permitted in systems without an SRAS. After executing `sras_off`, the OS can re-enable the SRAS at any time by executing a new privileged `sras_on` instruction.

The third option is to permit certain types of non-LIFO procedure control flow, such as returning to addresses located deep within the stack. This option requires re-compilation of some legacy programs. During re-compilation, the compiler must take precautions to ensure that the top of the SRAS will always contain the correct target address for an executed return instruction in programs that use non-LIFO techniques. We define new instructions, `sras_push` and `sras_pop`, which explicitly push and pop entries to and from the SRAS without actually calling or returning from a procedure. Compilers can employ these new instructions to return to an address deep within the SRAS (and to the associated frame in the memory stack) when using `longjmp`, C++ exception handling, or other non-LIFO routines.

The fourth option is to provide dynamic support for common non-LIFO behavior. This approach does not support all instances of non-LIFO behavior that the second option can handle via re-compilation, but it does allow execution of some legacy executables (where the source code is no longer available) that exhibit non-LIFO procedure control flow. First, we implement the `sras_push` and `sras_pop` instructions described above. We also need an installation-time or run-time software filter that strategically injects `sras_push` and `sras_pop` instructions (as well as other small blocks of code) into binaries prior to or during execution. The software filter inserts these instructions in recognized routines that cause non-LIFO procedure control flow. For instance, standardized functions like `setjmp` and `longjmp` can be identified at run-time via inspection of linked libraries such as `libc`. This option handles only executables that employ known non-LIFO techniques, however. For new manifestations of non-LIFO procedure control flow, the software filter may not identify some locations where the new instructions should be inserted.

Regardless of the method(s) used to handle non-LIFO procedure control flow, we require that the SRAS be "turned on" by default in order to provide built-in protection. To disable the SRAS, a user (at his own risk) must explicitly "turn off" the SRAS.

6.5 Security Analysis

The primary design goal is to prevent attacks in which hostile code is injected and executed on innocent hosts by exploiting buffer overflow vulnerabilities that corrupt procedure return addresses. The modifications described above accomplish this goal: an architecture based on a Secure Return Address Stack detects and prevents corruption of return addresses. In the proposed system, only call and return instructions can modify the contents of the SRAS. Hence, the correct return addresses will be preserved in the event of a standard buffer overflow attack that corrupts the values of return addresses in the memory stack. If such corruption does occur in memory, the processor detects this and can respond appropriately.

Since the SRAS is finite in size, its contents must be securely swapped to and from memory upon overflow and underflow, respectively, to guarantee security. In both the OS-managed and processor-managed SRAS schemes, we protect spilled SRAS contents by storing the addresses in physical pages that are not accessible by the virtual memory spaces of user-level applications. Because non-kernel processes cannot access the contents spilled from their respective SRASes, no software bug or buffer overflow vulnerability in such processes can affect the spilled return addresses.

To provide truly pervasive and comprehensive protection against buffer overflow and related attacks, the SRAS solution should be implemented in conjunction with existing software defenses. Such software-based defenses include static scanning methods and safe programming practices, which can identify and mitigate a variety of security vulnerabilities. The SRAS proposal complements these defenses by offering specialized dynamic protection for legacy code and by preventing potential attacks in new code that may be unrecognized by the software defenses. In addition, since we require changes to hardware, this proposal is meant to be a long-term solution. Software defenses, however, can and should be applied as they become available.

6.6 Performance Analysis

We now examine the performance impact of spilling and retrieving the contents of the SRAS to and from memory during program execution. The architectural enhancements that we propose enable security features for all programs, rather than just for network-based or cryptographic applications. Because of this, a performance study that examines the impact of the proposed architectural changes on all programs is more useful than one limited to specific applications. Thus, we use the SPEC2000 benchmarks as a representative computing workload, for this benchmark suite is commonly used to evaluate the performance of general-purpose processors [155].

6.6.1 Simulation Methodology

To obtain performance data for the benchmarks, we use SimpleScalar, a cycle-accurate out-of-order superscalar processor simulator [21]. We consider the scenario in which the operating system is invoked each time a SRAS swap is required and the scenario where the processor primarily handles SRAS swapping. The OS-managed swapping scheme is easier to implement, but the processor-managed scheme can provide better performance. We simulate the execution of 500 million instructions of 12 SPEC2000 integer benchmarks after skipping at least 1 billion instructions in order to capture steady state behavior [137].

The base processor model closely represents an Alpha 21264 processor [30]. We summarize the processor simulation parameters in Table 6.3. The base processor includes a hardware return address stack. In some situations, speculative execution based on the prediction of "non-return" branches, which we define to be any branch instructions that are not procedure return instructions, can pollute the RAS with invalid return addresses. Hence, to maintain the integrity of the SRAS, the processor must include a perfect repair mechanism to recover from SRAS corruption due to "non-return" branch mispredictions [150]. Such mechanisms involve saving the top-of-stack (TOS) pointer and the return address to which the TOS points following the prediction of a non-return branch instruction. If the processor discovers that a particular branch was mispredicted and the wrong program control path was followed, the processor can use information such as the saved TOS to restore the SRAS to its former, correct state (preceding the mispredicted non-return branch). Note that, as described in Section 6.3.2, the processor can distinguish between non-return branch instructions and return instructions by simply inspecting an instruction opcode.

Table 6.3: SimpleScalar simulation parameters

Parameters	**Characteristics**
Instruction window	64-entry RUU
Fetch/decode/issue width	4 instructions
Commit Width	8 instructions
Functional Units	4 integer ALUs, 1 integer mult. 4 FP ALUs, 1 FP multiplier
BTB	4K-entry, 2-way set associative
Branch Predictor	Hybrid: 4K 2-bit selector 4K 2-bit bimodal predictor 1K 2-bit local w/ 10-bit history
L1 data cache	64 KB 2-way set-associative 64 byte blocks, 2 cycle latency, 2 ports
L1 instruction cache	64 KB 2-way set-associative 64 byte blocks, 1 cycle latency
L2 unified cache	2 MB 4-way set-associative 64 byte blocks, 12 cycle latency
Main memory	100 cycle latency
Load/store queue	64 entries
I-TLB and D-TLB	128-entry fully associative

We gather performance results for all 12 benchmarks, SRAS sizes of 16, 32, 64, 128, and infinite entries, and page sizes of 8 KB, 16 KB, and 32 KB. To model the SRAS swapping code in the OS-managed swapping scheme, we wrote a swapping and memory management routine in C. We also wrote C code that models the allocation and deallocation of physical pages during OS invocations in the processor-managed swapping scheme. We compile the SPEC2000 benchmarks and the OS swapping code on an Alpha machine with full optimizations to produce Alpha executables.

