Implementing an Untrusted Operating System on Trusted Hardware

David Lie
Dept. of Comp. and Elec. Eng.
University of Toronto
Toronto, ONT M2J 1A2

Chandramohan A. Thekkath
Microsoft Research
1065 La Avenida
Mountain View, CA 94043

Mark Horowitz
Computer Systems Lab.
Stanford University
Stanford, CA 94305

ABSTRACT

Recently, there has been considerable interest in providing "trusted computing platforms" using hardware — TCPA and Palladium being the most publicly visible examples. In this paper we discuss our experience with building such a platform using a traditional time-sharing operating system executing on XOM — a processor architecture that provides copy protection and tamper-resistance functions. In XOM, only the processor is trusted; main memory and the operating system are not trusted.

Our operating system (*XOMOS*) manages hardware resources for applications that don't trust it. This requires a division of responsibilities between the operating system and hardware that is unlike previous systems. We describe techniques for providing traditional operating systems services in this context.

Since an implementation of a XOM processor does not exist, we use SimOS to simulate the hardware. We modify IRIX 6.5, a commercially available operating system to create XOMOS. We are then able to analyze the performance and implementation overheads of running an untrusted operating system on trusted hardware.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Cryptographic Controls, Access Controls; D.4.1 [**Operating Systems**]: Process Management; C.1.0 [**Processor Architectures**]: General

Keywords

XOM, XOMOS, Untrusted Operating Systems

General Terms

Security

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

1. INTRODUCTION

There are several good reasons for creating tamper-resistant software including combating software piracy, enabling mobile code to run on untrusted platforms without the risk of tampering or intellectual property theft, and enabling the deployment of trusted clients in distributed services such as banking transactions, on-line gaming, electronic voting, and digital content distribution. Tamper-resistant software is also useful in situations where a portable device containing sensitive software and data may fall into the hands of adversaries, and in preventing viruses from modifying legitimate programs.

Tamper-resistance can be enforced using software or hardware techniques. In the past, techniques such as software obfuscation have been explored, but with limited success [4]. There is a widespread belief that software-based solutions are relatively easier to attack than hardware-based solutions [3]. Thus, there have been several proposals for creating systems that rely on hardware, rather than on software to enforce the security and protection of programs [10, 11, 14, 17, 27, 29].

Trusting the hardware rather than the software has interesting implications for operating systems design. Since sharing hardware resources among multiple users is a difficult task, often requiring complex policy decisions, it is most naturally done in software by an *operating system*. But, if one is reluctant to trust anything but the hardware, then we must somehow arrange for an untrusted agent—the operating system software—to manage a trusted resource—the hardware.

This paper explores the design of an operating system that runs on hardware that supports tamper-resistant software. Our operating system is intended to work with an existing processor architecture called XOM, which stands for eXecute Only Memory [17]. In the XOM processor architecture, programs do not trust the *operating system* or *external memory*, but instead, trust the processor hardware to protect their code and data. In trying to design the operating system for the XOM processor, we found difficulties with the original architectural specification that made it impractical to provide certain operating system facilities, such as time-slicing, process forking, and user-level signal handling. This paper describes the design changes in the operating system and the architecture that were required to build a functioning system.

There is currently no hardware implementation of the XOM architecture. We therefore modify the SimOS [22] simulation system to model a XOM processor, based on an

in-order processor model using the MIPS R10000 [12] instruction set. XOMOS is implemented by modifying the IRIX 6.5 [25] operating system from SGI. Using XOMOS we can execute XOM-protected (and ordinary) programs. Our experiments demonstrate that it is possible to write an operating system that not only manages resources for applications that do not trust it, but also supports most, but not all, of the traditional services that one expects from an operating system.

The rest of the paper is organized as follows. Section 2 describes the XOM trust model and its implications for operating systems design in more detail. Section 3 describes related work. Section 4 provides background and summarizes the basic processor architecture originally described in [17]. Supporting an operating system on this interface required significant software and hardware co-design, resulting in several changes to IRIX and the instruction set architecture. These changes are described in Section 5. Section 6 describes the implementation effort required to create XOMOS, and discusses the implementation and performance of micro-benchmarks as well as two secure applications running on XOMOS: RSA operations in OpenSSL and mpg123, an MP3 decoder. In Section 7, we discuss how XOM supports many of the primitives that trusted computing platforms such as TCPA and Palladium provide. Finally, we present our conclusions in Section 8.

2. THE XOM TRUST MODEL

XOM can protect against a sophisticated attacker who may even have physical access to the hardware. In this scenario, the attacker either tries to extract secrets from a program by observing its operation, or she tries to tamper with the program's operation to make it reveal secrets. To this end, the attacker may exploit weaknesses in the operating system and use its privileged status to attack programs. To guard against this attack, the XOM model assumes that the operating system is not trusted. We have formally verified the XOM architecture using a model checker in previous work [16]. This work demonstrated that even an actively malicious operating system can only extract a limited amount of information about user programs. Because of its role as resource manager, the only major attack the operating system could perform is a denial of service attack. The verification also demonstrated that given a correctly working operating system, user programs are guaranteed forward progress.

The XOM attack model also assumes that main memory may be compromised by an adversary. The XOM processor not only encrypts values in memory, but also stores hashes of those values in memory as well. XOM continuously authenticates memory to ensure that an attacker who physically attacks memory or the memory bus cannot tamper with data stored off-chip. It will only accept encrypted values from memory if accompanied by a valid hash. The hash not only protects the actual value of the data, but also checks the virtual address that the data was stored to by the application, thus ensuring that an adversary cannot copy or move data from one virtual address to another.

2.1 Implications of the XOM Trust Model

The lack of trust in any software-based resource manager and external memory makes our system significantly more robust and quite different from well-known approaches like microkernels and capability-based systems that try to solve a similar problem. If one is willing to compromise and lower the barrier by conceding hardware based attacks on memory, then solutions such as secure booting [2, 15, 29], TCPA [28] and Palladium (also known as Next-Generation Secure Computing Base or NGSCB) [6, 7] are viable alternatives to our design. We defer a fuller discussion of related work to Section 3.

The XOM architecture prevents programs from tampering with each other by placing them in separate *compartments* [24]. The separation of compartments is enforced by a combination of cryptography and tagging data in hardware. The untrusted operating system runs in a separate compartment from user processes. Entities running in a compartment cannot access data in another compartment. However, to share hardware resources among a set of processes, the operating system must be able to preempt a process, as well as save and restore its context. With XOM, a correctly written operating system will be able save and restore user data. Yet, even a malicious operating system should not be able to read or modify data belonging to a user process.

In principle, an operating system should be able to virtualize and manage resources without having to interpret any of the values it is moving, but there are practical obstacles to supporting many of the paradigms that typical operating systems support, such as signal handlers, memory management, process creation and so on. In addition, the operating system must also manage resources used to store the cryptographic hashes that the XOM hardware uses. The challenge is to find ways of supporting traditional operating system functionality under the new division of trust. XOMOS is a first step at exploring these issues.

XOM does alter the way a certain set of services are supported by the operating system. For example, security sensitive operations such as shared memory, inter-process communication and program debugging are restricted in their usage. We discuss this further in Section 5.

3. RELATED WORK

Our work is related to previous work on secure booting [2, 15, 29]. Loosely speaking, secure booting is a technique that guarantees the integrity of higher level software running on a machine by ensuring the integrity of each of the lower layers (e.g., the runtime libraries, operating system kernel, firmware, hardware) on which it depends. At the bottom of the chain a secure entity, such as a CPU, is assumed to have a tamper-resistant secret embedded in it. On power up, this secure entity takes control of the machine and authenticates the next layer e.g., the firmware, and transfers control to that layer, which authenticates the next layer in turn, and transfers control to it, and so on.

A key difference between secure booting and XOM is that the latter does not trust the operating system or memory. As a result, bugs in the operating system cannot undermine the security of applications running on it. In a secure booting system, a determined and sufficiently sophisticated adversary could exploit the trusted memory to modify the instruction or data stream of an authenticated program during execution (say by using dual ported memory). In contrast, XOM encrypts all instructions and data to and from main memory and can detect tampering of the code or the data at all times. The details of this technique are described in Section 4.

Executing trusted code on a secure co-processor is another way of achieving some of the same goals as we do [27, 29]. While a co-processor approach is feasible, it has some limitations. For example, it is difficult to time-share the co-processor among mutually suspicious pieces of code. XOMOS allows mutually suspicious applications to be securely multi-tasked on the same processor without any special co-operation from the applications.

