NPS-CS-06-012
ISI-TR-2006-619

Trustworthy Commodity Computation and Communication

SecureCore Technical Report

Integration of User Specific Hardware for SecureCore Cryptographic Services

Ganesha Bhaskara, Timothy E. Levin, Thuy D. Nguyen, Terry V. Benzel, Cynthia E. Irvine and Paul C. Clark

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0430566 and CNS-0430598 with support from DARPA ATO. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or of DARPA ATO.

Author Affiliations

Naval Postgraduate School:

Cynthia E. Irvine, Timothy E. Levin, Thuy D. Nguyen, and Paul C. Clark

Center for Information Systems Security Studies and Research
Computer Science Department
Naval Postgraduate School
Monterey, California 93943

USC Information Sciences Institute:

Terry V. Benzel and Ganesha Bhaskara

Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, Ca 90292

Table of Contents

I. Introduction

The objective of this document (internal to ISI/NPS/Princeton) is to hash out details and design issues that may arise while integrating Secret Protected (SP) [rlee05] with the SCHW and the SecureCore architecture. This document describes the use of cryptographic hardware like SP and Trusted Platform Module (TPM) [tcgp05] within the context of the SecureCore project to provide cryptographic services. We start with describing the assumptions about the architecture, hardware, software and usage of the SecureCore device. We then describe the hardware requirements for virtualization of SP and how the virtualized SP is integrated into and used in the SecureCore architecture.

A. *Overview of SecureCore Architecture*

The SecureCore architecture (Illustration 1) is designed around a Least Privilege Separation Kernel (LPSK) running on SecureCore Hardware (SCHW), which along with the Secure Core Security Services (SCSS) and the SecureCore Operating System (SCOS) provides a platform to perform commercial as well as highly trusted transactions via the trusted SC Process [irvi05]. The LPSK partitions all resources into distinct blocks. The LPSK and the SCSS together form the Trusted Management Layer (TML) underlying all active blocks.

The SCHW provides a ring architecture (with at least 4 positive and 2 negative rings) to provide least privilege domains. It also supports hardware isolation between its blocks as well as individual tasks or processes. ("Threads" are execution streams within the context of a task.) The LPSK, of which the SCHW is an integral part, exploits hardware privilege levels as well as the hardware tasks to manage and enforce separation between the blocks. Resources are allocated to blocks at the time of initialization or startup. The SCSS layer virtualizes the static resources exported by the LPSK and exports interfaces that hide the static nature of the LPSK from the OS that resides in that block. It provides cryptographic, networking, user I/O and storage services.

The secureCore architecture is configured to support one [confidentiality-level/integrity-level] pair per block. Similar to all exported resources (e.g., subjects, objects, devices), keys too are assigned different security levels. The multiplicity of blocks and levels of information that must be kept separate motivates the virtualization of SP services. With an appropriate configuration, the LPSK along with the SCSS layer enforces MLS policies like controlled read down, write up and supports asynchronous notifications and event counters. Arbitrary write up is not allowed but is supported for certain encapsulated objects, such as keychains, sequencers and event counters. [reed79] For example eventcounts can be used to signal a high level subject to read down. The LPSK along with the SCHW, also provides a fine grained low level temporal access control and revocation mechanism using time-based events to grant, deny, or revoke access. Design principles [benz05], objectives and tradeoffs may dictate further partitioning of the SCSS in each block into trusted and untrusted layers. In the current form, the entire SCSS is assumed to be trusted. The SCOS provides a trusted environment to host the trusted services including the SC Process.

Context switches between blocks is user and process driven. The user is active in one block at a time, and if there are no background processes in the other blocks, the user's

block remains active until the user logs out or changes to another block. If background processes are running in other blocks when the user is logged in, then all active blocks will be scheduled with fixed time slices to prevent covert timing channels that could be exploited by Trojan horse applications.

When a user wants to change security/integrity levels or otherwise utilize services from a different block than the current one, the user presses the secure attention key (an external non-spoofable physical input). The LPSK services this interrupt and passes control to the SC trusted path application (SCTP) in the SCOS block. SCTP interacts with the user to negotiate a new level of security and integrity, and activates a user session in the new block.

