Virtual Secure Coprocessing on General-purpose Processors

John P. McGregor and Ruby B. Lee
Princeton Architecture Laboratory for Multimedia and Security (PALMS)
Department of Electrical Engineering
Princeton University
{mcgregor, rblee}@ee.princeton.edu

Abstract

Cryptographic processing is a critical component of secure Internet-connected computing systems. Furthermore, the methods employed to manage, store, and exercise a user's cryptographic keys greatly affect the security offered by cryptographic operations. Software-only user key management schemes contain numerous security weaknesses. Thus, many systems protect keys with distributed protocols or supplementary hardware devices, which include smart cards and cryptographic coprocessors. However, these key protection mechanisms suffer from various combinations of user inconvenience, inflexibility, performance penalties, and high cost.

In this paper, we propose architectural features for general-purpose processors that facilitate virtual secure coprocessing. We describe modest hardware modifications and a trusted software library that allow general-purpose processors to perform flexible, high-performance, and protected cryptographic computation. The hardware additions include a small key store in the processor, encryption engines at the cache-memory interface, a few new instructions, and minor hardware platform modifications. With these enhancements, users can store, transport, and employ their secret keys to safely complete cryptographic operations in the presence of insecure software. In addition, we enable users to securely access their keys on any Internet-connected computing device (that includes our modifications) without requiring auxiliary hardware such as smart cards.

1. Introduction

A wide variety of systems have been designed to protect network communications, stored data, and access to electronic resources. These systems generally employ cryptographic algorithms to provide many critical security functions such as confidentiality, integrity, and authentication. The security provided by most cryptographic algorithms depends on the secrecy and integrity of small pieces of data known as *cryptographic keys*. For the purposes of this discussion, cryptographic keys may consist of any secret information used to perform encryption or authentication, such as AES keys [21], private encryption exponents, passwords, 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 platforms such as information appliances and personal computers, users often employ their keys to perform cryptographic operations "in the clear". This means that the users temporarily or permanently store their secret keys and associated sensitive information in unprotected system RAM or other storage devices. When a user exercises secret keys in the clear, an unauthorized party may inspect the contents of memory to obtain the secret keys by defeating the security polices of the operating system or certain software applications. Such system penetration can be realized by exploiting one of the numerous security vulnerabilities that occur in software [9, 26]. 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 secret key compromise, 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 improper disclosure of medical records, theft of private correspondence, and access to copyrighted audio and video. Furthermore, when an attacker employs a compromised secret key to reveal a user's sensitive data, no measure can be taken to re-conceal that data. If cryptographic keys enable access to valuable assets such as online banking accounts or corporate IP repositories, the results of key compromise can be truly devastating.

The management and protection of cryptographic keys is therefore a critical component of secure computing systems. Due to the numerous security vulnerabilities that continue to plague software, local software-only key protection techniques are unsatisfactory. A software intrusion that exploits a common may enable an attacker to remotely penetrate a network-connected device and expose keys that provide access to all of a user's secrets and information. Therefore, the most secure key management schemes involve a set of distributed hosts or a protected hardware device.

1.1. Our Proposal

In this paper, we describe new architectural and software enhancements for general-purpose processors and platforms that protect users' secret keys during storage and use. With processor transistor counts approaching 1 billion, we believe that a small percentage of the transistor budget should be applied to improve security. Our enhancements effectively enable the general-purpose processor to operate as a *virtual secure coprocessor* when needed. We identify a minimal set of protected registers, system states, and algorithms to enable secure and efficient key utilization and storage in the presence of insecure networks, application software, and operating systems. We define a Concealed Execution Mode for general-purpose processors that protects computations involving users' secret keys. In addition, we describe a special trusted software library, the Cryptographic Operations Library, which is used in the Concealed Execution Mode to safely perform computation using secret keys. To further improve the security offered by virtual secure coprocessing, we propose methods for securely transporting key material to protected storage within the processor for future use in the Concealed Execution Mode. The performance and implementation costs of our enhancements are modest. Users can employ concealed execution while simultaneously running non-secured threads on a system. Also, we only require low cost changes to the general-purpose processor and the hardware platform, whereas some existing schemes require the addition of special trusted chips or security modules.

Our solution provides many benefits for individual users. First and foremost, we provide high security. Users can employ secret keys to perform computations on general-purpose platforms without leaking any sensitive key material to the insecure software and hardware environment. We seek to ensure the security of all cryptographic keys, whereas some key management schemes only protect limited classes of keys such as RSA keys. In addition, our system furnishes ubiquitous and convenient key access. Users can securely access their cryptographic key ring from *any* network-enabled device (that contains our enhancements) without having to carry and use a smart card or other protected, auxiliary hardware device. These network-enabled devices also do not need to be pre-authorized in order to securely utilize secret keys, as may be required in existing systems. Furthermore, since the cryptographic operations that we provide to applications are implemented in software instead of hardware, the system can support a wide range of security functions. Users also benefit from the higher performance of general-purpose processors as opposed to the low performance of resource-constrained cryptographic processors found in smart cards and other cryptographic tokens.

Examples of application targets for our proposal include wireless computing devices and distributed shared file systems. With mobile Internet-connected devices, individual users often require protected use of many cryptographic keys to access sensitive information or to communicate securely. However, these individual users may desire to safely access their keys from a wide variety of devices without requiring auxiliary components such as smart cards. In corporate environments, multiple users may need to access confidential stored data from many computers and information appliances. These users can benefit from key management services that are built into all computing devices and that enable protected use of keys that may be shared among several individuals.