There is much literature on security in the context of security kernels [23], capability-based operating systems [26], microkernels [1], and exokernels [8] that is related to our work. All of these approaches assume that the operating system and memory are trusted, and as a result are very different from XOMOS.

The TCPA [28] and Palladium or NGSCB [6, 7] initiatives are also related to our approach. One difference between these systems and XOM is that XOM provides the ability for programs to hide secrets in their binary images. Programs can use these secrets to authenticate themselves to third parties. Both TCPA and Palladium rely on a trusted monitor to perform the attestation (The TPM in the case of TCPA and the Nexus in the case of Palladium). TCPA provides a chain of cryptographic hashes so that an a third-party can reliably determine the complete boot sequence of the machine until it was loaded. It can therefore decide if the boot sequence has been tampered with and terminate. In some ways, TCPA is similar to the authenticated boot process found in a secure booting system, and neither system can protect itself from an attack that modifies memory contents after attestation. Both TCPA and the Palladium share the notion of *Sealed Storage* [6, 28], which uses a mechanism similar to XOM to protect data. We discuss sealed storage in greater detail in Section 7.

4. THE ORIGINAL XOM ARCHITECTURE

The XOM architecture is designed to address a shortcoming with implementing security in software. The problem is that it is easy to tamper with and observe software, and as a result, it is impossible to hide secrets in software. Instead, the XOM architecture uses the tamper-resistant properties of hardware, to protect a *master secret* that is different for every XOM-enabled processor. This is then used to secure and protect other secrets in software. For example, a program may want to protect its code and keep it secret. By hiding the encryption key for the program in the processor, adversaries cannot see or modify the program without mounting an attack on the processor hardware. Consequently, the XOM processor never allows the master secret to leave the chip; so all operations that use the master secret must be implemented on the processor.

XOM uses its master secret to protect programs by supporting *compartments*. A compartment is a logical container that prevents information from flowing into or out of it. A process in a compartment is immune to both *modification* and *observation*. Preventing unauthorized observation of the protected process is important because the process code and data may have secrets that its owner does not want to divulge. Conversely, by modifying the process data or code, an adversary may induce the process to leak secrets, so this must be prevented as well. We rely on the XOM hardware to implement compartments efficiently using the secret hidden in the processor. To do this, XOM uses both cryptographic and architectural techniques.

In this section, we will briefly summarize the original XOM processor hardware described in [17]. For the most part, a XOM processor behaves like a typical modern processor, and is simply a set of extensions to such a processor. It has registers, memory, and executes instructions in the usual fashion. The operating system and other processes can access the XOM extensions through a set of instructions. The next two subsections detail how the XOM architecture compartments are enforced and how the processor permits the operating system to handle program state without violating the security.

4.1 Implementing Compartments

XOM uses both asymmetric and symmetric ciphers to implement compartments[1]. The master secret hidden in the XOM processor is actually the private part of an asymmetric cipher pair. To support multiple compartments, each compartment has a distinct symmetric key, called the *compartment key*, which is used to encrypt its contents. The compartment key, in turn, is encrypted with the public key matching the processor, allowing the processor to recover the compartment key and decrypt the program. The encrypted compartment key need not be kept in the processor; it can be stored with the encrypted program code.

The use of compartments affects the software distribution model. A software producer who wishes to distribute programs that use compartments will first encrypt the program with a randomly chosen compartment key. She can then distribute the encrypted image, but no image will be executable until combined with a compartment key encrypted for that processor. Thus, she must encrypt the compartment key with the public key of the processor she wishes to allow the program to execute on. In a commercial setting, this could be done when the customer either purchases or registers the software online.

Data generated during program execution also must be isolated in the program's compartment. This is done by having the XOM processor encrypt data that a program stores to memory with the compartment key when it leaves the CPU chip. However, when data doesn't leave the chip, the XOM processor can skip the encryption. All values in processor caches and registers are stored in plain text[2] to increase efficiency. The XOM processor uses architectural support to enforce compartments without paying the cryptographic penalty. Hardware ownership tags are added to on-chip storage, such as registers and caches, to indicate which compartment data and code belongs to. A hardware table, called the *XOM Key Table*, maintains the mapping between compartment keys and ownership tags. On a cache eviction, the XOM processor looks up the compartment key in the XOM Key Table and uses it to encrypt the data to memory.

If code is encrypted, we say it is in a compartment. Principals who don't know the key cannot access the program code or data. There is a single distinguished compartment, called the *NULL compartment*, which has no compartment key. Programs that are not encrypted, exist and run in this

[1]Asymmetric key cryptography uses pairs of keys, a public key for encryption and a private key for decryption, while symmetric ciphers use a single key for both encryption and decryption

[2]Unencrypted values are referred to as plain text, encrypted values as cipher text

compartment. Code and data in the NULL compartment can be accessed from any compartment, and programs may use this compartment as an insecure channel for sharing data with other programs within the constraints of traditional operating system address space protection. It is important to note that the protection provided by XOM compartments is in addition to traditional address space and virtual memory protection provided by the operating system and processor architecture.

To protect against tampering of data while it is in memory, XOM processors employ a keyed cryptographic hash, or message authentication code (MAC), to check for the integrity of data and code stored into memory [13]. Each time a cache line is written to memory, a hash of it is generated, and both the hash and the cache line are encrypted. The hash pre-image contains both the virtual address and the value of the cache line. When decrypting the cache line, a matching hash must also be loaded before the XOM processor will accept the encrypted value as valid. Because the granularity of encryption is a cache line, it is important to note that programs cannot store encrypted and unencrypted data on the same cache line. Similarly, encrypted code and plain text code must be padded to be placed on separate cache lines.

The XOM architecture adds a few new instructions to the instruction set architecture of the base machine. XOM programs execute the `enter xom` instruction to enter their XOM compartment. This instruction causes the processor to decrypt the compartment key and enter it into the XOM Key Table. All instructions following `enter xom` are in the compartment and must be decrypted before execution. Programs leave secure execution and return to the NULL compartment by executing the `exit xom` instruction. This causes the compartment key to be unloaded from the XOM Key Table and following instructions are no longer decrypted. This pair of instructions allows a program to execute some code sections in the NULL compartment and others in a private compartment. This allows for easier application development and better program performance as XOM code incurs the overhead of cryptographic operations.

XOM programs explicitly indicate whether data they store to memory belongs to the NULL compartment or in their private compartment. The `secure store` and `secure load` instructions store and load data in a private compartment. Data thus stored is tagged with the same value as the program code when in the on-chip caches and will eventually be encrypted with the same compartment key as the program if flushed to off-chip memory. On the other hand, the standard load and store instructions save data in the NULL compartment.

XOM processors provide the `move from NULL` and `move to NULL` instructions to move register data between compartments. These instructions simply change the value of the tag on each register value.

4.2 Handling Program State

When a XOM program is interrupted, the operating system needs a way to save the context of the interrupted program, and restore it at a later time. However, at interrupt time, the contents of the registers are still tagged with the identity of the interrupted program. As a result, the operating system is unable to read those values to save them. The XOM processor provides two instructions for the operating system to use in this situation. A `save register` instruction directs the XOM hardware to encrypt the register, create a hash of the register, and store both to memory. A complementary `restore register` instruction takes the encrypted register and hash, verifies the hash, and restores them back to the original register, setting its ownership tags appropriately. The hashes detect when a malicious operating system attempts to tamper with register values by either spoofing in a fake value or restoring valid values back to a different register. In the event that the register is in the NULL compartment, these instructions do not perform any cryptographic operations. These instructions, and the instruction discussed in Section 4.1, are summarized in the first column of Table 1.

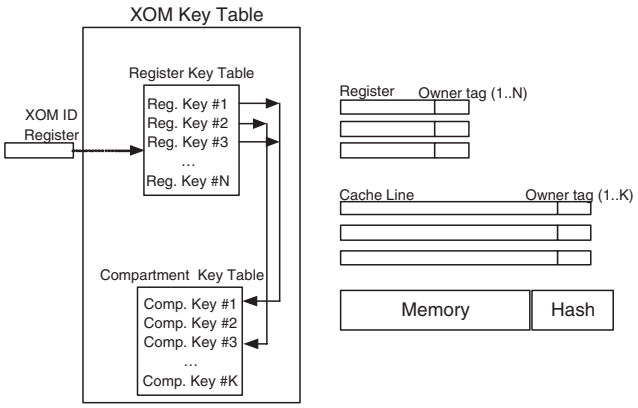

Figure 1: XOM Key Table Design. The XOM ID register refers to the Register Key entry for the currently executing XOM process. The ownership tags on the registers are used by XOM to encrypt and decrypt register contents under program control, using specialized machine instructions described in Table 1. The ownership tags on the caches are used by XOM to encrypt and decrypt cache contents in response cache evictions and fills, which are not under explicit program control.