We simulate the execution of the SRAS swapping and page allocation routines assuming the caches are initially cold. We obtain cycle counts for all four OS routines: spilling the SRAS to memory in the OS-managed scheme, filling the SRAS from memory in the OS-managed scheme, allocating a new physical page in the processor-managed scheme, and deallocating a physical page in the processor-managed scheme. We find that each of these four routines requires between 23,000 and 25,000 cycles to complete. These cycle counts vary with the size of the SRAS.

In addition, we simulate the spilling and filling of SRAS contents directly to and from physical memory in the processor-managed SRAS swapping scheme by stalling the processor for a number of cycles. This stall time represents the number of cycles required to transfer $n/2$ SRAS entries directly to or from main memory. The bus from the processor to main memory in the system model can transfer one 64-bit value every two cycles after an initial latency. In the event of underflow, the first load experiences the main memory latency of 100 cycles, and then the $(n/2 - 1)$ subsequent addresses arrive at the processor every other cycle for $(n-2)$ cycles. Hence, the total stall time is $(98+n)$ cycles. Similarly, in the event of overflow in this scheme, the processor stalls for $(98 + n)$ cycles to store $n/2$ addresses to main memory.

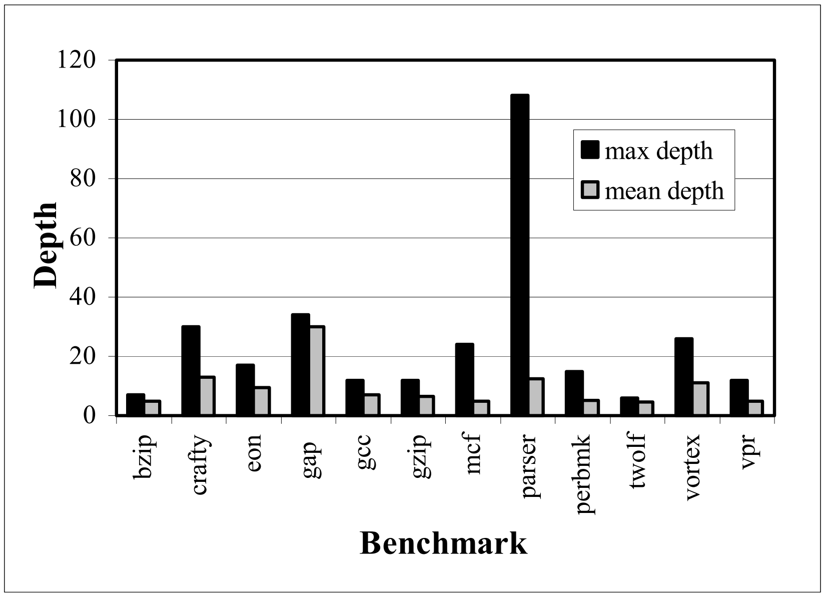


Figure 6.7: Return address stack depths

6.6.2 Performance Results

Figure 6.7 illustrates maximum and mean depths of the return address stack associated with the benchmarks. We observe that the return address stacks for the SPEC2000 benchmarks never exceed depths of 108 addresses. The mean depths of the return address stacks for the SPEC2000 benchmarks range from 4.72 to 29.94 addresses. Although Figure 6.7 provides an appropriate guide for choosing the SRAS size, the figure does not present precise information concerning the performance impact of the SRAS on the benchmarks. The SRAS negatively impacts performance only when its contents are swapped to or from memory. Hence, instead of hinging on maximum and mean stack depths, performance primarily depends on the rates at which the memory stack (and thus the SRAS) grows and shrinks.

The performance penalties caused by SRAS swapping in the SPEC2000 benchmarks are presented in Table 6.4. These statistics represent percent performance degradation

Table 6.4: Performance impact of SRAS swapping

Benchmark	**Percent Performance Degradation for Different SRAS Sizes**							
	OS-managed				**Processor-managed**			
	16	**32**	**64**	**128**	**16**	**32**	**64**	**128**
`bzip`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
`crafty`	**5.8**	0.1	0.0	0.0	0.0	0.0	0.0	0.0
`eon`	**67.9**	0.0	0.0	0.0	1.0	0.0	0.0	0.0
`gap`	**4.7**	**1.5**	0.0	0.0	0.0	0.0	0.0	0.0
`gcc`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
`gzip`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
`mcf`	**7.3**	0.0	0.0	0.0	0.0	0.0	0.0	0.0
`parser`	**64.2**	**29.3**	**9.2**	0.1	0.8	0.2	0.1	0.0
`perlbmk`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
`twolf`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
`vortex`	**36.7**	0.0	0.0	0.0	0.3	0.0	0.0	0.0
`vpr`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

caused by an n-entry SRAS relative to the base machine model that includes an n-entry return address stack. The entries listed in bold indicate the situations in which the performance degradation exceeds 1%. For an SRAS size of 16 entries in the OS-managed scheme, 6 of the 12 SPEC2000 integer benchmarks experience performance reductions ranging from 4.7% to 67.9%. If the SRAS contains 64 entries, the performance degradation caused by OS-managed swapping is negligible (i.e., 1% or less) for all benchmarks except for `parser`, which sustains performance degradation of 9.2%. The `parser` benchmark is a syntactic parser of English in which the memory stack grows and shrinks quickly; thus, SRAS swapping penalties can be significant. When the SRAS contains 128 or more entries, the performance impact is negligible for all of the benchmarks.

In the processor-managed scheme, however, the performance degradation is less than or equal to 1% for all of the benchmarks when using a SRAS of size 16 entries or greater.

We therefore conclude that the processor-managed SRAS swapping scheme is superior to the OS-managed SRAS swapping scheme. The processor-managed scheme achieves much higher performance than the OS-managed scheme at the cost of a small, incremental implementation effort. Furthermore, the incremental effort is partially offset by the smaller number of entries needed in the SRAS to achieve acceptable performance.

6.7 Comparative Analysis

We compare the characteristics of the SRAS solution to past work in Tables 6.5 and 6.6. Table 6.5 summarizes the system changes required by the different solutions. As shown in the table, unlike safe programming languages and static analysis techniques, the SRAS does not require source code changes. Furthermore, unlike StackGuard, the SRAS does not require compiler changes (unless certain non-LIFO handling features are needed). The SRAS and the other dynamic solutions (i.e., StackGhost, `libsafe`, and `libsafe`) involve OS changes, and only the SRAS requires hardware enhancements.

Table 6.6 summarizes the capabilities and system impact of the different solutions. We observe that the SRAS is the only solution that combines the features of strong protection against procedure return address corruption, wide applicability to various platforms, a negligible increase in code size, and low performance impact. Furthermore, the SRAS provides these features for both legacy code and new code.

6.8 Summary

This chapter examines hardware-based defenses against buffer overflow attacks. These defenses help mitigate vulnerabilities in application and operating system software that runs on general-purpose platforms.

