Protecting Cryptographic Keys and Computations via Virtual Secure Coprocessing

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

Abstract

Cryptographic processing is a critical component of secure networked computing systems. The protection offered by cryptographic processing, however, greatly depends on the methods employed to manage, store, and exercise a user's cryptographic keys. In general, software-only key management schemes contain numerous security weaknesses. Thus, many systems protect keys with distributed protocols or supplementary hardware devices, such as smart cards and cryptographic coprocessors. However, these key protection mechanisms suffer from combinations of user inconvenience, inflexibility, performance penalties, and high cost.

In this paper, we propose architectural enhancements for general-purpose processors that protect core secrets by facilitating virtual secure coprocessing (VSCoP). We describe modest hardware modifications and a trusted software library that allow common computing devices 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 provide 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.

1. Introduction

Security systems generally employ cryptographic algorithms to provide many critical security functions such as confidentiality, integrity, authentication, and privacy. For example, various implementations of secure electronic voting, distributed data storage, and virtual private networks use encryption and related tools to achieve essential security goals. The utility provided by most cryptographic operations is generally based upon the secrecy and integrity of small pieces of data known as *cryptographic keys*. For the purposes of this paper, cryptographic keys may consist of any secret information used to perform a security service, such as AES keys [26], 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. 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 key material. Such system penetration can be realized by exploiting one of the numerous security vulnerabilities that occur in operating systems and application software [11, 30]. 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 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 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 vulnerability 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. However, existing hardware-based key protection mechanisms suffer from a variety of disadvantages, including high cost, inflexibility, and inconvenience to users.

This work was supported in part by NSF CCR-0208946.

1.1. Our Proposal

In this paper, we describe new architectural and software enhancements for general-purpose processors and platforms that protect users' secrets. 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* (VSCoP) when needed [23]. 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 (CEM) for general-purpose processors that protects computations involving users' keys. In addition, we describe a special trusted software library, the Cryptographic Operations Library (COL), which is used in the CEM to safely perform computation using secret keys. To further improve the security offered by virtual secure coprocessing, we propose methods for securely transporting keys to protected storage within the processor for future use in the CEM. 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.

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 devices. 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. 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 constrained cryptographic processors found in smart cards and other cryptographic tokens.

1.2. Outline

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

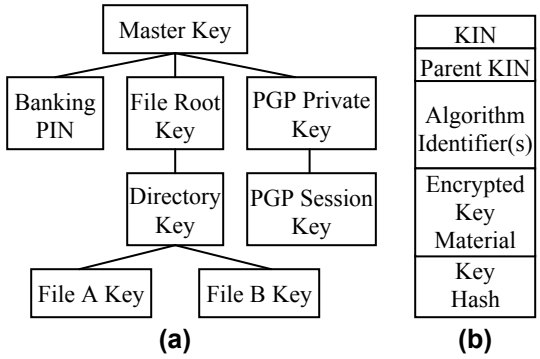

Figure 1. (a) Key ring and (b) key structure

2. Protecting Cryptographic Keys

2.1. Cryptographic Key Rings

We now 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, 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 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.

In this paper, we define master keys to be 128-bit keys for use in symmetric-key encryption or hashing algorithms. Furthermore, we define this master key to be the output of a cryptographically-strong one-way hash of the user's passphrase (although this could be supplemented with hardware token information in practice). Hence, users should carefully select passphrases with sufficient entropy to thwart off-line attacks [31, pp. 87-94].