A special provision must also be made to prevent the operating system from mounting a replay attack by taking the register values from one interrupt and repeatedly restoring it. This is done by revoking the key used to encrypt and hash register values each time a XOM compartment is interrupted. The hardware performs this revocation by regenerating a new key for the compartment when it takes a trap. Since this key is continually changing, we can't use the compartment key to encrypt register values; rather, we use a separate key, called the *register key*.

Memory values must also be protected against replay attacks. Previous work [16] has shown a way of preventing replays by storing a hash of memory in a replay-proof register. We don't directly support this scheme in the current system; instead, we rely on the applications to maintain the hash themselves. This adds a significant burden to the applications. We are aware of proposals to add hardware support to XOM that would maintain hashes with a performance penalty of about 25% [9]. The main obstacle to the addition of such hardware is that there must be a hash tree per *virtual* and not *physical* address. Since it is the operating system that maintains the virtual to physical mapping,

a new interface must be created to allow the hash tree hardware to retrieve these mappings from the operating system. Implementing this interface, which involves exception handling, will incur additional operating system and hardware overheads. We have a preliminary design, but we have not implemented it yet.

Note that XOM does not prevent incorrectly written programs from leaking secrets, nor is that its intent. XOM simply provides the necessary support so that correct programs can secure their secrets against a range of attacks.

Figure 1 describes some of the important aspects of the XOM hardware. The XOM Key Table actually consists of two sub-tables: the *Register Key Table*, which holds the register keys and the *Compartment Key Table*, which holds the compartment keys. The Register Key Table is indexed by a special register called the *XOM ID Register* that refers to the register key of the currently executing XOM process. The value of this register is set and unset via the `enter xom` and `exit xom` instructions. The index of a key in the Register Key Table is used as a shorthand (instead of the key itself) to tag the register contents. We call these tags the *register ownership tags*. Likewise, cache lines are tagged with the indices into the Compartment Key Table. These tags are called the *cache ownership tags*. When the context is unambiguous we use the common term *ownership tags* to refer to the tags on the registers and the caches. Every register key in the Register Key Table corresponds to a single entry in the Compartment Key Table, although multiple register keys may point to the same entry in the Compartment Key Table. The reason for this mapping is to allow multiple instances of the same program to execute as would occur as a result of a fork. This is discussed in more detail in Section 5.6.

5. SUPPORTING AN OPERATING SYSTEM

The purpose of the XOM hardware is to protect the master private key and to provide the basic functionality to enforce compartments. Higher-level tasks such as resource allocation and management, hardware virtualization, and implementing system call functionality are still the domain of the operating system. While these tasks may have some security implications, they are simply too complex to be implemented in the hardware.

XOM programs do not trust the operating system with their data. On the other hand, the operating system does not trust XOM programs to behave properly, and must be able to interrupt and remove resources from a misbehaving XOM program. Accordingly, the contract between the XOM architecture and the operating system must satisfy two requirements. First, given a properly working operating system, it should make resource management efficient and effective. Second, it should ensure that if the operating system is malicious, its privileged position does not allow it to violate the isolation of a compartment.

Given these requirements on the interface between the hardware and operating system, the hardware must provide exception and interrupt functionality as is found on ordinary processors. This allows the operating system to limit the execution time of programs and interpose when programs access resources. On the other hand, when the operating system moves the physical location of resources, it must adhere to the XOM compartments. This means, when saving process state, it must use the special instructions provided by XOM that encrypt and hash process registers. When relocating data in memory, the operating system must also relocate the respective hashes.

As mentioned in Section 1, there are some operating system services which cannot be implemented the same way they are in non-XOM systems. For example, it would be impossible to support external debugging or profiling as XOM would prevent any other program from gaining access to the state of the program. In addition, shared memory and inter-procedure communication through memory can only be implemented in the NULL compartment. To communicate securely, two programs should establish a shared key and encrypt all communications through the NULL compartment.

A significant number of changes to IRIX were required to implement XOMOS. We classify these changes into three categories:

- **Modifications for XOM Key Table maintenance:** The hardware and operating system must have support for programs to use the XOM Key Table, and the operating system must manage the limited number of entries it has.
- **Modifications for dealing with encrypted data:** When the operating system is managing system resources such as CPU time or memory, it must deal with user data that is encrypted as well as the accompanying hashes.
- **Modifications for traditional operating system mechanisms:** Various features in a traditional operating system such as shared libraries, process creation, and user defined signal handlers require special support.

For the most part, these modifications were implemented in the operating system. However, in some cases, we found that modifications to the XOM hardware architecture were also necessary. A summary of the hardware modifications is given in Table 1 and a sketch of the important XOM hardware data structures is shown in Figure 1.

5.1 XOM Key Table System Calls

The original architecture specifies a single `enter xom` instruction to enter XOM operation. This allocates an entry in the XOM Key Table, which is freed when the program executes an `exit xom` instruction. While this is adequate, it is inefficient if a program wishes to enter and exit its XOM compartment frequently, since the hardware would have to perform an expensive public key operation every time. In addition, since `enter xom` is privileged, the program has to drop into the kernel every time it wishes to enter a compartment. However, if `enter xom` is unprivileged, the operating system cannot prevent a malicious application from mounting a denial of service attack by allocating all entries in the XOM Key Table. To satisfy these conflicting requirements, we decouple the operations of loading and unloading Key Table entries from entering and exiting compartments.

We split each of the `enter xom` and `exit xom` instructions into two smaller primitives. The `xalloc` and `xinval` instructions allocate and invalidate XOM Key Table entries, while `xentr` and `xexit` instructions enter and exit a XOM compartment. When a program wants to enter a new XOM

Original Hardware	New Hardware	Description
`enter xom`	`xalloc $rt,offset($base)`	Privileged. Decrypt compartment key at `memory [$base + offset]` and enter it into the Compartment Key Table at the entry `$rt`. Allocate a new register key in the Register Key Table and return the index into table in `$rt`.
	`xentr $rt,$rd`	Use `$rt` as an index into the Register Key Table and enter the corresponding compartment. The current register key is placed in `$rd`.
`exit xom`	`xinval $rt`	Privileged. Mark the entry in the XOM Register Key Table indicated by `$rt` as invalid and disassociate it from the compartment key. Clear all registers that are tagged with the tag `$rt`.
	`xrclm $rt`	Privileged. Reclaim the entry `$rt` in the XOM Compartment Key table. (All entries in the Register Key Table pointing to this compartment key must have been previously invalidated.) Invalidate caches.
	`xexit $rt`	Executed in XOM mode. Exit XOM compartment and return to the NULL compartment. The hardware sets the register key of the exited compartment to the value in `$rt`.
`secure store`	`xsd $rt,offset($base)`	Executed in XOM mode. Stores `$rt` into `memory [$base + offset]`. The cache line is tagged with the executing process' compartment.
`secure load`	`xld $rt,offset($base)`	Executed in XOM mode. Loads `$rt` with `memory [$base + offset]`. Validate the accompanying hash. `$rt` is tagged with the executing process' compartment.
`save register`	`xgetid $rt,$rd`	Get the register tag value of `$rt` and place it in `$rd`.
	`xenc $rt,$rd`	Check that the ownership tag of `$rt` matches that of `$rd`. If so, use the contents of `$rd` to index the Register Key Table to locate the corresponding register key. Encrypt the contents of `$rt` with this key and place it in XOM co-processor registers $0...$3
	`xsave $rt,offset($base)`	`$rt` is one of the XOM co-processor registers $0...$3 which store the encrypted register created by `xenc`. The register contents are saved to `memory [$base + offset]`. The cache line is tagged with the NULL compartment, (c.f. `xsd`).
`restore register`	`xrstr $rt,offset($base)`	`$rt` is one of the XOM co-processor registers $0...$3 that stores the encrypted value to be restored. Fill the register with the value at `memory [$base + offset]`.
	`xdec $rt,$rd`	Use the contents of `$rd` to index the Register Key Table to locate a register key. Decrypt the 256 bit value set by `xrstr`, validate the result and restore to register `$rt`. Set the ownership tag on `$rt` using the contents of `$rd`.
`move to NULL`	`xmvtn $rt`	Executed in XOM mode. Set the ownership tag of `$rt` to NULL.
`move from NULL`	`xmvfn $rt`	Executed in XOM mode. Set the ownership tag of `$rt` to the value in the XOM ID register.