Table 6.5: Comparison of required system changes

Technique for defending against return address corruption	**Required changes**			
	Source code	**Compiler**	**OS**	**Processor**
Safe programming languages	Yes	Yes	No	No
Static analysis techniques	Yes	No	No	No
StackGuard	No	Yes	No	No
StackGhost	No	No	Yes	No
`libsafe`	No	No	Yes	No
`libverify`	No	No	Yes	No
SRAS	**No**	**No**[a]	**Yes**	**Yes**

[a]Compiler changes may be required for certain programs depending on how non-LIFO procedure control flow is handled (see Section 6.4).

Table 6.6: Comparison of capabilities and system impact

Technique for defending against return address corruption	**Provides complete protection**[b]	**Applies to many platforms**	**Application code size increase**	**Adverse performance impact**
Safe prog. languages	Yes	Yes	Can be high	Can be high
Static analysis techniques	No	Yes	Varies	Varies
StackGuard	No	Yes	Low	Moderate
StackGhost	Yes	No	None	Low
`libsafe`	No	Yes	Low	Low
`libverify`	Yes	Yes	High	Moderate
SRAS	**Yes**	**Yes**	**None**[c]	**Low**

[b]By "complete protection", we mean complete protection against buffer overflow addresses that directly corrupt procedure return addresses.

[c]Depending on how non-LIFO procedure control flow is handled, some programs may experience a very small increase in code size (see Section 6.4).

Malicious parties routinely exploit buffer overflow vulnerabilities to enable the insertion or execution of hostile code on an innocent user's machine by corrupting procedure return addresses in the memory stack. Such attacks can lead to disastrous consequences, especially when the buffer overflow affects cryptographic software modules that are privy to critical secrets such as cryptographic keys. Hence, addressing such buffer overflow vulnerabilities is a security priority. Although software-based countermeasures are available, a processor architecture defense is justified because of the fact that major security problems stemming from buffer overflows continue to plague networks and computer systems.

This chapter presents a processor-based, built-in, and low-cost layer of protection against common buffer overflow vulnerabilities. Some processors contain return address stacks to reduce performance penalties due to delayed branch resolution. We detail how a modified hardware return address stack can be used to protect return addresses. This modified hardware-based stack, the Secure Return Address Stack (SRAS), detects and prevents corruption of procedure return addresses resulting from buffer overflow and other attacks.

The SRAS requires only minor changes to the operating system and to the branch prediction structures found in many microprocessors, so legacy and future software can enjoy the security benefits without requiring application source code modifications. Since the SRAS is of finite size, various management and swapping methods are described and evaluated. This chapter demonstrates that the SRAS and the proposed management techniques cause a negligible performance impact in most applications. Furthermore, since the SRAS leverages branch prediction structures that may already be incorporated into the processor, the hardware requirements for implementing this mechanism are minor.

This proposal is not meant to be the only defense against such attacks on cryptographic and general software, however. We recommend that this proposal be used in conjunction with other techniques to provide more robust protection against potential software vulnerabilities.

Chapter 7

Conclusion

Many systems and proposals use cryptographic processing to realize certain essential security requirements. Examples of such requirements include confidentiality, integrity, and authentication. However, important issues remain unaddressed in cryptographic system designs and implementations. These problems include lack of protection for critical secrets such as cryptographic keys, poor performance of cryptographic primitives, and general dependability issues endemic to complex software such as latent buffer overflow vulnerabilities.

This thesis studied a set of architectural techniques for addressing these problems by providing more secure and efficient cryptographic processing. In this chapter, we summarize and reflect on the contributions of this thesis and propose directions for future research.

7.1 Securing and Accelerating Cryptographic Processing

This thesis presented four contributions towards enabling secure cryptographic processing. These contributions were not intended to address all known cryptographic software security and performance issues. Rather, these contributions address four important open problems in security.

The first contribution, described in Chapter 3, is a software and hardware system for protecting users' cryptographic keys on general-purpose computing devices. This system, which is called Virtual Secure Coprocessing (VSCoP), allows users to safely and flexibly store and exercise their cryptographic keys. Furthermore, VSCoP provides key protection on devices with OS software and application software that may contain security vulnerabilities. The system provides protection against several classes of previously unaddressed threats without requiring ancillary hardware. VSCoP requires a new cryptographic software library and a few changes to the processor, the hardware platform, and the operating system. The VSCoP modifications and enhancements enable protection for keys by effectively modifying the access control paradigm and restricting the trusted boundary to the physical boundary of the processor chip. Although modifications are made to several components of the computing system, the modifications are low-cost and efficiently provide an improved level of security for users' keys.

The second contribution, described in Chapter 4, is a system for protecting certain cryptographic keys that are distributed to other parties. The proposed system, called Traitor Tracing using RSA (TTR), enables the traceability of decryption keys in broadcast encryption scenarios. In general, it is difficult to build commodity decryption devices that prevent users in the field from extracting keys stored in the device and from subsequently divulging those keys. Thus, in this chapter, instead of focusing on hardening decoding devices, resources are applied to enabling traceability of decryption keys and related information. The proposed TTR scheme guarantees traceability when the pirate decoding devices are created by traitor collusions of size fewer than a threshold k. Following the confiscation of a pirate device, at least one "traitor" that supplied key information that enabled the creation of the device can be uniquely identified. Thus, TTR relies on deterrence and mechanisms for attribution to protect keys. Given the identities of one or more traitors, the legitimate owner of the cryptographic keys can seek retribution or proper remuneration in other venues, e.g., the judicial system. Furthermore, the new scheme achieves higher performance for decryption than that of previously proposed traitor tracing systems.

The third contribution, described in Chapter 5, consists of architectural enhancements for fast bit-level permutations and mappings. Such bit-level operations consume significant shares of the computation in software implementations of several cryptographic algorithms. Hence, by accelerating these operations, the performance of critical security primitives can be significantly increased. The architectural enhancements are based upon two new processor instructions, `swperm` and `sieve`. This thesis demonstrated that these instructions can be implemented and exercised via low-cost, high-performance processor hardware and simple application software. Furthermore, this thesis showed that we can use the new instructions to significantly improve the performance of the Data Encryption Standard, especially when we use the algorithm as a hash function.

The last contribution, described in Chapter 6, is a processor-based method for dynamically mitigating common software vulnerabilities. This proposal, called the Secure Return Address Stack (SRAS), provides built-in, transparent protection against buffer overflow attacks involving procedure return address corruption. Despite past proposals for software-based remedies for these attacks, this type of buffer overflow vulnerability continues to pose a serious threat to software modules. The SRAS prevents any undetectable procedure return address corruption via low-cost processor and operating system enhancements. In addition to improving the dependability of cryptographic software, we can apply the SRAS to provide buffer overflow protection for any software.

