Memory-centric Security Architecture

Weidong Shi Chenghuai Lu Hsien-Hsin S. Lee

College of Computing
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332

Abstract. *This paper presents a new security architecture for protecting software confidentiality and integrity. Different from the previous process-centric systems designed for the same purpose, the new architecture ties cryptographic properties and security attributes to memory instead of each individual user process. The advantages of such a memory centric design are many folds. First, it provides a better security model and access control on software privacy that supports both selective and mixed tamper resistant protection on software components from heterogeneous sources. Second, the new model supports and facilities tamper resistant secure information sharing in an open software system where both data and code components could be shared by different user processes. Third, the proposed security model and secure processor design allow software components protected with different security policies to inter-operate within the same memory space efficiently. Our new architectural support requires small silicon resources and its performance impact is minimal based on our experimental results using commercial MS Windows workloads and cycle based out-of-order processor simulation.*

1 Introduction

Recently, there is a growing interest in creating tamper-resistant/copy protection systems that combine the strengths of security hardware and secure operating systems to fight against both software attacks and physical tampering of software [2, 3, 6, 7, 11–13]. Such systems aim at solving various issues in the security domain such as digital rights protection, virus/worm detection, intrusion prevention, digital privacy, etc. For maximum protection, a tamper-resistant/copy protection system should provide protection against both software and hardware based tampering including duplication (copy protection), alteration (integrity and authentication), and reverse engineering (confidentiality).

Many the aforementioned copy protection systems achieve protection by encrypting the instructions and data of a user process with a single master key. Although such closed systems do provide security for software execution, they are less attractive for real world commercial implementations because of the gap between a closed tamper-resistant/copy protection system and real world applications that are mostly multi-domained where a user process often consists of components coming from heterogeneous program sources with distinctive security requirements. For instance, almost all the commercial applications use statically linked libraries and/or dynamically linked libraries (DLL). It is quite

natural that these library vendors would prefer a separate copy protection of their own intellectual properties decoupled from the user applications. Furthermore, it is also common for different autonomous software domains to share and exchange confidential information at both the inter- and intra- process levels. The nature of de-centralized development of software components by different vendors makes it difficult to enforce a process centric protection scheme.

Traditional capability-based protection systems such as Hydra [1] and CAP [4] although provide access control on information, they were not designed for tamper resistance to prevent software duplication, alternation, and reverse engineering. Specifically, they do not address how access control interacts with other tamper resistant protection mechanisms such as hardware based memory encryption and integrity verification.

In this paper, we present a framework called *MEmory-centric Security Architecture* or *MESA* to provide protection on software integrity and confidentiality using a new set of architectural and OS features. It enables secure information sharing and exchange in a heterogeneous multi-domain software system. The major contributions of our work are summarized as follows:

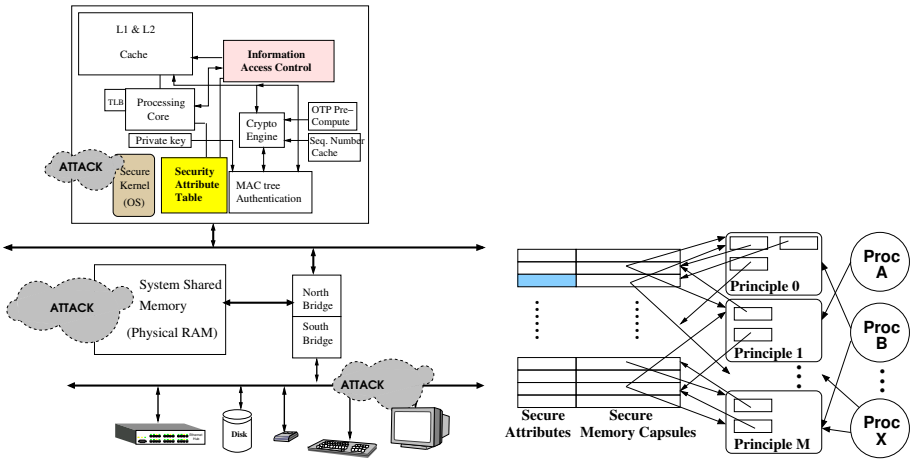
- We presented and evaluated a unique memory-centric security model for tamper resistant secure software execution. It distinguishes from the existing systems by providing better support for inter-operation and information sharing among software components in an open heterogeneous multi-domain software system.
- We introduced architecture innovations that allow efficient implementation of the proposed security model. The proposed secure processor design incorporates and integrates fine-grained access control of software components, rigorous anti-reverse engineering, and tamper resistance.
- We discussed novel system mechanisms to allow heterogeneous program components to have their own tamper resistant protection requirements and still to be able to inter-operate and exchange information securely.

The rest of the paper is organized as follows. Section 2 introduces MESA which is extended in Section 3 that details each MESA component. Evaluation and results are in Section 4. Discussion of related work is presented in Section 5 and finally Section 6 concludes.

