Architecture for Protecting Critical Secrets in Microprocessors

Ruby B. Lee Peter C. S. Kwan John P. McGregor Jeffrey Dwoskin Zhenghong Wang
Princeton Architecture Laboratory for Multimedia and Security
Department of Electrical Engineering, Princeton University, NJ 08544
rblee@princeton.edu

Abstract

We propose "secret-protected (SP)" architecture to enable secure and convenient protection of critical secrets for a given user in an on-line environment. Keys are examples of critical secrets, and key protection and management is a fundamental problem – often assumed but not solved – underlying the use of cryptographic protection of sensitive files, messages, data and programs.

SP-processors contain a minimalist set of architectural features that can be built into a general-purpose microprocessor to provide protection of critical secrets and their computations, without expensive or inconvenient auxiliary hardware. SP-architecture also requires a trusted software module, a few modifications to the operating system, a secure I/O path to the user, and a secure installation process. Unique aspects of our architecture include: decoupling of user secrets from the devices, enabling users to securely access their keys from different networked computing devices; the use of symmetric master keys rather than more costly public-private key pairs; and the avoidance of any permanent or factory-installed device secrets.

1. Introduction

In the Internet and wireless world, users want to be able to store and access their secrets or sensitive information securely across public networks, using different networked computing devices. This includes document files, messages, programs, rights-managed applications and media content. Adversaries should not be able to decipher or tamper with the sensitive data as it is transported or stored. This can be achieved by encrypting and hashing the data using well-studied cryptographic algorithms and security protocols. However, the security provided by cryptography depends on the safeguarding of cryptographic keys from adversaries. Therefore, keys are critical secrets. If keys can be protected adequately, confidentiality and integrity of sensitive data can be achieved.

Because software-only solutions, e.g., [8] [9] [18], have many weaknesses, high-assurance financial and military systems use specialized secure co-processing boxes to protect both the key materials and the computations that use the keys from physical and software attacks [31] [32]. Although such a protection mechanism is effective, it is expensive and impractical for use on mobile devices. Other applications use a smartcard to protect a core set of private keys [5]. The Trusted Computing Group (TCG) [36] uses a combination of hardware and software to create a more secure Personal Computer (PC) platform.

There are two important characteristics in these solutions: they require separate hardware; and the hardware contains permanent secrets, usually a factory-installed public-private key pair. Trust is derived from the uniqueness and provability of the device secret, and the device's association with a given user. Apart from privacy concerns, the latter attribute limits the portability of trust from one device to another.

We propose a new paradigm where the sensitive data follows the users and is not associated with any particular device, but can be accessed using a multitude of devices with built-in security features described in this paper. We investigate an architecture that can scale to protect an unbounded number of cryptographic keys, and where retrieval of a key does not require establishing trust using factory-installed secrets. In addition, no separate hardware is required. This architecture can be used to protect any type of user secrets. In this paper, we focus on the protection of keys and investigate what constitutes a minimalist set of architectural additions to the general-purpose microprocessor and platform.

Our proposal makes use of a hierarchically encrypted key chain to allow flexible storage of keys over publicly accessible networks. Computations involving the key materials are protected by a trusted software module running in a concealed execution environment. Concealed execution allows a process to execute without fear of its state being tampered with or observed by other processes, including the Operating System (OS). Finally, secure regis-

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

0-7695-2270-X/05/$20.00 (C) 2005 IEEE

ters and cryptographic engines are added to the general-purpose processor to support concealed execution. We call this an SP-processor, or "Secret Protected" processor.

The rest of the paper is organized as follows: Section 2 reviews related work and draws comparisons with our own. Section 3 states our threat model and defines our new trust model and reduced security perimeter. Section 4 describes our architecture. Section 5 explains the procedures for utilizing an SP-processor. Sections 6 and 7 give a security analysis and a performance analysis. Section 8 summarizes the paper and points to future research.

2. Past Work

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, e.g., [8] [9] [18]. Although effective in defending against certain attacks involving limited classes and types of keys, these solutions engage remote servers in real-time, providing an avenue for denial-of-service attacks.

Hardware-based solutions have also been proposed and used in industry. For example, secure co-processors [31] [32] [37] [12] are used in financial and military applications. Secure coprocessors encapsulate general-purpose subsystems within physically secured casings. The private key of a factory-selected public-private key pair is securely stored inside the casing during manufacture. While IBM's secure co-processor platform [32] also allows its key pair to be replaced *during* service using a revocation procedure, it still relies on a factory installed secret and third-party authentication. Software that runs in the subsystems need to be carefully written to never leak the private keys via the device's interface. Untrusted hosts can utilize this guarantee to enable a wide variety of applications [31] [37] [7]. Smartcards can be regarded as specialized versions of secure co-processors in that they possess private keys that never leave the smartcard package, and contain computation elements that perform cryptographic operations using these keys. However, smartcards have limited computational and storage capability, and can be easily lost or stolen, while secure coprocessor subsystems may be expensive and inconvenient for mobile devices.

Architectures, such as XOM [16], enable copy- and tamper-resistant software distribution [4] [11] [14] [33]. Application binaries are encrypted using symmetric-key encryption, and the keys are distributed specifically to each processor using its public-private key pair. These proposals provide a software tamper-resistant execution environment by compartmentalizing shared resources, implemented by either tagging or encryption. While not requiring additional hardware, the goal is to prevent software piracy, not secure and convenient on-line key storage and protection, as in this paper.

AEGIS [33] has mechanisms similar to XOM, but provides stronger memory integrity guarantees. In newer versions of AEGIS [35], the physical unclonable function (PUF) replaces the need for a factory-installed secret. The PUF is still a permanent secret of the chip, tying trust to the device rather than to the user's secrets, as in this paper.

Commercial initiatives from the Trusted Computing Group (TCG) [36], Microsoft's Next Generation Secure Computing Base (NGSCB) [24] and Intel's LaGrande [13] attempt to provide a more secure execution environment using *curtained* memory, separating the memory space of trusted and insecure applications. They also enable attestation of the integrity of the software stack in the system.