7.2 Directions for Future Research

There exist several possible directions for both immediate and long-term extensions to the contributions of this thesis.

Several of the components of VSCoP can be enhanced to provide additional capabilities and features. We consider two of many possible enhancements here. For example, we can improve the security provided by VSCoP by strongly authenticating any software that invokes the Cryptographic Operations Library (COL) software prior to servicing a COL request. Furthermore, by abstracting and generalizing the VSCoP design, it is possible to support multiple security compartments for protecting cryptographic keys and sensitive information. Each of the compartments would be associated with an independent and cryptographically enforced access and usage policy, which would be especially valuable in multi-user systems. In the long term, we intend to investigate the efficient integration of VSCoP with trusted computing platforms and related technologies. Through such integration, we can provide a more comprehensive foundation for secure computing.

Various aspects of the traitor tracing scheme can be improved and enhanced. By refining the security analysis of the scheme, we can achieve tighter bounds on certain system parameters that will enable more efficient operation and decreased overhead. Furthermore, we can investigate adding new features to the traitor tracing scheme. Such features include support for public-key encryption or integration with digital fingerprinting techniques. Future work in this area may include the exploration of different decoder hardware and software architectures that incorporate the traitor tracing functionality.

This thesis proposed techniques for hardware-based acceleration of a limited set of operations (i.e., bit-level mappings). However, other opportunities exist to improve the performance of other security operations via architectural methods. One possible research direction involves the exploration of new subword-parallel and other types of instructions for accelerating a variety of cryptographic algorithms. Furthermore, future work may include applying series of the bit-level mapping and permutation instructions proposed by this thesis to construct new ciphers that achieve the cryptographic property of diffusion more rapidly. By achieving diffusion faster, less computation would be required to reach a desired level of security. This could translate into further performance acceleration of cryptographically-enabled secure systems.

The SRAS proposed by this thesis successfully protects cryptographic and general software against a common class of buffer overflow attacks. To provide robust security against diverse vulnerabilities, however, we can and should implement additional mechanisms to provide built-in protection. Future work in this area includes the investigation of transparent hardware and software defenses against other classes of buffer overflow attacks (e.g., heap overflows) and of other common vulnerabilities.

The ultimate goal of security research is achieving robust system security against known and future threats. Such robust security would extend from the global computing infrastructure down to the electrons moving between individual transistors in general-purpose processors. Given the exponentially increasing complexity of information processing systems and security threats, this is clearly a difficult task. However, it is evident that security functionality should be incorporated into systems at a fundamental level in order to approach this ultimate security goal. Thus, computer architects must play a larger and more central role in the design and implementation of secure systems. With architectural techniques such as those proposed by this thesis, systems that protect networks, computers, and data can enjoy a higher degree of security.

Bibliography

[1] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack*, vol. 7, no. 49, 1996.

[2] American National Standards Institute, "American National Standard X9.17: Financial Institution Key Management," 1985.

[3] Amphion Corporation, "AES Encryption/Decryption," available at http://www.amphion.com/cs5265.html, 2002.

[4] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, New York: John Wiley and Sons, Inc., 2001.

[5] R. Anderson and M. Kuhn, "Low Cost Attacks on Tamper Resistant Devices," *Security Protocols: 5th International Workshop*, *Lecture Notes in Computer Science*, vol. 1361, Springer-Verlag, pp. 125–136, 1997.

[6] ARM Corporation, "A New Foundation for CPU Systems Security: Security Extensions to the ARM Architecture," available at http://www.arm.com/pdfs/TrustZone.pdf, May 2003.

[7] M. Artin, *Algebra*, Upper Saddle River, NJ: Prentice-Hall, Inc., 1991.

[8] D. Atkins, W. Stallings, and P. Zimmermann, "PGP Message Exchange Formats," IETF RFC 1991, August 1996.

[9] D. Balfanz and E. W. Felten, "Hand-Held Computers Can Be Better Smart Cards," *Proceedings of the 1999 USENIX Security Symposium*, 1999.

[10] A. Baratloo, N. Singh, and T. Tsai, “Transparent Run-time Defense against Stack Smashing Attacks,” *Proceedings of 9th USENIX Security Symposium*, June 2000.

[11] M. Bellare and P. Rogaway, “Optimal Asymmetric Encryption,” *Advances in Cryptology — EUROCRYPT ’94*, *Lecture Notes in Computer Science*, vol. 950, Springer-Verlag, pp. 92–111, 1995.

[12] R. M. Best, “Preventing Software Piracy with Crypto-Microprocessors,” *Proceedings of IEEE Spring COMPCON ’80*, pp. 466–469, 1980.

[13] E. Biham, R. Anderson and L. Knudsen, “Serpent: A New Block Cipher Proposal,” *Proceedings of the 5th Workshop on Fast Software Encryption*, *Lecture Notes in Computer Science*, vol. 1372, Springer-Verlag, pp. 260–271, 1998.

[14] M. Blaze, “High-Bandwidth Encryption with Low-Bandwidth Smartcards,” *Proceedings of the 3rd Workshop on Fast Software Encryption*, *Lecture Notes in Computer Science*, vol. 1039, Springer-Verlag, pp. 33–40, 1996.

[15] M. Bond and R. Anderson, “API-Level Attacks on Embedded Systems,” *IEEE Computer*, vol. 34, no. 10, pp. 67–75, Oct. 2001.

[16] D. Boneh, “Simplified OAEP for the RSA and Rabin Functions,” *Advances in Cryptology — CRYPTO ’01*, *Lecture Notes in Computer Science*, vol. 2139, Springer-Verlag, pp. 275–291, 2001.

[17] D. Boneh, “Twenty Years of Attacks on the RSA Cryptosystem,” *Notices of the American Mathematical Society*, vol. 46, no. 2, pp. 203–213, 1999.

[18] D. Boneh and M. Franklin, “An Efficient Public Key Traitor Tracing Scheme,” *Advances in Cryptology — CRYPTO ’99*, *Lecture Notes in Computer Science*, vol. 1666, Springer-Verlag, pp. 338–353, 1999.

[19] D. Boneh and J. Shaw, “Collusion-Secure Fingerprinting for Digital Data,” *Advances in Cryptology — CRYPTO ’95*, *Lecture Notes in Computer Science*, vol. 963, Springer-Verlag, pp. 452–465, 1995.

[20] Bulba and Kil3r, “Bypassing StackGuard and StackShield,” *Phrack Magazine*, vol. 10, issue 56, May 2000.

[21] D. Burger and T. Austin, “The SimpleScalar Tool Set, Version 2.0,” University of Wisconsin-Madison Computer Sciences Department Technical Report, no. 1342, June 1997.