2 Memory-centric Security Architecture

In this section, we overview our *Memory-centric Security Architecture* (MESA). Using novel architectural features, MESA enables high performance secure information sharing and exchange for a multi-security domain software system. Figure 1(a) shows MESA and its operating environment.

Now we present MESA from system perspective. One critical concept of MESA is the *memory capsule*. A memory capsule is a virtual memory segment with a set of security attributes associated with it. It is an information container that may hold either data or code or both. It can be shared by multiple processes. For example, a DLL is simply a code memory capsule. A set of security attributes are defined for each memory capsule besides its location and size. These security attributes include security protection level, one or more symmetric memory



(a) Memory-centric Security Model

(b) Principles and Security Memory Capsules

Fig. 1. Memory-centric Security Architecture (MESA)

encryption keys, memory authentication signature, accesses control information, etc. For each process, the secure OS kernel maintains a list of memory capsules and their attributes as process context. During software distribution, software vendors encrypt the security attributes associated with a memory capsule using the target secure processor's public key. The secure processor authenticates and extracts the security attributes using the corresponding private key. A secure processor never stores security attributes in the exposed physical RAM without encryption protection.

Another important concept is *principle*. A principle is an execution context smaller than a process. It is defined as the execution of secure code memory capsules having the same security property within a user process. Principles are associated with memory capsules. They can be considered owners of memory capsules. Based on the associated principles and security protection levels, access control of memory capsules can be carried out. An active principle is a currently executing principle. When an active principle accesses some memory capsule, the access will be checked. If the active principle is allowed to access the memory capsule, the access will be granted, otherwise, security exception of access violation will be raised. Note that the *principle* in MESA is different from the *principle* defined in capability system such as [1] and [4].

MESA allows different keys been used to protect separate memory capsules and enforces access control during program execution. However, the scenarios of fine-grained dynamic information sharing frequently happens during program execution. For example, 1) An application calls OS services such as fwrite, fread and passes pointers to data buffer owned by the application; 2) An application

calls OS and system services to get a pointer to data structures owned by the OS or system libraries; 3) A principle calls a routine of another principle and the caller requires the callee to either operate or modify some data content in a data memory capsule it owns.

2.1 Secure Capsule Management

Most functionality of the secure capsule management is achieved by a secure OS kernel. Among the major services provided by the secure kernel are, process and principle creation, principle authentication, principle access control.

First, during a process creation, the secure OS kernel will create a list of memory capsules associated with the process. The secure kernel creates application process from the binary images provided by software vendors. Each binary image may contain one or more protected code and data sections, each one with its own security attributes. Security attributes of binary images for the application, middleware, and system shared libraries are set independently by their corresponding vendors. The secure kernel creates a secure memory capsule context based on the binary images. Each memory capsule represents an instantiation of either a code module or data module and is uniquely identified with a randomly generated ID. For DLLs, a different capsule is created with a different ID when it is linked to a different process. However note that the code itself is not duplicated. It is simply mapped to the new process's memory space with a different capsule entry in the capsule context table.

Execution of a process can be represented as a sequence of executing principles. Heap and stack are two types of dynamic memory that a principle may access. Privacy of information stored in the heap and stack is optionally protected by allocating private heap and stack memory capsule to each principle. Another choice is to have one memory capsule to include both protected code image and memory space allocated as private heap and stack. When execution switches to a different principle, the processor's stack pointer is re-loaded so that it will point to the next principle's private stack. Details of private stack are presented in the subsection of intra-process sharing.

With the concept of private heap, it comes the issue of memory management of private heaps associated with each principle. Does each principle require its own heap allocator? The answer is no. Heap management can be implemented in a protected shared system library. The key idea is that with hardware supported protection on memory integrity and confidentiality, the heap manager can manage usage of each principle's private heap but cannot tamper its content.

To provide a secure sharing environment, support for authenticating principles is necessary. MESA supports three ways of principle authentication. The first approach is to authenticate a code memory capsule and its principle through a chain of certification. A code memory capsule could be signed and certificated by a trusted source. If the certification could be verified by the secure kernel, the created principle would become a trusted principle because it is certified by a trusted source. Another approach is to authenticate a principle using a public key supplied by software vendors. This provides a way of private authentication. The third way is to certify principle using secure processor's public key. For example, application vendors can specify that the linked shared libraries must be

certified by the system vendors such as Microsoft and the middle-ware image must be certified by the known middle-ware vendors. Failure of authenticating a code image will abort the corresponding process creation.