1.2. Outline

The rest of the paper is organized as follows. In Section 2, we discuss prior related work. In Section 3, we describe our design approach and the high-level implications of our proposed solution. In Section 4, we present virtual secure coprocessing. We explain how a general-purpose processor can also serve as a virtual secure coprocessor via user initialization, device initialization, and protected operation. In Section 5, we describe the details of our proposed processor, hardware platform, OS, and cryptographic software features needed to achieve our security goals. We investigate the performance impact of our proposal in Section 6, and we conclude in Section 7. In Appendix A, and we discuss architectural alternatives and extensions to our proposal.

2. Related Work

Researchers have proposed several hardware and software techniques for protecting cryptographic keys against unauthorized observation, modification, and use. We now 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 (e.g., [6, 14, 17, 20, 30, 31]), whereas other techniques protect users from physical theft and attacks by malicious code (e.g., [5, 10, 11, 12, 18, 20, 22, 28, 29, 30, 31, 32]).

2.1. Software-based Techniques

Distributed software-only approaches seek to protect certain types of cryptographic keys by requiring an adversary to quickly compromise several hosts or by enabling effective revocation mechanisms when key information is exposed. Some proposals allow a user to reconstruct cryptographic keys directly preceding use by engaging in a secure protocol that involves the participation of several servers (e.g., [11, 12, 22]). Due to the threshold cryptographic techniques involved, an adversary must compromise multiple servers in order to expose a key. In other solutions, users can perform certain cryptographic operations that employ secret keys with the aid of untrusted servers; when a client device or an untrusted server is compromised, the secret keys can be disabled (e.g., [18]). These and other distributed schemes effectively defend against several attacks involving limited classes and types of keys. Unlike our proposal, however, these schemes do not prevent attacks in which an adversary attempts to expose arbitrary secret keys while in use on a client platform in the presence of potentially insecure software.

2.2. Secure Coprocessors and Cryptographic 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 [6]. Since that time, researchers have proposed a rich variety of applications and architectures for such hardware (e.g., [31]). 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 [10, 28, 29]. These products offer exceptional physical security for cryptographic keys, but they are too costly, inconvenient, and bulky for mobile computers and information appliances. For instance, an IBM secure coprocessor subsystem can cost more than an average desktop personal computer. Remote access to the secure coprocessor and its secrets may be limited, as the secrets are stored within a single device on a single machine. Also, the secure coprocessor module is rather large, which would prohibit incorporation into small information appliances.

Extremely low-cost, portable alternatives to cryptoprocessors and secure coprocessors are cryptographic tokens. These devices include smart cards and other small, physically tamper-resistant hardware components [3]. Researchers have also shown that information appliances such as PDAs can serve as flexible cryptographic tokens [5]. Some tokens simply protect user secrets by requiring a password to access the information stored within the token, and other tokens perform cryptographic operations using the stored secrets without leaking key information to the untrusted environment [3]. These devices cannot provide the same degree of security as powerful cryptoprocessors, but they cost much less and they facilitate increased user convenience. For users who are willing to carry these devices, cryptographic tokens can provide highly portable access to secret keys. However, these constrained devices have restricted capabilities: performance can be poor and the number of supported cryptographic operations is often limited. Also, physical tamper resistance is difficult to implement. Many low-cost attacks exist that enable the extraction of secrets from cryptographic tokens [4, 16].

Our proposal differs from these cryptographic processors and tokens, as we facilitate the high-performance and secure use of key rings from any Internet-connected device. In addition, our system does not require any potentially expensive, auxiliary hardware devices such as on-board coprocessors or smart cards; we only require low-cost modifications to the processor and platform.

2.3. Trusted Computing Bases

The Trusted Computing Group (TCG) [30], which was formerly known as the Trusted Computing Platform Alliance (TCPA), and Microsoft's Next Generation Secure Computing Base (NGSCB) [20], which was formerly known as Palladium, seek to provide a trusted computing base for general-purpose and mobile computing devices. The TCG supports system attestation, limited protection of user secrets, and secure booting. NGSCB seeks to enable the features of the TCG and also provide resources for secure (i.e., validated and isolated) code execution. These systems embed secret information that is inaccessible to the end user in tamper-resistant hardware modules such as on-board cryptographic coprocessors. With operating system support, these modules use the secret information to complete operations such as verifying the integrity of installed software and preventing unauthorized access to copyrighted media and proprietary code.

As currently defined, these systems only provide limited protection for user cryptographic keys, however. A user can employ certain hardware resources to encrypt a sensitive key for storage, but keys must be used in the clear on the general-purpose processor to perform computations. Although the TCG and NGSCB may ensure that cryptographic keys are only released to trusted environments, these trusted environments might not be secure. That is, "trusted" certainly does not imply "dependable", and the trusted software environment is vulnerable to software bugs that could lead to the unauthorized exposure of sensitive cryptographic keys. In addition, TCG and NGSCB do not defend against certain hardware-based attacks. For instance, by physically monitoring and/or modifying data in the system buses and main memory, some security features of the trusted computing bases can be defeated. The Aegis project [32] seeks to address this problem by cryptographically protecting certain code and data that enters or exits the general-purpose processor. Intel's LaGrande project may also address such hardware attacks, but technical details are not yet available. Also, TGB and NGSCB individually encrypt user keys for unique devices; users must complete an authorization process to enable a new device to employ their keys.