Our proposal for SP-processors significantly improves upon our initial work, VSCoP [19] [20], providing more

Table 1: Comparison of proposed and implemented secure hardware architectures

	SP-Processors, VSCoP	XOM, AEGIS, Gilmont, Best	Secure Co-processors, e.g., IBM4758	TCG, NGSCB, LaGrande
Goal	Secret protection and flexible transportation and storage of a user's critical secrets	Copy and tamper-resistant software distribution and execution (in processor)	Secret protection for carefully-crafted security applications	Copy-resistant software distribution; curtained computing environment (outside processor)
Require separate hardware?	No	No	Yes	Yes
Require permanent device secret?	No	Yes (public-private key pair or PUF)	Yes (public-private key pair)	Yes (public-private key pair)
Source of User's trust	Secrecy of User Master Key, integrity of trusted software module and computation	Integrity of software and its computation result	Secrets stored in casing never leak out	Integrity of software at load time only
Trust ties users to devices?	No	Yes	Yes	Yes
Security perimeter	Processor chip boundary	Processor chip boundary	Casing of subsystem	Processor, DRAM, chipset, TPM chip, and buses
Common Characteristics	Assumes hardware is correct and infallible under security attacks. In the next generation of complex chip design and verification, will this assumption be valid?			

flexible security at lower cost. Example improvements are the new Trusted Software Module, reduced processor requirements, improved handling of interrupts, memory and cache protection, the addressing of new attacks, the new threat model and associated security analysis.

Table 1 provides some comparisons between these proposals and our SP-processor. Secure co-processors, XOM, AEGIS, and the trusted platform module (TPM) chip in TCG all require device secrets to be stored permanently on chip. In addition to the larger storage for asymmetric keys and the computational complexity of public-key ciphers, these systems often assume that the processor or platform secrets are never compromised. Our solution protects keys, without requiring permanent processor secrets. We also use much shorter symmetric keys and faster symmetric-key ciphers.

3. Threats and Trust

3.1. Threat Model

The main threats we are concerned with are the exposure or undetected corruption of a user's critical secrets (i.e., keys, in this paper). An adversary should not be able to observe, delete, replace or modify any key without detection.

We focus on software threats, typically launched by a remote attacker across the network. We assume an attacker can get his code executed locally by exploiting security vulnerabilities, by manipulating the execution of existing software, or by direct execution of hostile code inserted statically or dynamically. Software binaries in memory or on disks may also be modified. Hence, all existing software, including the OS, is vulnerable and untrusted.

Although we consider mainly software attacks, we also consider a few simple physical attacks [2], including probing of external buses and reading or modifying external memory and disk storage. We do not consider more difficult physical attacks, such as probing the internal components of the microprocessor chip itself. We also do not consider physical tampering of secure I/O paths.

We only consider operational threats, which are those that occur during use of the system. This paper does not consider developmental threats, such as insider threats during the design and manufacture of the hardware and software components. In addition, we do not consider threats based on incorrect or malfunctioning hardware. Denial of Service attacks is also beyond the scope of this paper, although the user's keys should still be protected even if system availability is disrupted.

3.2. Some Specific Attacks

Once attackers gain control of the OS, they have privileged access to memory, allowing them to observe and modify any location. They might also tamper with OS functions, such as system calls and interrupt handlers, and can trigger arbitrary interrupts. Control over interrupts gives the attackers access to register values. Together, these allow the attackers to tamper with the code base and execution state of any process.

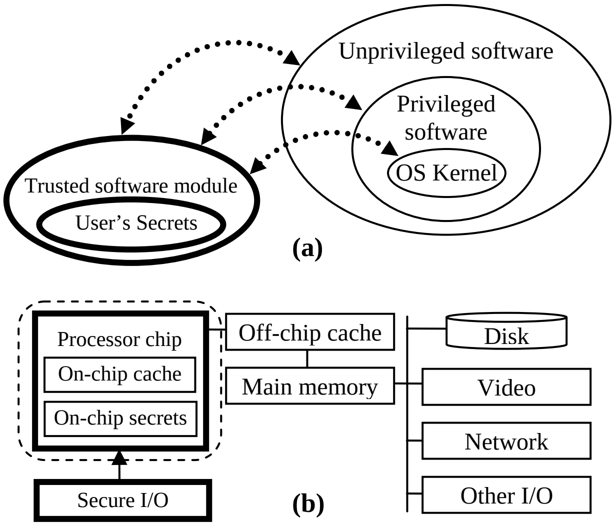

Figure 1. Trusted components (in bold). (a) Orthogonal software access model, and (b) Reduced physical security perimeter.

In addition to subverting the OS, the attackers can gain access to memory by manipulating memory pages stored on disk, or by using DMA to directly access physical memory, bypassing the microprocessor entirely.

These capabilities lead to a variety of attacks, in addition to simple observation of data, code or state. In a spoofing attack, the attackers generate data and try to pass it off as being valid. In a splicing attack, fragments of valid data might be duplicated or rearranged. Entire blocks of data might be copied or moved to incorrect locations. In a replay attack, valid data is saved by the attacker and reused at a later time in the same location. In a dynamic hostile code insertion attack, malicious code is initially inserted as data in the stack or heap during runtime.

3.3. New Trust Model

In order to protect a user's critical secrets in the context of the threat model and to allow convenient access to these secrets from different networked devices, we propose a new trust model built into client computing devices. Figure 1 depicts the trusted components of our solution. It assumes that most software and hardware components are untrusted, including the OS and main memory. The trusted components protect a user's secrets and are assumed to be implemented correctly.

Software only accesses a user's secrets through the trusted software module, which forms a new disjoint region in the access paradigm. Traditionally, access control is based on hierarchical rings of protection, with the innermost ring having the most access and the greatest protection. User software runs in the outermost ring, middleware in the middle rings, and privileged OS software in the innermost rings. Our new trusted software module (TSM) runs orthogonally to these protection rings. It is not part of the OS because operations that are permitted to execute in the new region do not require and should not be allowed to access all system information. Conversely, the TSM operates independently of the OS because the OS should not be allowed to access user secrets. We call this a "Virtual Secure Co-processing" trust model since it is a secure co-processing paradigm, but supported by the microprocessor itself rather than a separate coprocessor.