For the purposes of attestation of the device (potentially including properties such as hardware and software configuration), SecureCore architecture utilizes an embedded TPM engine or its equivalent that is co-resident or integrated with the SCHW. The TPM typically contains cryptographic keys and hashes that it uses to digitally sign its outputs, protect itself and verify the integrity of the boot sequence. The TPM also provides secure storage containers with confidentiality and integrity protection.

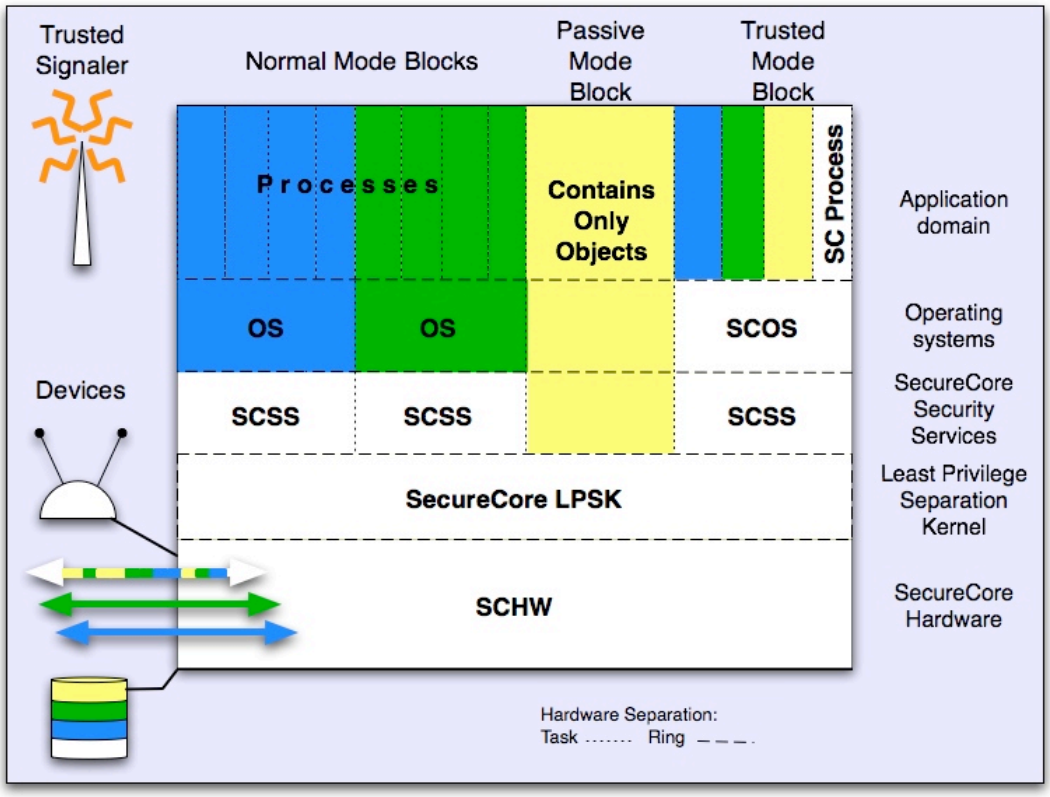


Illustration 1: SecureCore system architecture, phase II

B. Overview of SP

Objective: SP [rlee05] is a design for a processor subsystem that is applicable for incorporation into a broad range of processors. SP aims to enable secure and convenient protection of the user's critical secrets in an on-line environment, with minimal changes to hardware. SP protects the user's secret keys from various hardware and software threats that could result in the compromise of the key's confidentiality. SP also provides portable access to user keys across all SP-enabled devices (Illustration 2) by avoiding the use of permanent factory installed keys to protect critical user secrets. Additionally, a device-specific Device Master Key (DMK) is used to protect certain trusted software modules (TSMs) which implements all the control and data interfaces related to key chains. The DMK is loaded into the non-volatile "DMK register" before the system software is installed. When a TSM is installed, its individual instructions are hashed using the DMK. The hashed values are stored inline with the code. Since SP instructions are a part of the processor ISA, SP also aims to mitigate performance concerns associated with the traditional secure co-processor architectures. The user enters the User Master Key (UMK) directly into the volatile UMK register through a secure I/O channel.