Our solution differs from NGSCB-like systems as follows. First, unlike proposed trusted computing bases, we provide mechanisms for protecting user keys at all times: keys in our system are protected against potentially insecure hardware or software during transport, storage, or computation. Second, we facilitate the secure use of key rings from any Internet-connected device, and key ring access from a particular device in our system is not dependent on any pre-authorization procedure for that device. Third, our system does not require any auxiliary hardware devices such as on-board coprocessors or smart cards. Since the hardware components of our proposal only entail inexpensive modifications in the processor and platform, we can provide high security at low costs.

We emphasize that our proposal is ***not*** designed to replace the TCG and NGSCB components. By enabling additional protection for the most sensitive pieces of information (i.e., cryptographic keys), our proposal complements rather than supplants the security services provided by these systems. Unlike the TCG and NGSCB, our solution does not provide services such as secure bootup and attestation. These services are essential to achieving robust system security, and therefore our solution should enhance rather than replace these trusted computing base proposals.

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 [13, 15, 32]. In addition, by adding encryption and data authentication capabilities to general-purpose processors, it is possible to enable shielded program execution [14, 17, 32]. Such systems, e.g., eXecute Only Memory (XOM), preclude unauthorized modification and observation of software by unsecured or untrusted components outside of the processor chip. This involves obfuscating and authenticating instructions and program dataflow. The primary objective of shielded execution in XOM and related proposals is the prevention of software piracy and exposure of valuable proprietary code. Whereas XOM enables external parties to protect sensitive information when the external parties' software is being employed on an untrusted user's machine, our proposal enables a user to protect his secret information on his machine from external parties. Although some components of XOM and our proposal overlap, these two solutions differ in fundamental design goals, benefits to users, and several implementation issues.

We list a few of the security and implementation distinctions here. First, XOM does not support the mechanisms we enable to securely transport user secrets to protected storage within the processor. In addition, although XOM can obfuscate the program execution, XOM cannot restrict the operations performed by software. XOM allows any software to potentially access and employ all of a user's secrets, which enables malicious code or buggy programs (such as the SSL module in a web browser) to reveal sensitive user information. In contrast, our proposed system defends against such attacks by limiting access to user keys and related information. Concerning implementation, XOM requires public-key encryption at run-time, whereas our proposal only employs symmetric-key techniques during program execution. Also, XOM requires a significant amount of on-chip protected memory. In order to ensure that the asymmetric encryption routines will not run prohibitively slowly, the on-chip protected memory for XOM must be several kilobytes in size. This rivals the size of the L1 caches in many processors. In our proposal, however, the required on-chip protected storage is less than 100 bytes. Our virtual secure coprocessing architecture provides a valuable, novel security framework into which elements of XOM and other general-purpose architecture developments can be added and shared.

3. Protecting Cryptographic Keys

We first describe the characteristics and structure of a user's cryptographic key ring. Figure 1a shows an example of the hierarchical organization of a cryptographic key ring, which can potentially contain thousands of keys. A key ring includes a single master key that is used to encrypt and authenticate the integrity of all of the first level keys. In this paper, we define master keys to be 128-bit keys for use in symmetric-key cryptographic algorithms. Since all the keys in a key ring are cryptographically protected by the 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 or compromise. The security of the key ring depends on the measures taken to protect the master key.

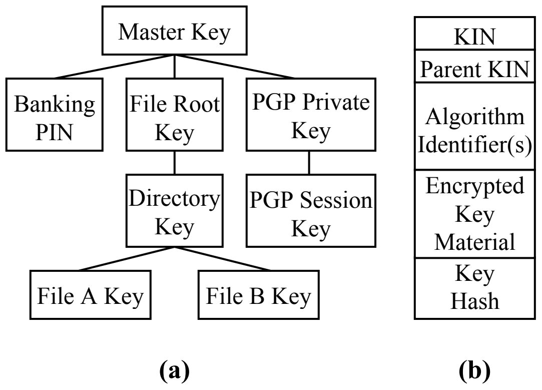

Figure 1. (a) Key ring and (b) key structure

Figure 1b depicts the data organization of an individual key. Each key consists of a key identification number (KIN), the key's parent KIN, an algorithm identifier, the key itself (in encrypted form), and the key hash. The KIN is a non-secret 128-bit integer that uniquely identifies the key. The key's parent KIN is the identifier of the key used to encrypt and authenticate the current key, and the algorithm identifier specifies the algorithm (or set of algorithms) permitted to use the key. The key hash is the keyed cryptographic hash message authentication code (HMAC) for the entire key data structure (minus the key hash) that can be used to verify the integrity of the key. This guards against adversaries that seek to forge and inject bogus keys into a user's key ring. Examples of algorithms that can be used to perform the encryption and hashing include AES and SHA-1, respectively [19].

Our goals are to enable the secure storage and ensure the secure utilization of a user's secret keys by special trusted software routines on general-purpose processors. First, we define the terms "secure storage" and "secure utilization" in the context of potential attacks. The security perimeter of a computing device is the boundary that separates the trusted domain from the untrusted environment. Protection is assumed for any information stored or utilized within the security perimeter. In most systems, the boundary is defined as the network or I/O interfaces of a device, as depicted by the dashed line in Figure 2a. This boundary is vulnerable to two classes of attacks, however: physical probing of the device and software-based attacks. Physical probing includes passive and active attacks on the device following theft in which an adversary attempts to physically extract secrets from the device. For example, the attacker may monitor components inside the security perimeter (such as a PCI bus) to discover sensitive data.