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

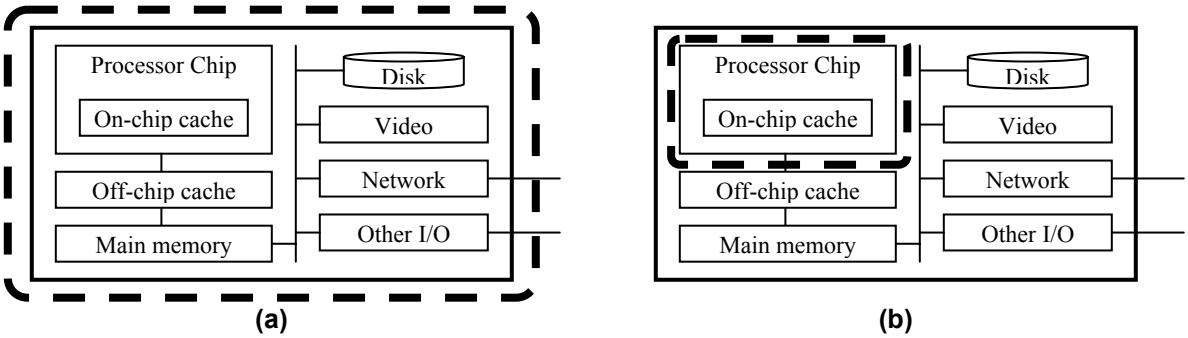

Figure 2. (a) Traditional and (b) proposed security perimeters for critical secrets

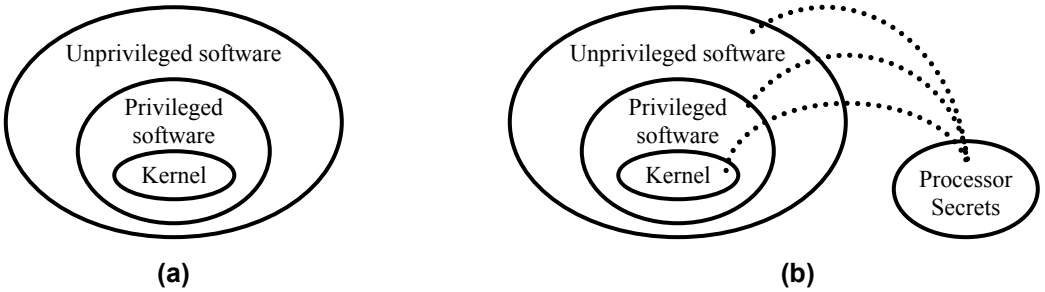

Figure 3. (a) Traditional and (b) proposed access control paradigms

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 [24].

2.2. Our Approach

Our goal is to secure the storage and utilization of a user's secret keys on general-purpose platforms. That is, we seek to defend keys against physical and software-based attacks that involve one or both of the following:

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

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.

To achieve these security goals, we propose restricting the device security perimeter and modifying the traditional access control paradigm with respect to users' cryptographic keys. The security perimeter of a computing device is the boundary that separates the trusted domain from the untrusted environment. 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 2. Memory that is off the processor chip, network interface cards, disks, buses, and any other peripherals will now be treated as being insecure and untrusted. This change can prevent many physical probing attacks that occur outside of the processor, such as attempts to read sensitive information from system buses or from memory swap files stored on disks.

In addition, we create a new disjoint region in the access control paradigm, as shown in Figure 3. 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 system information. This change can prevent many software attacks that regularly circumvent software-based security mechanisms to expose secrets.

Although trusted computing bases (TCBs) seek to achieve similar paradigm shifts for general software (e.g., [35]), they do not ensure special protection for users' critical secrets. That is, the compartmentalization features of TCBs can only provide the long-term protection of keys if all "trusted" software and hardware external to the processor proves to be perpetually impenetrable. Our proposal does not rely on this critical assumption.