Before we enumerate the details of how SP is used in SecureCore, we introduce the following definitions:

- Semantic Integrity refers to the association of the key with a particular application or service in the context of SecureCore and SP. A typical example would be the key associated with an email account.
- Syntactic integrity refers to the soundness of the bits of the keys.

In SecureCore, SP is used to protect confidentiality and syntactic integrity of keys. SP does not protect semantic integrity of the key or the key chain. SP depends on the applications, the individual TSMs, or their execution environments to ensure the right key is used, and to control access to TSM interfaces and encrypted keys, if those services are required. Thus, an attacker cannot, without being detected, modify the bits of any key without using the interfaces provided by the TSM. Further, in SecureCore, SP is used to provide cryptographic services to subjects in rings 0 and above.

The model in this document for keychain construction and access is as follows (SP does not depend on this model). The user keys are organized in a logical "key chain". The UMK is the root key for cipher-block chain encryption of the keys. If a user wants to use one of the keys to perform a cryptographic operations, then the keys on the path to that key are first decrypted using the UMK and the subsequent keys along the path, and the selected key is used to perform the required operations. Thus, by protecting the UMK, the entire key chain is protected.

An important consequence of this trust model is that the UMK becomes associated with the device until the user erases the UMK. For another user to use the device or for the same user to use the device with a key chain encrypted with a different UMK, the old UMK must be cleared, so that the association between the old UMK and the device is

broken and the user can enter a new UMK to establish association with the device.

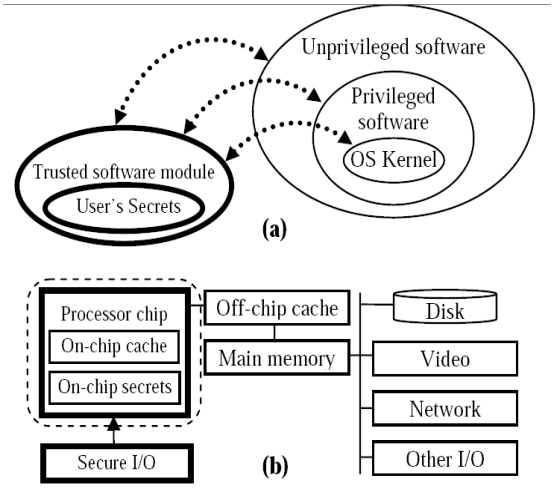
All instructions that involve user keys and related SP management functions must be executed in the Concealed Execution Mode (CEM) (Illustration 3), as part of a TSM. Before a TSM interface is invoked by a process, a CEM session is first started by executing the “begin_cem” instruction. At the end of the CEM session the “end_cem” instruction is executed to switch the processor back to normal mode. The “begin_cem” and “end_cem” instructions manipulate an “in CEM” bit in the CEM Status register that indicates that the CEM is in use. Once CEM is started, the SP processor continuously performs on the fly checks for the instruction's integrity with respect to it's hash until the CEM sessions ends.

Any data that needs to be moved off of the processor chip is also hashed and encrypted using the DMK as depicted in Illustration 2 and 3. If an interrupt occurs when the processor is in the CEM, the SP encrypts the contents of all general purpose registers, collectively hashes them and then stores the hash and the return address in internal SP registers inaccessible to software. Finally control is transferred to the kernel to service the interrupt. Doing so ensures

- integrity of the return address and the content of the general purpose registers
- confidentiality of the content of the general purpose registers

The two bits that indicate the status of the CEM act as a hardware lock that ensures that only one CEM session can be in active or interrupted mode at any given time i.e., each TSM is intended to be an interruptible critical section . When a task is restored on a context switch and CEM-interrupted is set, after the new context is loaded, SP checks the new PC address against the value in the CEM return address register; it checks the collective hash of all the new general purpose registers against the hash of registers that was stored in the internal hash register. The processor proceeds with the CEM mode if both the checks pass. Since the TSM is designed to be as simple as possible, it relies on its environment (i.e., the SecureCore architecture, and in particular, the hierarchical privilege domains, as described below) to protect it from unauthorized use, if that is a concern, and relies on the OS and / or SCSS to manage queuing and dispatch of requests, if those services are required.