In addition, we reduce the physical security perimeter from the "box" containing the computing device (e.g., PC, notebook, PDA, or game-machine) to just the microprocessor chip, due to the few physical attacks we consider in our threat model.

4. Architecture

4.1. Overview of our Approach

We first provide an overview of the main components of our SP-architecture and their interdependencies. Each component is further described in subsequent sub-sections.

Key Chain (Section 4.2)

The key chain is a hierarchical structure that stores all of a user's keys in encrypted form (Figure 2). Each key in the structure is encrypted by its parent key. At the root is a User Master Key. This construction allows an unbounded number of keys to be associated with a user. The encryption allows the key chain, except for the User Master Key, to be stored on-line and accessed over public networks.

Trusted Software Module (Section 4.3)

The key chain can only be utilized by the trusted software module (TSM). Only this module can access the User Master Key and use it to decrypt other keys in the key chain. The TSM interfaces with user applications and the OS by exporting a set of functions that carry out cryptographic computation using the key chain, such as signing an email message. In addition, the module might provide access control to its own functions by demanding that callers demonstrate their trustworthiness. Another important attribute of this module is that it should be small enough so that its correctness can be fully verified.

The Concept of Concealed Execution (Section 4.4)

Concealed execution allows the TSM to execute without fear of its state being observed or modified by malicious software. Based on the assumption that off-chip memory and OS are untrusted, concealed execution involves encrypting and hashing data going to off-chip memory, encrypting and hashing register contents during interrupts, preventing illegal accesses to on-chip cache lines that contain sensitive data, and dynamically checking the integrity of the instruction stream before execution.

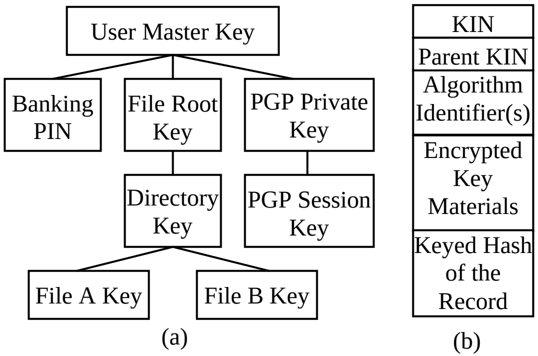

Figure 2. (a) Hierarchical key chain and (b) Structure of a key record

New Hardware Features (Section 4.5)

Various new processor features are required. New instructions are created for entering and exiting the concealed execution environment, for the manipulation of user and device secrets, and for encrypted memory accesses. Cryptographic engines are introduced to accelerate instruction stream integrity verification and memory data encryption, decryption, hashing and integrity checking. In addition, cache lines are tagged when they are associated with data from concealed execution.

Secure I/O (Section 4.6)

We provide mechanisms for users to log in and to re-initialize the device. To prevent software attacks, these critical functions must be tied to physical actions by the user. We illustrate a secure I/O path with simple input-output mechanisms.

OS Support (Section 4.7)

In this proposal, the TSM runs in a single thread. Therefore, the OS acts as a wrapper around the TSM and queues requests to it. Alternatively, this can be provided as an extension to the OS as a device driver.

4.2. Key Management

We use a hierarchical data structure to facilitate secure storage and distribution of keys. We call this a key chain. Figure 2(a) illustrates such a hierarchical key chain structure, which may contain thousands of keys. All child keys are encrypted by their respective parent key. The root of the tree is the User Master Key. Only a leaf key can be used to encrypt users' data. Using this structure, the entire key chain, except for the User Master Key, can be stored in publicly accessible (untrusted) repositories for retrieval. Therefore, the protection mechanism for the User Master Key is extremely important. In addition, this User Master

Key is only associated with the user; it is not permanently associated with any computing device.

Figure 2(b) depicts the detailed structure of a key record. Each key has a key identification number (KIN), parent's KIN, and an encryption algorithm identifier. These fields are stored in plaintext. The key data is encrypted using the parent key. The hash is used to guarantee the integrity of the entire key record. It is a keyed cryptographic hash, using the parent key. Examples of algorithms that can be used to perform the encryption and hashing include AES [25] and SHA-1 [23], respectively.

We define the User Master Key to be generated based on secrets possessed by the user. For instance, we can use the output of a cryptographically-strong one-way hash of the user's passphrase as the master key. The passphrase is entered via secure I/O directly to the trusted domain (microprocessor and trusted software module) without OS intervention. The selection of the passphrase should be carefully considered to provide sufficient entropy [30]. As suggested in [6], at least 80-bits of protection should be provided. If a random string consisting of numbers and characters (including both upper-case and lower-case letters) is used as the passphrase, a string of 14 characters is sufficient. If English text is used as the passphrase, 60 to 70 lower-case letters are needed to provide sufficient entropy [6] [29]. Though this is long, users can choose sentences that are easy for them to remember.

Alternatively, a hardware token or biometric information (possibly in conjunction with a hardware token or passphrase) can be used to generate the User Master Key. Reliable key generation using biometrics is an active research topic, and some commercial biometric products are already available, such as fingerprint readers in the keyboard or in the mouse. Applying multiple sources of inputs to authenticate a user is known as multi-factor authentication. Without loss of generality, in the rest of this paper, we will only employ a user's passphrase input for User Master Key generation.

In order to utilize a key in the key chain, a user only needs to present a passphrase to begin a session. An SP-enabled computing platform can convert the passphrase into the User Master Key, which in turn is used to retrieve the keys in the user's key chain. When the session is over, the user *disassociates* from the device by zeroizing the register used to store the User Master Key in the device. This process *decouples* the user's secrets from the device.

