ARCHITECTURAL SUPPORT FOR COPY AND TAMPER-RESISTANT SOFTWARE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

David J. Lie
December 2003

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dr. Mark Horowitz
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dr. Dan Boneh

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dr. Chandramohan A. Thekkath
(Microsoft Research, Silicon Valley)

Approved for the University Committee on Graduate Studies.

Abstract

Recently, there has been intense interest in the implementation of a trusted computing platform. Industry projects such as the Trusted Computing Platform Alliance, Microsoft's Palladium Project, and Intel's LaGrand Technologies all aim to embed hardware to support some amount of protection for applications so that they can be tamper-resistant.

In this work, we propose a new processor architecture called "XOM", which stands for eXecute Only Memory. XOM provides copy and tamper-resistance for software by supporting compartments, which protect both the code and data of programs. Compartments are implemented by a combination of architectural methods, in the form of on-chip access control tags, and cryptographic methods, in the form of ciphers and hashes that protect data off-chip. The trust model of the computing system is changed so that applications trust the hardware, instead of the operating system, to protect their code and data. A XOM processor was simulated by extending a MIPS-based processor model in the SimOS simulator.

An operating system, XOMOS, was constructed run on the XOM architecture. Because the applications do not trust the operating system with their data, this presents an interesting challenge for operating system design. This work shows that an untrusted operating system can be implemented on top of trusted hardware, such that the operating system has sufficient rights to manage resources, but does not have the rights to read or modify user application code or data. This is demonstrated by a port of the IRIX 6.5 operating system to the XOM processor, to create XOMOS. We were able to run some applications on XOMOS in our simulator and found overheads to be less than 5%.

We used a model checker to verify the security of the XOM processor architecture. A realistic "actual" processor was modeled along with an adversary, and compared against a "idealized" model that has no adversary. Inconsistencies between the two models are flagged as failures in the protection guarantees that the processor aims to provide. We thus demonstrate that the processor is able to provide tamper-resistance, and that the most difficult attack to defend against is a memory replay attack.

Acknowledgments

This work would not have been possible without the help and support of a large number of people.

First and foremost, I would like to thank my advisor, Prof. Mark Horowitz. There is little I can say to add to his already impeccable reputation as a researcher, scientist and teacher. As my supervisor, he has been patient and kind, and has always encouraged me to push myself further. It has truly been a privilege to work with him.

Dr. Chandramohan (Chandu) Thekkath has also been a great collaborator. He has generously donated his time and energy over the last four years. The numerous discussions we have had have been enjoyable, and I have benefited greatly from his knowledge of operating systems. I would also like to thank Prof. Dan Boneh, my associate advisor, who was responsible for sparking my interest in computer security and cryptography, as well as educating me in those fields. Thanks also go to Prof. John Mitchell, who always made himself available to proof read papers and help clarify my ideas. Prof. Dawson Engler and Prof. David Dill first introduced me to the concept of using formal techniques to check models for correctness, which subsequently proved very useful for my dissertation. Numerous thanks also go to Prof. Bill Dally, Prof. Mendel Rosenblum, Prof. Monica Lam, and President John Hennessey for allowing me to draw upon their experiences and technical knowledge.

This work would not have been possible without the support of Charles Orgish and Joe Little, who worked tirelessly to maintain a stable computing environment for research at the Computer Systems Laboratory at Stanford. Of no less importance has been the administrative support of Terry West, Deborah Harber, Taru Fisher, Penny Chumly, Teresa Lynn, Darlene Hadding, and Claire Ravi.

I would also like to thank DARPA for their financial support, and Stanford University for providing fantastic facilities and a comfortable and relaxing environment, which allowed me to focus my attention on my research.

There are too many colleagues and friends to mention all who have made my time at Stanford memorable. Thus, in the limited space I will mention the graduate students whom I worked with in the Gates Building at Stanford. Robert Kunz and Jeff Gibson who worked next door helped show me my way around Stanford, helping me pass quals and giving me technical advice. They were good companions on the road as well, providing excellent company at conferences we attended. Andy Chou has also been a great friend and co-author, and I will always admire his patience and ability to write large amounts of code quickly, without sacrificing elegance and clarity. I would also like to thank Dean Liu, Francois Labonte, my office mates Alex Solomatnikov, and Amin Firoozshahian, as well as all the other members of the Horowitz group for their support and help.

I would like to thank the members of my family, who first encouraged me to apply and come to Stanford. And finally, words cannot express the gratitude I have for Xiaoli Liang, who has been my loving and caring companion for nearly all the time I have been at Stanford. It is impossible to imagine how this thesis would have been possible without her help and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There are many 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, as well as for preventing viruses from modifying legitimate programs. This dissertation looks at one approach of providing this capability by modifying the base processor hardware to support programs that can only be executed, but cannot be read or modified. However, even without such a high level of tamper-resistance some applications assume it exists. For example, banking transactions still assume the tamper-resistance of software on the clients, even though there is no such assurance. While protocols such as Secure Sockets Layer (SSL), can reliably secure communications across the Internet, the user has no assurance that their web browser or underlying operating system is not maliciously monitoring or tampering with their transactions. If for example, a banking customer is unknowingly using a malicious web browser[1], the web browser may record all of the customer's personal information and use that to impersonate the customer.

In another case, imagine the web browser is legitimate, but the underlying operating system has been compromised. The operating system may allow the banking customer

[1]A customer may unknowingly download a modified client — in other words a Trojan Horse.

to initiate a transaction, but at an opportune time, simply hijack the network session by extracting the appropriate information from the address space of the web browser. This example shows that protecting the web browser from tampering is necessary, but not sufficient — it is also necessary to protect the web browser's execution from observation, otherwise the identity of the browser can be forged.

Systems can require various levels of tamper-resistance, which range from simply preventing an adversary from reading the instructions of a program, to protecting the instructions, data and execution of the program entirely. The level of security that an application requires depends on the sophistication of the adversary the system is intended to withstand. On the other hand, increased security may also require increased costs in performance overhead and hardware.

A very simple and cheap form of tamper-resistance can be provided by *software obfuscation*. Software obfuscation requires no hardware support and is implemented by adding entropy to the instructions of a program. Typically this involves adding extra code and transforming existing code so that it is difficult for humans or code analysis tools to understand, but can still be executed to produce results similar to the original program. In the past, software obfuscation has been explored and implemented, but with limited success [9]. Further, there is theoretical evidence that a general software obfuscation scheme does not exist [6]. As a result, software obfuscation is only effective against an unsophisticated adversary. Given enough time and resources, a reasonable adversary will be able to de-obfuscate the program by brute-force analysis — the reason being that if there exists a machine that can execute the code, then there likely exists another machine, possibly more complex, that can decode its operation.

A stronger way of obfuscating software is to encrypt the executable with a computationally strong cipher that will thwart a brute-force attack. Some proposals do this by placing cryptographic functions in the memory controller hardware [21, 22]. However, such systems are prone to software-based attacks — because the execution of the program is not protected, an attacker may examine the dynamic state of a program using a debugger or other such tool, and surmise the instructions that are being executed. While this requires slightly more intelligence on the part of the adversary, it is not a major hurdle for attackers. Such systems fail because there is no concept of *isolation* between different programs running on the processor.

The simplest way to provide isolation is to simply execute the tamper-resistant code on another processor. Systems described in [59, 62] support security through the use of *secure coprocessors*. These coprocessors are used to execute tamper-resistant portions of programs and are placed in physically hardened packages to prevent tampering by an adversary. The limited programming interface protects programs against software-based attacks while the physical packaging defends against hardware-based attacks. However, these coprocessors typically run much slower than the rest of the system and must be provided with their own memory and I/O interfaces. As a result, this security comes at the cost of both decreased performance and additional hardware.

The performance and hardware cost can be reduced by incorporating the features of the secure coprocessor into the main processor. Systems such as these combine architectural methods such as access control tags to protect code and data while it is on the main processor, and cryptographic techniques to protect code and data while it is in memory or on disk [35, 39]. The modifications are generally restricted to the main processor. The physical security relies on the hardening of the main processor package, but does not trust anything out of that package.

To support copy and tamper-resistant software, we propose a set of processor extensions, which are called “XOM”, pronounced “zom”, an acronym for eXecute-Only Memory. We create a mechanism, where code stored on disk or other media can be made so that it can only be executed, but cannot be read or modified, making it tamper-resistant. Such code, which is referred to as “XOM code” will only execute on certain hardware. In other words, the code *authenticates* the hardware it is running on. This authentication, combined with the fact that the code cannot be read, makes it copy-resistant.

The design of such systems must address issues with security, performance and flexibility. First and foremost, the system must meet its security requirements. This dissertation assumes a sophisticated adversary who has a wide variety of software and hardware tools available to her. She can execute code in privileged mode as the operating system. She can also use hardware techniques to modify and observe values in memory. The XOM hardware must be able to defend against all such attacks. Second, XOM must also operate at a reasonable level of performance. It is difficult to quantify what performance cost is reasonable for such a high level of security, so this work aims to keep the cost as low as possible without compromising the security of the system. Finally, XOM must provide flexibility. Facilities must be provided for programs to perform operations that they would be able to perform in a regular system without XOM, so long as those operations are compatible with the security requirements of the system. In other words, XOM should not impose any unnecessary restriction on software the utilizes its tamper-resistant support.

In this dissertation, we find that highly tamper-resistant software that can even defend against a compromised operating system, requires that some amount of specialized hardware be added to the processor. We will show that such hardware is modest in size and has a modest impact on performance. We do this by proposing a hardware architecture and studying its performance via a simulator, and its security via formal verification methods.

1.1 Overview of this Dissertation

This dissertation comprises six chapters. This chapter introduced the problem and gave a brief overview of the design space. Chapter 2 will introduce some of the concepts discussed in this dissertation. It will talk about the cryptographic mechanisms that are used, as well as *compartments*, which are logical containers in which we place our tamper-resistant software. They are what provides the *isolation* that allows programs to execute free from unauthorized observation or modification.

We then proceed to present the hardware we need to support copy and tamper-resistant software. The XOM processor bases its security in hardware, so we discuss the hardware mechanisms that we use to extend a standard processor to enable it to support compartments. These involve a combination of the cryptographic methods discussed in Chapter 2, and architectural methods such as tags and caching. In Chapter 3, we discuss the functionality required by defining an abstract XOM machine. We then discuss implementation alternatives, as well as a cycle accurate simulator that was constructed.

Operating systems inevitably have errors in them that make them vulnerable to attack. Under XOM, applications trust the processor to protect their data and execution, but do not trust the operating system. However, sharing hardware resources among multiple users is a difficult task, often requiring complex policy decisions. Thus, resource management is most naturally done in software by an *operating system*. Chapter 4 discusses the implementation of an untrusted operating system on the XOM architecture. Using the simulator discussed in Chapter 3, this chapter also presents the performance results for our modified operating system running a set of test applications.

However, while the implementation and performance may be reasonable, the XOM architecture is off little worth unless its security can be justified. In Chapter 5, we discuss the attack model XOM defends against, and discuss the mechanisms XOM uses to defend against an adversary. The security is analyzed using *formal verification* techniques in the form of a model checker.

Finally, we make our conclusions in Chapter 6 and propose future work that may be of interest.

Chapter 2

Cryptographic Concepts

This chapter discusses some cryptographic concepts that are used in this dissertation. XOM protects programs by controlling access to the program code and data. Access Control Lists, the natural data structure for performing this task, are well understood. However, XOM uses a slightly different abstraction, called a *compartment* [53]. The origin and motivation behind using compartments will be discussed in Section 2.1. XOM uses standard architectural tags to implement compartments for values in the trusted hardware of the processor. To extend this control into untrusted storage such as memory or disk, XOM uses cryptographic techniques, specifically symmetric ciphers, asymmetric ciphers and message authentication codes. These will be discussed in turn in the following sections.

2.1 Compartments

The purpose of XOM is to control access to data and code such that a program with the appropriate rights can access the code for execution, as well as the static and dynamic data the code requires. At the same time, XOM should both hide the value of that code and data from adversaries, and prevent such an adversary from modifying those values. Normally, access control refers to guarding data, but in this case, XOM is protecting an active principal such as a program together with its data. This kind of access control is more appropriately called *isolation*. In [53], Saltzer and Schroeder define *complete isolation* as:

Complete Isolation: A protection system that separates principals into compartments between which no flow of information or control is possible.