2.2 Intra Process Sharing

Intrinsic	Parameters	Explanation
sec_malloc(s,id)	s: *size* ; id: *principle id*	allocate memory from a principle's private heap
sec_free(p)	p: *memory pointer*	free memory to its owner's private heap
sec_swap_stack (addr)	addr: *address*	switch the active stack pointer to another principle's private stack. Addr points to a location of the target principle. Save <active stack pointer, active principle id> to the stack context table
sec_get_id (name)	name: *capsule name*	get id of a principle (secure kernel service)
sec_push_stack_ptr()		read the current executing principle's stack pointer from the stack context table and push it into its private stack
sec_save_ret_addr (addr)	addr: *address*	assign addr to a return address register (RAR)
sec_return()		assign RAR to PC and execute
sec_add_sharing_ptr (p, s, id, rw)	p: *pointer;* s: *size* ; id: *principle id* ; rw: *access right*	allow target principle (id) to access memory region [p, p+s) with access right rw, return a security pointer that can be passed as function parameter (secure kernel service)
sec_remove_ptr (p)	p: *security pointer*	remove access right granted to security pointer p
sec_save_security_ptr (reg, addr)	reg: *register holding security pointer* ; addr: *address*	save security pointer to memory
sec_load_security_ptr (reg, addr)	reg: *register holding security pointer* ; addr: *address*	load security pointer from memory

Fig. 2. MESA Security Intrinsics

There are many scenarios that pointers need to be shared by multiple principles. One principle calls a routine of another principle and passes pointers to data belonging to a confidential data memory capsule. In this scenario, information security will be violated if the callee's function attempts to exploit the caller's memory capsule more than what is allowed. For instance, the caller may pass a memory pointer, *mp* and length *len*, to the callee and restricts the callee to access only the memory block *mp[0, len-1]*. However, the callee can spoil this privilege and try to access to the memory at *mp[-1]*. This must be prevented. Passing by value may solve the problem but it is less desired because of its cost on performance and compatibility with many popular programming models. MESA facilitates information sharing using explicit declaration of shared subspace of memory capsules. The owner principle of a memory capsule is allowed to add more principles to share data referenced by a pointer that points to its memory capsule. However, the modification is made on a single pointer basis. The principle can call sec_add_sharing_ptr(addr, size, principle_p) intrinsic to declare that *principle_p* is allowed to access the data in the range [addr, addr+size]. The declaration is recorded in a pointer table. Dynamic access to the memory capsule is checked against both the memory capsule context and the pointer table.

```
// application: my_app
// middleware: my_middle as DLL
void* p;
unsigned int my_id, middle_id;
security void* sp;
int ret;

my_id = sec_get_id("my_app");
p = sec_malloc(0x20, my_id);
...
middle_id = sec_get_id("my_middle");
sp = sec_add_sharing_ptr(p, 0x20,
      middle_id, RD|WR);
ret = my_middle_foo(sp, 0x10);
sec_remove_ptr(sp);
...
sec_free(p);
```

(a) Printer Sharing

```
// CALLER SIDE
push ebp //save stack frame pointer
sec_swap_stack addr_of_my_middle_foo
// stack pointer switched to the callee's stack
// caller's esp saved to the context table
push 0x10; // input parameter
call my_middle_foo
//push return PC to the callee's stack
pop ebp //get stack frame pointer back

// CALLEE SIDE
sec_push_stack_ptr
ebp = esp //ebp stack frame pointer
...
r1 = [ebp + 4] //return address
sec_save_ret_addr r1 //save caller's return
esp = [ebp]
sec_swap_stack r1
// restore stack pointer to the caller's stack
// callee's esp saved to the context table
sec_return //return to the caller
//return to the caller by loading return PC
```

(b) Securely Maintain Correct Stack Behavior

Fig. 3. Cross Principle Function Call

After the pointer is consumed, it is removed from the pointer table by another intrinsic sec_remove_ptr(addr). This mechanism allows passing pointers of either private heap or private stack during a cross-principle call.

Aside from the above two intrinsics, MESA also proposes other necessary intrinsics for managing and sharing secure memory capsules as listed in Figure 2. Note that these security intrinsics are programming primitives not new instructions. Although a hardware implementation of MESA can implement them as CISC instructions, yet it is not required.

Figure 3(a) illustrates how to securely share data during a function call using MESA's security intrinsics. There are two principles. One is "my_app" and the other is a middleware library called "my_middle". "my_app" allocates a 32-byte memory block from its own private heap by calling sec_malloc(0x20, my_id). Then it declares a sharing pointer p using intrinsic sec_add_sharing_ptr that grants principle "my_middle" read/write access to the memory block pointed by p. The intrinsic call returns a security pointer, sp. After declaring the security pointer, principle "my_app" calls my_middle_foo() of principle "my_middle", passes the security pointer and another input argument. Inside my_middle_foo(), codes of principle "my_middle" can access the private memory block defined by the passed security pointer sp. After the function returns control back to principle "my_app", "my_app" uses intrinsic sec_remove_ptr to remove the security pointer thus revoking "my_middle"'s access to its private memory block. In addition to passing shared memory pointers, during the cross-principle call, program stack has to securely switch from the caller's private stack to the callee's private stack using stack-related security intrinsics.