Thus, the user only depends on the DMK of the device to protect the integrity of the TSM and confidentiality and integrity of the intermediate states of computation.



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

Illustration 2: Trust boundaries and access model of SP

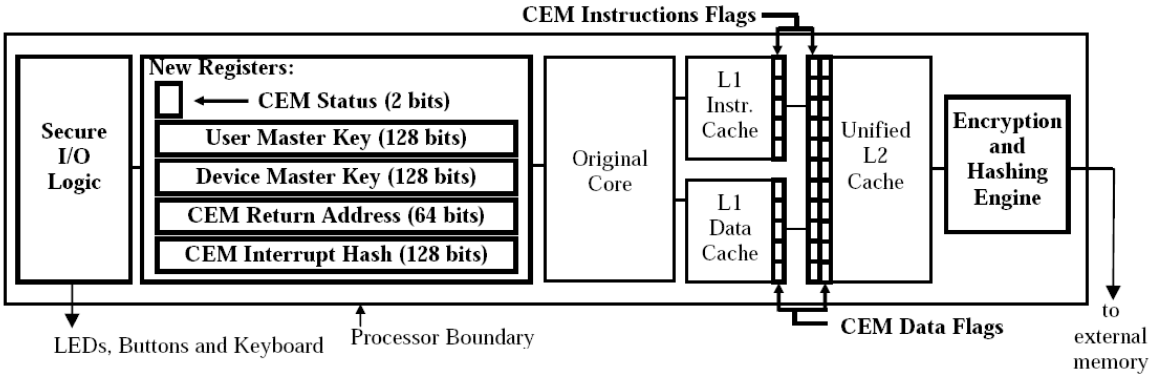


Illustration 3: Internal details of CEM

Though TPM and SP can perform many similar tasks, SP has performance advantages over TPM due to its tight integration with the processor ISA. Further, SP does not depend on factory installed secrets and also provides powerful mechanisms that can be used to protect confidentiality and integrity of data and / or code at runtime. In the SecureCore architecture, the mechanisms provided by SP and TPM are used to achieve objectives that are fairly orthogonal to one another.

II. Assumptions in SecureCore Architecture

Since SecureCore is targeted towards small hand held mobile devices, we make some typical assumptions about operational scenarios that are specific to mobile, embedded,

networked environment.

A. *Assumptions about usage scenarios*

The SecureCore networked mobile device is designed to be used by a single user at a time. This user may need to assume different roles, which translate into different security profiles for the user. The user will require access to cryptographic services that include secure storage of keys in addition to encryption, decryption, key generation, random number generation etc. In phase one, the SP extension for the SCHW is assumed to be capable of supporting one User Master Key (UMK) i.e., the SCHW has no mechanism to store and handle more than one UMK at any given time. Therefore, if a different user wants to use the device or if a different UMK is needed for a different role, the UMK must be cleared and a new UMK entered. Support for multiple UMKs will be investigated in SC phase two.

The user can assume roles that can only utilize the basic services provided by the device. Further, all actions performed by the user are constrained by MAC policy. Typically, the user does not assume roles that would be required for maintenance operations, including software installations, TPM key changes, configuration changes etc. Such maintenance operations including changes to hardware are performed at a secure facility and are not a part of normal operation of the device.

B. *Assumptions about HW*

We assume that the SCHW has sufficient functionality to support a type 1 VMM [gold72]. Further, the SCHW also incorporates SP functionality as described in [rlee05]. We term this integrated VMM and SP hardware platform as “SCHW with SP extensions”. For the purposes of attestation of the device (potentially including properties such as hardware and software configuration), we assume the presence of a TPM engine or its equivalent that is co-resident or integrated with the SCHW.