[22] J. Burke, J. McDonald, T. Austin, “Architectural Support for Fast Symmetric-Key Cryptography,” *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pp. 178–189, November 2000.

[23] K. Campbell, L. A. Gordon, M. P. Loeb, and L. Zhou, “The Economic Cost of Publicly Announced Information Security Breaches: Empirical Evidence from the Stock Market,” *Journal of Computer Security*, vol. 11, no. 3, pp. 431–448.

[24] Canadian Security Systems Centre, *Canadian Trusted Product Evaluation Criteria*, version 3.0e, January 1993

[25] B. Chor, A. Fiat, and M. Naor, “Tracing Traitors,” *Advances in Cryptology — CRYPTO ’94*, *Lecture Notes in Computer Science*, vol. 839, Springer-Verlag, pp. 257–270, 1994.

[26] B. Chor, A. Fiat, M. Naor, and B. Pinkas, “Tracing Traitors,” *IEEE Transactions on Information Theory*, vol. 46, no. 3, pp. 893–910, May 2000.

[27] C. Coarfa, P. Druschel, and D. S. Wallach, “Performance Analysis of TLS Web Servers,” *Proceedings of the 2002 Network and Distributed Security Symposium (NDSS)*, 2002.

[28] Common Criteria Editorial Board, *Common Criteria for Information Technology Security Evaluations*, Version 1.0, 1996.

[29] Compaq Computer Corporation, *Alpha 21164 Microprocessor: Hardware Reference Manual*, December 1998.

[30] Compaq Computer Corporation, *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[31] Computer Emergency Response Team (CERT), “CERT/CC Statistics 1988-2004,” available at http://www.cert.org/stats/cert_stats.html, accessed January 2005.

[32] Computer Emergency Response Team (CERT), http://www.cert.org/, 2005.

[33] Computer Security Institute, *2004 CSI/FBI Computer Crime and Security Survey*, 2004.

[34] M. Conover, “w00w00 on Heap Overflows,” available at http://www.w00w00.org/files/aritcles/heap-tut.txt, 1999.

[35] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman, “FormatGuard: Automatic Protection From printf Format String Vulnerabilities,” *Proceedings of the 2001 USENIX Security Symposium*, 2001.

[36] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman, “PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities,” *Proceedings of the 2003 USENIX Security Symposium*, pp. 91–104, 2003.

[37] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[38] I. J. Cox, J. Kilian, T. Leighton, and T. Sharmoon, "A Secure Robust Watermark for Multimedia," *Proceedings of the First Information Hiding Workshop – IHW '96*, *Lecture Notes in Computer Science*, vol. 1174, Springer-Verlag, pp. 185–206, 1996.

[39] S. A. Craver, J. P. McGregor, M. Wu, B. Liu, A. Stubblefield, B. Swartzlander, D. S. Wallach, D. Dean, and E. W. Felten, "Reading Between the Lines: Lessons from The SDMI Challenge," Princeton University Computer Science Technical Report TR-657-02, 2002.

[40] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," IETF RFC 2460, December 1998.

[41] J. M. DeLaurentis, "A Further Weakness in the Common Modulus Protocol for the RSA Cryptoalgorithm," *Cryptologia*, vol. 8, no. 3, pp. 253–259, 1984.

[42] Department of Defense, *Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD, 1985.

[43] Y. Desmedt and Y. Frankel, "Shared Generation of Authenticators and Signatures," *Advances in Cryptology — CRYPTO '91*, *Lecture Notes in Computer Science,* vol. 576, Springer-Verlag, pp. 457–469, 1991.

[44] K. Diefendorff, et al., "AltiVec Extension to PowerPC Accelerates Media Processing," *IEEE Micro*, vol. 20, no. 2, pp. 85–95, March/April 2000.

[45] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, November 1976.

[46] Y. Dodis, N. Fazio, A. Kiayias, and M, Yung, “Fully Scalable Public-Key Traitor Tracing,” *Proceedings of Principles of Distributed Computing (PODC-2003)*, July 2003.

[47] J. Dyer, R. Perez, S. Smith, M. Lindemann, “Application Support Architecture for a High-Performance, Programmable Secure Coprocessor,” *Proceedings of the 22nd National Information Systems Security Conference*, October 1999.

[48] T. ElGamal, “A Public-key Cryptosystem and a Signature Scheme Based on Discrete Logarithms,” *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, July 1985.

[49] A. Fiat and M. Naor, “Broadcast Encryption,” *Advances in Cryptology — CRYPTO ’93, Lecture Notes in Computer Science*, vol. 773, Springer-Verlag, pp. 480–491, 1993.

[50] A. Fiat and T. Tassa, “Dynamic Traitor Tracing,” *Advances in Cryptology — CRYPTO ’99, Lecture Notes in Computer Science*, vol. 1666, Springer-Verlag, pp. 354–371, 1999.

[51] A. M. Fiskiran and R. B. Lee, “Evaluating Instruction Set Extensions for Fast Arithmetic on Binary Finite Fields,” *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pp. 125–136, September 2004.

[52] A. M. Fiskiran and R. B. Lee, “Performance Scaling of Cryptography Algorithms in Servers and Mobile Clients,” *Proceedings of the Workshop on Building Block Engine Architectures for Computer Networks (BEACON)*, October 2004.

[53] W. Ford and B. S. Kaliski, Jr., “Sever-assisted Generation of a Strong Secret from a Password,” *Proceedings of the 5th IEEE International Workshop on Enterprise Security*, 2000.

[54] M. Frantzen and M. Shuey, “StackGhost: Hardware Facilitated Stack Protection,” *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[55] A. O. Freier, P. Karlton, P. C. Kocher, “The SSL Protocol,” Version 3.0, available at http://wp.netscape.com/eng/ssl3/ssl-toc.html, March 1996.

[56] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern, “RSA-OAEP Is Secure under the RSA Assumption,” *Advances in Cryptology — CRYPTO ’01, Lecture Notes in Computer Science*, vol. 2139, Springer-Verlag, pp. 260–274, 2001.

[57] E. Gafni, J. Staddon, and Y. L. Yin, “Efficient Methods for Integrating Traceability and Broadcast Encryption,” *Advances in Cryptology — CRYPTO ’99, Lecture Notes in Computer Science*, vol. 1666, Springer-Verlag, pp. 372–387, 1999.

[58] J. Garay, R. Gennaro, C. Jutla, and T. Rabin, “Secure Distributed Storage and Retrieval,” *Proceedings of the 11th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science*, vol. 1320, Springer-Verlag, pp. 275–289, 1997.

[59] B. Gassend, E. Suh, D. Clarke, M. van Dijk, and S. Devadas, “Caches and Merkle Trees for Efficient Memory Authentication,” *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, February 2003.