The operations XOM performs to protect programs provide isolation by implementing these compartments. When data is in a compartment, we say that the compartment *owns* that data. Since programs are active principals inside a compartment, we may also say that the program owns the compartment (and by extension, all the data inside the compartment). One should note that *complete* isolation means that *no* flow of information is possible. For practical systems, this is too restrictive since even programs that *want* to share information would not be able to. To implement a more pragmatic form of isolation, XOM provides a *shared compartment* that acts as a common medium available to all programs for information sharing. In contrast, all other compartments are called *private compartments*. Programs must explicitly move information between their private compartments and the shared compartment. Thus, private compartments are isolated from the shared compartment, but not vice-versa. If two programs wish to communicate information securely, they may do so by negotiating a shared secret key and treating the shared compartment as an insecure channel.

2.2 Symmetric Ciphers

Ciphers are algorithms that are used to obscure information in a way that only a principal who knows a certain value, called a *key*, can recover the original message. Information in its plain form is called *plain text*. The act of obscuring this information is called *encryption* and produces *cipher text*. The strength of modern ciphers is grounded on principles that are beyond the scope of this dissertation. Instead, we make common cryptographic assertion that the strength of ciphers is based on the length and secrecy of the keys.

Symmetric ciphers are a class of ciphers that use the same key for encryption and decryption. The sender uses the key to encrypt a message and send it to the receiver. The receiver then uses the key to decrypt and recover the original message. An adversary eavesdropping on the conversation will not be able to understand the message because she does not know the key that the sender and receiver share. Some common examples of symmetric ciphers are 3DES [5] and Rijndael [13].

Symmetric ciphers typically operate by permuting elements within the message, and by non-linear substitutions with the use of lookup tables. As a result, efficient hardware implementations of symmetric ciphers are possible. However, the draw back is that encryption and decryption are performed with the same key. This means that these ciphers can only be used if the sender and the receiver already share a secret key. *Key distribution* becomes a problem that is often solved by asymmetric ciphers, which are discussed in the next section.

2.3 Asymmetric Ciphers

Asymmetric Ciphers use a different key for encryption and decryption. The encrypting key is called a *public key* while the decrypting key is called a *private key*. The private and public key pair are randomly created during a *key generation* phase and then are stored for use after that. The principal that will receive the protected data (receiver) will generate one such pair and distribute the public key. Everyone, including the sender of the protected data (sender) and the adversary know the value of the public key, but not the private key. For this reason, asymmetric ciphers are also commonly referred to as public key ciphers. If a sender wants to send a message, she takes the public key of the intended recipient and uses it to encrypt the message. The receiver keeps the private key secret to herself and thus will be the only principal who will be able to recover the original message. An important property of asymmetric ciphers is that the public key reveals no information about the private key.

Asymmetric ciphers rely on a type of mathematical function called a "one-way trap door function." What this means is that once the function is applied, it is difficult to undo it unless there is knowledge of a special "trap door." In this case the trap door is the private key that the receiver has. For example, the El Gamal cipher [16], uses discrete logarithms as a trap door function, while RSA [50] is based on factoring. These types of functions do not lend themselves well to efficient hardware implementations, and as a result, asymmetric ciphers are orders of magnitude slower to compute than symmetric ciphers.

Typically, asymmetric ciphers and symmetric ciphers are used together to compensate for each other's short comings. A random symmetric key is chosen and the fast symmetric cipher is used to encrypt the message that is to be sent. The short key is then encrypted with the public key using the slow asymmetric cipher. The message is usually much longer than the symmetric key, and the asymmetric cipher is only used to encrypt the symmetric key for the message. As a result, for messages of sufficient length the symmetric cipher will dominate the cost of encrypting the message.

2.4 Message Authentication Codes

Using a cipher only prevents the adversary from learning the contents of a message. In other words, it prevents observation, but does not prevent or detect modification. Consider a scenario where an adversary substitutes some random cipher text in place of the actual cipher text in transit between a sender and a receiver. Even though the adversary cannot predict what plain text the receiver will recover when she decrypts the cipher text, the receiver will not be able to verify that the cipher text has been altered and may possibly use the incorrect data.

One could use one of the ciphers discussed in the previous sections to solve this problem. The sender can *sign* the message by encrypting it with a key known only to the sender and the receiver, and then sending both the plain text and cipher text to the receiver (note that we are not concerned about secrecy in this example, though secrecy could be acheived by encrypting with yet another key). The receiver can then verify that the plain text and cipher text match. Since only the sender knows the secret key required to create the cipher text, this prevents an adversary from *forging* the message. In this way, we provide a form of *authentication* for the message. A similar solution can be made using an asymmetric cipher instead of a symmetric one. The problem with either solution is that it is inefficient. Essentially, the message will have to double in length to include both the plain text and cipher text.

To create an efficient solution, we calculate a shorter hash of the message, and encrypt the hash of instead of the entire message. The receiver simply calculates the hash of the plain text and verifies that against the encrypted hash. A keyed hash of this type is commonly referred to as a *message authentication code* or MAC [34]. MACs are used by XOM to detect tampering of values in memory and disk.

Chapter 3

The XOM Architecture

XOM uses hardware methods to provide tamper-resistance for software. The goal is to allow multiple programs, to co-exist on the same hardware, but at the same time, to provide strong guarantees that they cannot violate each other's compartments as illustrated in Figure 3.1. Though the operating system has the ability to manage resources, it is untrusted, and thus should not be able to circumvent the protection provided by the compartments.

The XOM hardware architecture implements the primitives necessary to support compartments. These primitives exist as a set of extensions that can be added to any general processor architecture. However, these extensions do not fundamentally alter the overall operation of a the system, and for the most part, the processor operates in a fashion that is very similar to the original. Regular applications will continue to be able to execute on a XOM processor unmodified. To become copy and tamper-resistant though, they must be altered to make use of the XOM extensions.

This chapter looks at the hardware and performance overheads that compartments create. It starts by describing the essential operations that a XOM machine must perform: creating, utilizing, and ultimately destroying a compartment, and protecting the compartment's memory and register data. These operations will use both asymmetric and symmetric ciphers for efficiency. We will also discuss protection in external memory, as well as implementation issues for operating systems, and effects on the distribution model for software. To better understand what is required to protect the compartments, this chapter will briefly preview the types of attacks the machine needs to protect against — but detailed

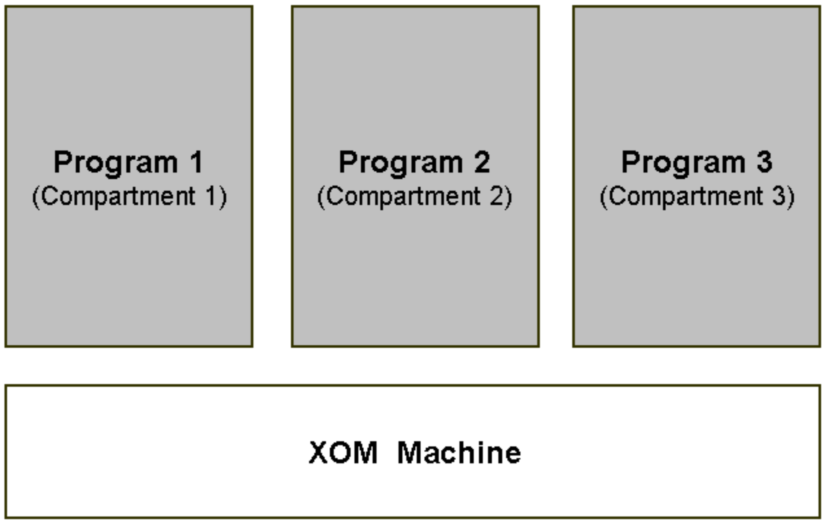

Figure 3.1: Compartments. The XOM Machine keeps each program in its own compartment. The compartments isolate the programs from each other, preventing the flow of information between them.

discussion of attacks will be deferred to Chapter 5.

The hardware required to implement this machine can be quite small, since most of the functionality can be provided by a trusted virtual machine monitor to create a virtual XOM machine. Unfortunately, this simple machine needs to encrypt/decrypt data on each memory operation. The chapter then describes how this overhead can be greatly reduced by adding tags to both the registers and the on-chip cache memory, as well as the use of hardware accelerators for cryptographic operations.

To study hardware implementation issues, we created a XOM hardware simulator. The simulator is built on top of the SimOS [28] using an architecture based on a MIPS R10000 processor [27]. We then go on to discuss issues on maintaining XOM systems, such as upgrading hardware to which software has been tied, as well as recovering software from a hardware failure. Finally, this chapter finishes with a discussion of related work that supports copy and tamper-resistance.

3.1 The Abstract XOM Machine

We begin this section by describing an Abstract XOM Machine. This is a set of extensions that when added to a generic machine, allow it to securely execute code in compartments. The XOM machine implements compartments as described in Section 2.1 to isolate programs executing on the machine from each other. To implement these compartments, the XOM machine will track which compartment a particular piece of code or data belongs to. To do this, the XOM machine associates a unique identifier, called a *XOM ID* with each compartment. Data and code in the machine are tagged with the XOM ID of the compartment to which they belong. When code is executing in a compartment, the XOM ID for that compartment becomes the *currently active XOM ID*. As a result, all data accessed by that program must be tagged with the same XOM ID as the compartment, otherwise an exception occurs and the program is halted. When a program writes data to a location, the tag on that location is set to the XOM ID of the active compartment. In this way, the XOM machine associates executable code with data that is *owned* by that program in its compartment. The XOM machine extends the base processor architecture with a *XOM ID register* that holds the XOM ID of the currently active compartment, as well as adding *XOM Compartment Tags* to all registers. Operating in a compartment penalizes programs with some amount of performance overhead. As a result, the XOM machine provides programs the ability to choose whether to execute in a compartment or not. An `enter xom` instruction is added to the instruction set architecture (ISA) of the machine that allows programs to indicate that they are going to start executing in their compartment and that all following instructions should be protected. Likewise, an `exit xom` instruction executed from within a compartment indicates that the program wishes to leave protected execution. With this mechanism, a program can even execute partially protected in a compartment, and partially in the clear with no protection. The actual choice of when to execute securely and when not to depends on the characteristics of the application, as executing in a compartment incurs certain penalties that will be discussed later.

Compartments provide a way for programs to execute without fear of observation or tampering. However, the compartment also prohibits the intentional flow of information. For example, if a program in a compartment wishes to share a result with another program in another compartment, it would not be able to do so since the other program's XOM ID does not match the tags on the registers. To allow controlled communication, XOM provides a *shared compartment* as described in Section 2.1. This compartment has a XOM ID of zero and programs are by default unprotected in the shared compartment before they execute an `enter xom`. On an `exit xom` instruction, programs return to the shared compartment. To allow protected programs to use the shared compartment, the XOM Abstract Machine extends the base ISA with two additional instructions to allow communication between programs. This communication is provided by the `move to shared` and `move from shared` instructions. These instructions provide a controlled way to change the tags associated with a piece of data. The `move to shared` instruction takes data that is tagged with the active XOM ID and changes the tag on it to the shared XOM ID. Executing this instruction on data that is not owned by the program results in an exception. The `move from shared` instruction changes data tagged by the shared XOM ID to the active XOM ID. Once again data has to be tagged with shared before this instruction can be executed. These two instructions, in combination with `enter xom` and `exit xom`, allow a program to keep part of its execution and data in a compartment, while keeping another part in the shared compartment as illustrated in Figure 3.2. While in its private compartment, the program can move data back and forth between its shared and private compartments with the `move to shared` and `move from shared` instructions.

Thus, a program can transmit data from its compartment to another through the shared compartment as follows:

1. Program A puts the data to be shared in a register. The register is tagged with Program A's XOM ID.

2. Program A executes `move to shared` on the register, moving the data into the shared compartment. The register is now tagged with the shared XOM ID.

3. Program B executes `move from shared` on the register, moving the data from the shared compartment to Program B's compartment. The register is now in Program B's compartment.