C. *Assumptions about SW*

Though OSs in the untrusted blocks can support multiple users and the SecureCore device supports multiple blocks, only that device's single user will be logged into any OS on any of the blocks.

The LPSK along with the SCSS is assumed to have the capability to prevent a remote user from connecting to a guest operating system and executing commands Such a restriction not only ensures that, only the actions of the user in possession of the device can invoke commands directly or indirectly, but also ensures that the UMK entered by the user in possession of the device is not utilized by remote users. When a user logs into the device and enters the UMK, the single key effectively becomes associated with each of the user blocks on the device.

III. Virtualization of SP interfaces with single UMK support

Hardware support for virtualization is assumed to be similar to [inte05], [adva05]. The

following table gives an overview of the allocation of rings in the SecureCore architecture.

Ring	*Ring (-2)*	*Ring (-1)*	*Ring (0)*	*Ring (1+)*
SC Layer	LPSK	SCSS	OS	Applications
Domains	all blocks with no restrictions	all blocks with restrictions on operations	Single block with no restrictions within that block	Single block and restricted to single task / address space

The TML provides isolation and virtualization of hardware resources. Only LPSK and SCSS have the access rights to request negative ring HW operations. The guest OSs have access to data and control instructions at ring 0.

Instructions that are used to manage the block control structure (BCS) and scheduling of blocks are assumed to be accessible only to software in ring -2. The BCS is assumed to contain information about the state of each block. When the LPSK interrupts one block and schedules another, it saves the complete state of the processor before restoring the state of the next block. This is done such that states of one block do not affect the operation of another block.

If a task in one block is using a resource and LPSK schedules the next block for execution, tasks in the next block should not be affected – through either timing or resource access variability -- by the fact that a task in the previous block was using that resource. In particular, the TML must be able to virtualize the status of the CEM. A simple approach would be to provide privileged instructions for TML to load and store the CEM state (CEM return address, CEM hash and CEM status bits). The LPSK would not need to be able to interpret the CEM state, so the state could be encrypted.

A. *Hardware support for virtualization of SP*

To facilitate virtualization of the SP, the following functional requirements, over and above the currently described functionality of SP, need to be met to enable virtualization of SP

SP : 1 SP extensions to SCHW *must* support tamper proof mechanisms that prevent unauthorized access to SP instructions. Further, these mechanisms (e.g., hierarchical privilege domains) must ensure that:
- SP : 1.1 The LPSK (ring -2) may
  - SP : 1.1.1 Write and read data to and from all registers that represent the state of the processor (in non-CEM, CEM and interrupted CEM.)
  - SP : 1.1.2 Execute instructions that manage the virtualization of the CEM
- SP : 1.2 Rings 3, 2, 1, 0, -1 or -2) may
  - SP : 1.2.1 Manage registers that are relevant within a given block (in normal, CEM, and interrupted CEM.)

SP : 2 Exceptions *must* be raised in the event of
- SP : 2.1 access violations
- SP : 2.2 Other mode of failures

SP : 3 All data operations *must* be preemptable to avoid covert timing channels.

Future extensions to the SP architecture, while maintaining the overall HW / SW architectural simplicity, should support additional functions to eliminate the need for TPM-like support from off the chip.

SP : 4 Enable secure boot
- SP : 4.1 Enable secure storage of keys not associated with the user
- SP : 4.2 Generate high quality asymmetric and symmetric keys as well as high quality random numbers

SP : 5 Support multiple UMKs
- SP : 5.1 Ensure that only SCSS or LPSK (rings -1 or -2) may manage additional registers and instructions for selecting the appropriate UMK.

Integrating the TPM functionality with SP will obviate the requirement to establish secure communication channels between SP/TSM and the peripheral TPM.