Table 1: Summary of modifications to the original XOM hardware architecture. Original hardware represents the primitives that are in the original XOM specification. New hardware details the instructions that were added to the ISA of the processor in our simulator implementation of XOM. All instructions, unless otherwise specified, are executed outside XOM mode.

compartment, XOMOS executes `xalloc` on behalf of the program to load a compartment key. XOMOS specifies an entry in the Compartment Key Table to load the key into. The XOM hardware also allocates an entry in the Register Key Table corresponding to the compartment key and returns an index into the Register Key Table. This index is returned by XOMOS to the program, which it then uses with the `xentr` instruction to begin execution in that compartment. Code following the `xentr` instruction must be properly encrypted and hashed to execute properly. Executing `xexit` from a compartment exits the compartment, but the XOM Key Table entry is not removed until the program invalidates it, so subsequent entries into the compartment only require an `xentr`.

Because `xalloc` and `xinval` access a limited hardware resource, they are privileged instructions, and are executed on behalf of the program by XOMOS via the system calls `xom_alloc()` and `xom_dealloc()`. This scheme allows the operating system to interpose and prevent misbehaving applications from allocating too many XOM Key Table entries.

5.2 Virtualizing the XOM Key Table

XOMOS manages the XOM Key Table to allow as many applications as possible to run simultaneously. However, the table is a limited resource and there must be a mechanism to reuse its entries. Recall that the internal storage in the machine is protected by ownership tags that correspond to indices into the Register Key Table and the Compartment Key Table, so reusing table entries could compromise the data of the previous owner, since both new and old owner would share the same tag value.

To ensure that old entries are not reused inappropriately, we allow XOMOS to invalidate and reclaim table entries. `xinval` invalidates entries in the Register Key Table entry causing that particular register key to be destroyed, but preserves the Compartment Key Table entry associated with it. `xrclm` is used to reclaim entries in the Compartment Key Table. Recall that multiple Register Key Table entries may refer to the same entry in the Compartment Key Table. As a result, an entry in the Compartment Key Table can only be evicted with the `xrclm` instruction if all the Register Key Table entries referring to it have been previously invalidated with the `xinval` instruction. XOMOS maintains the necessary data structures in order to do this. It is easy to maintain the data structures since XOMOS knows which entries are in the invalid state since all table operations require system calls into the kernel. When a Compartment Key Table entry and the corresponding Register Key Table entries are reclaimed, the hardware must ensure that no data protected by the old keys still exists on the processor.

When a Register Key Table entry is invalidated, the processor clears all registers in the register file that may be tagged with the evicted register key. Later, as the Compartment Key Table entry is reclaimed, all register data related to that compartment key has already been flushed, leaving only data in the cache. However, it is too complex for the hardware to check every cache entry so it invalidates all on-chip caches to prevent old data in the caches from leaking out. It is the operating system's responsibility to make sure any dirty data in the cache is written back first, or it will be lost.

The operating system maintains a mapping between process IDs, Register Key Table indices, and encrypted compartment keys. When a process requests a XOM Key Table entry via the `xalloc` system call, but none is available for reclamation, the operating system forcibly reclaims an entry with the `xinval` and `xrclm` instructions. Note that the operating system should select an entry whose processes are not interrupted while in a compartment (since invalidating the register key makes any process state protected by the key unrecoverable). When the process that just lost its entry is subsequently restarted, the operating system reallocates the Key Table entry using the encrypted compartment key.

5.3 Saving and Restoring Context

As discussed in Section 4, the operating system saves the state of an interrupted process with the aid of additional hardware instructions. However, the original architecture overlooked one subtlety. When saving the register value with the `save register` instruction, the operating system has no way of reading the ownership tag of the register it is saving. When the operating system restores registers with the `restore register` instruction, it needs to tell the hardware which compartment to restore the register to with a suitable tag. To fix this, we add a new instruction, `xgetid` that gets the ownership tag value of the compartment that owns that register. XOMOS uses this to determine a register's ownership tag before saving it. Without this ability, XOMOS cannot identify the owner of data, and thus cannot manage the register.

```
li     $k1,BASE_OF_EFRAME  # save cntxt
xgetid $s0,$at             # get tag
                           # of $s0->$at
xenc   $s0,$at             # encrypt $s0
                           # into $x0...$x3
xsave  $0,EF_S0($k1)       # save
xsave  $1,(EF_S0+8)($k1)   # encrypted
xsave  $2,(EF_S0+16)($k1)  # values
xsave  $3,(EF_S0+24)($k1)
sw     $at,(EF_S0_XID)($k1)
...                        # restore cntxt
xrstr  $0,EF_S0($k1)       # restore
xrstr  $1,(EF_S0+8)($k1)   # from memory
xrstr  $2,(EF_S0+16)($k1)
xrstr  $3,(EF_S0+24)($k1)
lw     $at,(EF_S0_XID)($k1)# load XOM ID
xdec   $s0,$at             # decrypt
```

Figure 2: XOMOS context switch code.

The encrypted register is larger than a 64-bit memory/register word on our processor due to the additional information that must be saved. XOM uses a 128-bit cipher text that contains the encrypted register value, register number, and the register ownership tag. This is then combined with a 128-bit hash for integrity resulting in a 256-bit value. Saving the entire value to memory in one instruction would result in a multi-cycle, multi-memory access instruction, which is difficult to implement in hardware.

Instead of the single `save register` instruction, we change the architecture to implement an `xenc` instruction that will encrypt and hash the register contents with the register key and place them in four special XOM registers. These can be accessed via the `xsave` instruction, which takes an index pointing to one of the four registers and saves it to a memory location. Similarly, to replace the `restore register` instruction, an `xrstr` instruction restores values in memory to the four XOM registers and an `xdec` instruction is used to decrypt the value in the XOM registers with the register key, verify the hashes, and return the value to a general-purpose register.

The low-level trap code in XOMOS includes the XOM register access instructions. Figure 2 illustrates the code to save and restore a register. This sequence saves and restores register `$s0`. `$k1` points to the base of the exception frame while `EF_S0` is the offset into the exception frame where the register value of `$s0` is stored. A similar sequence is required for every register. Processing traps for code in a compartment represents a large instruction overhead — where 2 instructions are required to save and restore a register for an application with no protected registers, 13 instructions are

required to save and restore each protected register. To preserve the performance for applications that are not executing in a compartment, XOMOS checks if an interrupted process is in XOM mode, and only executes the extra instructions if it is required.

Aside from new context switch code, changes are also required to the exception frame structure, where XOMOS stores the interrupted process state. The exception frame must be enlarged to allow room to hold the ownership tag of each register as well as the larger cipher text.

Some parts of the interrupted process state cannot be protected by XOM and are left tagged with the NULL compartment. For instance, data such as the fault virtual address in a TLB miss, or the status bits that indicate whether the interrupted thread was in kernel mode or not, must be available to the operating system for it be to handle these exceptions. While this process state reveals some information about the application, the nature of such information is limited. For example, a malicious operating system can obtain an address trace of every page an application accesses while in a XOM compartment by invalidating every page in the TLB and recording every fault address.

5.4 Paging Encrypted Pages

XOM uses cryptographic hashes to check the integrity of data stored in memory. The operating system also must virtualize memory, which means that it must be able to relocate encrypted data and hashes in physical memory. It is impossible to store the hashes in the ECC memory bits as suggested in [17] because to virtualize memory, the operating system must be able to access the hashes. We store the hashes on a different page from the data so as to retain a contiguous address space.

A malicious operating system cannot take advantage of this separation between the hashes and the data. A XOM application will not proceed with a secure memory load if a valid hash is not supplied to it. To tamper with data, the operating system must be able to create the correct hash for the fake data. Using sufficiently strong cryptographic algorithms can make this computationally difficult.