Note that data in the shared compartment can be read by any program, not just the intended recipient. If Program A and Program B wish to ensure that the data is transmitted without

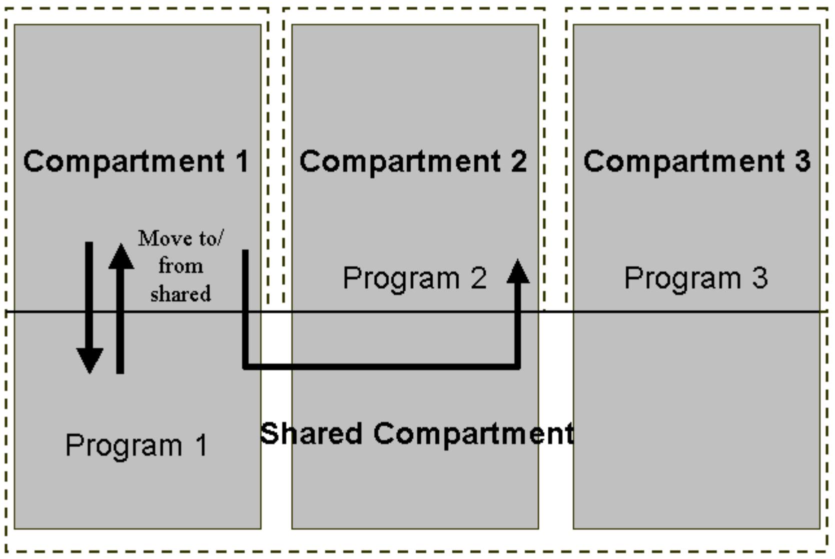

Figure 3.2: The Shared Compartment. The Shared Compartment is used as a common compartment through which programs with separate private compartments can communicate. Programs enter and exit their compartments with the `enter xom` and `exit xom` instructions, and move data between their compartment and the shared compartment with the `move to shared` and `move from shared` instructions. Data can also flow between private compartments through the shared compartment.

fear of tampering or observation by an adversary, they can set up a secure channel with a shared key.

3.1.1 Supporting External Memory

A XOM machine such as the one above would require all secure code and data to fit onto the processor. This is unlikely given the size of today's programs. While it would be possible to extend the tagging scheme into memory, this presents possible security flaw. Communication between the processor and memory usually occurs on a bus implemented as traces on a circuit board. This can be probed by an adversary with methods orders of magnitude cheaper than those that would be required to probe on-chip signals [31, 49]. As a result, the XOM design must assume that any data that has been transmitted off the chip, can be potentially read or altered by an adversary. In other words, hardware that is not in the same chip package as the main processor is assumed to not be tamper-resistant. The security of tags relies on the tamper-resistance of the chip package. To protect memory we turn to cryptography. XOM encrypts data leaving the processor and decrypts it when it is loaded back onto the processor. To allow the machine to check the integrity of the data as well as protect it from observation, a MAC is associated with all values stored in memory. If an external agent tampers with the data, then the MAC will not verify and the instruction will cause an exception. As discussed in Chapter 2, encryption algorithms require a key with which to encrypt or decrypt text. As a result, every compartment must have a unique *compartment key*. Since the program code is initially in the compartment and in memory, it must be encrypted with the compartment key and accompanied by a MAC before execution. Compartment code is decrypted with the compartment key before execution. The XOM machine also uses this compartment key to encrypt and decrypt data stored by the program in memory.

Because these cryptographic operations incur some overhead, the XOM machine allows programs in a compartment to indicate whether data going to memory should be kept in the compartment and encrypted, or whether it should be left outside the compartment in plain text. This is done by adding a `secure store` instruction, which stores data securely to memory, and a `secure load` instruction, which will load secure data from memory. These do not replace regular loads and stores, which interact with data in memory without encrypting or decrypting it. The `secure load` instruction takes a destination register and memory location as operands. It decrypts the contents of memory using the active compartment key, verifies the MAC, loads the decrypted value into the register and changes the tag on the register to the active XOM ID. If there is a mismatch in the MAC, the instruction will cause an exception. The `secure store` instruction stores data to memory securely, by encrypting it with the compartment key and adding a MAC. It can only be executed on a register tagged with the active XOM ID, while a regular store can only be executed on a register tagged with the shared XOM ID. The `secure store` instruction encrypts the register contents as well as creating a MAC, and stores both to memory. If the register sourced in a `secure store` is tagged with a XOM ID other than the active one, the instruction raises an exception.

3.1.2 Supporting an Operating System

Now we have the essential primitives to allow a single program to run securely and use memory. However, programs must typically run alongside other programs, sharing resources on the same processor. The allocation of resources is governed by an operating system. To manage resources effectively, the operating system must be able to arbitrarily interrupt the program, store its state to memory, and then restart the program with that state at a later time. The operating system must also be able to relocate program data that is in memory. However, XOM does not trust the operating system. This is because there are many methods with which an adversary could compromise the operating system and gain control of it [10]. When a XOM program is interrupted, the contents of the registers are still tagged with the XOM ID of the interrupted program. As a result, the operating system is unable to read those values to store them to memory. We need to add two more instructions to the ISA — the `save register` and `restore register` instructions. To allow the operating system to relocate secure data in memory, we make the MACs available to the operating system so they can be moved together with the encrypted data.

The `save register` and `restore register` instructions are used by the operating system to move data that it does not own and does not belong to the shared compartment. The `save register` instruction takes the contents of a register and creates an encapsulated version of this data the operating system can move, but cannot manipulate. It first encrypts the register contents and then calculates a MAC that includes the identity of the register. It places the encrypted data, MAC, and the XOM identifier into a set of special registers, which are owned by the operating system. This data can then be stored to memory.

The `restore register` instruction is the inverse of the `save register` operation and it is used to restore the data back to the registers before restarting a program. The instruction uses the special registers that hold the encrypted data, MAC, and destination key identifier. The operating system then indicates with the instruction, which register and what compartment XOM ID it wishes to restore the register value to. The XOM machine decrypts the data and checks the MAC. If the MAC verifies both the decrypted data and

Instruction	**Description**
`enter xom`	Enter secure execution and set the currently active XOM ID to the current compartment. All following instructions are in the compartment and should be encrypted and accompanied by a MAC.
`exit xom`	Exit XOM compartment and return to the shared compartment. Set the currently active XOM ID to shared.
`secure store`	Stores register to memory securely.
`secure load`	Loads memory securely from memory to a register.
`save register`	Encrypts and saves a register to memory so that the operating system can save a program's state.
`restore register`	Decrypts a value from memory and places it a register to restore a program's state.
`move to shared`	Sets the XOM ID tag of register to the shared compartment.
`move from shared`	Sets the XOM ID tag of register to that of the executing program.

Table 3.1: Instructions in the Abstract XOM Machine. This table summarizes the instructions that the XOM Abstract Machine adds to the base instruction set architecture.

destination register were not tampered with, the decrypted data is written into the destination register and the register tag is set to the target XOM ID. In this way, we ensure that the register contents are not altered and that the values are restored back to the same register from which they were saved.

The other issue is to enable the operating system to relocate data that has been stored and encrypted in memory. This is done by making the MACs stored by the XOM processor available to be copied or moved along with the encrypted data. Thus when the operating system wishes to relocate values in memory or swap them to disk, it copies both the MAC and the cipher text together. This does not require any special instructions as the cipher text in memory can be handled with regular memory operations.

At this point we have described all the instruction set additions XOM needs to make to the underlying architecture. We summarize these instruction in Table 3.1. We will now discuss how this architecture affects the software distribution model. Further, we will discuss the security of the XOM machine, by examining the types of attacks XOM defends against.

3.1.3 Software Distribution Model

Since programs are typically created on a system other than the one that will execute the program, the compartment keys must be transmitted from the program producer to the processor that does the execution. Asymmetric ciphers are the perfect tool to accomplish this. Figure 3.3 illustrates a flowchart for this process. By hiding a private key in the hardware of the processor and distributing the public key, programs have a way of transmitting the compartment key to the processor for execution. As discussed in Section 2.3, the software producer simply encrypts the program image with the compartment key, and then encrypts the compartment key with the public key of the target processor. Since this private key is used to protect all compartment on a machine, it is referred to as the *master secret*. If every XOM machine is initialized with a different public/private key pair, then this provides a way for a program to authenticate the processor it is executing on, as it will only be possible for the processor with the correct private key to decrypt and access the compartment key. This provides *copy protection* for programs. Without the compartment key, a processor cannot decrypt and execute the program. Without knowledge of the master secret, even an adversary with sophisticated virtualization or simulation technology, will not be able to fake or forge a XOM machine, and fool software into executing.

To execute compartment code, the XOM processor must first decrypt and recover the compartment key. As a result, what the `enter xom` instruction actually does to enter the compartment, is to decrypt the compartment key with the private key or master secret of the XOM machine, and then assign a XOM ID to it. The decrypted compartment key is then stored in a hardware table called the *XOM Key Table*, which maps the compartment keys onto XOM IDs. The XOM machine uses the XOM Key Table to find the appropriate compartment key to decrypt instructions and for use with the `secure load` and `secure store` instructions. For example, during a `secure load` the XOM machine does a lookup on the XOM Key Table using the value in the currently active XOM ID register. With this, it gets the correct compartment key, and uses that to decrypt the value that is coming from memory.

The `exit xom` instruction unloads the compartment key from the XOM Key Table. This frees up entries in the XOM Key Table for use by other programs. However, this opens

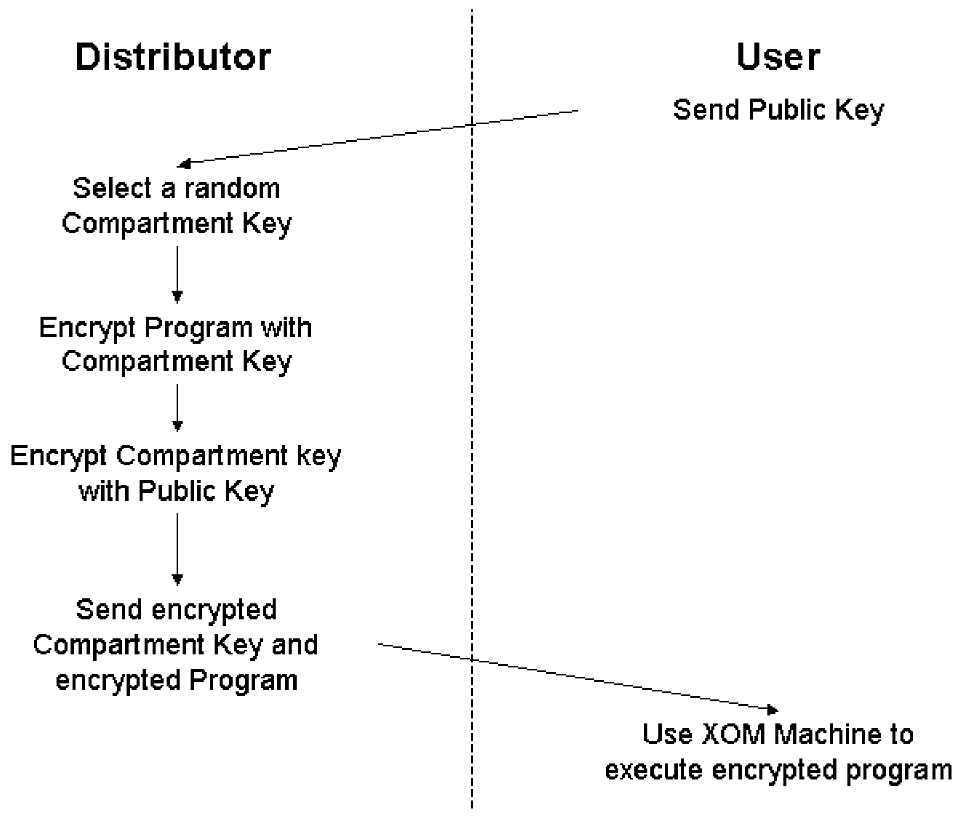
Distributor
User
Send Public Key
Select a random Compartment Key
Encrypt Program with Compartment Key
Encrypt Compartment key with Public Key
Send encrypted Compartment Key and encrypted Program
Use XOM Machine to execute encrypted program

Figure 3.3: Software Distribution in XOM. The software distributor uses asymmetric keys to transmit the compartment key to the processor. The processor can then recover the compartment key and use it to execute code inside the compartment.

up a potential security hole. An entry in the XOM Key Table could be freed and reallocated to another compartment, and it is possible for the new compartment to be assigned the same XOM ID as the previous compartment. The new compartment could thus gain illegitimate access to data belonging to the old compartment due to recycling of the XOM ID value. To ensure that the new compartment doesn't gain unauthorized access to data in the previous compartment's data, the XOM machine clears all registers that are tagged with a XOM ID before it can be allocated to a new compartment.

XOM affects the commercial model for software distribution. For a given program, a compartment key must be encrypted specifically for each processor. As a result, the distributer will typically encrypt all copies of the program with a single compartment key and distribute the encrypted copies. When a customer wishes to purchase the software, as part of the installation process, she contacts the software distributor to pay for or register the software. At this point, the distributor will return a compartment key specifically encrypted for the customer's processor after he is satisfied that the customer has paid for the software.

One issue where key distribution is a bit more complicated is if multiple CPUs execute the same binary. This is the case in systems with multiple processors. Multiprocessor systems have several CPUs that share a common pool of physical memory and are managed by a single operating system image. As a result, each CPU has to be able execute the same encrypted binaries. However, each XOM CPU has a different master secret. To allow all CPUs to run the same binaries, the compartment key is encrypted separately for each CPU and software support is added so when a thread wishes to allocate a XOM Key Table Entry, it allocates it with the compartment key encrypted for the CPU it is running on.