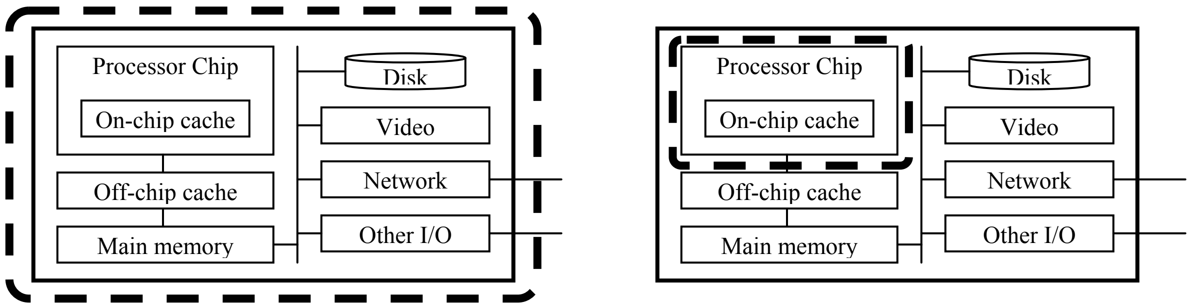

Figure 2. (a) Traditional and (b) proposed security perimeters

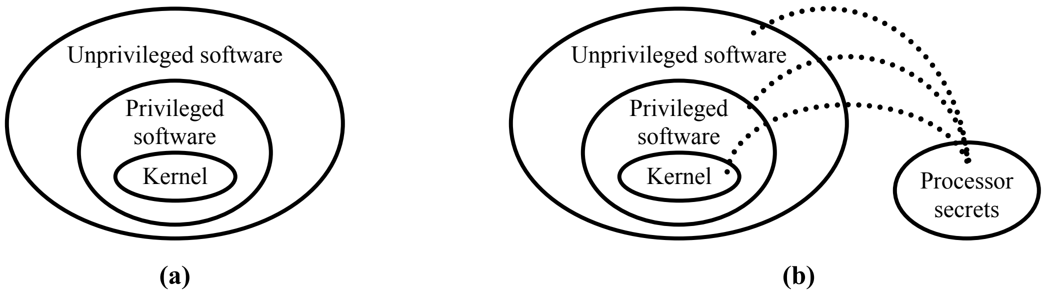

Figure 3. (a) Traditional and (b) proposed access control paradigms

Software-based attacks include both local software attacks following the physical theft of the device and remotely launched, network-enabled attacks that can occur at any time. These attacks can expose sensitive information by exploiting the traditional access control paradigm, which we depict in Figure 3a. The OS kernel (included in the innermost ellipse in Figure 3a) enjoys access to all information and resources in the system. Consequently, a software-based attack that compromises the kernel can access all sensitive information available to the system. Because of increasing network connectivity and the escalation of software security vulnerabilities, remotely launched software attacks are our principal concern. Although we hope to prevent some physical attacks, our efforts are focused on software-based attacks.

We propose protecting users' cryptographic keys by changing the device security perimeter and by modifying the traditional access control paradigm. We restrict the security perimeter for cryptographic keys in the system to the physical boundary of the general-purpose processor chip, as shown in Figure 2b. With respect to secret keys, memory that is off the processor chip, network interface cards, disks, buses, and any other peripherals are treated as being insecure and untrusted. We do not require physical tamper resistance of the processor chip packaging to thwart more sophisticated physical attacks, however. Although this added layer of protection is desirable, implementing physical tamper resistance for a general-purpose processor may be difficult: research has demonstrated that motivated adversaries can defeat such defenses [4, 16].

In addition, we create a new disjoint region in the access control paradigm, as shown in Figure 3b. The new region consists of processor-protected secrets that are inaccessible to the OS kernel and application software. 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 3b. The new region is not included within the kernel ellipse because operations that are permitted to execute within the new region do not require and should not be allowed to access all information in the system.

To implement the changes to the security perimeter and access control paradigm, we must defend against hardware-based and software-based attacks that involve one or both of the following:

- Unauthorized exposure or corruption of *data* that represents secret keys or that can be used to infer nontrivial information concerning secret keys
- Corruption, unauthorized insertion, or unauthorized execution of *code* that directly performs computations on secret keys

Since all hardware external to the processor is untrusted, some manifestations of the these attacks include:

- Software or physical probing attacks that seek to explicitly expose or modify secret keys that would normally reside in external memory or on disks

- Inspection of information leaked from the processor that enables an external observer to determine the values of user and device secrets
- The injection of malicious code that may reveal user or device secrets during cryptographic computations
- Application software or operating system software eavesdropping of a user's secret key input to a device

We note that our proposed system can detect and respond to data and code corruption but not necessarily prevent such corruption. That is, we do not claim resilience against all denial of service attacks.

4. Virtual Secure Coprocessing

We realize the new access control paradigm and the new security perimeter by enabling what we call *virtual secure coprocessing* (VSCoP) in general-purpose computing platforms. A virtual secure coprocessor is a general-purpose processor that functions as a secure coprocessor when needed. We provide new processor architectural and software features that enable a user to safely employ cryptographic keys in the presence of insecure software. We now present an introduction to the components and processes involved in our proposal.