We reserve a portion of the physical address space for the *xhash* segment, where the cryptographic hashes for XOM will be stored. The starting location of the XOMOS kernel is adjusted to be just below the *xhash* segment. In our XOM processor, L2 cache lines are 128 bytes long and require a 128-bit hash, making the *xhash* segment one-eighth the size of physical memory. To facilitate data address to hash address translation, we locate the segment at the top of the physical address space. The offset of the hash in the segment can then be calculated by dividing the physical address of the first word in the cache line by eight.

Whenever the XOMOS pager swaps a page in physical memory out to the backing store, it also copies the matching values in the *xhash* segment onto a reserved space on swap. When faulting a page back in, the operating system copies the hash data of the page being faulted in, and places it at the correct offset in the *xhash* segment. The operating system gives similar treatment to XOM code pages since XOM code also has hash values protecting it. These are stored in a separate segment in the executable file. When a code page is faulted in, the appropriate hash page is also read in from the executable file image and placed in the *xhash* segment.

Since not all applications may actually use XOM facilities, our simple design is wasteful as it reserves a fixed portion of memory for hashes. Unencrypted values will not have hash values that need to be saved. The design could be made more efficient with additional hardware.

5.5 Shared Libraries

Linking libraries statically is relatively straight forward as the library code can be placed in the XOM compartment by encrypting and hashing it with the compartment key after linking. On the other hand, if linked dynamically, shared library code cannot be encrypted since it must be linkable to many applications, and encrypting it with a certain key would make it linkable to only one. While it is possible to have code in the compartment encrypt the library code at run time, thus bringing it into the compartment, this is complicated, and there is no way to authenticate the unencrypted code without additional infrastructure (In the simplest case, the library would have to be signed). Instead, we chose to design an interface where XOM encrypted code must call unencrypted library code with the assumption that the call is insecure — the caller cannot be sure that the library code has not been tampered with.

To support dynamically linked libraries in a way that is transparent to the programmer, the compiler must be altered to use a *caller-save* calling convention to deal with secure data. To see why, recall that in a callee-save calling convention, the dynamic library subroutines are expected to push the caller's registers on the stack. However, since the subroutine is not in the same compartment as the XOM code calling it, it will not have the ability to access those values. Thus, the caller, rather than the callee, must save all secure registers. In addition, before calling the subroutine, the calling XOM code must first move, as necessary, register values such as subroutine arguments, the stack pointer, frame pointer, and global pointer to the NULL compartment so that the callee can access them. After this it must exit its XOM compartment with the `xexit` instruction.

Encrypted data cannot be stored on the same cache line as unencrypted data. When making a function call across a XOM boundary, we can either realign the frame pointer for local variables to cache line boundaries, or simply use a separate stack when executing in a XOM compartment. Similarly, the start of the unencrypted code must be aligned to be on a different cache line than that of the encrypted code.

When returning from the subroutine call, the above sequence must be reversed. The application re-enters its XOM compartment, moves the stack pointers back from NULL, replaces them to the values before alignment and restores the caller saved register values. Similar code must be executed before a system call since the system call arguments and program counter must be readable by the kernel.

We have implemented and tested this method by manually saving the registers and adding the wrapper code around calls to the C standard library (*libc*). An example of such wrapper code is given in Figure 3.

Libraries that perform security sensitive routines should be statically linked and encrypted. An example of this is the OpenSSL library, which contains cryptographic routines. On the other hand, it does not make sense to encrypt shared libraries that consist of input or output routines. The program should check values from these libraries to see if they

```
# compiler has saved all registers
# ownership tag value is in $s0
sd      $fp,0($sp)   # push fp
and     $fp,$fp,~0xF # align fp
xmvtn   $fp          # move pointers
xmvtn   $sp          # to null
xmvtn   $gp
xmvtn   $a0          # move
xmvtn   $a1          # subr. arguments
xmvtn   $t9
xexit                # exit XOM (aligned)
jal     $t9          # subroutine call
...
xentr   $s0          # reenter XOM (aligned)
xmvfn   $fp          # move pointers
xmvfn   $gp          # back
xmvfn   $sp
xmvfn   $v1          # move return value
ld      $fp,0($sp)   # restore old fp
# now compiler restores all
# caller save regs.
```

Figure 3: Exiting and entering a Compartment.

are sensible since they could potentially be coming from an adversary.

5.6 Process Creation

Naively implemented, a XOM application that forks will cause the operating system to create a child that is the exact copy of the parent, with the child inheriting the parent's register ownership tag value. If the operating system interrupts one process, say the parent, and restores the other, an error will occur since the current register key will not match the register state of the child.

The solution is to allocate a new register ownership tag for the child. Because there are two different threads of execution, we need two different register keys (and two different register ownership tags). A new `xom_fork()` library call is created for programs where both the parent and child of a fork will be using compartments. `xom_fork()` is similar to regular UNIX `fork()` except it will use the `xom_alloc()` system call to allocate for the child, a second register key with the same compartment key as the parent. They must have the same compartment key because the child needs to access the memory pages it inherits from the parent. After the new Register Key Table entry is acquired, the parent requests the operating system to do a normal `fork()`. When the parent returns, it continues using the old register ownership tag, while the child will use the new register ownership tag.

Register data is tagged with register ownership tags, which distinguish ownership between the parent and the child. The situation with the data in the cache is more subtle. Since both parent and child have the same compartment key, secure data in the caches must be tagged with the same value for both. Clearly, we cannot use the register ownership tags, which are different for each process; instead a different set of tags, called the cache ownership tags, are used in the caches. It is this relationship that implies that a single entry in the compartment key table, which is indexed by a cache ownership tag, can have several register keys and register ownership tags associated with it. The XOM hardware records the mapping between register keys and compartment keys. When a process executes a secure store, the internal XOM ID register and the mappings between register and compartment keys are used to determine the process' cache ownership tag, which is then used to tag the data in the cache. If this cache line is flushed to memory, the value is encrypted with the compartment key that corresponds to the tag. On a secure load, a similar translation is done to obtain the correct cache ownership tag value, which is checked against the tag in the cache.

5.7 User Defined Signal Handlers

A user defined signal handler may access the state of the interrupted process. It may also modify that state and then restart the process with the altered state. However, when a process executing in its XOM compartment is delivered a signal, the state of the interrupted thread will be encrypted. XOMOS saves the register state of the process using `xgetid`, `xenc`, and `xsave` instructions much like the context switch code in Figure 2. The interrupted state is copied into a *sigcontext* structure and delivered to the user-level signal handler. However, to support XOM, the fields of the *sigcontext* structure are enlarged the same way the exception frame is, to accommodate the larger encrypted register values and hashes.

To process the signal, the signal handler requires the register key that the *sigcontext* structure is encrypted with. To be secure, the hardware must only release this key to a handler in the same compartment as the interrupted thread, which means the signal handler code must also be appropriately encrypted and hashed with the same compartment key as the interrupted thread. Entry into the signal handler within the XOM compartment and the retrieval of the register key must be a single atomic action. Otherwise, we can get the following race: If the signal handler has entered the compartment and gets interrupted before it retrieves the register key, then that key will be destroyed by the hardware before the handler can ever get to it.

The XOM hardware guarantees the required atomicity by writing the register key into a general-purpose register when a program executes a `xentr` instruction. This way, the signal handler in the XOM compartment always has the required register key, even if it is subsequently overwritten in the key table by an interrupt. With the register key, the signal handler can then decrypt and verify the cipher texts in the *sigcontext* structure, and even modify and re-encrypt them if necessary.

The simplest way for the signal handler to restart the thread is to restore the new register state and jump to the interrupted PC. However, IRIX requires the restart path for the signal handler to pass through the kernel so that it can reset the signal mask of the process. The kernel uses the contents of the *sigcontext* structure returned by the handler to restart the process. Thus, the signal handler requires a way to set the register key so that it matches the key used in the modified *sigcontext* structure. To do this, we modify `xexit` to take a register value, which the hardware will use as the current value of the internal XOM ID register. XOM makes signal restarts that pass through the kernel more expensive because the signal handler must re-encrypt all modified register values in the *sigcontext* structure and the hardware must decrypt all those values when the operating system restarts the thread.

In fact, if the signal handler modifies any of the *sigcontext* registers, it should select a new register key and re-encrypt

Function	Number of	
	Lines	Files
Key Table System Calls	63	2
Key Table Reclamation	28	2
Save and Restore Context	907	16
Paging Encrypted Pages	40	1
Signal Handling	802	2

Table 2: Number of lines and files changed in the kernel.