4.3. Trusted Software Module

The key chain is shielded from the untrusted software by the Trusted Software Module (TSM). The TSM is the only software module in the system that is allowed to access a user's keys (critical secrets). It is written such that it never leaks secrets outside the trusted domain. It is the only module that carries out direct computation involving the keys. All other applications, including the OS, depend on interfacing with this module for operations related to the secrets. The TSM is not limited to cryptographic operations on user keys. In fact, it is up to the vendor of the TSM to decide what functionalities should be made available in the module. In this paper, we focus our discussion on operations involving keys, to illustrate the architecture.

```
int Encrypt(input, output, isize, osize,
    keychain, KIN, algorithm, initial_info)
int KeyedHash(input, output, isize, osize,
    keychain, KIN, algorithm, initial_info)
int AddKeyToChain(output, osize, keychain,
    parent, KIN, algorithm, initial info)
```

Figure 3. Example functions in the TSM API

The TSM provides encrypt and decrypt functions and functions that allow an application to generate and add keys to the user key chain. Figure 3 lists a few sample functions. Consider the function `Encrypt()`. Before the function is called, the user must determine which key is to be used. The keychain can be managed by an application, a software library or the OS. It retrieves the subset of the keychain from the root to the desired key, and passes this to the TSM. The TSM decrypts this, until the desired user key is decrypted and its integrity verified. It then applies that key to perform the desired encryption operation. Upon completion, the TSM exits concealed execution, and returns control to the calling application.

In order to exclude any vulnerabilities due to untrusted code, we require that the TSM be entirely self-contained. It should not call functions in external libraries nor make OS system calls. All necessary libraries should be statically linked into the TSM at compile time, and memory for storing intermediate data statically allocated.

Consistent with our minimalist architecture goal, the TSM currently executes in a single thread. The OS queues requests to the TSM. All other non-CEM threads still run multithreaded. Cost-benefit tradeoffs of a multithreaded TSM can be studied in the future.

Another important attribute of the TSM is its code size. Software bugs are inevitable in a complex code base. In order to give confidence of its correctness, the code size of the TSM needs to be small enough to be fully verified.

Even when the TSM code is verified and trusted, it is possible for subverted software to make malicious use of its functions. Although malicious software cannot obtain the user's actual secrets, it can make use of those secrets while the user's key chain is associated with the device. In many scenarios, a verified and trusted TSM provides sufficient protection. Otherwise, the TSM should also attempt to verify the trustworthiness of the caller to its functions. In high security applications, a secure bootup mechanism [3] may be used, such that the software platform is trusted. Or we may trust a small, verified kernel of the OS to correctly identify caller processes, and then restrict access

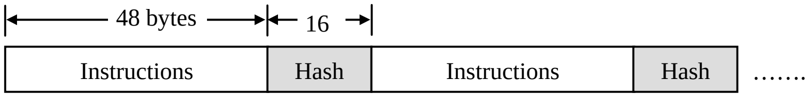

Figure 4: Instruction stream of the trusted module with integrity hashes inserted at the cache line boundary, assuming 64-byte cache line.

based on this identification. Such methods can be applied without modification to our architecture.

4.4. Concealed Execution Mode

The execution of the trusted software module (TSM) is protected from software attacks by a new Concealed Execution Mode (CEM). This has four aspects: (1) protection of the code binaries from malicious modification, (2) protection of data generated during CEM computation from malicious observation and modification, (3) protection of register contents during interrupt handling, and (4) clearing of CEM data upon exit from CEM.

4.4.1. Integrity of the Code Binaries. To support CEM, we introduce another critical secret called the Device Master Key. This is associated with the device for the lifetime of the installation of the TSM.

During device initialization, the TSM is installed with a brand new Device Master Key. This Device Master Key is used to generate keyed hashes of the TSM binaries. Hashes of the TSM are generated per instruction cache line, and stored inline with the code. They also include the address of the first instruction in the cache line. Figure 4 illustrates the resulting instruction stream. Since the hashes occupy instruction space, the compiler for the TSM needs to take into account the insertion of hash values to produce branch instructions correctly.

During CEM, when instructions are fetched from the off-chip memory into the L2 cache, the processor verifies the integrity of the cache line by comparing the reference hash with a hash calculated on the fly over the instructions in the cache line. If the instruction stream has been tampered with, the two values will disagree, and an exception is thrown. Furthermore, dynamic hostile code insertion is prevented by tagging protected instruction and data cache lines in the on-chip L2 unified cache and L1 caches (described in Section 4.5). Therefore, all executed instructions are valid and the integrity of the TSM is guaranteed.

After integrity checking, the hash is replaced in cache with NO-OP instructions so that it does not interfere with the flow of the non-branching instructions.

4.4.2. Memory Data Protection. Any CEM data going to the off-chip memory is also protected by encryption and hashing. When an on-chip data cache line marked as containing CEM data is evicted, the processor first encrypts the cache line using the Device Master Key, then computes a keyed hash over the encrypted memory contents and the starting address of the cache line. This sequence of encryption-then-hash has been proven secure in [15]. The encryption prevents adversaries from observing the memory contents, either by privileged software or by probing the buses. Hashing prevents undetected tampering with data associated with CEM execution.

On fetching a protected data cache line into the on-chip caches, the processor compares its saved hash value with one calculated on the fly. The two will not match if the memory was tampered with, causing an exception to be thrown. Like the instruction hashes, there are many alternatives for the storage of data hashes. For example, they can be stored in a separate memory region in the virtual address space. Memory usage needs to take into account the space requirement of the hashes and make allocation accordingly. In the simplest realization, the memory required by the TSM is statically allocated. The loader can partition the virtual address space into data space and hash space, perhaps using the most significant bit of the virtual address, to form a direct mapping relationship. Thus the processor always knows where to fetch or write the reference hashes.

4.4.3. CEM Interrupt Handling. We assume an untrusted OS handles the interrupts. On an interrupt, the contents of the register file (and any other architected registers in the original ISA) of the CEM thread are protected by encryption and hashing of the register contents in hardware, prior to transferring control to the interrupt handler in the OS. Our solution does not require any changes to the existing OS interrupt handler.