We build the virtual secure coprocessor around two secrets stored 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 and is maintained in volatile memory. The processor employs the device secret to perform cryptographic and security functions that enable protected storage and utilization of the user's secret keys. The device secret is stored in non-volatile memory and never leaves the processor die. With these secrets, utilizing the virtual secure coprocessor involves three steps: device initialization, user initialization, and protected operation.

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 (COL), which is the only software module that will be permitted to access users' keys. To install this library, the user first clears the device secret stored in the processor by writing all zeros to it. Then, the processor will generate a new device secret, and this secret will be used to authenticate the library using message authentication codes based upon keyed one-way cryptographic hash functions [19]. This guards against attacks in which an adversary attempts to modify the trusted COL code. Note that the device does not need to be fresh from the factory, however. The new user can employ a previously used device to securely store and utilize his cryptographic key ring without the risk of exposing his secrets or the previous user's secrets.

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. The master key is the output of a cryptographically-strong one-way hash of a user-supplied passphrase, biometric data, and/or information supplied by a smart card. We wish to liberate users from smart card dependence, so the master key should generally be based upon a combination of a passphrase and biometric data. In the remainder of this paper, we assume that the master key is based upon a passphrase only. The security of the system therefore depends on the secrecy and quality of the passphrase used to generate the master key. Hence, users should carefully select passphrases with sufficient entropy to thwart off-line attacks [27]. 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 Internet-enabled device.

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. The processor or platform hardware then computes the user's master key without OS intervention, and the processor stores the result as the user secret. Next, when a software application needs to perform a cryptographic operation 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.

The Concealed Execution Mode (CEM) enables software to securely employ the user secret within the processor to perform cryptographic operations. In order to prevent a potential attacker from exposing any user secrets during the CEM, the processor must maintain the secrecy and integrity of all sensitive data that is available to other processes or is released from the processor chip. Sensitive data includes key bits, intermediate results of cryptographic computations, or processor state information that may reveal the values of user secrets. We prevent

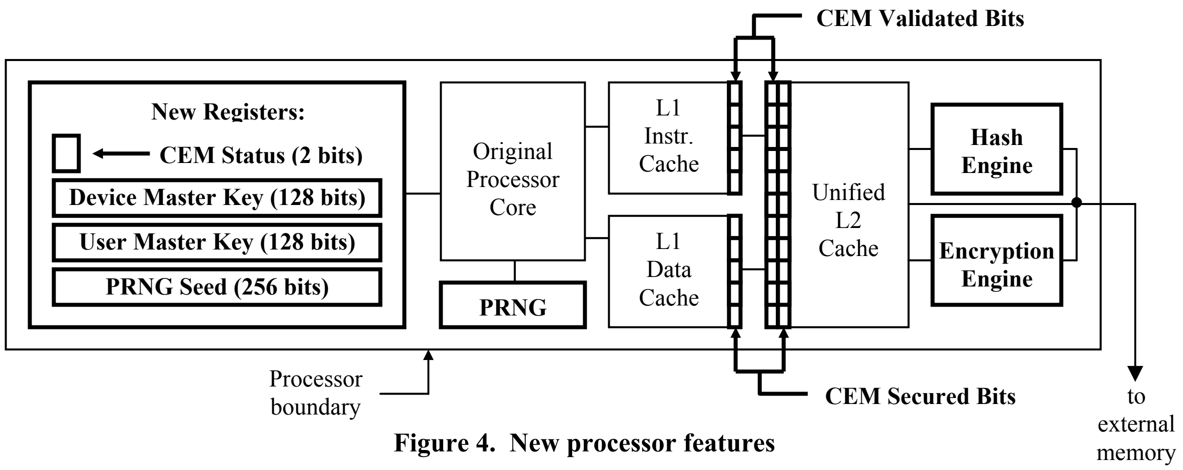

Figure 4. New processor features

unauthorized modification or transposition of data using the same hash techniques that we employ to verify instructions in the COL. To preclude attackers from observing the contents of concealed data, we use encryption to protect data that leaves the security perimeter of the chip [19].

Invoking the Concealed Execution Mode does not require the suspension of ordinary threads. Our proposal enables secure context switching between CEM and non-CEM threads; multitasking capabilities are not sacrificed. In addition, the user can employ the CEM to securely perform cryptographic computations without the participation of an auxiliary hardware device such as a smart card. A user's secret keys are not bound to any particular device, so a user can successfully and securely employ his cryptographic key ring from any computing device without conducting a pre-authorization procedure. Furthermore, we only require hardware-based symmetric-key cryptographic operations to implement the CEM. Slower, more expensive public-key cryptographic algorithms are not needed in hardware.

5. Architectural Enhancements

5.1. New Processor Features

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 4 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 and device secrets. We implement a minimum of 4 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 any of these four registers to exit the processor in unsecured (i.e., unencrypted and unauthenticated) form. Also, none of these register values are set at the factory; the register contents are defined by the end user in the field.

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 the CEM is in use for the current instruction stream and whether any thread on the system is currently employing the CEM. We do not need or want to preserve these two registers in the device when power is turned off, so we implement these registers using volatile SRAM. When power is removed, the contents of these registers will be drained (i.e., zeroized).