Function	Num. of Lines
Shared Library Wrappers	64
Signal Handling	136
Fork & Process Creation	72

Table 3: Line count of user level changes.

all of them with that key. Otherwise, if the signal handler reuses the old key, a malicious operating system may choose to restore the old value and ignore the new value. In addition, a malicious operating system may deliver signals with faulty arguments. This will not pose a security problem the contents in the sigcontext structure will only be accessible if they were encrypted and hashed properly.

6. RESULTS

At the time of writing, we are not aware of any applications that use the security features provided by TCPA or Palladium. As a result, it is difficult to do a comparison of application and operating system characteristics between XOM, Palladium and TCPA. Instead, we attempt to document the overheads of XOM by comparing it to the basic MIPS with IRIX based system that does not implement XOM. We begin by quantifying the implementation effort of the modifications discussed in Section 5 is discussed. We then proceed to examine the performance overheads of our modifications. The performance impact of XOM appears in two aspects. First, there is the overhead that results from the modifications that were performed on the base IRIX 6.5 operating system. The operating system overheads are studied with a series of micro-benchmarks, which stress the parts of XOMOS that have been modified. The performance is compared to the original, unaltered, IRIX 6.5 operating system. The other source of overhead is the cost of encrypting and decrypting memory accesses, as well as the cost of entering and exiting a compartment. Secure compartments are used by XOM applications, so the cryptographic overheads are a factor for them. We thus examine the end-to-end performance of a XOM-enabled MP3 audio player and RSA operations in the OpenSSL library.

6.1 Implementation Effort

To implement XOMOS, we added approximately 1900 lines of code to the IRIX 6.5 kernel. The breakdown of these lines of code is shown in Table 2. In addition to the kernel changes, dealing with process creation, shared libraries, and user level signal handling required changes at the user level, as shown in Table 3.

One qualitative observation we made was that most of the kernel modifications were limited to the low-level code that interfaces between the operating system and the hardware.

System Call	Cycles	Instrs.	Cache Misses
`xom_alloc()`	413752	3625	13
`xom_dealloc()`	5691	3841	4.2

Table 4: Overhead due to new system calls in XOMOS.

As a result, much of the higher-level functionality of the operating system, such as the resource management policies, kernel architecture and file system were left unchanged. This reduced the side effects of these modifications considerably and suggests that the changes are not operating system dependent. While some modifications such as signal and fork are UNIX specific, the concepts of saving state to handle a trap, paging and process creation are common to most modern operating systems. This suggests that it would also be possible to port other operating systems to run on the XOM architecture.

To quantify the overhead of XOMOS and XOM, we now present the performance of both micro-benchmarks, as well as complete applications running in the SimOS simulator.

6.2 Basic Processor Model Parameters

Our simulator models an in-order processor where all instructions complete in one cycle unless stalled by a cache miss. The processor model has split 16 KB L1 caches and a unified 128 KB L2 cache. While these caches are small for a typical modern processor, the micro-benchmarks that we simulate are also small, so scaling down the caches helps put a conservative upper bound on what the performance will be. In the case of complete applications, we present results from several larger cache sizes. The memory latency is set at 150 processor cycles, and the memory system models bus contention as well as read/write merging. We use the AES (Rijndael) block cipher [5], which is composed of 14 "rounds." Each round takes approximately 1 cycle to complete, so we conservatively assume that encryption or decryption will add 15 cycles to the memory access time. This cost is incurred whenever a XOM memory access misses in the cache, or when a cache line in a private compartment is flushed to memory. All the instructions introduced in Table 1, execute in a single cycle with the exception of the `xalloc` instruction, which requires on the order of 400,000[3] cycles to perform the public key decryption. During this time, the processor is stalled.

6.3 Operating System Overhead

The operating system modifications add overhead in several areas; Tables 4 and 5 summarize these overheads.

First, recall that *XOMOS* introduces two new system calls to manipulate the XOM Key Table entries. The execution time for a `xom_alloc()` is dominated by the time to execute the `xalloc` instruction, in addition to the standard overhead of crossing into the kernel. The `xom_dealloc()` system call has the same execution time as a null system call in IRIX 6.5, since the kernel only executes an `xinval` before returning to the application.

Second, additional instructions are required by the operating system to save and restore context, resulting in more executed instructions. In addition, since encrypted regis-

[3]This is time required to perform an RSA decryption as measured using the OpenSSL library routines.

Benchmark	Total Cycles			Total Instructions			Kernel Instructions			Cache Misses		
	IRIX	XOM	OV	IRIX	XOM	OV	IRIX	XOM	OV	IRIX	XOM	OV
System Call	5691	6343	11%	3841	4018	5%	3799	3837	1%	4.2	5.6	33%
Signal Handler	31168	39593	27%	11114	14635	32%	11037	14425	31%	38.4	48.3	26%
XOM_Fork	13872170	12610085	-9%	119297	122722	3%	117940	121340	3%	1035	1058	2%

Table 5: Micro-benchmark overhead of XOMOS vs. IRIX. The Signal Handler benchmark performs more XOM operations and as a result incurs more overhead.

ters are larger than unencrypted registers, operating system data structures that store process state such as the exception frame or *sigcontext* data structures have a larger memory footprint. This can increase the cache miss rate and cause more overhead.

Another source of overhead comes from the additional I/O operations that are performed to save hash pages to disk. In our implementation, a hash page accompanies every data page, and thus the I/O requirements for paging operations are increased by the size of the hash pages. In this case, this resulted in a bandwidth increase of one eighth. This should not be an issue for applications that are not memory bound.

Reclaiming XOM Key Table entries also results in some operating system overhead. Since this requires flushing on-chip caches, this can be an expensive operation. However, note that each time a XOM Key Table entry is allocated, the XOM processor needs to perform an expensive public key operation (see Section 6.2). Typically, several such operations will occur before the XOMOS needs to reclaim entries, so we have a reasonable assurance that the percentage of cycles spent on XOM Key Table reclamation will not be large.

We wrote three micro-benchmarks that exercised the portions of the operating system kernel that had been modified. These benchmarks exercised a NULL non-XOM system call, signal handling and process creation in the modified kernel. The NULL system call benchmark makes a system call in the kernel that immediately returns to the application. The signal handling benchmark installs a segmentation fault (SEGV) signal handler and then causes a SEGV to activate the handler. The handler simply loads the program counter from the sigcontext structure, increments it to the next instruction and then restarts the main thread. Finally, the process creation benchmark calls `xom_fork` to create new XOM processes. The benchmarks do not perform any secure memory operations, so the overheads incurred are purely from the extra instructions executed and any negative cache behavior. Table 4 shows the results from these benchmarks.

The overhead for making (non-XOM) system calls is modest and the number of extra instructions in the kernel is actually very small. As discussed in Section 5.5, system calls cannot be made from inside a compartment. To make a system call, the XOM application must exit the compartment, make the system call and then return to compartment. The kernel only needs to check that the system call is not made while inside a compartment or the system call will fail. Because of this, about 95% of the extra instructions occur in user code. The remaining cycles are caused by additional cache misses. Each time a program enters or exits a compartment, an event we call a *XOM transition*, the compiler must pad the instruction stream with `nop`'s so that encrypted code and unencrypted code boundaries are aligned to cache lines in the machine. This not only increases the instruction count, but also the code footprint which may hurt instruction cache behavior.

The signal handler overhead experiences the most kernel overhead, with the majority of the extra instructions executed occurring on the kernel side. Because the signal is delivered while the application is in a compartment, the kernel must use the longer XOM save routines shown in Figure 2 to save every register. In addition, when the kernel populates the sigcontext structure, the kernel requires more instructions to copy the larger encrypted register values. The additional instructions and larger data structures also result in an increase in cache misses.

Finally, the xom_fork benchmark actually has negative overhead. Fork is already a long operation in IRIX, so the overhead imposed by XOM negligible. The majority of the extra instructions in fork are actually due to the extra `xom_alloc()` system call that is used to allocate a new Register Key Table entry. However, in this case more favorable behavior in the L1 cache makes up for the additional instructions and L2 cache misses.