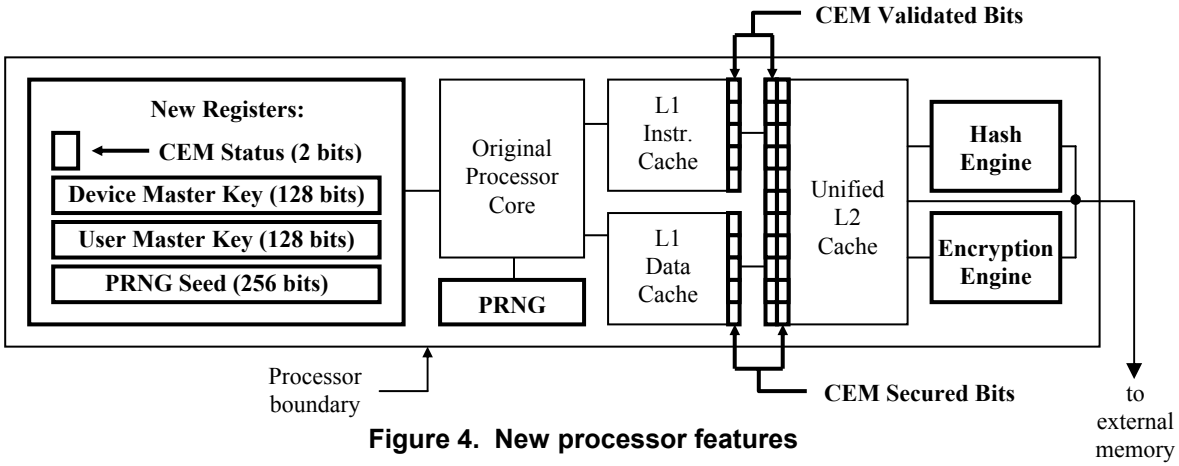

Figure 4. New processor features

3. Virtual Secure Coprocessing

We realize the new access control paradigm and the new security perimeter for cryptographic keys by enabling what we call *virtual secure coprocessing* (VSCoP). A virtual secure coprocessor is a general-purpose processor that functions as a secure coprocessor when needed. We provide new processor architecture and software features that enable a user to safely employ cryptographic keys for a limited period of time in the presence of potentially insecure application software and operating systems. By preventing unauthorized exposure or use of sensitive keys, VSCoP can enhance security and privacy for many applications.

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 by the processor in special secured 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. With OS, platform, and processor support, the two secrets can be used to enable the Concealed Execution Mode (CEM). The CEM protects the execution of a special software library called the Cryptographic Operations Library (COL) that will be the only module in the system privileged to access a user's cryptographic key ring.

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.

We now describe the architectural and software enhancements needed to enable virtual secure coprocessing.

3.1. New Processor Features

In the processor, 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. 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 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., 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 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 [1].

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 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 MAC of 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 [24, 26], 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. In addition, we can implement an optional 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.

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 [26] or some other symmetric-key encryption algorithm in cipher block chaining (CBC) mode [24]. 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.

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. There are many solutions to this problem that experience varying degrees of security and implementation cost [23]. For example, the memory authentication system presented in [15], which is based upon Merkle hash trees, could be cleanly integrated with our proposal to protect against such replay attacks.

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 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 [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.

3.2. New Hardware Platform Features

The processor additions facilitate most of the concealed execution functionality, but the platform assists in moving secrets to/from the processor using simple new features.

Upon receiving a used or new device, one should 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 used to protect a new user's secrets. Also, for similar reasons, before transferring the device to an untrusted party, it would be desirable to reset the device secrets to zeroes. Thus, we must provide support for resetting the Device Master Key and the PRNG seed in the processor.

However, this feature should be tied to a physical action in order to prevent a software attacker from replacing the Device Master Key with one used to authenticate a malicious COL that could 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 reset the Device Master Key and PRNG seed 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 secrets equal zero. Upon writing new values to the device secret registers, which the processor will only permit to occur 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 to respond to the Device Reset button or influence the VSCoP status light.

The hardware platform (rather than the potentially insecure OS) also assumes responsibility for gathering and hashing the user authentication information to generate the User Master Key. 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. A user initiates this procedure by pressing a special "Authenticate" button on the device. While the user authentication information is being inputted, the VSCoP Status Light blinks "green", and after the operation is complete, 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 should 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 secret keys. To achieve this goal, the "Authenticate" button can be pressed again to inform the processor to clear user key data and COL state information from all relevant locations in cache memory and new processor registers. Following the successful wipe, the platform turns the VSCoP Status Light from "green" to "blue" to indicate that user keys are no longer contained in the device.

Table 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.

3.3. New Instructions

We extend the Instruction Set Architecture (ISA) with new instructions to exercise the new hardware features to enable the Concealed Execution Mode. 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 [21]. We summarize the functionality of our proposed instructions in Table 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; 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. The CEM Status registers are then checked and used to ensure that only one process employs the CEM at any given time.