[60] T. Gilmont, J.-D. Legat, and J.-J. Quisquater, “An Architecture of Security Management Unit for Safe Hosting of Multiple Agents,” *Proceedings of the International Workshop on Intelligent Communications and Multimedia Terminals*, pp. 79–82, November 1998.

[61] R. Golliver and D. Bhandarkar, “The Road to Billion Transistor Processor Chips in the Near Future,” available at http://www.lanl.gov/orgs/ccn/salishan2003/pdf/golliver.pdf, 2003.

[62] P. Gutmann, “An Open-source Cryptographic Coprocessor,” *Proceedings of the 2000 USENIX Security Symposium*, 2000.

[63] D. Harkins and D. Carrel, “The Internet Key Exchange (IKE),” IETF RFC 2409, November 1998.

[64] L. Hornof and T. Jim, “Certifying Compilation and Run-time Code Generation,” *Proceedings of the ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, January 1999.

[65] K. J. Houle, G. M. Weaver, N. Long, and R. Thomas, “Trends in Denial of Service Attack Technology,” CERT Coordination Center, October 2001.

[66] Intel Corporation, *IA-64 Application Developer’s Architecture Guide*, Intel Corporation, 1999.

[67] Intel Corporation, *The IA-32 Intel Architecture Software Developer’s Manual, Volume 2: Instruction Set Reference*, 2001.

[68] Intel Corporation, “LaGrande Technology Architectural Overview,” available at http://www.intel.com/technology/security/, September 2003.

[69] International Organization for Standardization (ISO), *ISO 15408: Evaluation Criteria for IT Security*, available at http://www.iso.org, 1999.

[70] ITSEC Working Group, *ITSEC: Information Technology Security Evaluation Criteria*, Version 1.2, September 1991.

[71] D. R. Kaeli and P. G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proceedings of the International Symposium on Computer Architecture*, pp. 34–41, May 1991.

[72] J. Kahn, J. Komlos, and E. Szemeredi, "On the Probability that a Random $+/-1$ Matrix Is Singular," *Journal of the American Mathematical Society*, vol. 8, no. 1, pp. 223–240, January 1995.

[73] K. Kant, R. Iyer, and P. Mohapatra, "Architectural Impact of Secure Socket Layer on Internet Servers," *Proceedings of the 200 IEEE International Conference on Computer Design (ICCD)*, September 2000.

[74] S. Kent and R. Atkinson, "IP Authentication Header," IETF RFC 2402, November 1998.

[75] S. Kent and R. Atkinson, "IP Encapsulating Security Payload," IETF RFC 2406, November 1998.

[76] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," IETF RFC 2401, November 1998.

[77] Auguste Kerckhoffs, "La cryptographie militaire," *Le journal des sciences militaires*, vol. IX, pp. 5–38, January 1883, pp. 161-191, February 1883.

[78] A. Kiayias and M. Yung, "Breaking and Repairing Asymmetric Public-Key Traitor Tracing," *Proceedings of the ACM Workshop on Digital Rights Management*, 2002.

[79] A. Kiayias and M. Yung, "Self Protecting Pirates and Black-box Traitor Tracing," *Advances in Cryptology — CRYPTO '01*, *Lecture Notes in Computer Science*, vol. 2139, Springer-Verlag, pp. 63–79, 2001.

[80] A. Kiayias and M. Yung, “Traitor Tracing with a Constant Transmission Rate,” *Advances in Cryptology — EUROCRYPT ’02, Lecture Notes in Computer Science*, vol. 2332, Springer-Verlag, pp. 450–465, 2002.

[81] D. Kirovski, M. Drinic, and M. Potkonjak, “Enabling Trusted Software Integrity,” *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002.

[82] klog, “The Frame Pointer Overwrite,” *Phrack Magazine*, vol. 9, no. 55, Sept. 1999.

[83] P. Kocher, “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems,” *Advances in Cryptology — CRYPTO ‘96, Lecture Notes in Computer Science*, vol. 1109, Springer-Verlag, pp. 104–113, 1996.

[84] P. Kocher, J. Jaffe, and B. Jun, “Differential Power Analysis,” *Advances in Cryptology — CRYPTO ’99, Lecture Notes in Computer Science*, vol. 1666, Springer-Verlag, pp. 388–397, 1999.

[85] H. Krawczyk, “HMAC: Keyed-Hashing for Message Authentication,” IETF RFC 2104, February 1997.

[86] K. Kurosawa and Y. Desmedt, “Optimum Traitor Tracing and Asymmetric Schemes,” *Advances in Cryptology — EUROCRYPT ’98, Lecture Notes in Computer Science*, vol. 1403, Springer-Verlag, pp. 145–157, 1998.

[87] X. Lai and J. Massey, “A Proposal for a New Block Encryption Standard,” *Advances in Cryptology — EUROCRYPT ’90, Lecture Notes in Computer Science*, vol. 473, Springer-Verlag, pp. 389–404, May 1990.

[88] R. B. Lee, “Accelerating Multimedia with Enhanced Microprocessors,” *IEEE Micro*, vol. 15, no. 2, pp. 22–32, April 1995.

[89] R. B. Lee, “Multimedia Extensions for General-purpose Processors,” *Proceedings of the IEEE Workshop on Signal Processing Systems: Design and Implementation*, pp. 9–23, November 1997.

[90] R. B. Lee, “Precision Architecture,” *IEEE Computer*, vol. 22, no. 1, pp. 78–91, January 1989.

[91] R. B. Lee, “Subword Parallelism with MAX-2,” *IEEE Micro*, vol. 16, no. 4, pp. 51–59, August 1996.

[92] R. B. Lee, A. M. Fiskiran and A. Bubshait, “Multimedia Instructions in IA-64,” *Proceedings of the International Conference on Multimedia and Expo*, August 2001.

[93] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, “An Architectural Approach to Mitigating Distributed Denial of Service Attacks Resulting from Buffer Overflow,” Princeton University Department of Electrical Engineering Technical Report CE-L2001-003, November 2001.

[94] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, “Enlisting Hardware Architecture to Thwart Malicious Code Injection,” *Proceedings of the International Conference on Security in Pervasive Computing (SPC-2003)*, *Lecture Notes in Computer Science*, vol. 2802, Springer-Verlag, pp. 237–252, March 2003.

[95] R. B. Lee, P. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, “Architecture for Protecting Critical Secrets in Microprocessors,” *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, June 2005.

[96] R. B. Lee, R. L. Rivest, M.J.B. Robshaw, Z.J. Shi, and Y.L. Yin, “On Permutation Operations in Cipher Design,” *Proceedings of the International Conference on Information Technology (ITCC)*, vol. 2, pp. 569–577, April 2004.

[97] R. B. Lee, Z. Shi, and X. Yang, "Efficient Permutation Instructions for Fast Software Cryptography," *IEEE Micro*, vol. 21, no. 6, pp. 56–69, December 2001.