One thing we noticed from these benchmarks is that it is important to avoid performing unnecessary XOM operations in the kernel. In our implementation, we were careful to always test if the interrupted application was running in a compartment or not. If it wasn't, the extra instructions to save and restore the larger encrypted registers were left out. We can see this in the difference between the kernel instructions executed for the NULL system call benchmark, which exits the compartment before trapping into the kernel, and the signal handling benchmark, which traps while in a compartment. Another factor in the overheads is that IRIX is a highly performance tuned operating system. By increasing the size of the code and data structures, our modifications destroyed a part of that tuning and resulted in more cache misses.

6.4 End-to-end application overhead

To measure the end-to-end application overheads, we added XOM functionality to two applications that would benefit from secure execution. The first, called *XOM-mpg123*, was created by modifying mpg123 — a popular open source MP3 audio player. This application simulates a scenario where a software distributor may wish to distribute a decoder for a proprietary compression format. The other is the OpenSSL [20] library, an open source library of cryptographic functions, which is used in an array of security applications. In OpenSSL, we tested the performance of RSA encryption and decryption, by using the `rsa_test` benchmark that is included in the OpenSSL distribution to create the XOM-RSA benchmark.

We wished to study the effects of varying the amount of code in the XOM compartment with these experiments.

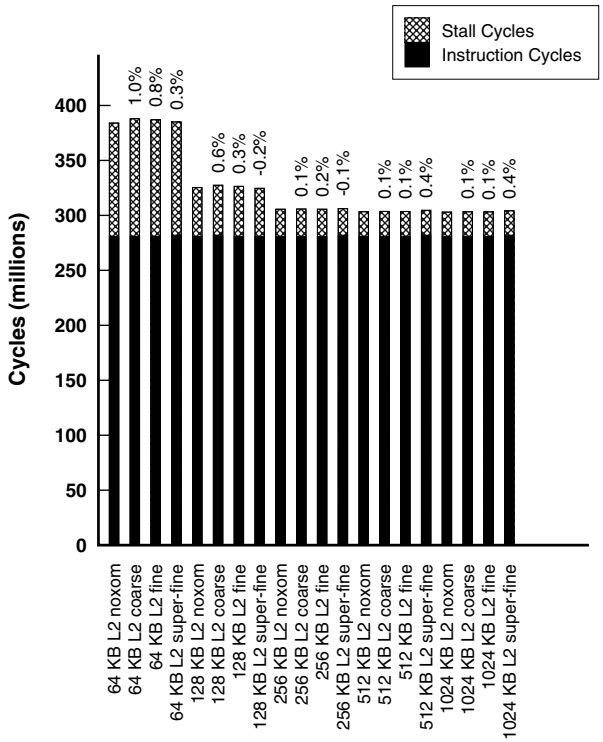

Figure 4: Performance of XOM-mpeg. Percentages above the bars show the increase relative to the non-XOM case for each cache size.

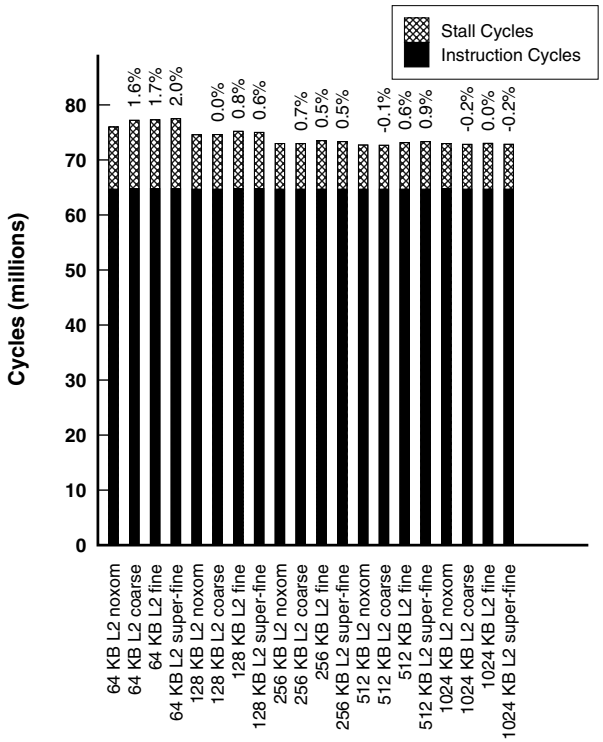

Figure 5: Performance of XOM-RSA. Percentages above the bars show the increase relative to the non-XOM case for each cache size.

XOM slows applications down in two ways. First, each XOM transition requires padding in the instruction stream, which can result in extra cache misses. Second, secure accesses to memory incur encryption or decryption latency. Minimizing these events will result in lower overhead imposed by using XOM compartments.

These performance considerations are balanced against security requirements. Placing a large portion of the application in the compartment reduces the amount of code visible to the adversary. We refer to this as *coarse-grained* XOM compartment usage. On the other hand, minimizing the portion in the compartment reduces the overheads associated with memory accesses, but may allow the adversary to infer more information about the application. We refer to this as *fine-grained* XOM compartment usage.

To study these effects, we created three versions of XOM-mpg123 and XOM-RSA, each at a different granularity of XOM compartment code. The *coarse* benchmarks encompassed the entire application except the initial start-up code. The *fine* benchmarks just protect the main algorithms that the application is using, and try to avoid making any system calls from inside the compartment to reduce the number of XOM transitions. For example, in XOM-mpg123, the code that decodes each frame of data is protected. This would expose the format of the MP3 file to an attacker, but would not expose the actual decoding algorithm. The fine grained version of XOM-RSA has each encryption and decryption function protected, but the code to setup those operations is in the clear. Finally the *super-fine* benchmarks seek a small operation to protect. This operation usually makes no system calls and has little or no memory accesses. In XOM-mpg123, only the Discrete Cosine Transform (DCT) function used in MPG decode is placed in the compartment. On the other hand, XOM-RSA only embeds the *bignum* implementation it uses to manipulate large integers. In this case, neither the cryptographic algorithms nor keys are protected, only the bignum implementation.

For each application, we identified the sections that were to be placed in the compartment and inserted `xentr` and `xexit` instructions to delimit the boundaries. In reality, the programmer must decide which loads and stores within the compartment code need to be protected, by identifying data structures that must be kept private. This requires significant compiler support, so to approximate the memory bus overheads due to encryption, the simulator was modified to mark addresses that had been written to while in the compartment. Subsequent loads and stores made to those addresses from the compartment incur cryptographic overheads that XOM imposes on memory operations. The rationale is that any data that is stored while in a compartment is to be kept secure by XOM until read again by a compartment. All instruction fetches from inside a compartment also require cryptographic operations. However, when not inside a compartment, instruction or data loads and stores proceed as normal without any XOM overhead. As mentioned in Section 4.2, protection against memory replay attacks is expected to be implemented by the application itself, but our ported applications do not implement this.

All three coarseness levels are simulated for each application. Both transition overhead and encryption overhead are affected by the cache behavior of the applications. To see how dependent it is, we also varied the cache size of the machine. The execution time results are shown in Figure 4 and Figure 5, broken down into cycles spent executing instructions and cycles spent stalled on memory.

The overall execution time is given in processor cycles. For the most part, the overhead is lower than the previous section's micro-benchmarks since the operating system overhead is diluted over a longer execution time. Adding XOM functionality adds very little overhead in general. This is not surprising for the following reasons: The XOM transitions do not result in many extra executed instructions since number of instructions cycles remain roughly the same, regardless of compartment coarseness. The XOM transitions may also negatively impact cache behavior, but this effect is minimal. This is because the number of XOM transitions is small (less than one for every 3000 instructions executed in the worst case). As a result, the simulations showed that the compartment granularity had no effect on the cache miss rate. For the MP3 application, the cache miss rate was about 20% for the 64KB L2 cache, about 7% for the 128KB L2 cache and less than 1% for all other L2 cache sizes. For the RSA application, the cache miss rate was about 4% for the 64KB L2 cache, and less than 1% for L2 cache sizes greater than 128KB. It is interesting to note that coarse grained security can sometimes result in a larger number of XOM transitions due to all the system calls that are made in the compartment.

Similarly, for the memory encryption overhead, note that both applications spend less than 30% of their execution time stalled on memory. Since the encryption overhead for each memory access is 10% of the access time (15 additional cycles to 150 cycles), this means that at most, the XOM encryption overhead will add about 3% to the overall execution time. In reality, this is further reduced by the fact that on average, only about 30% of the misses in the L2 cache actually required XOM cryptographic operations. It is interesting to note that in the XOM-RSA benchmark, there is a small slow-down as the compartment granularity gets finer. This is in spite of a decrease in the number of XOM memory operations. The reason for this is the additional cache misses caused by the larger number of XOM transitions.