XOM also affects the software development process. Software developers use debugging tools inspect or even alter the state of running programs during development. XOM would prevent any of these debuggers from working since the processor is unable to differentiate between a valid user trying to debug a program and a malicious attacker trying to use a debugger to extract secrets from a program. The solution is to note that the developer should know the value of the compartment key. With this knowledge she may decrypt and alter values in memory. In addition, with knowledge of the compartment key, she can construct a program that can inspect and alter the register state of the program being debugged. This can be done with a technique akin to that used to create user-defined signal handlers as explained in Section 4.4.3

3.1.4 Security

The Abstract XOM Machine has a number of useful capabilities at this point. It can support the execution of programs protected in private compartments. It also allows the use of an untrusted operating system to manage resources for such programs. Next, we will examine some common attack strategies that an adversary may employ and see how the XOM machine defends against them.

Simply encrypting data in memory is not sufficient to make it tamper-resistant. Encryption only ensures the secrecy of the data in memory, but does not guarantee its integrity — an adversary can still alter values randomly in memory. To address this, the XOM processor adds a MAC to every piece of data that is written to memory and verifies this MAC for both data and instructions that are read from memory as mentioned in Section 3.1.1.[1] The MAC is then stored to memory at the same time the cipher text is. Taking a MAC of the data prevents an adversary from substituting tampered data in the place of real data. This attack is referred to as a *spoofing attack*. Because the adversary cannot create a valid MAC, she cannot forge a valid piece of data in memory.

A slightly more complex attack is one where the adversary does not need to create a fake cipher text and MAC. Instead, she simply copies both the cipher text and the MAC from one memory address to another. We refer to this as a *splicing attack*. A splicing attack is countered by including the virtual address along with the data in the pre-image[2] of the MAC. Thus, when verifying the hash, the XOM machine can ensure that the data it is being loaded from the same virtual address it was originally written to. Since the virtual address must be included in the pre-image for the MAC calculation, the caches must be extended to include the virtual address of the data in each cache line.

Still another attack exists for a determined adversary. This attack involves the adversary recording cipher text and MAC values at a point in the execution of a program and then "replaying" them at a later point in the execution of the same program. This attack is aptly named a *replay attack*. An adversarial operating system is capable of performing this attack on both register and memory values. To replay a register value, the adversarial operating system interrupts a running process and saves the register state using the `save register` instruction. The adversary than restores the process state and restarts the process. At a later time, the adversarial operating system interrupts the process again, but instead of restoring the register values from the second interruption, it restores the values from the first interruption. When the process restarts, it will be using the replayed register values. To defend against such an attack, the XOM machine uses a key other than the

[1]If desired, a separate key could be used for the MAC, and everything would still follow, but we do not discuss that here.

[2]The pre-image is the value that is used in the hash calculation.

compartment key for encryption in the `save register` and `restore register` instructions. This key, called the *register key*, is regenerated every time a particular XOM compartment is interrupted. As a result, the register key that is used to save the register at the time of the first interrupt, will have been destroyed and regenerated when the adversary tries to restore the state from the second interrupt. Thus, trying to restore the stale value will result in an exception.

Instead of trying to replay values in registers, an adversarial operating system may try to replay data in memory. To do this, the operating system records cipher texts and MACs in memory and then overwrites the same location at a later time with the old cipher text and MACs. To defeat this attack the application keeps a hash for a region of memory in one of the registers. To replay this region, the adversary must also be able to replay the hash kept in the register. However, the regenerating register key will protect the register from replay, thus defeating the memory replay attack. The drawback with this approach is that every time a value in the region changes, the hash kept in the register must be updated. If the region of memory is large, or if the values in this region change frequently, this results in a large overhead as the entire region must be read to update or verify the hash. The performance impact can be mitigated by using Merkle trees to perform memory authentication. For now, we will defer the discuss Merkle trees to Section 5.2.

3.2 Virtual Machine Implementation

A XOM machine can be implemented as a virtual machine with a minimum amount of hardware. Such a virtual machine would operate between the operating system and the hardware. The XOM virtual machine takes the underlying hardware and presents a XOM Abstract Machine image to the operating system. However, a virtual machine that uses emulation often suffers slow downs on the order of 3-13 times [64]. The performance will be substantially improved with the implementation of a *XOM Virtual Machine Monitor (XVMM)*. With a virtual machine monitor, software is compiled for the underlying hardware, and runs natively on the underlying hardware for the most part, with the processor hardware invoking the virtual machine monitor only when certain events occur. Virtual machines have been implemented in the past without additional hardware [8, 24, 63]. However, supporting the security features of XOM in a virtual machine monitor does require a small amount of extra hardware.

The main hardware additions to the CPU include special microcode that stores the private key, private on-chip memory, the ability to trap on instruction cache misses and a special privileged mode under which the XVMM runs. The actual XVMM could be implemented in either software or in microcode. Software implementations must be authenticated by a secure booting mechanism such as those described in [37, 61, 62]. Either way, the XVMM executes as a trusted, authorized, and privileged program. There are special hardware facilities that only the XVMM can access, such as the private key, secure on-chip memory and the revectoring of certain interrupts. This is why the XVMM must run at a privilege level higher than that of the untrusted operating system.

A XVMM implementation also requires the hardware to invoke it on certain events. When a program in a XOM compartment misses in the instruction cache, the XOM machine must fetch the required instruction from memory. The XVMM must be invoked by the hardware on instruction cache misses so that it may decrypt the data coming from memory and place it in the cache. In this way, the XVMM can use the instruction cache to hold decrypted instructions. This is possible because *all* instructions in a compartment must be encrypted. However, an XVMM cannot store data in plain text in the data cache. Because compartments have the ability to choose whether data is stored to the compartment, via the `secure store` instruction, or stored to the shared compartment using a regular store, the XVMM would have no way of telling what compartment data in the caches is in, without additional hardware in the data cache.

An XVMM requires the processor to be configured to trap on instruction cache misses so that the XVMM may decrypt data and instructions that are coming from memory. An *informed memory operation* [30] is a mechanism that interrupts the processor on all cache misses, and can be used to implement the control transfer required for our XVMM. The XVMM configures the informed memory operation to transfer control to code specific to the memory operation that missed. This code will then perform the required decryption to recover the instructions for execution.

Data in the machine (such as caches and registers) must be tagged in some way to implement compartment access control. The obvious solution is to add a hardware tag to each unit of hardware storage that requires one. However, this additional hardware is not strictly needed by an XVMM. Instead of explicitly adding hardware tags, the XVMM could simply remove the compartment data from the machine every time the program leaves a compartment (either due to an explicit `exit xom` instruction, or implicitly due to an interrupt). This is done by flushing the instruction cache and clearing all registers.

Flushing the instruction cache ensures that instructions that have been decrypted and placed in the cache are executed after the program leaves its compartment. Because the instruction cache contains no modified data, clearing the contents is a straightforward operation. However, clearing the registers requires support from the XVMM. A *shadow register file* is needed to hold data required for XOM instructions such as `move to shared` and `move from shared`, as well as register save instructions `save register` and `restore register`. The XVMM maintains a set of shadow registers for the each compartment in the XOM Key Table. These are managed entirely by the XVMM, requiring no special hardware support. The shadow registers store the value as well as an ownership bit indicating whether the register is in the private or shared compartment. The `move to shared` and `move from shared` instructions cause the XVMM to update the ownership state in the shadow registers from owned by the private compartment to owned by the shared compartment. When a compartment is interrupted, the XVMM moves data that is owned by the compartment from the registers into the shadow registers, thus preventing illegal access by the operating system's interrupt handler. Later, when the operating system restarts the compartment, the XVMM moves the data from the shadow registers back to the actual registers.

The limitation of this model is that the XVMM cannot unambiguously differentiate between an operating system that legitimately alters the register state of a compartment, from one that does so maliciously. For example, the operating system may interrupt a XOM program running in a compartment, and store the register state of the compartment using `register save` instructions. However, it might not restore a certain register when it restarts the compartment, but may instead leave a value of its choosing in that register. The XVMM cannot safely restart the process in this case since the unrestored register may hold tampered data, which could be used by the compartment code if restarted. A conservative

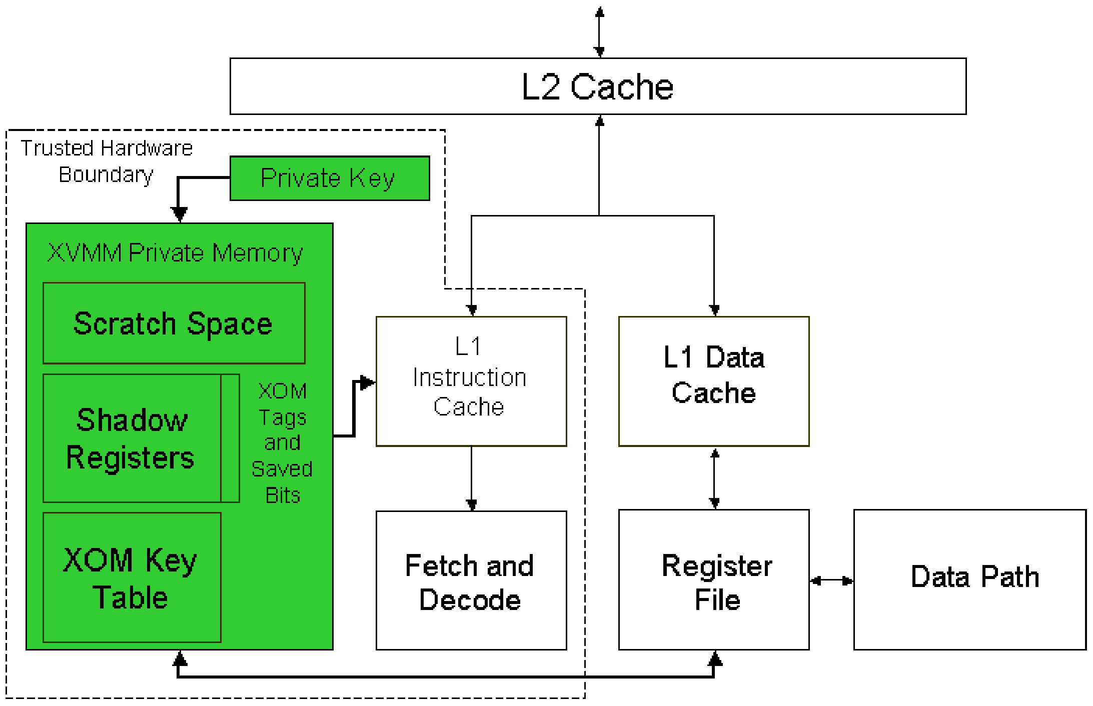

Figure 3.4: The XOM Virtual Machine Monitor (XVMM). The XVMM requires some additional hardware in the form of storage for the masters secret (private key) and private memory for the XVMM to use.

solution would be for the XVMM to refuse to restart the compartment, until the operating system restores the state of all compartment registers using the `restore register` instruction. However, this would prevent the operating system from returning data from after an exception, as it may do after a system call. The solution to this is to note that during an asynchronous interrupt, the operating system has no reason to alter the register state of the interrupted compartment. However, on a synchronous interrupt such as a system call, the compartment knows a priori, that a certain register will have a return value in it. As a result, the burden is placed on the compartment to make sure the result register has had its ownership changed to shared with the `move to shared` instruction before a system call. Otherwise, the operating system will not be able to set the return value. This burden on the application can be removed with additional hardware as we see later.

The additional hardware and structure of the XVMM is summarized in Figure 3.4. The

XVMM implements the instructions as follows:

`enter xom`: The program indicates an address where the XVMM will find the program's encrypted compartment key. The XVMM loads the encrypted compartment key and performs an asymmetric decryption to recover the compartment key. The XVMM then stores the compartment key into the XOM key table, assigns a XOM ID, and notes that this is the "currently active XOM ID". Instructions after an `enter xom` must be encrypted and accompanied by a valid MAC, otherwise the XVMM will not load them into the instruction cache for execution. The XVMM initially assumes all registers are in the shared compartment after an `enter xom`. The program must indicate to the XVMM which registers it will use to store private data with the `move from shared` instruction.