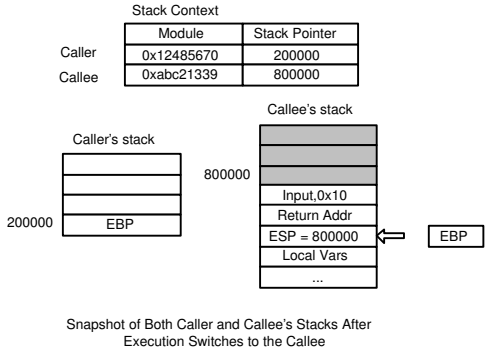


Fig. 4. Snapshot of Stacks After a Cross-Principle Call

To show how private stacks are protected during cross-principle call, we show what happens at assembly level when "my_app" calls my_middle_foo() and the exit code of my_middle_foo() in Figure 3(b). The assembly code uses x86 instructions and MESA intrinsics. In the example, the caller switches stack pointer to the callee's private stack using sec_swap_stack intrinsic. Input addr is either a function entry address or return address if the intrinsic is used to switch stack pointer from the callee's private stack to the caller's. MESA maintains a table of stack pointer context for all the running principles. When sec_swap_stack(addr) is executed, it will save the current active stack pointer as a <principle, stack pointer> pair and set the active stack pointer to the target principle's. Then it pushes values to the callee's stack memory capsule. When the callee's stack capsule requires information to be encrypted, the pushed stack values will be encrypted using the callee's key. Since the caller and the callee use different stacks, to maintain compatibility with the way how stack is used for local and input data, the callee uses sec_push_stack_ptr intrinsic at the function entry point to push its stack pointer from the context table into its private stack. Figure 4 shows both the caller and the callee's stacks after the discussed cross-principle call. When execution switches back from "my_middle" to "my_app", the callee copies the returned value to the caller's stack capsule. Since the return PC address is saved in the callee's private stack (happens during execution of function call instruction after stack swap), the callee has to read the return address and put it into a temporary register using sec_save_ret_addr. Then the callee switches the stack pointer to the caller's. Finally, the callee executes sec_return intrinsic that assigns the returned address stored in the temporary register to the current program counter. In the case where a large amount of data need to be returned, the caller could reserve the space for the returned value on its stack by declaring a security pointer pointing to its stack and passes it to the callee.

Note that MESA protects against tampering on the target principle's stack by only allowing one principle to push values to other principle's stack. A principle can not modify another principle's stack pointer context because only the owner principle can save the active stack pointer as its stack pointer context according

to the definition of sec_swap_stack. Explicitly assigning values to the active stack pointer owned by a different principle is prohibited.

2.3 Inter Process Sharing - Shared Memory

MESA supports tamper-resistant shared memory through access control by the secure kernel. In MESA, each memory capsule can be owned by one or more principles as shown in Figure 1(b). Access rights (read/write) to a secure memory capsule could be granted to other principles. For secure sharing, the owner principle specifies the access right to be granted only to certain principles that are authenticated. Using this basic secure kernel service, secure inter-process communication such as secure shared memory and secure pipe can be implemented. For example, assume that P1 and P2 are two user principles belonging to different processes that want to share memory in a secure manner. Principle P1 can create a memory capsule and set a sharing criteria by providing a public authentication key. When principle P2 tries to share the memory created by P1, it will provide its credential, signed certificate by the corresponding private key to the secure kernel. The secure kernel then verifies that P2 can be trusted and maps the capsule to the memory space of P2's owning process.

3 Architectural Support for MESA

In this section, we discuss the architectural support for MESA. Inside a typical secure processor, we assumed a few security features at the micro-architectural level that incorporate encryption schemes as well as integrity protection based on prior art in [2, 7, 6]. In addition, we introduce new micro-architecture components for the MESA including a *Security Attribute Table (SAT)*, an *Access Control Mechanism (ACM)*, and a *Security Pointer Buffer (SPB)*. Other new system features are also proposed to cope with MESA in order to manage the security architecture asset.

3.1 Security Attribute Table

Secure memory capsule management is the heart of MESA. Most of the functionality for secure memory capsule management is implemented in the secure OS kernel that keeps track of a list of secure memory capsules used by a user process. Security attributes of frequently accessed secure memory capsules are cached on-chip in a structure called Security Attribute Table (SAT). Figure 5 shows the structure of a SAT attached to a TLB. Each entry stores a set of security attributes associated with a secure memory capsule. The crypto-key of each SAT entry is used to decrypt or encrypt information stored in the memory capsule. The secure kernel uses intrinsic sec_SATld(addr) to load security attributes from the memory capsule context stored in memory to SAT. The crypto-keys in the memory capsule context are encrypted using the secure processor's public key. The secure processor will decrypt the keys when loading them into the SAT.