The Device Master Key, which is used to authenticate software, 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. Rewriteable non-volatile memory is probably the most difficult aspect of implementing our proposed virtual secure coprocessor in a microprocessor chip.

The other new components support concealed execution of trusted COL software. The processor performs the hash verification of trusted 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 HMAC of the data or instruction memory address, the secret Device Master Key, and the data or instruction cache line itself. This keyed HMAC can be a 16-byte AES-CBC-MAC [19, 21], 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 code and data, respectively. 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 percent.

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 HMAC, 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 [21] or some other symmetric-key encryption algorithm in cipher block chaining (CBC) mode [19]. 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 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.

In addition, we implement a simple address translator in hardware that converts hashed code and data addresses to and from regular code addresses so programs are not required to accommodate the awkward 16-byte hash values. Compilers therefore do not need to modify control flow or data structures to deal with 16-byte code and data "holes". However, the trusted COL software must take precautions to avoid inadvertently writing CEM secured data to regions of memory that are not allocated for concealed execution. In addition, certain branch instructions in the COL must be modified at compile-time to accommodate the address translator and avoid branching to incorrect targets.

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 discuss solutions for this problem in Appendix A. Furthermore, an attacker could 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 and any other information related to instruction sequence from external observation. We achieve this goal by never allowing such sensitive information to reach the processor package's pins while in the Concealed Execution Mode. In addition, we must disable testing scan chains and other processor hardware debugging features that may dump secret information from the processor during the CEM. 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 pseudorandom number generator can all be implemented using a single AES module, which requires as few as 25,000 gates [2]. 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 complexity is small. In addition, as described in Section 6, processor simulations indicate that the performance impact of the modifications is negligible.

5.2. New Instructions

We extend the Instruction Set Architecture (ISA) with new instructions to employ the new processor features. These instructions include `begin_cem`, `end_cem`, `cem_store`, `cem_load`, `device_key_mv`, and `user_key_mv`. Some of these instructions operate similarly to ISA additions described in [17]. We summarize the functionality of our proposed instructions in Table 1.

Table 1. New instructions

Instruction	Function
`begin_cem`	Enters the CEM. CEM Status register bits are set. All subsequently fetched instructions are cryptographically validated before execution.
`end_cem`	Exits the CEM. CEM-secured cache lines invalidated; general-purpose registers are zeroized. 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 from/to a register to/from individually addressable 64-bit chunks of the User Master Key.

At device initialization time, `device_key_mv` is used to securely seed the PRNG seed and Device Master Key registers. The `device_key_mv` instruction cannot be employed to read the contents of those registers, however. All operations that require reading the Device Master Key and PRNG seed registers are implemented in processor hardware, not software. When a user logs into a device, the master key of his key ring is securely computed and stored in the processor's User Master Key register via hardware with `user_key_mv`. Also, a function can obtain bits of the User Master Key via the `user_key_mv` instruction. Only COL software functions running in the CEM are privileged to employ that instruction.

When an application wishes to enter the Concealed Execution Mode by calling a function in the COL, the `begin_cem` instruction is executed to initiate virtual secure coprocessing on the general-purpose processor. In this paper, we only allow one process to employ the CEM at any given time to avoid complexities caused by sharing secured memory. If bit 1 of the CEM Status register equals 1 when a `begin_cem` instruction is issued, then the CEM is already in use, and therefore the processor denies the CEM request. Otherwise, both bits of the CEM Status register are set to ones, and the processor begins concealed execution. Bit 0 of the CEM Status register is used to perform secure context switching, which we describe below. 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.

In the CEM, privileged software securely transfers data to and from memory using the `cem_load` and `cem_store` instructions. These instructions prevent spoofing or 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 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.

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 zeroized, and both bits of the CEM Status register are reset to 0. Cache lines that contain secured CEM data are invalidated rather than flushed to prohibit reuse of results from previous CEM invocations to prevent data replay attacks.

5.3. New Hardware Platform Features

The new processor features facilitate most of the concealed execution functionality, but we have yet to address the issue of loading a user's master key into the processor. As explained above, the master key is a function of a passphrase and perhaps biometric and smart card information. We wish to liberate users from smart card dependence, so the authentication information should generally be a combination of a passphrase and biometric data. The most straightforward approach for transferring such information to the processor would be OS mediation. That is, the OS would prompt the user for the authentication information, and the OS would hash this information and pass the resulting master key to the processor. The problem with this method is the fact that OS software is insecure; that is a primary reason for implementing the proposed security enhancements.

Instead, the hardware platform (rather than the OS) should assume responsibility for gathering and hashing the user authentication information. We propose a hardware mechanism with which the platform could temporarily prevent keyboard or similar input from reaching OS I/O buffers. The platform would instead send these user inputs directly to the processor chip. The processor could then hash the information to obtain the user's master key. A user can initiate this procedure by pressing a special "Authenticate" button on the device, and the OS could provide software support that would indicate that authentication information is being received.

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

Upon receiving a used or new device, one may desire to reset the device secrets in order to guarantee that neither the factory nor a previous owner will have knowledge of the PRNG seed or the Device Master Key. Thus, we must provide support for resetting the Device Master Key and the PRNG seed in the processor. However, a software attacker may attempt to replace the Device Master Key with a key used to authenticate a malicious COL that would ultimately expose user 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 zeroize the Device Master Key and PRNG seed registers in the processor. The processor will allow a new, non-zero Device Master Key and PRNG seed to be written to the registers using `device_key_mv` only if the current register values are zero. This reset button could also be used to clear a device of a user's secrets before reselling or transferring the device to a different user.