B. Integrated SecureCore Architecture

In this initial phase of design we intend to support only one process per block, one CEM session per block and use a single TSM per block. Though one can envision architectures which allow multiple CEM sessions per block and multiple TSMs in a block, distributed across different rings, this is beyond the scope of this document. Furthermore, SP/TSM is used to provide cryptographic services to subjects in and above ring 0 in each block – use of CEM to protect the kernel itself will be described elsewhere.

Individual key-management: We define a straw man TSM that provides crypto-service interfaces. For example, for encryption of data, an application submits the data, a keychain and a key indicator. A key manager or key management library assists the application in selecting the appropriate key. A TSM transition function invokes begin_cem. The TSM decrypts the sequence of keys from the root to the requested key and then uses it to perform the requested cryptographic operation. Alternatively, keychain management could occur outside of the TSM, in which case the TSM interface would accept the encrypted key directly.

The ring policy requires that the TSM must be at least as privileged as the most privileged subject using it. If the TSM is used to encrypt and decrypt objects in ring 0, which is allocated to the OS, the TSM is placed in ring 0. In this case, access to the TSM from lesser-privileged rings is controlled by the OS – the applications call interfaces exported by the OS, which in turn uses the application's parameters to invoke the interfaces of TSM. The OS additionally performs TSM-related request queuing and dispatch, to provide an orderly execution environment for the applications. In general, the ring level of a TSM is dependent on the customer-site security policy for execution of that particular TSM.