A secure memory capsule could be bound to one or many memory pages of a user process. When the entire user virtual memory space is bound to only

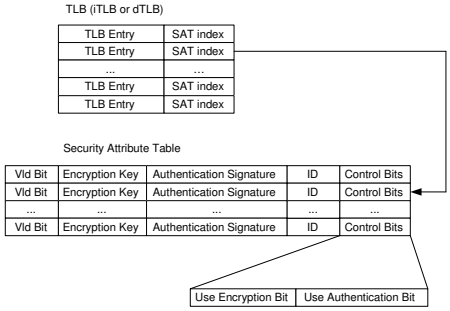


Fig. 5. Security Attribute Table (SAT) and TLB

one secure memory capsule, the model is equivalent to a process-centric security model. As Figure 5 shows, each TLB entry has an index to SAT for retrieving the security attributes of its corresponding memory page. During context switches, the secure kernel authenticates the process's memory capsule context first, then loads security attributes into SAT. SAT is accessed for each external memory access. For a memory access missing in the cache, data in the external memory will be brought into the cache, decrypted using the encryption key in the SAT and its integrity verified against the root memory authentication signature, also stored in the SAT using hash tree [7]. On-chip caches maintain only plaintext. SAT is also accessed when data is evicted from the on-chip caches. The evicted data will be encrypted using keys stored in the SAT and a new memory capsule root signature will be computed to replace the old root signature.

If the required security attributes could not be found in the SAT, a SAT fault is triggered and the secure kernel will take over. First, the secure kernel flushes the cache, then load the required security attributes into the SAT. The SAT indexes stored in the TLB are also updated accordingly.

3.2 Access Control Mechanism

Efficient hardware-based access control plays a key role for protecting memory capsules from being accessed by un-trusted software components. It is important to point out that having software components or memory capsules encrypted does not imply that they can be trusted. Adversaries can encrypt a malicious library and have the OS to load it into an application's virtual space. The encrypted malicious library despite encrypted can illegally access confidential data.

Access management is achieved by associating principles with memory capsules they can access. The information is stored in a table. Based on the security requirements, the secure processor checks the table for every load/store operation and the operation is allowed to be completed only if it does not violate any access restraint. To speed up memory operations, frequently accessed entries of the rule table can be cached inside the processor by an *Access Lookup Cache (ALC)*. Entries of ALC are matched based on the current executing principle's ID and the target memory capsule's ID. To avoid using a CAM for ALC implementation, executing principle's ID and the target memory capsule's ID can be

XORed as the ALC index shown in Figure 6. For a memory operation and ID tuple <running principle ID, memory capsule ID, rd, wr>, if a matching ALC entry can not be found, an ALC miss exception will be triggered and program execution will trap into the secure kernel. The secure kernel will check the complete access control table to see whether a matching entry can be found there. If an entry is found, it will be loaded into the ALC. Failure to find an entry in both the ALC and the complete access control table implies the detection of an illegal memory operation. Upon detection of such an operation, the secure kernel will switch to a special handling routine where proper action will be taken either terminating the process or nullifying the memory operation.

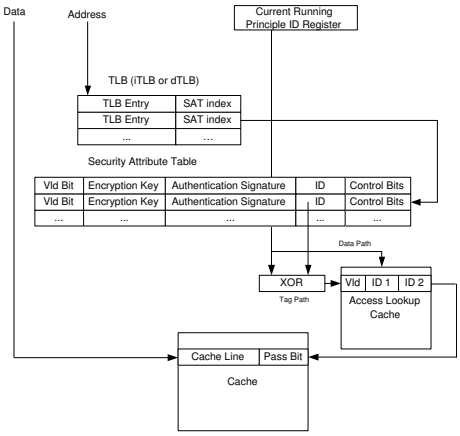


Fig. 6. Information Access Control Mechanism

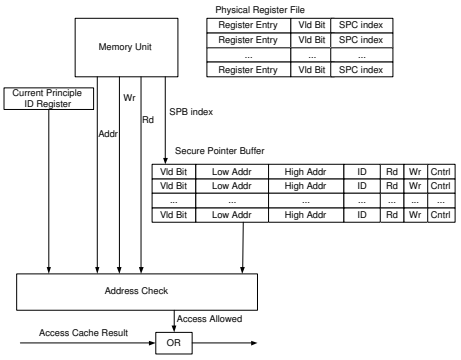


Fig. 7. Security Pointer Buffer

To minimize performance impact of ALC lookup on cache access, ALC checking can be conducted in parallel with storing information into the cache, thus incurs almost no performance loss since stores are typically not on the critical path. As shown in Figure 6, there is one pass bit in each cache line indicating whether the stored data yet passes access control security checking. When data is written to a cache line, the bit will be clear. When ALC lookup returns a hit, the bit will be set. To guarantees that a secure memory capsule is not updated by instructions whose principles do not have write access right, cache lines with the checked bit clear are inhibited from being written back to the memory. Furthermore, read access to a cache line with the check bit set is stalled until the bit is set. The performance impact of the above mechanism on cache access is minimal because as our profiling study indicates the interval between the same dirty line access is often long enough to hide the latency of ALC lookup.