This allows the system to avoid complexities caused by sharing secured memory. 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 can securely transfer 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 reset to zeroes, and the CEM Status register is reset to 0. Cache lines that contain secured CEM data are invalidated using existing cache line flags to prohibit reuse of results from previous CEM invocations. Alternatively, cache line contents could be cleared to zeroes for extra security.

3.4. 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 secure such preemptive context switches by using the CEM Status registers, the PRNG, and the on-chip encryption and hash engines [23]. The PRNG is employed to generate a session key that encrypts and authenticates 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 is used to generate the session key.

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. Also, 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. 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 related proposals and devices, such as the IBM secure coprocessors [32], also require that secure execution requests be performed serially.

```
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 in the COL API

3.5. Software Support

Most legacy application code does not need to be changed to implement VSCoP. Only applications that wish to invoke the CEM would possibly need to be modified to call the Cryptographic Operations Library (COL).

The 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.

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 [29]. A software application could interpret the COL API like the PKCS #11 interface: 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. 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 COL functions must be constructed carefully to avoid leaking any keys or sensitive intermediate information [9]. 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.

Also, 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 computation with secret keys (although an attacker cannot obtain the actual 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 bases (e.g., [19, 25, 35]).

4. Applying VSCoP

We now provide a summary of the steps involved in applying the new enhancements to protect secret keys. We define three major 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, the only software module that will be permitted to access users' keys. First, if the device secrets have not already been reset to zero by 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. Next, the installation procedure writes new random values to the Device Master Key and PRNG seed registers.

At this point, the user verifies the authenticity of the COL by checking its digital signature using software-based Public Key Infrastructure (PKI) techniques. Then, the COL is signed 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 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.

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.

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 system hardware then computes the user's master key and 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. 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 a user has completed an operation that requires the use of his key ring, the user can elect to clear the device of all information related to his secret keys by pressing the Authenticate button.

5. 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 only contains 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 [28], the AES encryption algorithm [26], and the MD5 one-way hash function [27].

To obtain the results, we use a modified version of the SimpleScalar cycle-accurate superscalar processor simulator [10]. The processor model is based upon the enhanced processor and memory system described in Section 3. 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.

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 [24, 26]. 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 we conservatively estimate that one AES round can be completed in at most two processor cycles. 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. 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 incurred by VSCoP (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.

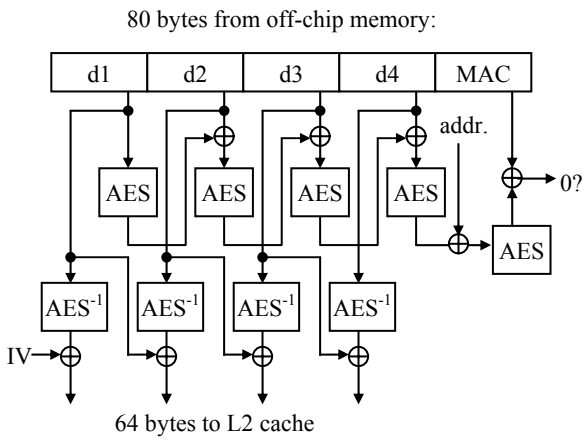

Figure 6. Secure data cache line load

6. Related Work

Researchers have proposed several hardware and software techniques for protecting cryptographic keys against unauthorized observation, modification, and use. 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, whereas other techniques protect users from physical theft and attacks by malicious code.

Unlike VSCoP, no related 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.

6.1. 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., [13, 14]). 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., [22]). These and other distributed schemes can effectively defend against certain attacks that involve limited classes and types of keys.

6.2. 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 [7]. Since that time, researchers have proposed a rich variety of applications and architectures for such hardware (e.g., [17, 36]). 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 [12, 32, 33]. These products offer exceptional physical security for cryptographic keys, but they are too costly, inconvenient, and bulky for mobile computers and information appliances.