On an interrupt, the processor, while in the CEM, first encrypts all registers as one unit, using the Device Master Key. The resulting ciphertext is stored back to the registers. A hash is calculated over the ciphertext and stored on-chip in the CEM Interrupt Hash register. Then the processor temporarily exits CEM and transfers control to the OS. This allows the OS to handle the register values as in normal interrupts, but not to observe their true values. Storing the hash on-chip prevents it from being tampered with by an adversary, allowing the processor to detect any modifications to the register data when they are restored. This CEM interrupt scheme also prevents replay attacks using a prior copy of the encrypted registers – the only set that can be successfully replayed is identical to the one just saved.

Finally, the processor needs to be able to identify a return back into the CEM, so as to carry out decryption of the register contents and verify their integrity. Our solution is to store the return address in an internal CEM regis-

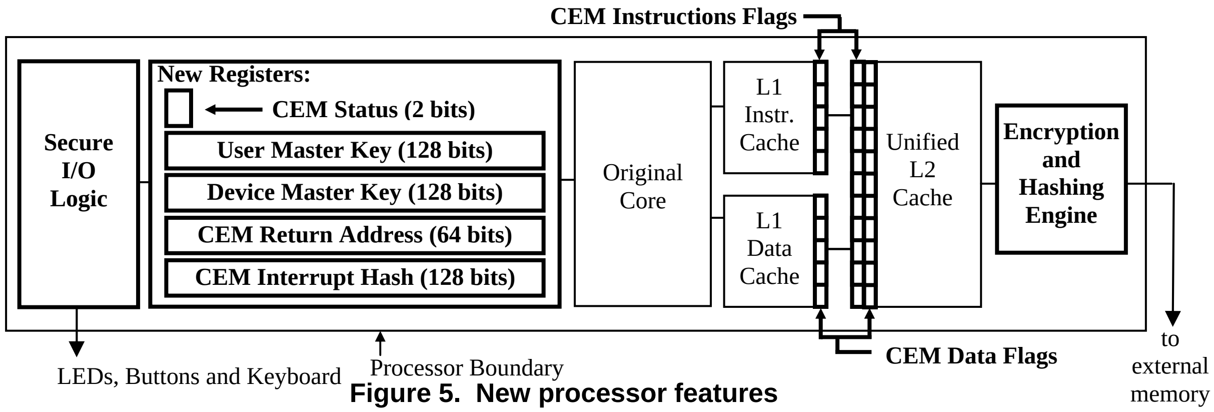

Figure 5. New processor features

ter, called the CEM Return Address register. Every time the processor executes a return instruction, it checks whether the target address is the same as this register's value. If true, the return goes to the interrupted CEM session. The *in situ* registers are decrypted, their integrity verified against the on-chip CEM Interrupt Hash value, and the processor returns to the CEM.

4.4.4. CEM Exit. When the CEM exits, control returns to the caller. At this point, the general purpose registers might contain sensitive data involved in the CEM session. Therefore, the processor must clear the registers before returning from CEM sessions.

4.5. New Hardware Features

We now describe the minimalist set of new hardware features required in the microprocessor in order to support a secure CEM thread amongst other simultaneously running insecure threads. The basic idea is that whenever there is resource sharing, such as the on-chip caches and registers, we need to look into the security implications and take steps to remedy them. There are two general strategies. If the resources are under the direct control of the processor hardware, tagging can be used to distinguish between concealed execution and normal execution. Using the tag, the processor can grant or deny accesses to the resources according to the current execution mode. If the resources are managed by software, such as the off-chip memory, encryption and hashing can be employed to ensure confidentiality and integrity. Figure 5 illustrates a typical processor with the new components shown in bold.

New CEM Registers

We define 5 new registers for SP-processors: a 128-bit User Master Key, a 128-bit Device Master Key, a 64-bit CEM Return Address, a 128-bit CEM Interrupt Hash, and a 2-bit CEM Status register. All but the last have already been introduced in earlier sections. One bit in the CEM Status register indicates whether CEM is in use in the current instruction stream. This is required so that the processor can access CEM instructions, validate the integrity of the TSM instruction stream, and handle interrupts differently. The other bit in the CEM Status register indicates whether any thread on the system is currently employing the CEM (but may have been interrupted). This is used to enforce only one CEM thread running at a time, and prevent an untrusted OS from spawning more than one CEM thread.

Only the User Master Key register can be accessed by the TSM in CEM. The system does not permit the contents of any of the other registers to be visible to any software, including the TSM. Hence there is no instruction to read the contents of these registers. Also, none of these register values are set at the factory; the two key registers are defined by the user in the field, and the others are only set within the processor for CEM processing.

The CEM Return Address, CEM Interrupt Hash, User Master Key and CEM Status registers should not be preserved when the power is off, so they are implemented as regular volatile registers. In fact, the User Master Key is cleared whenever a user session ends. On the other hand, for efficiency reasons, the Device Master Key is implemented using non-volatile memory. Since the Device Master Key is associated with the installation of the TSM, keeping it in non-volatile memory means that the TSM does not have to be reinstalled every time the power is turned off.

Tagging CEM Cache Lines

Two new cache line flags, CEM Data and CEM Instructions, indicate whether the cache line belongs to the CEM thread, and whether it contains protected data or instructions. When insecure threads attempt to access the cache line with one of the flags set, an exception is thrown. In addition, we ensure that the CEM thread cannot treat data in the L2 cache as instructions, either intentionally or because of software bugs.

During a CEM session, when there is an L2 cache miss for a secure data load, fresh content from off-chip memory is decrypted and its integrity checked. If the integrity check is successful, the L2 CEM Data flag is set and L2 CEM Instructions flag is cleared. Otherwise, an exception is thrown and the cache line remains invalid. When the data is brought into the L1 data cache, only the CEM Data flag is preserved. Similarly, an instruction miss in the L2 cache triggers the integrity check of the instructions coming from off-chip memory. The L2 CEM Data flag is cleared, and the CEM Instructions flag is set. Only the CEM Instructions flag is copied to the L1 I-cache.

Once data is in cache, exceptions are raised if non-CEM threads attempt to access data or instructions in a cache line that has its CEM data or instructions flag set.

Our new definition of the CEM cache line flags also allows detection of some dynamic hostile code insertion attacks [17] caused by stack or heap smashing. The CEM Instructions flag in L1 I-cache will remain a "0" if the cache line contents indicated CEM data in the L2 cache. The processor throws an exception if the CEM thread ever attempts to execute an instruction when the CEM Instructions flag is a "0" in L1 I-cache.