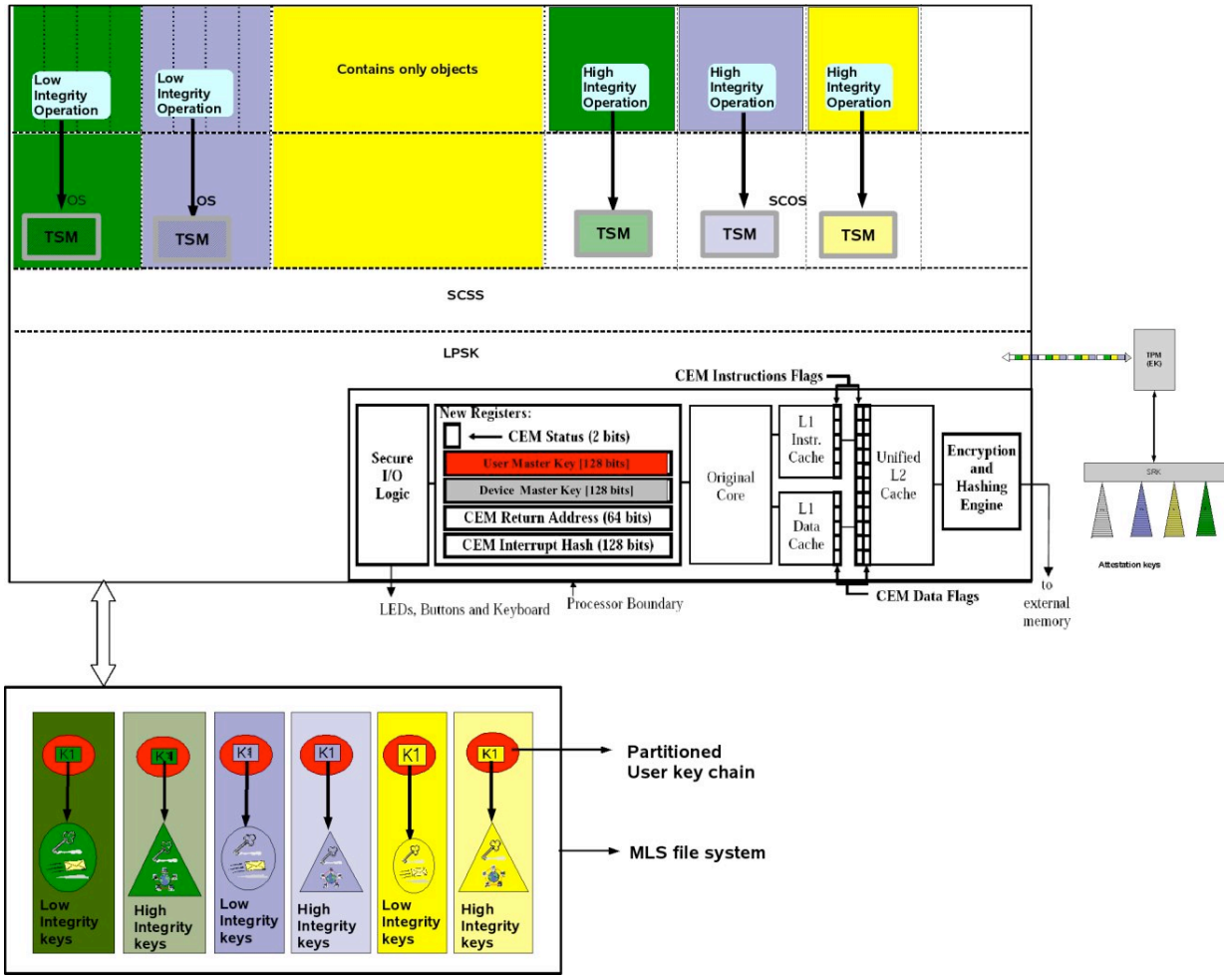


Illustration 4: SP integration with SecureCore MLS architecture

To protect against covert channels, the SC architecture must ensure that no block can lock shared resources, such as the CEM, from access by other blocks. SP instructions are added for the LPSK to virtualize the state of SP across kernel blocks, including instructions for saving and restoring the state of the SP mechanism.

The OS is unable to read or write to the SP internal hash/return address/state registers as the instructions required to perform such operations are privileged and can only be invoked from ring -2. If an interrupt for a context switch between blocks occurs during a CEM session, the LPSK saves the entire state of the processor, including SP state/return address/internal hash registers, and restores the state of the next block.

The LPSK, SCSS and SCOS are carefully designed so that on a block-level context switch, residual data from the previous block is not available to the user, OS or applications, if that availability would violate the MLS security policy.

Covert storage channels are eliminated by wiping clean any state of data present in the

caches in the processor before the state of the next block is restored. Timing channels are avoided, since the CEM will not be locked when the next block executes, and the time for block context switching is made consistent, regardless of CEM activity.

3.2.1 Key Chain Management and Enforcement of MLS policies

Key chains are resources that can be shared by applications within a block as well as between blocks. Unlike the assumption about key chain location and access in [rlee05], the keys and key chains are stored in LPSK-managed resources according to their confidentiality and integrity levels, as shown in Illustration 4, and the MLS policy governs the access rights of different blocks to the key chains. This approach is free of covert storage channels and is the same as used in some secure multi level file systems [irvi95]. Thus applications in each block will only be able to perform read and write operations on key chains if the MLS policies permit.

Since access to keys and key chains is controlled by the LPSK, the key chain managers are not MLS aware and consequently need not be trusted with enforcing the MLS policy [irvi95]. Further, only the key chain managers in the high integrity blocks, which hosts high integrity applications need to be high integrity .

IV. SecureCore Usage Scenarios

SecureCore can be configured to allow read down and controlled write up of exported resources with respect to confidentiality, and read up and controlled write down with respect to integrity. With these capabilities, read down and write up of encrypted data by the user can be permitted as a part of normal operation. In this section, several scenarios illustrate how this is managed in SC.

In these examples, we assume that the existence of a kernel k_append operation for performing a "blind write" to a higher level. The application A has handles A.k, A.m for each key K or memory segment M in its address space, which provide the only access to those resources for A. For keys, the handle can be as simple as the numerical index into the keychain, or can reflect the semantics of the application, such as a mail-signing key and a mail-encryption key. The confidentiality level of the handle is the user's session level, and we will assume here that: (1) the keys used for encryption are labeled the same as the data they encrypt, and (2) users are not allowed to use keys that are labeled at a higher confidentiality level than their session level.

OS operations are prefaced with "os_," SCSS operations with "scss_" and kernel operations with "k_".

Scenario 1: A user at a low confidentiality level wants to write encrypted data to a higher level.