[98] R. B. Lee, X. Yang, and Z. J. Shi, "Validating Word-oriented Processors for Bit and Multi-Word Operations," *Proceedings of the Asia-Pacific Computer Systems Architecture Conference (ACSAC)*, pp. 473–488, September 2004.

[99] R. B. Lee, Z. Shi, and X. Yang, "How a Processor Can Permute n Bits in $O(1)$ Cycles," *Proceedings of Hot Chips 14 — A Symposium on High Performance Chips*, August 2002.

[100] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *Proceedings of ASPLOS-IX*, pp. 168–177, 2000.

[101] P. MacKenzie and M. Reiter, "Networked Cryptographic Devices Resilient to Capture," *Proceedings of the 22nd IEEE Symposium on Security and Privacy*, pp. 12–25, 2001.

[102] R. McEliece, "A Public-key Cryptosystem Based on Algebraic Coding Theory," DSN Progress Report, #42–47, Jet Propulsion Laboratory, 1978.

[103] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, "A Processor Architecture Defense against Buffer Overflow Attacks," *Proceedings of the IEEE International Conference on Information Technology: Research and Education (ITRE 2003)*, pp. 243–250, August 2003. Best Student Paper Award.

[104] J. P. McGregor and R. B. Lee, "Architectural Enhancements for Fast Subword Permutations with Repetitions in Cryptographic Applications," *Proceedings of the International Conference on Computer Design (ICCD 2001)*, pp. 453–461, September 2001.

[105] J. P. McGregor and R. B. Lee, “Architectural Techniques for Accelerating Subword Permutations with Repetitions,” *IEEE Transactions on Very Large Scale Integration Systems*, vol. 11, no. 3, pp. 325–335, June 2003.

[106] J. P. McGregor and R. B. Lee, “Performance Impact of Data Compression on Virtual Private Network Transactions,” *Proceedings of the IEEE Conference on Local Computer Networks (LCN 2000)*, pp. 500–510, November 2000.

[107] J. P. McGregor and R. B. Lee, “Performance Impact of Data Compression on Virtual Private Network Transactions (Extended Version),” Princeton University Department of Electrical Engineering Technical Report CE-L2000-002, May 2000.

[108] J. P. McGregor and R. B. Lee, “Protecting Cryptographic Keys and Computations via Virtual Secure Coprocessing,” *Proceedings of the Workshop on Architectural Support for Security and Antivirus held in conjunction ASPLOS-XI*, October 2004, and *Computer Architecture News*, vol. 33., no. 1, pp. 16–26, March 2005.

[109] J. P. McGregor and R. B. Lee, “Virtual Secure Coprocessing on General-purpose Processors,” Princeton University Department of Electrical Engineering Technical Report CE-L2002-003, November 2002.

[110] J. P. McGregor, Y. L. Yin, and R. B. Lee, “Efficient Traitor Tracing Using RSA,” Princeton University Department of Electrical Engineering Technical Report CE-L2003-004, May 2003.

[111] J. P. McGregor, Y. L. Yin, and R. B. Lee, “A Traceability Scheme for Broadcast Encryption Based on RSA: Security Proofs and Performance Comparisons,” Princeton University Department of Electrical Engineering Technical Report CE-L2005-003, May 2005.

[112] J. P. McGregor, Y. L. Yin, and R. B. Lee, “A Traitor Tracing Scheme Based on RSA for Fast Decryption,” *Proceedings of the 2005 Conference on Applied Cryptography and Network Security*, *Lecture Notes in Computer Science*, vol. 3531, Springer-Verlag, June 2005.

[113] A. J. Menezes, *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[114] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, LLC, Boca Raton, FL, 1997.

[115] Microsoft, “Next-Generation Secure Computing Base,” available at http://www.microsoft.com/resources/ngscb/, June 2004.

[116] D. Moore, C. Shannon, and K. Claffy, “Code-Red: A Case Study on the Spread and Victims of an Internet Worm,” *Proceedings of the 2002 Internet Measurement Workshop*, pp. 273–284, November 2002.

[117] D. Naor, M. Naor, and J. Lotspiech, “Revocation and Tracing Schemes for Stateless Receivers,” *Advances in Cryptology — CRYPTO ’01, Lecture Notes in Computer Science*, vol. 2139, Springer-Verlag, pp. 41–62, 2001.

[118] M. Naor and B. Pinkas, “Threshold Traitor Tracing,” *Advances in Cryptology — CRYPTO ’98*, *Lecture Notes in Computer Science*, vol. 1462, Springer-Verlag, pp. 502–517, 1998.

[119] National Bureau of Standards, “Data Encryption Standard,” FIPS Publication 46-2, December 1993.

[120] National Institute of Standards and Technology, “Advanced Encryption Standard,” FIPS Publication 197, November 2001.

[121] National Institute of Standards and Technology, “Secure Hash Standard,” FIPS Publication 180-2, August 2002.

[122] National Institute of Standards and Technology, “Security Requirements for Cryptographic Modules,” FIPS Publication 140-2, May 2001.

[123] S. Oberman, G. Favor, and F. Weber, “AMD 3DNow! Technology: Architecture and Implementations,” *IEEE Micro*, vol. 19, no. 2, pp. 37–48, 1999.

[124] A. Peleg and U. Weiser, “MMX Technology Extension to the Intel Architecture,” *IEEE Micro*, vol. 16, no. 4, pp. 42–50, August 1996.

[125] B. Pfitzmann, “Trials of Traced Traitors,” *Proceedings of the First Information Hiding Workshop – IHW ’96, Lecture Notes in Computer Science*, vol. 1174, Springer-Verlag, pp. 49–64, 1996.

[126] B. Pfitzmann and M. Waidner, “Asymmetric Fingerprinting for Larger Collusions,” *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 145–157, 1997.

[127] J. Postel, “Internet Protocol,” IETF RFC 791, September 1981.

[128] J. Postel, “Transmission Control Protocol,” IETF RFC 793, September 1981.

[129] N. Potlapally, S. Ravi, A. Raghunathan, and G. Lakshminarayana, “Optimizing Public-Key Encryption for Wireless Clients,” *IEEE International Conference on Communications (ICC)*, pp. 1050–1056, May 2002.

[130] B. Preneel, “Analysis and Design of Cryptographic Hash Functions,” Ph.D. dissertation, Katholieke Universiteit, Leuven, Belgium, January 1993.

[131] R. L. Rivest, “The MD5 Message Digest Algorithm,” IETF RFC 1321, April 1992.

[132] R. L. Rivest, “The RC5 Encryption Algorithm,” *Proceedings of the 2nd Workshop on Fast Software Encryption*, *Lecture Notes in Computer Science*, vol. 1008, Springer-Verlag, pp. 86–96, December 1994.