Hardware Encryption and Hashing Engine

We provide encryption and hashing of cache lines by hardware for performance reasons. If AES-CBC-MAC is used for hashing, both the encryption and the hashing can be implemented using a single hardware AES module.

New SP Instructions

The new SP instructions and their functionality are summarized in Table 2. Some of them operate similarly to instructions in [16].

At device initialization (Section 5.1), `device_key_mv` is used to write values to the Device Master Key register. There is no instruction for reading the contents of this register. All operations that require using the Device Master Key register are implemented in hardware. Only the TSM running in CEM can obtain contents of the User Master Key register via the `user_key_mv` instruction. However, there is no instruction that can be used by software to write values to the User Master Key; only the processor hardware can write values to that register during user initialization (described in Section 5.2).

When an application wishes to enter the CEM by calling a function in the TSM, the `begin_cem` instruction is executed. The processor determines whether another CEM thread may have been interrupted by checking the CEM Status flags, and throws an exception if there is one. Otherwise, the processor proceeds by setting the CEM Status flags to 1's. All instructions that enter the processor following the execution of `begin_cem` are cryptographically validated using the Device Master Key.

In the CEM, the TSM can securely transfer data to and from memory using the `cem_load` and `cem_store` instructions. Spoofing and splicing attacks are prevented by encryption and hashing. Programs running in CEM can also complete normal (unsecured) memory loads and stores, which are essential for transferring the inputs and results of the cryptographic function from and to the relevant software applications. For example, an encryption function running in 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 TSM function, the function executes the `end_cem` instruction to exit CEM. At this time, all of the general-purpose register values used by the TSM are cleared, the CEM Status register and the CEM Return Address are cleared, and the CEM data cache lines are flushed and cleared. The CEM instruction cache lines remain tagged in the on-chip caches to avoid observation from unsecured processes, but still allow the next CEM session to benefit from instructions already loaded in cache from the previous CEM session.

Table 2. 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. General-purpose registers are cleared. CEM Status bits are set to 0's
`cem_store`	Stores a 64-bit datum to secured memory. The CEM Data cache line bit is set for every cache line touched by this instruction.
`cem_load`	Loads a 64-bit datum from secured memory. The CEM Data cache line bit is set to indicate that the cache line content is data, not instructions.
`device_key_mv`	Transfers information from a register to individually addressable 64-bit chunks of the Device Master Key.
`user_key_mv`	Transfers 64-bit blocks of information to a register from individually addressable 64-bit chunks of the User Master Key.

Area Costs

The new registers consume only 450 bits. The additional 2-bit cache line flags are insignificant increases to the size of the on-chip caches. The encryption and hash engines can be implemented using a single AES module, requiring as few as 25,000 gates [1]. The Secure I/O Logic, described later, is not large. Hence, the only implementation complexity may be the non-volatile memory for the Device Master Key register.

4.6. Secure I/O

Secure I/O channels are required for user authentication and device initialization. We propose a very simple secure I/O interface comprising two LEDs and two buttons. One button and LED is for Device Master Key initialization, while the other button and LED are for User Master Key initialization. A "Device Reset" button clears the Device Master Key, readying the device for a new TSM installation. The Device Reset LED is red when the Device Master Key is cleared (zeroized) and changes to blue once the

Device Master Key is set by the installation software. The other button, the "Authenticate" button, allows the user to initiate a session utilizing his or her key chain. After the user presses the Authenticate button, the platform switches the keyboard to a secure mode and begins diverting all keystrokes to the processor directly. Security against software attacks for this input path is provided by encryption from the keyboard to the processor. The processor's Secure I/O Logic unit decrypts the keystrokes and uses the hashing engine to compute the hash of the passphrase. This hash is then moved to the User Master Key register. The Authentication LED is red when the User Master Key is cleared to zero, and changes to green once a User Master Key is successfully entered.

4.7. OS support

The OS implements functionalities outside of the concealed execution environment to support the single-threaded nature of TSM and to improve performance.

Function Wrapper
The OS provides an entry point for applications to utilize the exported functions of the TSM. It saves and restores the states of the applications before transferring control to and from the TSM.

Queueing TSM Requests
The minimalist processor feature list above requires that the TSM be executed in a single thread. Therefore, the OS must queue requests to the TSM.

Special TSM Loader
Because data accesses of the TSM require special encryption, decryption and hashing, the system loader needs to handle it differently from other applications. Static data (which is the only data type we allow the TSM) already comes in encrypted and hashed format. In the simple scheme we proposed for storing the data cache line hashes, the loader needs to place the data and reference hashes in separate regions in the virtual memory space to create a one-to-one mapping, facilitating the integrity check for memory accesses.

Optional Key Chain Management
When calling the TSM, the user must select which key to use. Because the key identification numbers (KINs) and their ancestral relationships are all stored in plaintext, the management of the storage and transport of the key chain does not need to be carried out by the TSM. The OS or a user software library can do this by fetching all ancestors of the key prior to transferring control to the TSM.

5. Using SP-enabled Devices

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. The user creates a new Device Master Key and installs the TSM.

First, the user presses the "Device Reset" button to make sure the Device Master Key is cleared and to trigger the installation of the TSM. Next, the device must be booted up to a known correct and secure state. This can be achieved by platform attestation and secure bootup, using methods proposed in TCG [36], or more simply by a minimal secure BIOS. This secure BIOS possesses the functionality to install the TSM.

The installation BIOS verifies the authenticity of the TSM by checking its digital signature using software-based Public Key Infrastructure (PKI) techniques, verifying its integrity and authenticity. The installation procedure continues by packaging the TSM for later use in the CEM. This is done by:

1. Generating a new Device Master Key.
2. Signing the TSM by hashing its instruction stream on a per cache line basis.
3. Encrypting and hashing the static data in the TSM.
4. Issuing the `device_key_mv` instruction to copy the newly generated Device Master Key to the processor.