The XVMM registers a handler for cache miss events so that instruction cache misses incurred during the execution of XOM code will be correctly vectored to it. Similarly, it also re-vectors all CPU exceptions and interrupts to itself. If the compartment takes an exception that must be delivered to the operating system, the XVMM copies all registers that are in the private compartment into the set of shadow registers corresponding to that compartment. Naturally, if the compartment was active, the program counter is one of the registers that the XVMM saves. Finally, before execution of secure code, the instruction cache is flushed to clear out any instructions that are not in the compartment. Later, when the operating system returns execution to the user process, the XVMM must again be invoked. It restores the private registers from the shadow registers for the compartment, but leaves the registers in the shared compartment untouched. This allows the operating system to modify some user registers in order to to handle events such as system calls.

`exit xom`: The XVMM unregisters the handler for cache miss faults and restores handlers for all CPU interrupts and exceptions. The contents of the instruction cache are flushed.

`secure store`: The XVMM ensures that the register is marked as private in the shadow register file. Then it encrypts the register and calculates a hash. Both the register and hash are stored to memory.

`secure load`: The XVMM decrypts the value from memory and verifies it against the hash. If this operation succeeds, the value has not been tampered with and is written into the register. The status of the register in the shadow register file is set to private.

`move to shared`: The XVMM verifies that its shadow registers indicate that the register is currently in the private compartment. If it is not, the XVMM raises an exception. Otherwise, it tags the shadow register as shared.

`move from shared`: The XVMM checks that the register is currently marked as in the shared compartment in its shadow registers. If so, it marks the register as being in the private compartment.

`save register`: When the compartment is interrupted, the XVMM moves all values marked private in the shadow register file from the architectural registers into the shadow registers. However, the XVMM still must note which registers were saved by the operating system so it knows which ones to restore when restarting that compartment. To do this, it keeps another bit, called the *saved bit* for each register in the shadow register file the XVMM maintains for each compartment. On an interrupt, the XVMM clears all the saved bits in the shadow register file of the compartment. When the operating system executes a `save register` instruction, the XVMM sets the saved bit for the corresponding register in the shadow register file.

`restore register`: When the operating system uses the `restore register` instruction to restore a register value, the XVMM clears the saved bit in the shadow register file, indicating that the register is to be restored when the compartment is restarted. When the operating system restarts the XOM program, the XVMM checks that all saved bits are cleared in the shadow register file. If any are still set, the XVMM throws an exception. This prevents the operating system from injecting data into the compartment. If all saved bits are cleared, the XVMM copies all private registers from the shadow register file to the architectural registers and restarts the compartment. Any registers not marked private in the shadow register file are left unaltered.

The XVMM trades off performance for a simpler hardware implementation where most of the complexity is moved to software. The additional hardware requirements are secure storage for the private key and some on-chip memory that the XVMM can use for its private data structures.

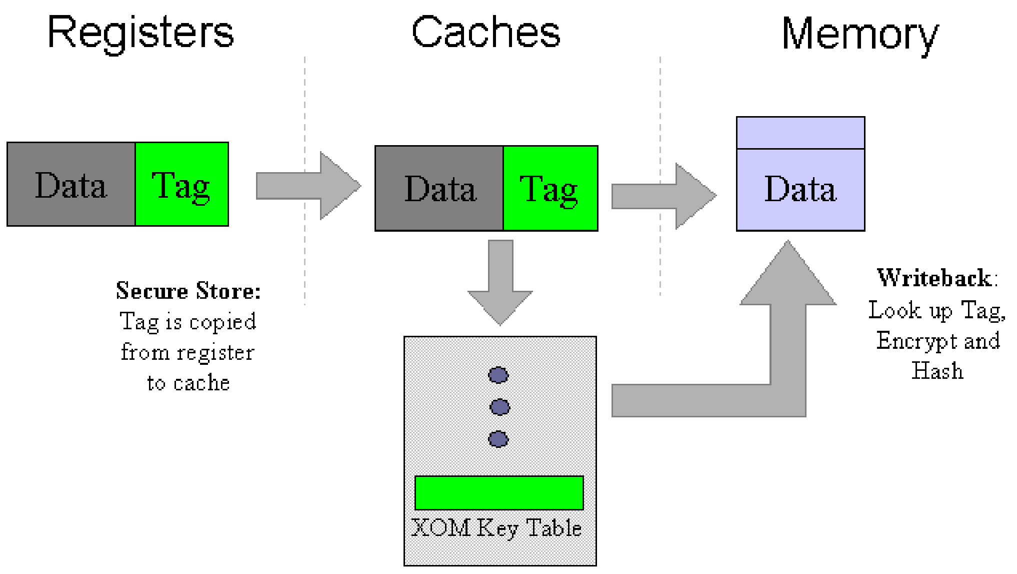
Registers
Caches
Memory
Data
Tag
Data
Tag
Data
Secure Store: Tag is copied from register to cache
Writeback: Look up Tag, Encrypt and Hash
XOM Key Table

Figure 3.5: Memory Support for Secure Store. Data is first stored into the cache. On cache eviction, the XOM Cache Ownership Tag is used as an index into the Compartment Key Table. The appropriate compartment key is then used to encrypt and MAC the data.

3.3 A Hardware Implementation of a XOM Machine

The XVMM described in the previous section suffers from three main performance penalties which can be mitigated with extra hardware. The first is that the XVMM is unable to cache compartment data, only instructions. In addition, the XOM machine must flush the instruction cache every time there is a trap. Both problems can be fixed by adding XOM ownership tags to the on-chip caches. The second source of overhead is the fact that the cryptographic operations are handled in software by the XVMM. Selecting fast algorithms and adding hardware acceleration reduces the cost of these operations. The final source of overhead is that the instructions that XOM provides are interpreted as opposed to implemented in hardware. A full hardware implementation would remove the need to trap to the XVMM as those operations would be handled directly in hardware.

The performance overhead of the memory encryption and MAC computation can be

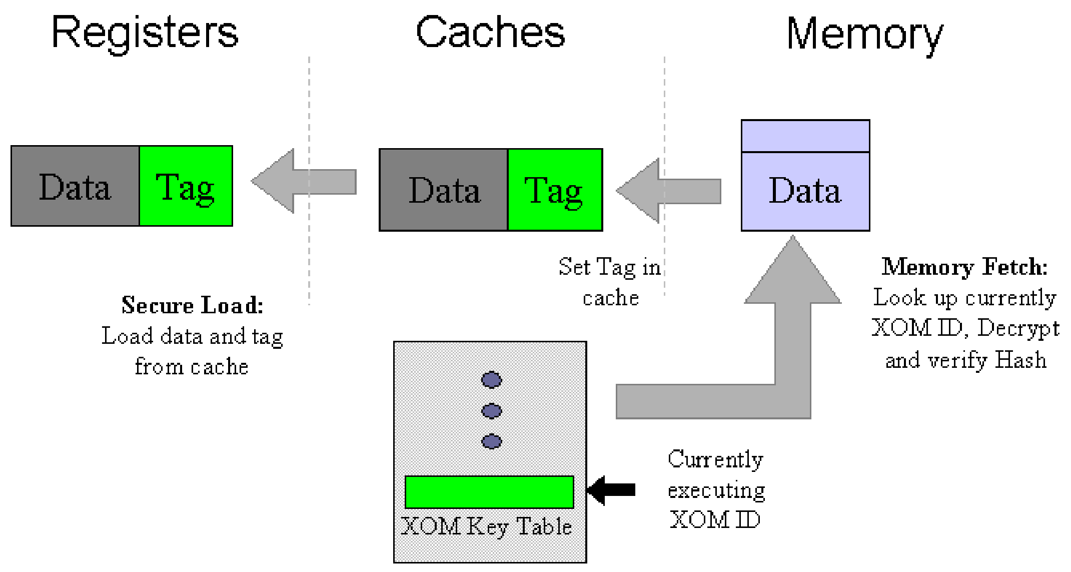
Registers
Caches
Memory
Data
Tag
Data
Tag
Data
Secure Load:
Load data and tag from cache
Set Tag in cache
Memory Fetch:
Look up currently XOM ID, Decrypt and verify Hash
XOM Key Table
Currently executing XOM ID

Figure 3.6: Memory Support for Secure Load. On a cache hit, the data is simply loaded from the cache. If the load misses, the cipher text and MAC is decrypted, verified and placed in the cache.

reduced by adding XOM compartment tags to the on-chip caches. Tags allow the processor to track the ownership of values in the caches, thus enabling the machine to cache data valves, and removing the need to flush the instruction cache on traps. This allows the XOM processor to utilize the on-chip data cache to defer encryption until the data is flushed to memory and also means that data or instructions need only be decrypted once as they are loaded from memory into the cache. Caching effectively reduces the number of cryptographic operations, meaning the program need only pay the cost when it misses in the cache. The overhead of the cryptographic operations are also reduced by tagging a cache line rather than a single data word. This larger block size reduces initialization overhead of the operations and also reduces the space overhead for the MACs as each MAC is protecting a larger section of memory.

Tagging cache lines and not individual words also introduces a security complication. There is a XOM ID tag for every register, and a XOM ID tag for every cache line. But a cache line typically contains several words, so as a result, the granularity of compartment ownership in the registers is different from that in the caches. This poses a security threat that allows an adversary to take ownership of data that does not belong to it. If the XOM machine naively sets the XOM ID to the last compartment that writes the cache line, this would allow an adversary to take ownership of all the data in a cache line by writing to a single word within that line. This attack is illustrated in Figure 3.7. Clearing the cache line with some value (such as zeros) before allowing the new owner to write to the line does

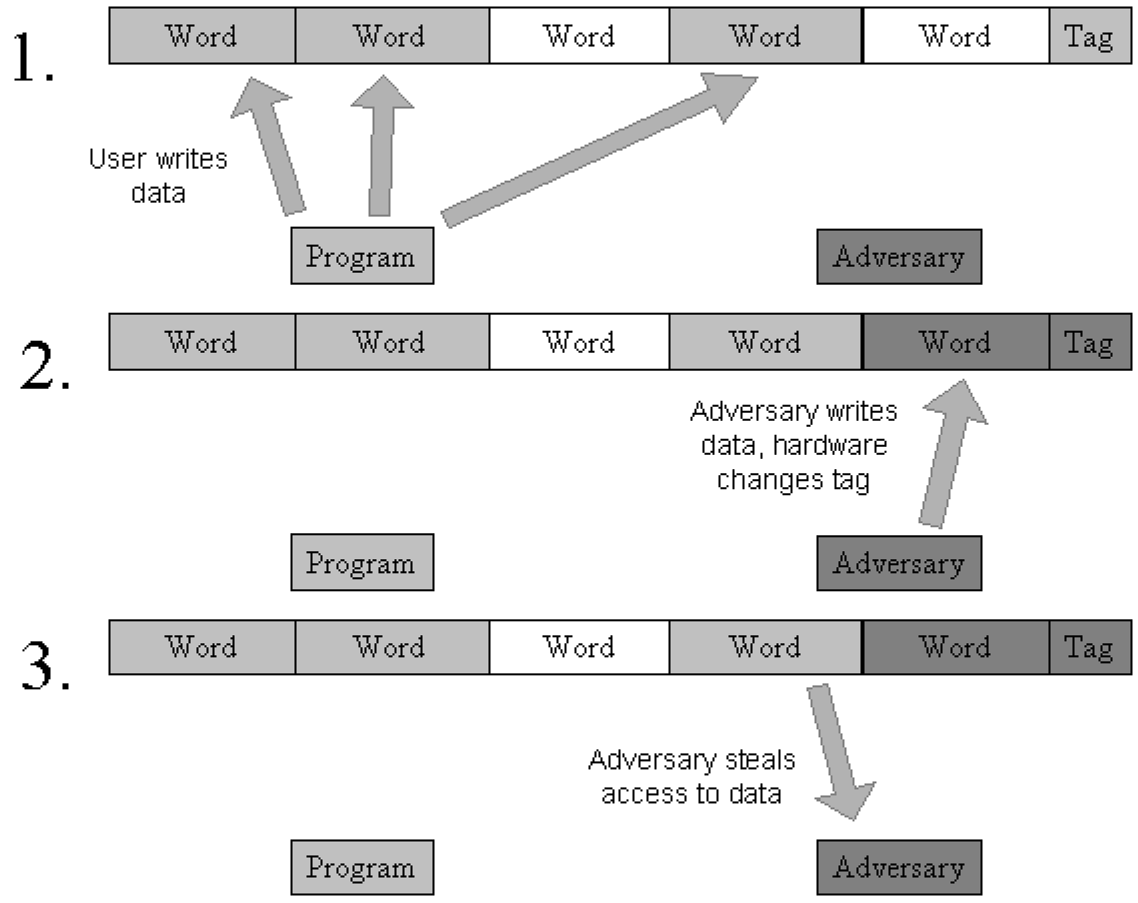

Figure 3.7: Valid Bits in the Caches. A clever adversary exploits that difference in protection granularity between registers and cache lines. The attacker writes to a single word to take ownership of all words in the cache line.

not solve the problem, as this would allow an adversary to arbitrarily inject values (such as zeros) into the compartment of the previous owner. The solution requires additional hardware to be added in the form of per-word valid bits in the cache. Valid bits are added to each cache line to indicate which words in the line are valid and which are not. When the XOM ID of a cache line changes, all valid bits are set to invalid, and only become valid when that particular word is written by the new owner. Data which is set to invalid in this way is lost. This implies that for correct operation, two compartments *cannot* share a cache line as each time the line changes ownership, the previous owner's data will be clobbered. When a cache line is flushed to memory, the valid bit information must also be stored along with the MAC and then returned when the cache line is loaded back into the cache, otherwise the adversary could get rid of the valid bit state by simply causing the line to be evicted.