Extremely low-cost, portable alternatives to cryptoprocessors and secure coprocessors are cryptographic tokens. These devices include smart cards, PDAs [6], and other small, physically tamper-resistant hardware components [3]. 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, 8]. 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 operations and protocols is often limited. Also, physical tamper resistance is difficult to implement at low costs [4, 20].

6.3. Trusted Computing Bases

As currently defined, trusted computing bases (TCBs) only provide limited protection for user cryptographic keys. The Trusted Computing Group (TCG) [35], which was formerly known as the Trusted Computing Platform Alliance, Intel's LaGrande Technology (LT) [18], and ARM's TrustZone technology [5] seek to provide a trusted hardware base for many types of computing devices. These technologies support varying degrees of system attestation, limited protection of user secrets and inputs, secure booting, and process isolation. In these systems secret information that is inaccessible to the end user is embedded in tamper-resistant hardware modules such as on-board cryptographic coprocessors or general-purpose processors. Microsoft's Next Generation Secure Computing Base (NGSCB) [25], formerly known as Palladium, seeks to provide resources for secure (i.e., validated and isolated) code execution via trusted hardware computing bases. 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 resources provided by TCBs to encrypt a sensitive key for storage, but keys must be used on in the clear in a single pre-specified device to perform computations. Although the TCB may ensure that cryptographic keys are only released to trusted environments, these trusted environments might not be secure. That is, *"trusted" 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. Such bugs in kernel and application software are inevitable and can enable the complete subversion of the TCB mechanisms that provide limited protection for user secrets.

In addition, most TCBs do not defend against 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 [34] seeks to address this problem by cryptographically protecting certain code and data that enters or exits the general-purpose processor.

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

6.4. 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 [15, 19, 34]. In addition, by adding encryption and data authentication capabilities to general-purpose processors, it is possible to enable shielded program execution [16, 21, 34]. 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 tampering and of valuable proprietary code exposure. 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 these proposals and our solution overlap, they differ in fundamental design goals, benefits to users, and several implementation issues.

7. Conclusion

The protection of cryptographic keys is essential for network, computer, and storage security. Many existing key protection 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 provide flexible, efficient, and protected use of users' cryptographic keys. In future work, we will explore closer integration of VSCoP with proposed TCBs and software verification systems.

Acknowledgements

The authors wish to thank Sean Smith and the anonymous referees for their helpful comments and suggestions.

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] 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.

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

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

[8] M. Blaze, "High-Bandwidth Encryption with Low-Bandwidth Smartcards," *Proceedings of the Workshop on Fast Software Encryption*, pp. 33-40, February 1996.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] Intel Corporation, "LaGrande Technology Architectural Overview," avail. at http://www.intel.com/technology/security/, September 2003.

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

[20] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," *Advances in Cryptology – CRYPTO '99*, Springer-Verlag LNCS, no. 1666, pp. 388-397, 1999.

[21] 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.

[22] 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.

[23] 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*, Nov. 2002.

[24] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, LLC, Boca Raton, FL, 1997.

[25] Microsoft, "Next-Generation Secure Computing Base," avail. at http://www.microsoft.com/resources/ngscb/, June 2004.

[26] National Institute of Standards and Technology, "Advanced Encryption Standard," FIPS Publication 197, Nov. 2001.

[27] R. L. Rivest, "The MD5 Message Digest Algorithm," RFC 1321, available at http://www.ietf.org/rfc/rfc1321.txt, April 1992.

[28] 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, Feb. 1978.

[29] RSA Security, Inc., "PKCS #11 v2.11: Cryptographic Token Interface Standard," available at http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/, Nov. 2001.

[30] The SANS Institute, "The Twenty Most Critical Internet Security Vulnerabilities," http://www.sans.org/top20/, Oct. 2002.

[31] R. E. Smith, *Authentication: From Passwords to Public Keys*, Addison-Wesley, 2002.

[32] S. W. Smith, E. R. Palmer, S. H. Weingart, "Using a High-Performance, Programmable Secure Coprocessor," *Proc. of the International Conf. on Financial Cryptography*, pp.73-89, 1998.

[33] S. W. Smith and S. H. Weingart, "Building a High-Performance, Programmable Secure Coprocessor," *Computer Networks*, 31(8), pp. 831-860, April 1999.

[34] 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.

[35] Trusted Computing Group, http://www.trustedcomputinggroup.org, June 2004.

[36] 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.