After this installation process, the device can be rebooted to the untrusted OS for normal usage.

User Initialization
User initialization occurs when a user instantiates his cryptographic key chain for use on an initialized SP-enabled device. The user presses the Authenticate button to ensure that the User Master Key register is cleared and to trigger the secure I/O mechanism: the keyboard begins encrypting and diverting all keystroke input to the processor directly until the carriage return is hit. The user enters a passphrase via this secure input mechanism. The user's key chain is now ready for use by the TSM.

Creating a new user key chain simply involves selecting a passphrase, by which the User Master Key is generated. As keys are added to the chain, a user can store the encrypted key chain locally or remotely.

Since the User Master Key is never permanently stored on the device, it is safe for a user to pass on a used device to another user.

Protected Operation
Protected operation is the secure use, via the TSM, of a user's key chain in an SP-device that has been device-

initialized and user-initialized as in the two procedures detailed above.

For example, a rights-managed digital video application may request the TSM to decrypt chunks of an MPEG data file, using a previously selected and secured key in the key chain. The application does not need to read the key but merely needs to utilize it to decrypt the content. Although the application runs in the untrusted domain, all computations involving the key are carried out by the TSM, without fear of leaking information of the keys when it runs under Concealed Execution Mode.

When completed, the user should clear the device of all information related to the User Master Key by pressing the Authenticate button again, which will zeroize the User Master Key register.

6. Security Analysis

SP-architecture addresses the threats in Section 3. The user's keys are protected from observation and tampering during storage and transport by encryption and hashing. They can only be used by calling the TSM, which is a trusted software module. This means that it is correct and free from software security vulnerabilities. Tampering with the TSM code base and hostile code injection can occur while the binaries are stored on disk, in main memory, or in cache. On disk and in main memory, the binaries are protected by signing with keyed hashes. As the code is loaded into cache through the instruction fetch path, the integrity is verified. These CEM instruction cache lines are tagged and cannot be modified. Anything that has not been verified and then tagged as CEM instructions will not be executed while in CEM.

Threats of manipulating the execution state of the TSM require tampering with the state of a CEM thread or its data. This includes register values, data cache, and main memory. CEM data is separated from unprotected data on a cache-line granularity in on-chip caches. Access of CEM cache lines by non-CEM threads causes an exception.

Observation, spoofing and splicing of TSM's main memory space are prevented by encrypting and hashing of all off-chip stores. The encryption prevents adversaries from observing the memory content. Integrity checks using hashes thwart spoofing attacks. Splicing is prevented by incorporating the memory addresses in the hashes. In addition, memory replay attacks can be prevented using known memory authentication systems, e.g., [10] [34]. The methodology in [10] uses Merkle hash trees, and can be cleanly integrated with our architecture.

Observation, spoofing, splicing and replay of register contents during interrupt handling and context switches are remedied similarly by encryption and hashing. By first encapsulating *in situ* the registers of the TSM thread before transferring control to OS interrupt handlers, our architecture provides protection without the need to modify the interrupt handling routine in the OS.

After a CEM session ends, the registers and CEM data cache lines are cleared to prevent intermediate data from leaking information about the user's secrets.

Some dynamic hostile code insertion attacks are also prevented by our new definitions of the CEM Data and CEM Instructions flags for the on-chip L1 and L2 caches. Hostile code cannot be brought in during execution as data, then later executed as code.

We also provide a secure I/O path to and from the trusted domain. Simple interfaces such as buttons and LEDs are connected directly to the processor. Keyboard inputs are first encrypted prior to transmission, preventing software attacks on the user's passphrase.

We address software attacks during device installation by requiring the system to boot up into a known trusted state. This can be implemented by a secure BIOS installation module, or more generally by platform attestation and secure bootup methods as presented in [3] and [36].

Finally, but importantly, the processor carries no factory-installed secrets. In fact, we provide mechanisms to clear or replace any values from the factory or previous owner. We therefore are also protected from compromise of the factory and its secrets database.

7. Performance Analysis

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

Since this paper concentrates only on protecting keys and their related cryptographic functions, we evaluate performance degradation associated with these cryptographic functions. Thus, we obtain performance statistics by simulating the execution of common cryptographic routines, e.g. [21], in the CEM. We use the RSA encryption algorithm [27], the AES encryption algorithm [25], and the MD5 one-way hash function [26] as representatives of public-key ciphers, symmetric-key ciphers and secure hashes. We will evaluate SP-processor performance against a wider range of benchmarks in the future.

We model our proposed enhancements to the interface between the L2 cache and external memory as follows. We use 128-bit AES-CBC for data encryption/decryption and 128-bit AES-CBC-MAC to provide code and data authentication [23] [25]. The AES-CBC encryption and decryption of 64-byte cache lines can be completed with 4 serial and 4 parallel AES operations, respectively. The initialization vector (IV) is equivalent to the address of the cache line. MAC computation for authenticating both 48-byte instruction and 64-byte data cache lines requires a latency of 4 and 5 AES operations respectively; in both cases, one extra AES operation is included to hash 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 decryption, encryption and MAC computation are at most 20, 80 and 100 cycles, respectively.

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 require a MAC computation, so the added latency is 4 serial AES operations. Hence, the maximum external memory access penalties incurred (per 64-byte cache line) for secure data loads, secure data stores, and authenticated instruction loads are 100, 120, and 80 cycles, respectively. Our previous results in [20] show a performance degradation of less than 1%.

8. Summary and Future Research

SP-architecture incorporates a minimalist set of processor and platform features that protect a user's critical secrets during storage, transmission and use in an on-line system. For example, cryptographic protection of sensitive data depends on the protection of keys. Keys are critical secrets that must be carefully protected and managed. In SP-architecture, keys conveniently follow their users and are not associated with any particular device. This allows a user to securely employ his keys on multiple devices, and allows a device to be used by different users.

We used a hierarchically encrypted data structure to efficiently protect user key chains that are stored in open networks. We proposed new features to the microprocessor to support a Concealed Execution Mode. We defined a Trusted Software Module, which performs protected computations on users' secret keys while running in the Concealed Execution Mode: the keys, their computations and intermediate state are all protected from observation and tampering by adversaries. Splicing, spoofing and replay attacks are thwarted.