3.3 Security Pointer Buffer

MESA requires tracking shared pointers in a security pointer table. To improve performance, values of the pointer table are cached in an on-chip *Security Pointer*

Buffer (SPB) shown in Figure 7. Each entry records the low and high address of the shared memory region and the principle ID that is granted temporary access right. A new programming data type, security pointer is required. The difference between a security pointer and a regular pointer is that aside from the address value, security pointer also maintains other information for identifying the declared pointer such as an index to the SPB, see Figure 7. If a memory address held by a register is added to the SPB, the involved register will be tagged with the index of the added entry in the SPB and a security pointer valid bit is set. When the register value is assigned to another register, the index is also passed. When a null value is assigned to the register, the valid bit is cleared. When address stored in a security pointer is used to access memory and its valid bit is set, the associated index is used to retrieve the corresponding entry in the SPB. The memory address is compared against the memory range stored in the SPB, and ID of the running principle is also compared with the ID stored in the corresponding SPB entry. If the memory address is within the range, the principle IDs are identical, and the type of access (read/write) also matches, the memory reference based on the security pointer is accepted. Otherwise, a security pointer exception will be raised.

Security pointer exception is raised only when memory access is issued using a valid security pointer and there is a mismatch in the SPB. The exception is handled by the secure kernel. There are two possible reasons for a SPB access failure. First, the entry may be evicted from the SPB to the security pointer table and replaced by some other security pointer definition. In this case, a valid entry for that security pointer can be found in the security pointer table stored in the external memory. If the access is found to be consistent with some security pointer declaration, it should be allowed to continue as usual. Second, the security pointer declaration is reclaimed and no longer exists in both the SPB and the security pointer table. This represents an illegal memory reference. Execution will switch to a secure kernel handler on illegal memory reference and proper action will be taken. Two new instructions, sec_save_security_ptr and sec_load_security_ptr are proposed to support save/load security pointers in physical registers to/from the external memory. SPB and security pointer definition table are part of the protected process context. They are securely preserved during process context switch.

3.4 Integrity protection under MESA

The existing integrity protection schemes for security architectures are based on an m-ary hash/MACtree [7]. Under the memory-centric model in which information within a process space is usually encrypted by multiple encryption keys, the existing authentication methods cannot be directly applied. Toward this, we generalize the integrity protection tree structure. We first protect individual memory capsules with their own hash/MAC (message authentication code) tree, and then, a hierarchical hash/MAC tree is constructed over these capsule-based hash/MAC trees. To speed up integrity verification, frequently accessed nodes of the hash/MAC trees are cached on-chip. When a new cache line is fetched, the secure processor verifies its integrity by inserting it into the MAC/hash tree. Starting from the bottom of the tree, recursively, a new MAC or hash value is

computed and compared with the internal MAC/hash tree node. The MAC/hash tree is always updated whenever a dirty cache line is evicted from the secure processor. The secure processor can automatically determine the memory locations of MAC/hash tree nodes and fetch them automatically during integrity check if they are needed. Root of the MAC/hash tree is preserved securely when a process is swapped out of the processor pipeline.

4 Performance Evaluation

4.1 Simulation Framework

We used TAXI [9] as our simulation environment. TAXI includes two components, a functional open-source Pentium emulator called Bochs capable of performing full system simulation, and a cycle-based out-of-order x86 processor simulator. Bochs models the entire platform including Ethernet device, VGA monitor, and sound card to support the execution of a complete OS and its applications. We used Windows NT 4.0 SP6 as our target. Both Bochs and the simulator were modified for MESA. Proposed enhancement such as SAT, ALC, and SPB were modeled. We measure the performance under two encryption schemes, XOM/Aegis-like scheme and our improved scheme called *M-Tree*. The XOM/Aegis scheme uses block cipher (triple-DES and AES) for software encryption. In our evaluation, we selected AES as one encryption scheme. The other scheme is based on stream cipher which can be faster than block ciphers as the crypto-key stream can be pre-computed or securely speculated [5].

Since we have no access to the Windows source codes, simplification was taken to facilitate performance evaluation. Software handling of the MESA architecture was implemented as independent interrupt service routines. These routines maintain the SAT, ALC, and SPB using additional information on memory capsules provided by the application. Executing process is recognized by matching CR3 register (pointing to a unique physical address of process's page table) with process context.