On the whole, we noted that neither XOM-mpeg nor XOM-RSA stressed the memory system heavily. To find the upper bound on the XOM encryption overhead, we ported and simulated the McCalpin STREAM benchmark [19]. This benchmark is meant to measure memory bandwidth by executing sequential reads and writes on a large memory buffer. We found that the memory stall time made up about 40-50% of the overall execution time. Since the overhead on a memory access is about 10%, we expected the benchmark to have an execution overhead of approximately 4-5%. This was confirmed by our simulator.

7. SUPPORT FOR TRUSTED COMPUTING

XOM can be used to implement a platform for trusted computing by protecting software from observation and tampering. Trusted computing offers three key security mechanisms called *attestation*, *curtained memory*, and *sealed storage*. The XOM architecture supports these mechanisms as well, and many of the techniques used in XOM are directly applicable to trusted computing.

Attestation is a mechanism that allows a remote party to verify some properties about a remote program and the platform it is running on. For example, a remote party may want some guarantees that it is talking to an unmodified version of a specific program before it continues with communications. XOM's support for attestation is subtly different from the hardware attestation supported by TCPA or Palladium. In those schemes, the hardware signs a hash of the state of the machine guaranteeing that the remote machine is in a known state. Using XOM to make software tamper-resistant gives software the ability to attest for itself, without the aid of any other component. The software simply hides a signing key in its code image and uses this key to sign messages in a challenge-response protocol. Because XOM applications can only be decrypted and executed on the correct XOM processor, the software attestation also attests for the hardware. However, because each piece of software must attest for itself, software that does not use compartments cannot attest for itself. Since the OS does not typically run in a XOM compartment, it cannot attest for itself.

Curtained memory is a mechanism where some portion of memory is protected from observation and tampering. Palladium provides this mechanism by making a portion of physical address space inaccessible to software without the proper credentials. XOM provides curtained memory through the use of compartments. Compartments can be located anywhere in physical or virtual memory and storage for data in compartments can be swapped to a backing store. In addition, compartments are implemented entirely in the processor and do not require any modifications to the memory or memory controller. Compartments are resistant to direct attacks on the hardware in the memory system, so even an adversary who has access to the memory bus or who can emulate memory, cannot compromise a compartment.

Sealed storage is a mechanism that allows programs to store data in memory or on disk so that only programs with the proper credentials can access it. XOM implements sealed storage by having programs hide keys in their program image. Programs may then use these keys to encrypt and decrypt data that is stored in the sealed storage. To authenticate the contents of sealed storage XOM can use hash trees [18]. The only caveat is that XOM, like Palladium and TCPA, will require some form of non-volatile RAM to store the root of the tree.

8. CONCLUSIONS

Currently, there exist various initiatives that place the trust in modern computing systems in a hardware component rather in the operating system. In these systems, the applications don't trust the operating system to protect their data, but the operating system also does not trust the application to properly use its resources. The result is that the interface between the operating system and applications must change to support the hardware security features, and some of the protection aspects of the operating system must be moved into the hardware. This paper studied how these changes can be implemented and what the impact of those changes on the performance of the system is. To do this, we modified the original XOM architecture to better support an operating system and implemented the XOMOS operating system for study.

We found that XOMOS could be written by modifying a standard operating system such as IRIX. The size of the modifications on the original operating system was modest — about 1900 lines in roughly 20 files were modified. As one would expect, most of the modifications dealt with the low-level interface between the operating system and the hardware, and with routines that copied and saved application state. Because of this, we feel that the same types of modifications could be applied to a wide range of operating systems. Since managing protected data is much more expensive than normal data, care must be taken to ensure that this processing is only done when needed, and also that it is needed infrequently. We were able to find techniques to achieve this.

Although the basic XOM architecture that was originally proposed already has the basic primitives required to support copy and tamper-resistance, we found that certain features in hardware are required to enable the implementation of XOMOS. To virtualize and manage resources, XOMOS must be able to relocate data in physical space while the XOM processor checks the integrity of data in virtual space. As a result, facilities must be provided for the operating system to identify the owner of data, and the security hashes of memory data must be available to the operating system, so that it may relocate data. Finally, the decomposition of complex functions into simple primitives, as in the case of register saves and restores, allows the operating system to better control resource usage. In the case of XOM Key Table management, the act of allocating a resource can be privileged, and thus decoupled from the act of accessing the resource, which remains unprivileged.

Our preliminary performance numbers look promising. The hardware overheads are not small — with memory encryption and decryption costing 15 cycles and saving and restoring a protected register requiring 13 instructions instead of 2. However, these costs are only incurred when the machine must do an even more expensive operation — either a memory fetch (which takes 150 cycles) or a trap into the kernel. As a result, we found that in reality, end-to-end application overheads are often less than 5%, regardless of the granularity of the XOM compartments. Since coarser compartments should be more secure, we conclude that the use of coarse compartments, where the majority of the application is executed securely, is viable. This reduces the burden on the developer to identify and secure the sensitive portions of an application.

These results have encouraged us to explore other issues such as using XOM to increase security in the file system. We believe with the current trend towards trusted computing platforms, the techniques explored in this paper will be valuable as guides to the design and implementation of such systems.

Acknowledgements

The authors would like to thank Robert Bosch, Bob Lantz and Kinshuk Govil, who were instrumental in getting SimOS working. The authors would also like to think John DeTreville for his thoughtful comments on an earlier draft of the paper. The research presented in this paper was performed with the support of DARPA Contract F29601-01-2-0085.

9. REFERENCES

[1] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of Summer Usenix*, pages 93–113, July 1986.

[2] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139:19–23, 2001.

[4] Business Software Alliance, 2003. `http://www.bsa.org`.

[5] J. Daemen and V. Rijmen. AES proposal: Rijndael. Technical report, National Institute of Standards and Technology (NIST), Mar. 2000. Available at `http://csrc.nist.gov/encryption/aes/round2/r2algs.htm`.

[6] P. England, J. DeTreville, and B. Lampson. Digital rights management operating system. U.S. Patent 6,330,670, Dec. 2001.

[7] P. England, J. DeTreville, and B. Lampson. Loading and identifying a digital rights management operating system. U.S. Patent 6,327,652. Dec. 2001.

[8] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, 1995.

[9] B. Gassend, E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 295–306, Feb. 2003.

[10] T. Gilmont, J. Legat, and J. Quisquater. An architecture of security management unit for safe hosting of multiple agents. In *Proceedings of the International Workshop on Intelligent Communications and Multimedia Terminals*, pages 79–82, Nov. 1998.

[11] T. Gilmont, J. Legat, and J. Quisquater. Hardware security for software privacy support. *Electronics Letters*, 35(24):2096–2097, Nov. 1999.

[12] J. Heinrich. *MIPS R10000 Microprocessor User's Manual*, 2.0 edition, 1996.

[13] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. `http://www.ietf.org/rfc/rfc2104.txt`, Feb. 1997.

[14] M. Kuhn. The TrustNo1 cryptoprocessor concept. Technical Report CS555, Purdue University, Apr. 1997.

[15] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.

[16] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 166–178, May 2003.

[17] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference Architectural Support for Programming Languages and Operating Systems*, pages 168–177, Nov. 2000.

[18] U. Maheshwari, R. Vingralek, and B. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 135–150, Oct. 2000.

[19] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec. 1995.

[20] OpenSSL, 2003. `http://www.openssl.org`.

[21] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(18):120–126, Feb. 1978.

[22] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, Jan. 1997.

[23] J. Rushby. Design and verification of secure systems. *ACM Operating Systems Review, SIGOPS*, 15(5):12–21, Dec. 1981.

[24] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[25] SGI IRIX 6.5: Home Page, May 2003. `http://www.sgi.com/software/irix6.5`.

[26] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Dec. 1999.

[27] S. W. Smith, E. R. Palmer, and S. Weingart. Using a high-performance, programmable secure coprocessor. In *Financial Cryptography*, pages 73–89, Feb. 1998.

[28] The Trusted Computing Platform Alliance, 2003. `http://www.trustedpc.com`.

[29] J. D. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. In *Harvard-MIT Workshop on Protection of Intellectual Property*, Apr. 1993.