Application *A* level: SECRET
Data *D* level: SECRET
Buffer tmp level: SECRET
Key *K* level: SECRET
Memory Segment *M* level : TOP SECRET (TS)
Application operations: `tmp := os_encrypt(A.d, A.k);`

```
k_append(A.m, tmp);
```

Scenario 2: A user at a high security level wants to read encrypted data that is at a lower security level.

Application *A* level: TS
Encrpted Data D^k level: SECRET
Buffer tmp level: TS
Key *K* level: SECRET
Application operations: `tmp := os_decrypt(A.d^k, A.k);`
`os_read(tmp);`

Scenario 3: A privileged user wants to downgrade data and encrypt it with a lower level key.

Application *A* level: TS
Data *D* level: TS
Buffer tmp level: TS
Key *K* level: SECRET
Memory Segment *M* level : TOP SECRET (TS)
Application operations: `scss_downgrade(A.d, SECRET);`
`os_encrypt(A.d, A.k);`

Scenario 4: A user at a low confidentiality level wants to append low level keys from her keychain to a high level keychain.

Application *A* level: SECRET
Key *K* level: SECRET
Keychain *C* level : TOP SECRET (TS)
Application operations: `scss_kchain_append(A.c, A.k)`

Note that Scenario 4 does not address integrity of the keys and keychain. For example, a low integrity SECRET key could corrupt the TS keychain, and a low integrity application could flood SCSS with requests, leading to denial of service for other processes needing to use SCSS, or exhaustion of the TS keychain space. SC can be configured to avoid these problems through the use of integrity labels, which would prevent writing low integrity data to a high integrity object. See scenario 6.

Scenario 5: A user wants to create and append a keys to her keychain.

Application *A* level: SECRET
Key *K* level: SECRET (initially empty)
Buffer tmp level: SECRET
Keychain *C* level: SECRET
Application A operation: `app_get_key(tmp), A.k =`
`OS_kchain_add(C, tmp)`

Scenario 6: A a high integrity user at a low confidentiality level wants to append low confidentiality, high integrity keys to a high confidentiality, high integrity keychain.

Application *A* level: SECRET/HIGH_INTEG
Key *K* level: SECRET/HIGH_INTEG
Keychain *C* level : TS/HIGH_INTEG
Application operations: `scss_kchain_append(A.c, A.k);`

References

[adva05] Advanced Micro Devices, AMD "Pacifica" Virtualization Technology, 2005.

[benz05] Benzel, T. V., Irvine, C. E., Levin, T. E., Bhaskara, G., Nguyen, T. D., and Clark, P. C., "Design Principles for Security", NPS-CS-05-010, Naval Postgraduate School, September 2005.

[golsd72] R. Goldberg, "Architectural Principles for Virtual", Computer Systems . Ph.D. thesis, Harvard University, Cambridge, MA, 1972.

[inte05] Intel Corp, "Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture", 2005.

[irvi05] Cynthia Irvine, Terry Benzel, et. al. , National Science Foundation Poster - CyberTrust meeting, Sept 26, 2005 (PDF)

[irvi95] Irvine, Cynthia E., "A Multilevel File System for High Assurance," Proceedings 1995 IEEE Symposium on Security and Privacy, Oakland, CA, pp. 78-87, May 1995.

[reed79] Reed, D.P., and R. K. Kanodia, "Synchronization with Event counts and Sequencers", Communications of the ACM, 22(2):115-- 123, Februrary 1979.

[rlee05] Ruby B. Lee, Peter C. S. Kwan, John Patrick McGregor, Jeffrey Dwoskin, and Zhenghong Wang, "Architecture for Protecting Critical Secrets in Microprocessors", Proceedings of the 32nd International Symposium on Computer Architecture (ISCA 2005), pp. 2-13, June 2005.

[smit05] Sean Smith, "Trusted Computing Platforms - Design and Applications", ISBN: 0-387-23916-2, Springer 2005.

[tcgp05] Trusted Computing Group, TPM Main, Part 1, "Design Principles, Specification Version 1.2, Revision 85", 13 February 2005.