5.4. OS Support

To 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. Bit 0 of the CEM Status register is reset to 0 for the non-CEM process following the context switch, and 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. Bit 0 of the CEM Status register is restored to 1 when the CEM thread begins to execute again.

In addition, we encrypt and compute the hash for all general-purpose and processor status registers before the OS transfers the sensitive context to memory using a fresh key provided by the pseudorandom number generator. If we were to employ the same key to encrypt the registers for every CEM context switch, the system would be vulnerable to data replay attacks. When the 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 value. Then, the fresh seed is used to deterministically generate an encryption and hash key. Many pseudorandom number generators exist; we suggest applying AES encryption to generate a pseudorandom number based upon the 256-bit PRNG seed register similar to the method described in [1].

As described above, the OS should include support for the processor and platform features associated with securely transporting user master key information to the processor. 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. 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 between processes. We note that 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. In future work, we will investigate techniques for supporting multiple simultaneous CEM threads.

5.5. Software Support

```
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 5. Example functions from the COL API

The Cryptographic Operations Library (COL) is a trusted, shared 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 Master Key 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. At installation time, the user can first verify the authenticity of the COL by employing existing software-based Public Key Infrastructure (PKI) techniques. Then, the user can use the reset button and the `device_key_mv` instruction to write new values to the Device Master Key and PRNG seed registers. Then, the user can "sign" the COL using the Device Master Key via a keyed HMAC. 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 HMAC generation process to facilitate the installation of a corrupted and dangerous COL. To prevent such attacks, the user should only install the COL 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.

We list a few functions from the COL API in Figure 5. The COL API is structured similarly to PKCS # 11, the Cryptographic Token Interface Standard [25]. A software application could interpret the COL API like the PKCS #11 interface: entry points to procedures implemented by a hardware device. Let us consider the high-level operation of the COL function `Encrypt`. When a software application calls `Encrypt`, the program jumps to the appropriate function and enters the Concealed Execution Mode. Starting with the master key stored in the processor, the COL traverses the cryptographic key ring until the desired user key is decrypted and authenticated. The COL then applies that 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.

The `Encrypt` function is completed without leaking any keys or sensitive intermediate information. 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. The COL also contains functions that allow an application to generate and add keys to the user key ring. 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.

Unfortunately, like regular application software, software designed for secure cryptographic processors and modules can suffer from bugs and API specification vulnerabilities [7]. Bugs in the COL may allow an attacker to illicitly extract secret information from the underlying hardware. Furthermore, the CEM protects the execution of COL instructions, but the CEM does not preclude attacks on the COL interface. An adversary may attempt to extract user secrets by strategically constructing calls to the COL that may result in the inadvertent leakage of key bits. Thus, we must take special precautions to prevent COL vulnerabilities. The most obvious defenses are diligent, cautious programming and the use of source code analysis tools. In addition, we minimize the COL API in order to avoid attacks that take advantage of the interface complexity. If a COL flaw is discovered, users can securely update or replace their COL via the COL installation procedures.

The CEM and the COL allow users to safely employ the virtual secure coprocessor without exposing their secret keys. However, attackers may compromise a system, install rouge application software, and then employ the rouge application to call the COL and perform cryptographic computation using the secret keys (although the attacker cannot obtain the actual key values). To thwart such malicious code execution, our solution should be integrated with the attestation, secure booting, and general code verification techniques provided by other proposed systems and trusted computing bases (e.g., [15, 20, 30]). For example, the processor-based code authorization scheme presented by Kirovski et al. in [15] would nicely complement our virtual secure coprocessor. At run-time, the processor could verify the integrity of authenticated, trusted software before allowing an application to enter the CEM.

6. 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. By hashing and possibly encrypting/decrypting some information at the processor boundary, we add latency to external memory accesses. We obtain performance statistics by simulating the execution of common cryptographic routines in the Concealed Execution Mode. These routines include the RSA encryption algorithm [24], the AES encryption algorithm [21], and the MD5 one-way hash function [23].

To obtain performance results, we use a modified version of the SimpleScalar cycle-accurate superscalar processor simulator [8]. The processor model is based upon the enhanced processor and memory systems described in Section 5. 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 with 64 KB 1-cycle L1 instruction cache, a 64 KB 2-cycle L1 data cache, and a 2 MB 12-cycle unified L2 cache. The initial external memory access latency is 100 cycles, and the memory bus can transfer 8 bytes every 4 cycles.