To model the usage of MESA, we assume that the application software and the system software are separately protected. The system software includes all the system DLLs mapped to the application space including kernel32.dll, wsock32.dll, gdi32.dll, ddraw.dll, user32.dll, etc. The middleware libraries, if used, are also separately protected. Note that all the system libraries mentioned above are linked to the user space by the OS and they are invoked through normal DLL call. To track the data exchanged among the application, middleware DLLs, and system DLLs, dummy wrapper DLLs are implemented for the DLLs that interface with the application. These dummy DLLs are API hijackers. They can keep track of the pointers exchanged between an application and the system and update the security pointer table and the SPB accordingly. Usage of dynamic memory space such as heap and stack are traced and tagged. The assumption is that each protected code space uses its own separate encryption key to guarantee privacy of its stack and dynamically allocated memory space. Application images are modified using PE Explorer tool so that they will link with the wrapper DLL functions.

Parameters	Values	Parameters	Values
Fetch/Decode width	8	Issue/Commit width	8
L1 I-Cache	DM, 8KB, 32B line	L1 D-Cache	DM, 8KB, 32B line
L2 Cache	4way, Unified, 32B line, 512KB	L1/L2 Latency	1 cycle / 8 cycles
MAC tree cache size	32KB	RM cache size	8KB
I-TLB	4-way, 256 entries	D-TLB	4-way, 256 entries
SHA256 latency	80ns / 80ns	ALC latency	1 cycle
SPB size	64, random replacement	SPB latency	1 cycle

Table 1. Processor model parameters

Table 1 lists the architectural parameters experimented. A default latency of 80ns for both SHA-256 and AES were used for the cryptographic engines in the simulation assuming that both units are custom designed. Seven NT applications were experimented: IE6.0, Acrobat Reader 5.0, Windows Media Player 2, Visual Studio 6.0, Winzip 8.0, MS Word, and Povray 3. The run of IE includes fetching webpages from yahoo.com using Bochs's simulated Ethernet driver. The run of Visual Studio includes compilation of Apache source code. Winzip run consists of decompressing package of Apache 2.0. Our run of Media Player includes play of a short AVI file of Olympics figure-skating. The input to Acrobat Reader is an Intel IA64 system programming document. The run includes search for all the appearance of the keyword "virtual memory". The run of MS Word consists of loading an IEEE paper template, type a short paragraph, cut/paste, and grammar checking. The run of Povray renders the default image.

4.2 Performance Evaluation

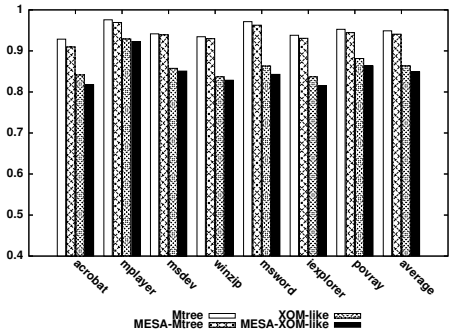


Fig. 8. Normalized IPC Results with MESA Support

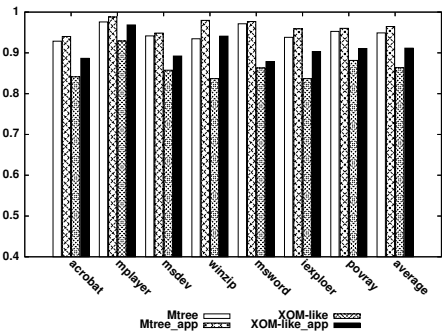


Fig. 9. Normalized IPC Results for Selective Protection

We first evaluate the performance of access control, and security pointer table. We use a 32-entry SAT for storing keys and security attributes. This is large enough to hold all the protected memory capsules in our study. The number of entries in the *security pointer buffer* is 64. We evaluated three scenarios, baseline where security protection is turned off, *M-Tree* protection on the whole virtual space with one memory capsule, and protection using MESA.

Performance results for all the applications are shown in Figure 8, in which the IPC results were normalized to that of the baseline. On average, the *M-Tree* lost 5% performance. When MESA is used, it further slows down by another 1%. Block cipher on average lost 13% performance and introducing MESA will cause another 1.8% loss. Note that when the whole user space is protected as one memory capsule (degenerated case), *security pointer buffer* and access control are no longer providing any valuable service and could be turned off. There will be no further performance loss under this scenario.

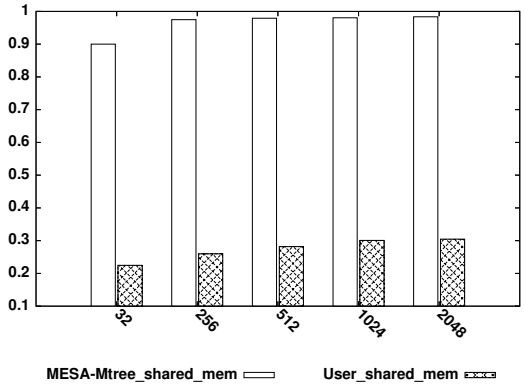


Fig. 10. Normalized Throughput of Protected Inter-process Shared Memory for MESA M-Tree under Different Shared Memory Sizes

One objective of MESA is to provide a secure environment so that software components co-existing in the same space could be selectively encrypted and protected. To evaluate the gain of selective protection, we selectively encrypt only application and its supporting DLLs, leave all the system libraries (e.g. gdi32.dll, wsock32.dll) un-encrypted. The normalized IPC normalized are shown in Figure 9. For *M-Tree*, the performance gain is about 1.6% comparing with encrypting everything in the memory space. This is substantial considering the slowdown of *M-Tree* is merely 5%. For the block ciphers, the gain is 5.5%. This means that by ensuring a secure environment for information sharing, MESA could actually improve the overall system performance, especially for the block cipher tamper resistant systems.