[133] R. L. Rivest, A. Shamir, and L. M. Adleman, “A Method for Obtaining Digital Signatures and Public-key Cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, February 1978.

[134] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, “The RC6 Block Cipher,” NIST AES Proposal, August 1998.

[135] RSA Security, Inc., “PKCS #11 v2.11: Cryptographic Token Interface Standard,” available at http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/, Nov. 2001.

[136] R. Safavi-Naini and Y. Yang, “Sequential Traitor Tracing,” *Advances in Cryptology — CRYPTO ’00*, *Lecture Notes in Computer Science*, vol. 1880, Springer-Verlag, pp. 316–332, 2000.

[137] S. Sair and M. Charney, “Memory Behavior of the SPEC2000 Benchmark Suite,” IBM T. J. Watson Technical Report RC-21852, Oct. 2000.

[138] The SANS Institute, “The SANS/FBI Twenty Most Critical Internet Security Vulnerabilities,” available at http://www.sans.org/top20/, October 2002.

[139] B. Schneier, *Applied Cryptography*, Second Edition, John Wiley and Sons, New York, 1996.

[140] B. Schneier, *Secrets and Lies*, New York: John Wiley and & Sons, Inc., 2000.

[141] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, “Twofish: A 128-bit Block Cipher,” NIST AES proposal, June 1998.

[142] S. Setia, S. Koussih, S. Jajodia, and E. Harder, "Kronos: A Scalable Group Re-keying Approach for Secure Multicast," *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pp. 215–228, 2000.

[143] C. E. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical Journal*, vol. 28, pp. 656–715, October 1949.

[144] Z. J. Shi, *Bit Permutation Instructions: Architecture, Implementation, and Cryptographic Properties*, Princeton University Department of Electrical Engineering Ph.D. Thesis, 2004.

[145] Z. J. Shi and R. B. Lee, "Bit Permutation Instructions for Accelerating Software Cryptography," *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 80–86, July 2000.

[146] Z. Shi, X. Yang and R. B. Lee, "Arbitrary Bit Permutations in One or Two Cycles," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2003)*, pp. 237–247, June 2003.

[147] V. Shoup, "OAEP Reconsidered," *Advances in Cryptology — CRYPTO '01, Lecture Notes in Computer Science*, vol. 2139, Springer-Verlag, pp. 239–259, 2001.

[148] A. Silverberg, J. Staddon, and J. Walker, "Efficient Traitor Tracing Algorithms using List Decoding," *Advances in Cryptology — ASIACRYPT '01, Lecture Notes in Computer Science*, vol. 2248, Springer-Verlag, pp. 175–192, 2001.

[149] G. J. Simmons, "A Weak Privacy Protocol Using the RSA Cryptosystem," *Cryptologia*, vol. 7, no. 2, pp. 180–182, 1993.

[150] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms," *Proceedings*

of the 31st International Symposium on Microarchitecture (MICRO-31), November 1998.

[151] R. E. Smith, *Authentication: From Passwords to Public Keys*, Boston, MA: Addison-Wesley, 2002.

[152] S. W. Smith, E. R. Palmer, S. H. Weingart, “Using a High-Performance, Programmable Secure Coprocessor,” *Proceedings of the International Conference on Financial Cryptography*, pp.73–89, 1998.

[153] S. W. Smith and S. H. Weingart, “Building a High-Performance, Programmable Secure Coprocessor,” *Computer Networks*, vol. 31, no. 8, pp. 831–860, April 1999.

[154] SPARC International, Inc., *The SPARC Architecture Manual*, 1992.

[155] The Standard Performance Evaluation Corporation, http://www.spec.org/, November 2001.

[156] J. Steiner, C. Neuman, and J. I. Schiller, “Kerberos: An Authentication Service for Open Network Systems,” *Proceedings of the 1998 Winter USENIX Conference*, pp. 191–202, March 1988.

[157] D. Stinson, *Cryptography: Theory and Practice*, Boca Raton, FL: CRC Press, LLC, 1995.

[158] D. Stinson and R. Wei, “Combinatorial Properties and Constructions of Traceability Schemes and Frameproof Codes,” *SIAM Journal on Discrete Mathematics*, vol. 11, no. 1, 1998.

[159] D. Stinson and R. Wei, “Key Preassigned Traceability Schemes for Broadcast Encryption,” *Proceedings of Selected Areas in Cryptology – SAC ’98, Lecture Notes in Computer Science*, vol. 1556, Springer-Verlag, pp. 144–156, 1999.

[160] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," *Proceedings of the 17th International Conference on Supercomputing (ICS)*, 2003.

[161] "Timeline of Linux Development," *Wikipedia*, available at http://en.wikipedia.org/wiki/Timeline_of_Linux_development, accessed January 2005.

[162] M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He, "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16, no. 4, pp. 10–20, August 1996.

[163] Trusted Computing Group, http://www.trustedcomputinggroup.org, June 2004.

[164] J. D. Tygar and B. Yee, "Dyad: A System for Using Physically Secure Coprocessors," Carnegie Mellon University Technical Report CMU-CS-91-140R, May 1991.

[165] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code," *Proceedings of the 2000 Annual Computer Security Applications Conference (ACSAC 2000)*, December 2000.

[166] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 156–169, 2001.

[167] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," *Network and Distributed System Security Symposium*, February 2000.

[168] D. Wallach, "Copy Protection Technology Is Doomed," *IEEE Computer*, vol. 34, no. 10, pp. 48–49, Oct. 2001.

[169] Y. Watanabe, G. Hanaoka and H. Imai, "Efficient Asymmetric Public-Key Traitor Tracing without Trusted Agents," *Progress in Cryptology – CT-RSA 2001*, *Lecture Notes in Computer Science*, vol. 2020, Springer-Verlag, pp. 392–407, 2001.

[170] C. F. Webb, “Subroutine Call/Return Stack,” *IBM Technical Disclosure Bulletin*, vol. 30, no. 11, April 1988.

[171] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, 2nd ed., Reading, MA: Addison-Wesley, 1993.

[172] R. S. Winternitz, “Producing One-way Hash Functions from DES,” *Advances in Cryptology — CRYPTO ’83*, pp. 203–207, 1984.

[173] X. Yang and R. B. Lee, “Fast Subword Permutation Instructions Using Omega and Flip Network Stages,” *Proceedings of the International Conference on Computer Design*, pp. 15–22, September 2000.

[174] X. Yang, M. Vachharajani and R. B. Lee, “Fast Subword Permutation Instructions Based on Butterfly Networks,” *Proceedings of SPIE: Media Processors 2000*, vol. 3970, pp. 80–86, January 2000.

[175] E. Young, libdes, available at http://www.shmoo.com/crypto/, accessed April 2005.