Adding tags to the caches also introduces another problem. 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 XOM ID 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 XOM ID for the child. Because there are two different threads of execution, we need two different register keys (and two different XOM IDs). Register data is tagged with XOM IDs, which distinguish between the compartment of 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 XOM ID tag, which is different for each process; instead a different set of tags, cache ownership tags, are used in the caches.

To support multiple instances of the same program using compartments, the XOM Key Table 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 the XOM ID register, which refers to the register key of the currently executing XOM ID. 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*,

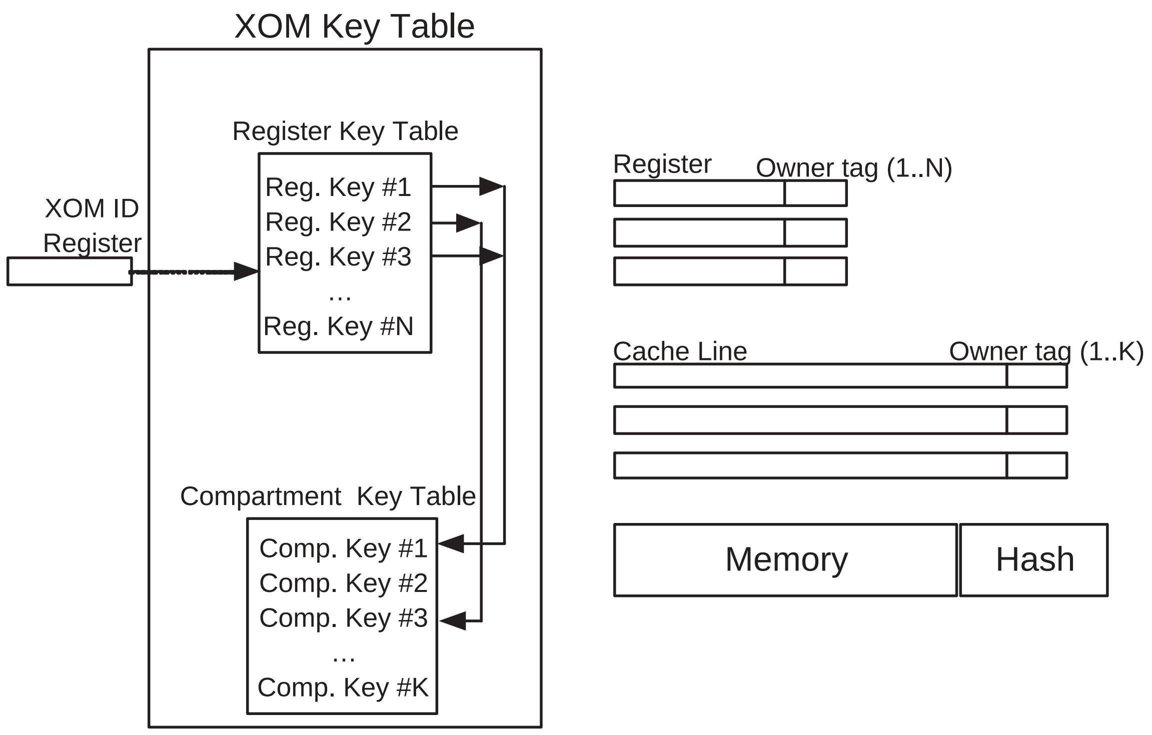

Figure 3.8: XOM Key Table Design
XOM Key Table Design. The XOM ID register refers to the Register Key entry for the currently executing XOM process. The ownership tags on the registers are used by XOM to encrypt and decrypt register contents under program control, using specialized machine instructions described in Table 3.1. 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.

and they hold the XOM ID of the compartment that owns the register. Likewise, cache lines are tagged with the indices into the Compartment Key Table. These tags are called the *cache ownership tags*. 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. This architecture allows several XOM IDs to map to the same cache ownership tag value, and thus use the same compartment key in memory. These structures are shown in Figure 3.8.

The performance overhead introduced by XOM comes primarily from the time to perform the cryptographic operations. Two types of cryptographic operations are used in the XOM machine. Asymmetric ciphers are used to protect the compartment key during transmission from the software producer to the execution processor, while symmetric ciphers are used to protect the actual compartments in memory. When a compartment is about to execute, an `enter xom` instruction causes an asymmetric operation in the XOM machine to recover the compartment key and enter it in the XOM key table. This means that every entry into a compartment incurs an asymmetric operation. RSA, a common asymmetric cipher, takes on the order of 300 million cycles to decrypt a value with a key-length of 4096 bits, according to the OpenSSL [48] crypto-library implementation. However, entry and exit into a compartment is an infrequent event, so the cost of the initial asymmetric operation will not have a large impact on the overall execution time of the XOM program. As a result, the benefit of adding hardware to accelerate asymmetric operations does not justify the cost. However, since the decrypted compartment keys are stored in the Compartment Key Table, it would be possible to cache the compartment key for later use. Programs that enter and exit their compartment frequently may benefit from caching the decrypted compartment keys. This requires some further extensions to the instruction set and will be discussed in Section 4.2.1.

On the other hand, symmetric operations are required to decrypt instructions on every cache miss. The symmetric cipher Rijndael [13] is a common cipher in use at the time of writing. Optimized software implementations can perform an encryption or decryption in Rijndael on the order of 300-1400 cycles. However, even with a optimistic cache miss rate of 1%, this would still have an average overhead of 3-14 cycles per memory operation. This motivates us to examine hardware acceleration for the symmetric operations in memory. There exist custom hardware implementations that can encrypt or decrypt a block in approximately 10 cycles [45]. Rijndael is an iterative algorithm which consists of 12-14 iterations of a basic “round” operation. In addition, since the encryption block, an L2 cache line, is larger than the basic block of the Rijndael cipher, an appropriate encryption mode must be selected. For efficiency, a mode which allows parallel or random access decryption, such as Output Feedback Mode (OFB) or Counter Mode (CTR) [41] is desirable. This allows for faster decryption with additional hardware, as well as optimizations such as *critical word first*. Thus, with a conservative latency estimate of 14 cycles, a pipelined implementation that provides a new encrypted or decrypted value every cycle is conceivable. Existing implementations of the Rijndael algorithm consists of bit permutations and lookup tables and use anywhere between 60 K to 280 K gates [45], which is an acceptable cost for a modern processor.

Each cache line requires a MAC to be generated to check for tampering in memory. To be effective, the MAC must be of sufficient length to guarantee a low probability of collision. In practice, most security systems rely on a MAC of at least 128 bits. The MAC algorithm can be made significantly faster by noting that the hash is going to be encrypted before being stored in memory. As a result, the XOM machine need not necessarily employ a hash with a strong one-way property since the adversary would need to break the encryption to exploit any reversibility in the hash. As a result, a faster hash algorithm such as CRC, may be used to take the hash for the MAC. More custom hardware could be added for the hash (which would be smaller than the symmetric cryptography hardware since it does not need to be pipelined). For cache evictions, the hash calculated for the MAC can happen in parallel with encryption. Unfortunately, this is not the case for cache fills since the hash can only be calculated from the plain text value. An alternative is to use a cipher that also checks for integrity such as Offset Codebook (OCB) [51]. This completely removes the need to calculate a MAC, but makes the encryption and decryption operation slower.

Another optimization deals with the latency of verifying the hash value on a load. Since the hash depends on the entire line, a simple implementation would delay returning any data to the processor until the entire line was fetched onto the processor. To eliminate this additional overhead the XOM hardware returns the requested word first, and speculatively starts the processor when the requested word is decoded. If the hash does not verify the XOM thread will abort, and does not need to be restarted, obviating the need for a precise exception in this case. All that is needed, is to ensure is that any operations that allow information to leak out of the machine such as stores cause the machine to stall until the check is complete.

Finally, adding XOM register ownership tags directly to the registers removes the need for shadow registers. The tags on the caches indicate explicitly which compartment can

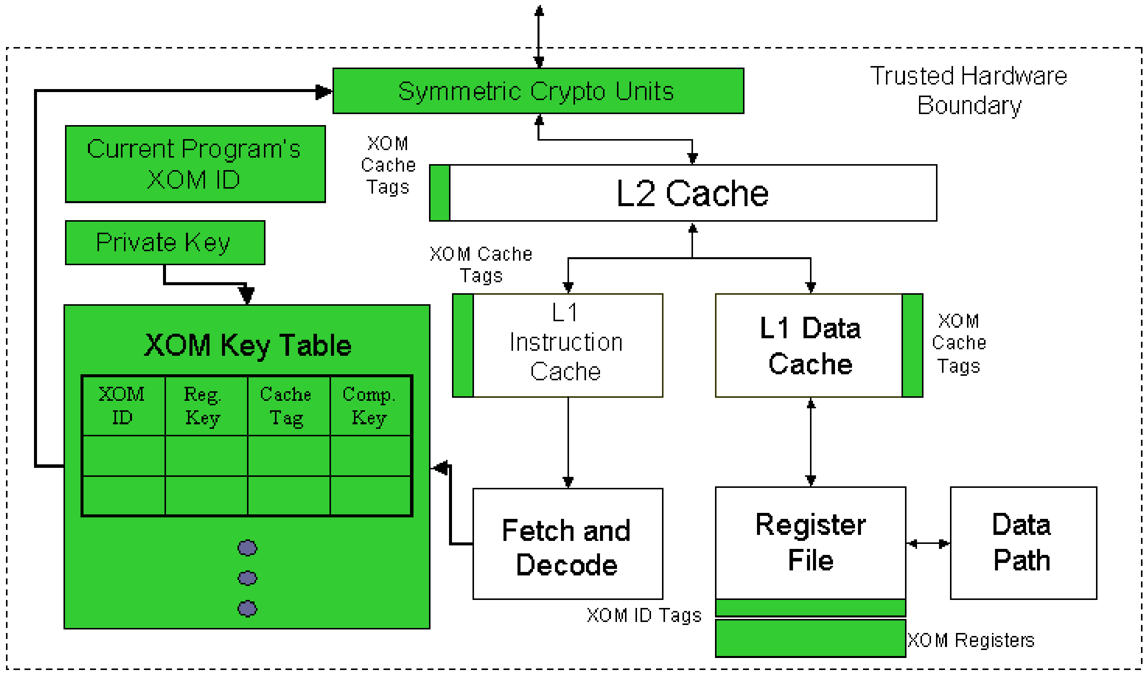

Figure 3.9: A XOM Machine Implemented in Hardware. A full hardware implementation of XOM may include tags in all on-chip caches and registers, as well as cryptographic accelerators on the memory bus.

access that register. This simplifies the actions the XOM machine must take during a trap. In addition, the requirement that the register used for a system call return value be in the shared compartment before the system call is also removed. The explicit tag on the register checks accesses that occur after the compartment is restored, so the XOM processor need not perform a saved bits check as it does with an XVMM. In addition, the tags allow the instructions such as `move to shared`, `move from shared`, `save register`, and `restore register` to be implemented in hardware, removing the need to call into the XVMM on these instructions.

A XOM machine can now be implemented by the bare hardware of the processor as shown in Figure 3.9. The machine maintains a register that holds the currently executing program's XOM ID. It uses this to control access to all registers and cache lines. The XOM key table stores mappings between XOM IDs, register keys, XOM cache tags and compartment keys. XOM instructions in the previously implemented by the XVMM can now be implemented directly into the instruction set architecture of the machine:

`enter xom`: This operation remains essentially unchanged from the XVMM. Asymmetric operations are still handled in software either by microcode or by a small VMM loaded at boot. The XOM processor assumes all registers are initially in the shared compartment and marks their XOM ID tags as such. However, when a trap occurs, the XOM processor only updates the XOM ID of the currently executing program to be shared.