Another advantage of MESA is that it supports secure and high performance shared memory for inter-process communication. Secure inter-process communication could not be naturally supported by process-centric tamper-resistant system. The strong process isolation demands that information be either exchanged through secure socket or encrypted by software via a negotiated key between the two involved processes. MESA supports protected shared memory with almost no loss on performance. We used a micro-benchmark to evaluate the performance of shared memory under MESA vs. a process-centric architecture. A pair of Windows producer and consumer programs using shared memory were written for this purpose. There are three scenarios, 1) the shared memory is

not protected, which is used as the baseline; 2) the shared memory is protected as an encrypted secure memory capsule; 3) the shared memory is protected by applications themselves through a strong cipher (AES) with a separately negotiated key by the producer and consumer. The pair of programs were simulated in TAXI and throughput results were collected for the above three scenarios under different size of shared memory. Figure 10 clearly shows the advantage of MESA for supporting confidentiality and privacy of shared memory. For small size shared memory, the loss of throughput is about 10%. But as the size of shared memory grows, MESA protected shared memory performs almost as fast as one without protection.

5 Related Work

Software protection and trusted computing are among the most important issues in information security. Traditionally, the protections on software are provided through a trusted OS. To improve the security model, some tamper resistant devices are embedded into a computer platform to ensure the loaded OS is trusted. A typical example of such endeavor is the TPM and the related OS. Although these systems provide authentication services and prevent some simple tampering on the application, they are not designed for protection of software confidentiality and privacy against physical tampering. To address this issue, new secure processor architecture, e.g. XOM and AEGIS, emerged. However, as mostly closed system solutions, they too fail to address some very important issues such as inter-operation between heterogeneous software components and information sharing. MESA is also different from the information flow security model such as RIFLE [8]. RIFLE keeps track of the flow of sensitive information at runtime by augmenting software's binary code. It prevents restricted information from flowing to a less secure or un-trusted channel. MESA is designed for a different purpose and usage model. The main purpose of MESA is to mitigate some of the drawbacks associated with the whole process based cryptographic protection of software instead of trying to solve the issue of secure information flow. MESA is also significantly different from the memory protection model called Mondrian [10]. Memory capsules in MESA are authenticated and encrypted memory spaces, concepts do not exist in Mondrian.

6 Conclusions

This paper describes MESA, a high performance memory-centric security architecture that is able to protect software privacy and integrity against both software and physical tampering. Different from the previous process-centric tamper-resistant systems, our new system allows different software components with different security policies to inter-operate in the same memory space. It facilitates software vendors to devise their own protections on software components therefore more flexible and suitable to open software system than the previous process-centric protection approaches.

7 Acknowledgment

This research was supported by NSF Grants CCF-0326396 and CNS-0325536 and a DOE Early CAREER PI Award.

References

1. E. Cohen and D. Jefferson. Protection in the hydra operating system. In *Proceedings of the 5th Symposium on Operating Systems Principles*, 1975.
2. D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectual Support For Copy and Tamper Resistant Software. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.
3. D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the Symposium on Operating Systems Principles*, 2003.
4. R. M. Needham and R. D. Walker. The Cambridge CAP Computer and its Protection System. In *Proceedings of the Symposium on Operating Systems Principles*, 1977.
5. W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *Proceedings of the International Symposium on Computer Architecture*, 2005.
6. E. G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings 0f the 36th Annual International Symposium on Microarchitecture*, 2003.
7. E. G. Suh, D. Clarke, M. van Dijk, B. Gassend, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing . In *Proceedings of the International Conference on Supercomputing*, 2003.
8. N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the International Symposium on Microarchitecture*, 2004.
9. S. Vlaovic and E. S. Davidson. TAXI: Trace Analysis for X86 Interpretation. In *Proceedings of the 2002 IEEE International Conference on Computer Design*, 2002.
10. E. J. Witchel. *Mondrian Memory Protection*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2004.
11. J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracty and Tampering. In *Proceedings of International Symposium on Microarchitecture*, 2003.
12. X. Zhuang, T. Zhang, and S. Pande. HIDE: an Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.
13. X. Zhuang, T. Zhang, S. Pande, and H.-H. S. Lee. HIDE: Hardware-support for Leakage-Immune Dynamic Execution. Technical Report GIT-CERCS-03-21, Geogia Institute of Technology, 2003.