SP-architecture incorporates several novel features. It proposes a new trust model based on what we call “virtual secure coprocessing”. It decouples user secrets from the devices. It does not rely on any permanent or factory-installed secrets. The root of trust is based on two independent master keys: one for the device and one for the user, both being symmetric-keys. Hence, hardware support is required only for symmetric-key operations rather than more costly public-key operations.

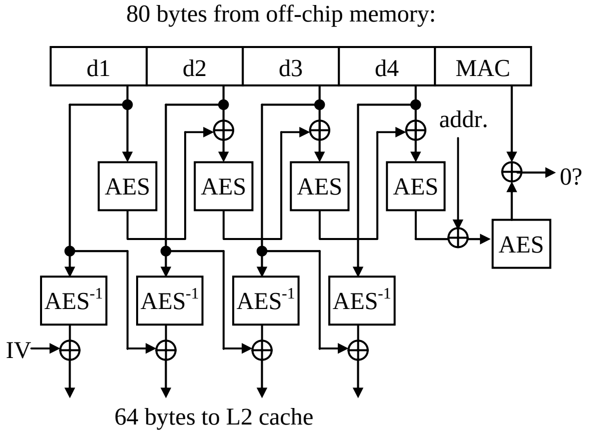

Figure 6. Secure data cache line load

While this paper focused on protecting keys as critical secrets, SP-architecture can also protect other critical secrets. In future work, we hope to evaluate SP-processing for different applications, such as digital rights management and privacy protection systems. Also, while various proposals exist for secure I/O and secure bootstrapping, we believe that more research is needed to study alternatives and integrate these into architectures like SP-processing. The simplicity of SP-architecture suggests that verification and security assurances may be facilitated. Future work should include how this can best be done for SP and other secure processor architecture proposals. We also assumed, like all other hardware-based proposals, that we can trust the correctness and integrity of hardware. Future research should look into how well this assumption holds as chips continue to grow in complexity.

There are many possible extensions, alternative mechanisms and applications of SP-processing. We hope to have provided a foundation for future research and evaluation of processor and platform architectures for more secure and convenient networked computing devices.

9. References

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

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

[3] W. Arbaugh, D. Farber, and J. Smith. “A Secure and Reliable Bootstrap Architecture.” *Proc. of IEEE Symp. on Security and Privacy*, pp 65–71, May 1997.

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

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

[6] W.E. Burr, D.F. Dodson and W.T. Polk, "Electronic Authentication Guideline: Recommendation of the National Institute of Standards and Technology," NIST Special Publication 800-63 Version 1.0.1, Sep 2004.

[7] J. Dyer, R. Perez, S. Smith, M. Lindemann, "Application Support Architecture for a High-Performance, Programmable Secure Coprocessor," *Proc. of 22nd Natl. Information Systems Security Conference,* October 1999.

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

[9] J. Garay, R. Gennaro, C. Jutla, and T. Rabin, "Secure Distributed Storage and Retrieval," *Proc. of the 11th Int'l. Workshop on Distributed Algorithms*, Springer-Verlag LNCS, no. 1320, pp. 275-289, 1997.

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

[11] T. Gilmont, J.-D. Legat, and J. J. Quisquater, "An Architecture of Security Management Unit for Safe Hosting of Multiple Agents," *Proc. of the Int'l Workshop on Intelligent Communications and Multimedia Terminals*, pp. 79-82, Nov 1998.

[12] P. Gutmann, "An Open-source Cryptographic Coprocessor," *Proc. of 2000 USENIX Security Symp.*, 2000.

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

[14] D. Kirovski, M. Drinic, and M. Potkonjak, "Enabling Trusted Software Integrity," *Proc. of the 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X),* October 2002.

[15] H. Krawczyk, "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)," *Proc of CRYPTO 2001,* 2001.

[16] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *Proc. of the 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX).*, pp. 168-177, 2000.

[17] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, "Enlisting Hardware Architecture to Thwart Malicious Code Injection," *Proc. of Int'l Conf on Security in Pervasive Computing (SPC-2003),* LNCS 2802, pp. 237-252, Springer Verlag, Mar 2003.

[18] P. MacKenzie and M. Reiter, "Networked Cryptographic Devices Resilient to Capture," *Proc. of the 22nd IEEE Symp. on Security and Privacy*, pp. 12-25, 2001.

[19] J. P. McGregor and R. B. Lee, "Virtual Secure Coprocessing on General-purpose Processors." *Princeton University Dept. of Electrical Engineering Technical Report* CE-L2002-003, Nov 2002.

[20] J. P. McGregor and R. B. Lee, "Protecting Cryptographic Keys and Computations via Virtual Secure Coprocessing." *Proc. of the 2004 Workshop on Architectural Support for Security and Antivirus*, Oct 2004; also in *Computer Architecture News*, ACM Press, Vol. 33. No. 1, pp 16-26, Mar 2005.

[21] mCrypt cipher suite, PAX Project, Princeton Architecture Laboratory for Multimedia and Security (PALMS), http://palms.ee.princeton.edu/PAX.

[22] Microsoft, "Fingerprint Technology," http://www.microsoft.com/hardware/mouseandkeyboard/features/fingerprint.mspx

[23] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[24] Microsoft, "Next-Generation Secure Computing Base," http://www.microsoft.com/resources/ngscb/, Jun 2004.

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

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

[27] R. L. Rivest, A. Shamir, and L. Adelman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Comm. of the ACM*, 21(2), pp. 120-126, Feb. 1978.

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

[29] B. Schneier, *Applied Cryptography*, J. Wiley & Sons, 1996.

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

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

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

[33] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," *Proc. of the 17th Int'l Conf. on Supercomputing (ICS),* 2003.

[34] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," *Proc. of the 36th Int'l Symp. on Microarchitecture*, Dec 2003.

[35] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and Implementation of a Single-Chip
Secure Processor Using Physical Random Functions," *MIT Technical Report* CSAIL CSG-TR-483, 2004.

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

[37] 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.