`exit xom`: Because the values in the caches and registers are protected by XOM tags, the processor does not need to flush the caches. It simply changes the XOM ID of the currently executing program to shared.

`secure store`: The XOM processor verifies that the source register's tag matches the XOM ID of the program executing the store. If these tests pass, the processor stores the register value to the cache. The XOM ID is translated to a XOM cache ownership tag, which is written to the cache line. This process is illustrated in Figure 3.5.

`secure load`: If the line hits in the cache, it compares the XOM cache tag of the cache line against the XOM ID of the program. If a mapping in the XOM key table agrees with the two values, both the value in the cache and matching XOM ID are written to the register. If the load misses in the cache, then the line is filled from memory. This is illustrated in Figure 3.6.

`move to shared`: The XOM processor ensures that the program is not executing with the XOM ID of shared. It changes the XOM ID tag on the register to shared.

`move from shared`: The XOM processor checks that the XOM ID of the program is not shared. It then sets the XOM ID tag of the register to the XOM ID of the program.

`save register`: The XOM processor uses the register key associated with the XOM ID tag on the register to encrypt and MAC the register value. This value will typically be larger than the plain text value since it will include a MAC. While saving this large value to memory could be implemented in several cycles, it is easier to place the larger value in several registers and have the operating system save each value individually. Instead of the single `save register` instruction, we change the architecture to implement an `xenc` instruction that will encrypt and MAC the register contents with the register key and place them in four special *XOM registers*. These can be accessed via the

Parameter	**Value**
Master Secret	4096 bits
Compartment Keys	128 bits
MAC Hashes	128 bits
L1 Cache Line	32 bytes
L1 Cache Size	16 Kbytes
L2 Cache Size	128 Kbytes
L2 Cache Line	128 bytes
Registers	64 bits
Time for each instruction	1 cycle
Time to decrypt a cache line	15 cycles
Time to access memory	150 cycles
Time for asymmetric key decryption	400,000 cycles

Table 3.2: XOM Simulator Parameters. These parameters are used in our SimOS based implementation of a XOM hardware simulator.

`xsave` instruction, which takes an index pointing to one of the four registers and saves it to a memory location.

`restore register`: 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 MAC, and return the value to a general-purpose register.

If any of the security checks above fail, the XOM processor throws an exception, switches the currently executing program's XOM ID to shared, and traps to the appropriately registered exception handler. This halts the execution of any private compartment code.

3.4 The XOM Hardware Simulator

A cycle accurate, detailed hardware simulator of a system with a XOM processor was built on top of the SimOS hardware simulator [28]. SimOS not only models the processor, but the memory system (including the memory bus), and disk. A processor model in SimOS, called Mipsy, is extended to contain XOM capabilities as explained in this chapter. Mipsy

Abstract Instruction	**Actual Instruction Implementation**
`enter xom`	`xalloc $rt,offset($base)`
	`xentr $rt,$rd`
`exit xom`	`xfree $rt`
	`xrclm $rt`
	`xexit $rt`
`secure store`	`xsd $rt,offset($base)`
`secure load`	`xld $rt,offset($base)`
`save register`	`xgetid $rt,$rd`
	`xenc $rt,$rd`
	`xsave $rt,offset($base)`
`restore register`	`xrstr $rt,offset($base)`
	`xdec $rt,$rd`
`move to shared`	`xmvtn $rt`
`move from shared`	`xmvfn $rt`

Table 3.3: Simulated XOM Instructions. Implementation of XOM Architecture primitives on top of our MIPS based architecture. Because of restrictions of the MIPS ISA, as well as requirements by the operating system, some primitives specified as single instructions in the XOM Abstract Machine, are actually implemented as several instructions in the simulator.

models an in-order processor where every instruction completes in one cycle unless stalled by a cache miss. Mipsy's instruction set is very similar to that of the MIPS R10000 Processor [27]. The modifications to the simulator closely follow the full hardware implementation of a XOM processor. A master secret and XOM Key Table are added to the processor model, along with a register to hold the currently active XOM ID. Tags are added to the on-chip caches and registers and cryptographic functions are added to the memory interface between the caches and memory system. Extra processing is performed by the processor each time an exception occurs and a new exception is created, which is thrown when an access violation occurs. Finally the additional instructions given in Table 3.1 are added to the instruction set. These follow the format given by the R10000 instruction set and are summarized in Table 3.3. Because of restrictions in the base MIPS instruction set architecture, as well as requirements in the operating system, some of the single instructions in the XOM Abstract Machine are implemented as several instructions in our simulator. Since part of the motivation for breaking down the instructions into several smaller ones is motivated by the operating system, details about what each of the simulated instructions does will be given in the next chapter, which addresses operating system issues. In our implementation, we use the parameters given in in Table 3.2. Other parameters such as CPU speed, memory bus bandwidth and latency, memory size, etc. are configurable.

3.5 Maintenance Issues

In this section we address two issues concerning how to maintain systems using XOM processors. The first discusses how to migrate software from one XOM processor to another one. This may happen legitimately when a customer buys new hardware, but would like to continue running the same software for example. The other issue concerns the related case where a processor is destroyed and the master key is not recoverable.

3.5.1 Processor Upgrade

Customers may wish to upgrade a processor to a newer model, but continue to use all trusted software purchased for the previous processor. To support this, there must be some way to migrate the master secret from the old processor to the new one. To allow this, we propose two schemes: one which requires a central authority and one which is decentralized. Both require that that the master secret should be stored EEPROM or some form of rewriteable non-volatile RAM on the processor, and that a trusted program can be created to overwrite the master secret with a new value.

In the first scheme, a trusted central authority, such as the processor manufacturer is required. The customer, when wishing to upgrade to a new processor returns the old processor to the authority. The authority then provides the customer with a trusted program that she runs on her new processor. The trusted program contains the master secret of the old processor embedded in its encrypted binary. The compartment key of the program is specifically encrypted for the new processor, so it can only be run on the processor the customer has just purchased. When she runs the program, it overwrites the the master secret on the customer's new processor with that of the old processor. After that, the new processor can run all programs encrypted for the old processor.

In the second scheme, no trusted authority is required. Each processor has a trusted program on it. The old processor has a *sender* program and the new processor has a *receiver* program. The programs have the ability to read and overwrite master secrets. They use a simple protocol to transfer the secret from the old processor to the new one. The main goal is to make sure that there is no way for an attacker to tamper with the messages such that there will be two processors with the same master key.

1. Receiver randomly creates a symmetric session key and encrypts it with Sender's public key. She appends a message indicating that she wishes to transfer the master secret from the Sender to the her. Receiver signs the message with a secret signing key to ensure the message cannot be forged.

2. Sender decrypts the message and recovers the session key. After this, both programs will encrypt all messages with the session key. Sender then reads the master secret from the old processor and sends it to Receiver over the encrypted channel.

3. Receiver receives the new master secret, stores it, but does not use it to overwrite the master secret on Receiver's processor yet. Instead, she sends an acknowledgment back to the Sender indicating that the master secret has been received.

4. When Sender receives the acknowledgment, he overwrites the master secret with some random value, destroying it on the old processor. Sender then sends a "secret destroyed" message to Receiver.

5. Receiver receives the "secret destroyed" message and then commits the new master secret to her processor overwriting the current master secret. The new processor can now run all binaries originally encrypted for the old processor.

The only attacks a malicious adversary can mount are to either prevent the transfer from happening, or to cause the old processor's master secret to be destroyed and never committed to the new processor.

3.5.2 Processor Key Recovery

If a XOM processor is ever physically destroyed, lost or otherwise rendered inoperable, then there should be away to obtain a new processor with the same master secret so that all software purchased for the destroyed processor is still available. A scheme similar to the first scheme for processor upgrades in the previous Section may be used. The only caveat is that the customer should be able to prove that the old processor was destroyed. However, even if she does not, the most damage the she can cause is to obtain two processors with the same master secret.

Note that this solution requires a trusted third party to hold the master keys for all XOM processors. While certain key infrastructures have such single points of failure, such a structure is not desirable. The alternative is to require that the customer to have the compartment keys for all software re-encrypted for the replacement processor.

3.6 Related Work

XOM is by no means the only system which tries to protect software from tampering. In this section we will address various hardware and software approaches that have been tried in the past.

We will then explore trusted computing options which are being developed concurrently. The most notable of these are Microsoft's Palladium system, now called the Next-Generation Secure Computing Base, and the Trusted Computing Platform Alliance (TCPA).

3.6.1 Hardware Approaches

There are several related hardware systems that support functionality similar to XOM. In [21], [22], and [23] Gilmont et al. outline a method to use support in the memory management unit to support the use of encrypted memory. Like XOM, they used a hybrid scheme with a private-public key pair used to encrypt a symmetric key. In their performance study, they found that the overheads were modest. Their system differs from XOM in that they do not provide any secure method for the operating system to manage resources, and thus implicitly trust the operating system.

In 1997, Kuhn proposed a system entitled "TrustNo1", which has many features similar to XOM [35]. He also proposes the use of encrypted memory, but allows the use of an untrusted operating system, by using hardware support to encrypt the state of the interrupted process before allowing the operating system to save it. XOM uses the same method, but allows the saving and restoring of state at a register granularity. In addition, XOM provides protection against replay attacks and addresses the accompanying problem of having two instances of the same program using the same compartment key.

An alternative form of hardware support to XOM, is to encase an entire system of standard components in a tamper-resistant package. The IBM 4758 [32] contains a complete system, including a processor, memory, a PCI interface and a serial port. However, the processor is a 100Mhz 486 processor and is slow compared to the 1GHz or more processors it is paired with. As a result programming environment incurs significant overhead and communication between the compute processor and the coprocessor is akin to a Remote Procedure Call (RPC) model. Despite this, secure-coprocessor cards have experienced some degree of success. Bennett Yee has been responsible for the Dyad [62] and Sanctuary [65] projects which explore the protection of mobile code against tampering via the use of secure coprocessors.

3.6.2 Software Approaches

For the most part, software approaches have centered around the concept of software obfuscation. The goal is to find a transformation that can be applied to a program such that the transformation does not alter the working of the code, but it would be difficult for an adversary to reverse that transformation. This allows the program to hide secrets in its code. To detect tampering, the program performs checks on itself that are rendered invisible to the adversary via obfuscation. Many of these techniques involve compiler-level transformations such as altering control and data structures, inserting dead or irrelevant code, or using lookup tables [12]. These methods have become quite popular and some have even been commercialized at the time of this writing. However, their effectiveness is an open question as many still believe that general software obfuscators do not exist [6].

3.6.3 Trusted Computing

The Trusted Computing Platform Alliance (TCPA) [61], Microsoft's Next Generation Secure Computing Base (also known as Palladium) [17, 18] and Intel's LaGrande technology all aim to provide a platform where a remote party can establish some level of trust. The goal of these systems was originally to provide functionality for Digital Rights Management (DRM) such that content providers could securely distribute content on the Internet without fear of piracy. As a result, these architectures provide tamper-resistant and tamper-evident functions such that a remote party, such as the content distributor, can establish whether software, such as a movie player, has been tampered or altered on a customer's machine. We will now discuss TCPA and Palladium. Unfortunately at the time of writing, we are not aware of any documentation on LaGrande.

TCPA uses a secure Trusted Platform Module (TPM), which exists as an extra IC soldered onto the motherboard of the system. The TPM performs various functions such as pseudo-random number generation, key and data storage and certain cryptographic functions. In addition, the TPM certifies the boot process ensuring that only a certified operating system can be run. This allows applications to trust the operating system. Finally TCPA provides the ability for *attestation*. This allows the hardware to prove to a third party that a certain piece of software is running on the platform. TCPA's feature set is much smaller than that of XOM's. While TCPA checks the integrity of programs much like XOM, it does not allow program's to hide secrets in their binary code. In addition, its secure boot architecture verifies software at boot time, but is vulnerable to an adversary who is able to alter the contents of memory after that time.

On the other hand, Microsoft's Palladium takes a different approach. Palladium uses a security kernel, known as the *nexus* that will run at a privilege level below that of the operating system. This prevents it from being tampered by the operating system. Programs can communicate directly with the nexus to have it performs functions on behalf of programs, or even have part of the program execute inside the nexus. Palladium provides many of the same functions as TCPA, including *sealed storage*, a form of storage that is only accessible to programs with the correct credentials. It also allows the platform to attest as to what software is running on the platform. Unlike TCPA, Palladium provides *curtained memory*, which divides the physical memory on the machine into secure and insecure domains. Finally, Palladium will also provide secure input and output channels so that a compromised operating system will not be able to tamper with input from the user or output going to the user. XOM is able to provides the same attestation and sealed storage function as it does with TCPA. In addition, compartments provide functionality similar to that of curtained memory. XOM does not provide any functions to secure I/O, but would be compatible with the functionality that Palladium provides.