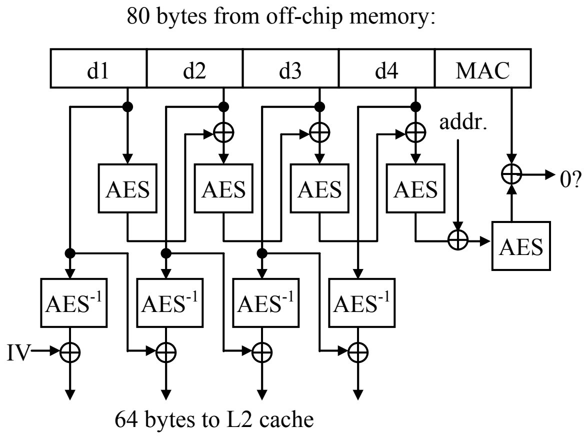

Figure 6. Secure data cache line load

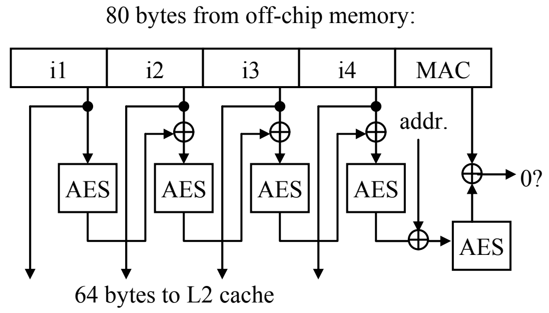

Figure 7. Secure instruction cache line load

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 [19, 21]. 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. Hence, 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. As shown in Figure 6, 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, but the first 16-byte AES encryption operation must be completed before the MAC computation begins. 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 7, authenticated instruction cache line loads simply require 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 memory 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 subsequent authentications) from external memory is relatively low.

7. Conclusion

The protection of cryptographic keys is essential for network, computer, and storage security. In current systems, users are required to employ supplementary hardware devices or distributed algorithms and protocols to safely access and store their keys. Many of these existing solutions suffer from poor performance, inconvenience, high cost, and incomplete security. We present a secure key management alternative for personal computing and embedded platforms through virtual secure coprocessing (VSCoP). We describe architectural and software enhancements that enable the general-purpose processor to serve as a virtual secure coprocessor (when needed) to provide flexible, efficient, and protected use of users' cryptographic keys. In future work, we will investigate alternative methods of implementing the virtual secure-coprocessor for increased security, improved performance, and reduced complexity. We will also explore closer integration of our solution with proposed trusted computing bases and software verification systems.

References

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

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

[3] R. Anderson, *Security Engineering*, John Wiley and Sons, Inc., New York, NY, 2001.

[4] R. Anderson and M. Kuhn, "Low cost attacks on tamper resistant devices," *Security Protocols: 5th International Workshop*, Springer Verlag LNCS, no. 1361, pp. 125-136, 1997.

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

[6] R. M. Best, "Preventing Software Piracy with Crypto-Microprocessors," *Proc. of IEEE Spring COMPCON '80*, pp. 466-469, 1980.

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

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

[9] CERT Coordination Center, http://www.cert.org/, 2002.

[10] 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.

[11] 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.

[12] J. Garay, R. Gennaro, C. Jutla, and T. Rabin, "Secure Distributed Storage and Retrieval," *Proc. of the 11th Inter. Workshop on Distributed Algorithms*, Springer-Verlag LNCS, no. 1320, pp. 275-289, 1997.

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

[14] 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.

[15] 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.
[16] P. Kocher, J. Jaffe, and B. Jun, “Differential Power Analysis,” *Advances in Cryptology – CRYPTO ’99*, Springer-Verlag LNCS, no. 1666, pp. 388-397, 1999.
[17] 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.
[18] 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.
[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, LLC, Boca Raton, FL, 1997.
[20] Microsoft Corp., “Microsoft Palladium Initiative – Technical FAQ,” available at http://www.microsoft.com/, August 2002.
[21] National Institute of Standards and Technology, “Advanced Encryption Standard,” FIPS Publication 197, Nov. 2001.
[22] M. K. Reiter, M. K. Franklin, J. B. Lacy, and R. N. Wright, “The Omega Key Management Service,” *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pp. 38-47, 1996.
[23] R. L. Rivest, “The MD5 Message Digest Algorithm,” RFC 1321, available at http://www.ietf.org/rfc/rfc1321.txt, April 1992.
[24] R. L. Rivest, A. Shamir, and L. Adelman, “A Method for Obtaining Digital Signatures and Public-key Cryptosystems,” *Communications of the ACM*, 21(2), pp. 120-126, February 1978.
[25] RSA Security, Inc., “PKCS #11 v2.11: Cryptographic Token Interface Standard,” available at http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/, Nov. 2001.
[26] The SANS Institute, “The Twenty Most Critical Internet Security Vulnerabilities,” http://www.sans.org/top20/, Oct. 2002.
[27] R. E. Smith, *Authentication: From Passwords to Public Keys*, Addison-Wesley, 2002.
[28] S. W. Smith, E. R. Palmer, S. H. Weingart, “Using a High-Performance, Programmable Secure Coprocessor,” *Proceedings of the 2nd International Conference on Financial Cryptography*, pp.73-89, 1998.
[29] S. W. Smith and S. H. Weingart, “Building a High-Performance, Programmable Secure Coprocessor,” *Computer Networks*, 31(8), pp. 831-860, April 1999.
[30] Trusted Computing Platform Alliance, http://www.trustedpc.org, 2002.
[31] 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.
[32] 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.

Appendix A. Extensions and Alternatives

Architectural alternatives exist for some components in our proposal. One important extension involves the data replay attacks introduced in Section 5.1. There are many solutions to this problem that experience varying degrees of security and implementation cost. For example, we could construct a processor-based session key using the pseudorandom number generator (PRNG) during concealed execution. This fresh session key would be used instead of 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. Alternatively, the memory authentication system presented in [13], which is based upon Merkle hash trees, could be cleanly integrated with our proposal to provide protection against such replay attacks.

Furthermore, we can 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 may 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, similar to the operation of TCG key protection mechanisms [30]. 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.

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.

Another 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. 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.