With the exception of the secure I/O provided by Palladium, XOM provides a superset of functionality of TCPA and Palladium, at the cost of more complex hardware. Both sealed storage and curtained memory are effectively implemented by the compartment architecture of XOM. For persistent storage across reboots, the XOM processor must be augmented with some non-volatile RAM much the same way TCPA and Palladium are. XOM provides attestation since programs have their integrity checked constantly during execution. In addition, a remote user can be assured that a program will only run on a certain piece of hardware since only that hardware will contain the correct master secret to execute that program. In general, XOM's memory encryption allows XOM to be resilient to an adversary who tampers with values in memory or on the memory bus, while both Palladium are vulnerable to such attacks.

3.7 Summary

This section explored the architecture of a XOM Abstract Machine that implements compartments via extensions to a generic processor architecture and instruction set. We showed that compartments can be implemented on chip via a tagging scheme that tracked the ownership of on-chip storage. While protected by tags, both data and instructions could be kept in plain text for efficiency. However, when utilizing memory or other storage where the integrity of tags could not be guaranteed, the XOM machine used cryptographic techniques in the form of ciphers and MACs to guarantee both the secrecy and integrity of such storage. This required that a compartment key and a regenerating register key be maintained for each compartment in a XOM Key Table structure. Finally, the compartment key must be transmitted from the software distributor to the customer's processor machine using asymmetric ciphers. This requires that a master secret in the form of a private key be added to the processor.

There are two implementation alternatives for the XOM Abstract Machine. In one alternative, we explore a virtual machine monitor implementation which can be implemented with a minimal amount of hardware additions to the base processor. While flexible and simpler, this implementation suffers from performance penalties. These are mainly due to its inability to fully utilize the on-chip caches to store compartment data in plain text, the length of time to perform cryptographic operations in software, and the overhead of interpreting XOM instructions and maintaining structures such as the XOM Key Table in software. These overheads are mitigated by a full hardware implementation that adds tags to the registers and caches of the machine, as well as adding hardware accelerators to perform the cryptographic operations.

Chapter 4

An Operating System for the XOM Architecture

The XOM architecture alters the traditional trust model that has existed in computer systems. Instead of applications trusting the operating system to safe guard them from tampering, applications now distrust the operating system and trust the hardware instead. This naturally changes the role of the operating system from both a root of trust and a resource manager, to just that of a resource manager, with the root of trust existing in the hardware.

We examine how this change in role affects the implementation of the operating system. As discussed in Chapter 3, the operating system must deal with user data and state differently. For example, to handle an interrupt, the operating system must use special instructions provided by the XOM machine to save user state in the registers. User data is encapsulated and protected by the hardware, but it is the operating system which determines where the data is to be stored. We will examine in more detail the changes that XOM requires in an operating system. We do this by porting the IRIX6.5 [55] operating system to the XOM architecture simulator described in Section 3.4, to create the operating system XOMOS.

The other concern is the effect XOM has on performance. The XOM hardware adds cryptographic delays to memory accesses. On the other hand, changes to the operating system also increases the latency of certain operations. In this chapter, we will also examine the performance implications XOM and XOMOS have, by measuring the performance of various applications and operating system micro-benchmarks.

4.1 Operating System Design Issues

The XOM architecture must satisfy two seemly conflicting requirements. It must protect program data from a malicious operating system, but must allow a non-malicious operating system to effectively manage resources amongst mutually untrusting programs. Since resource sharing is mostly done by the scheduler, the hardware must provide the same exception and interrupt functionality found in ordinary processors. This allows the operating system to limit the execution time of programs and interpose when programs access protected 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 in registers, it must use the special instructions provided by XOM that encrypt and MAC processor registers. When relocating data in memory, the operating system must also relocate the respective MACs.

XOMOScan be constructed by modifying a currently existing operating system. Since most of the modifications to the operating system deal with supporting new hardware, the higher level semantics of the underlying operating system, such as resource scheduling policies, user program interface and software architecture are not important. In practice, the design of XOMOS can be viewed more as a process of porting a standard operating system to a platform that supports secure execution. However, because the XOM architecture has some unique properties, this process requires solving some issues not found in ports to other architecture.

There is a significant amount of new hardware that XOMOS has to support. For applications to use this hardware, interfaces in the form of new system calls must be provided. In addition, the hardware prevents the operating system from reading some of the hardware state unless it uses a specific interface specified by the XOM architecture. The operating system must be modified to use this interface. These modifications fall into three categories:

- **Modifications for XOM Key Table support:** 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 and MACs:** When the operating system is managing system resources such as CPU time or memory, it must deal with encrypted data and the accompanying MACs. The MACs must also be stored and managed by the operating system as process data. The hardware and the operating system must work together to ensure that the hash values are saved properly.
- **Modifications for traditional operating system mechanisms:** Various features in a traditional operating system such as shared libraries, process creation, and user defined signal handlers must change because the operating system access to user data has been restricted.

We will now describe each of these modifications in more detail.

4.2 XOM Key Table Support

The XOM architecture specifies a XOM Key Table which is used to store both the compartment keys and register keys which are used to implement compartment. However, compartments are created by user applications, so the operating system must provide an interface through which user applications may manipulate entries in the XOM Key Table. This is implemented in XOMOS as a set of system calls which user applications may call to allocate and deallocate entries in the XOM Key Table.

Since the XOM Key Table is a finite hardware resource, it can limit the number of concurrent compartments in use by user applications. XOMOS circumvents this restriction by virtualizing the XOM Key Table entries.

4.2.1 XOM Key Table System Calls

The abstract XOM machine implements a single `enter xom` instruction to enter a compartment. 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.

An additional consideration is that `enter xom` and `exit xom` instructions may be executed by the programs themselves as unprivileged instructions. This makes a *XOM Transition*, the act of entering and exiting a compartment, efficient as it does not require kernel intervention. 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 make the operations of loading and unloading XOM Key Table entries separate from the operations of entering and exiting XOM compartments.

We split each of the `enter xom` and `exit xom` instructions into two smaller instructions. The `xalloc` and `xfree` 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 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`. Figure 4.1 illustrates this process of allocating an entry in the XOM Key Table and using it to enter a compartment.

Because `xalloc` and `xfree` access a limited hardware resource, they are privileged instructions, and are exported by XOMOS to user applications 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. It is important to note that even if the operating system executes `xalloc` on behalf of a user application, it cannot use the resulting XOM ID to execute code in that compartment since the operating system cannot forge a valid MAC without knowledge of the actual compartment key.

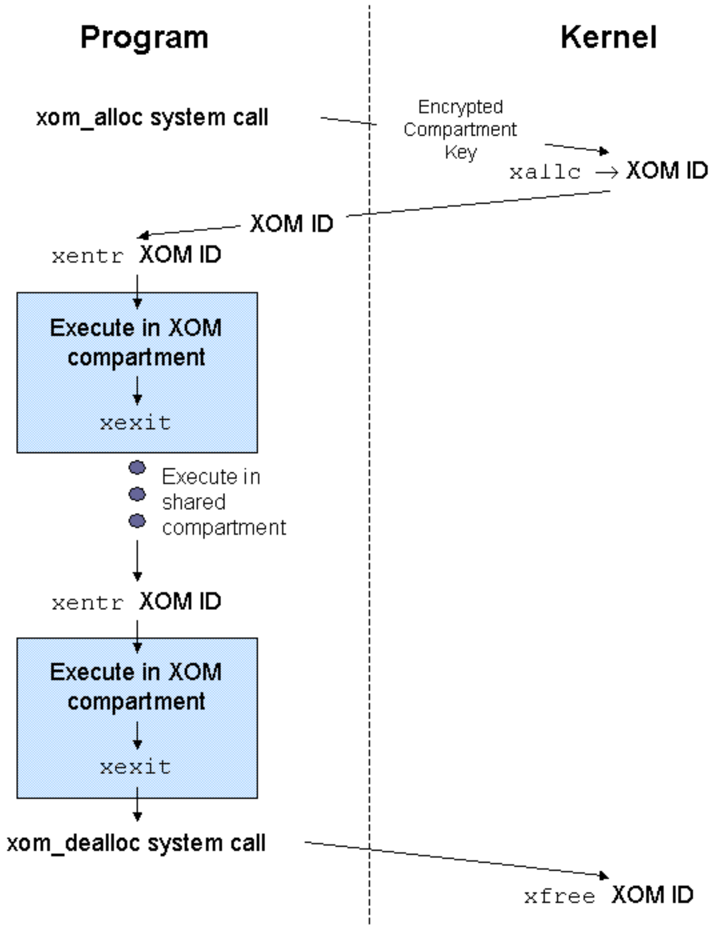

Figure 4.1: Allocating and Using XOM Key Table entries. When a program wishes to enter its compartment for the first time, it must first allocate an entry in the XOM Key Table and get a XOM ID. However, `xalloc` the instruction to allocate XOM Key Table entries is a privileged instruction. As a result, this functionality is provided to the program by the operating system via a system call. With a valid XOM ID, the program and enter and exit the compartment as it pleases. However, only the operating system can deallocate the XOM Key Table entry, which the program does via another system call.

4.2.2 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. `xfree` 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 `xfree` 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. This can be implemented by reserving a XOM ID tag value to indicate that a register is invalid and cannot be read by any program. 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 `xfree` 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. Should there be no process which is a suitable candidate for such a reclamation, XOMOS has two options: it can forcibly kill a process holding the resource, or it may stall the process that is requesting the resource until it becomes available. If XOMOS chooses to stall the requester, it must be careful not to cause any deadlocks amongst its processes.

4.3 Dealing with Encrypted Data and MACs

Because user data is made inaccessible to XOMOS in plain text, XOMOS can only move user data around when it is encrypted and accompanied by a MAC. This means that in some cases, such as dealing with user state on an interrupt, XOMOS must first invoke the hardware to encrypt the user state before it can save it away. In this section, we will explain how XOMOS uses the XOM hardware to save and restore user contexts. In other cases, XOMOS needs to relocate user data in memory to virtualize memory. However, when secure data is encrypted in memory with a MAC, the operating system must manage these MACs along with the encrypted data. This modification is also covered in this section.

4.3.1 Saving and Restoring Context

As discussed in Section 3.1.2, the operating system saves the state of an interrupted process with the aid of additional hardware instructions. However, 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.

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 4.2 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.

```
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 4.2: XOMOS Context Switch Code. In this code snippet, the operating system saves the state of the register `$s0`. To do this, it first tests the XOM ID tag on the register with `xgetid` to get the identity of the compartment it is in. It then encrypts the register with `xenc` and saves the encrypted and MAC'ed value to memory with `xsave`. Lastly it saves the compartment XOM ID. To restore the register, it does the inverse, loading the encrypted and MAC'ed value with `xrstr`, and decrypting it with `xdec`.

Some parts of the interrupted process state cannot be protected by XOM and are left tagged with the shared 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.

4.3.2 Paging Encrypted Memory

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 MACs in physical memory. Unfortunately, it is impossible to implement the storage of the MACs completely in hardware because to virtualize memory, the operating system must be able to access and move the MACs together with encrypted values in memory. 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 MACs and the data. The hardware will not let a XOM program with a secure memory load if a valid MAC is not supplied to it. To tamper with data, the operating system must be able to create a valid MAC 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 MACs 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 MAC, 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 MAC 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 MAC 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 MAC values protecting it. These are stored in a separate segment in the executable file. When a code page is faulted in, the appropriate MAC 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 MAC. Unencrypted values will not have hash values that need to be saved. The design could be made more efficient with additional hardware in the form of support for supporting flexible hash address to physical address mappings.

4.4 Supporting Traditional Operating System Mechanisms

The XOM architecture affected three mechanisms normally found in operating systems. The first was the use of shared libraries. Since shared libraries are executed directly by many applications, they cannot be used by an application in a compartment directly. We will address this problem first. We will then address the problem of running multiple instances of the same compartment code. An example of this is if an application in a compartment makes a UNIX “fork” system call. Finally, certain operating systems allow applications to define their own signal handlers. However, when the operating system delivers a signal to the application, the state of the interrupted context will be encrypted. The signal handler requires special processing to access the encrypted state, as well as to modify it and restart the context.

4.4.1 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 shared compartment so that the callee can access them. The key point is that all data that the library will need, must be placed in the shared compartment for the library to access. After this, the program exits 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 is reversed. The application re-enters its XOM compartment, moves the stack pointers back from the shared compartment, 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 4.3.

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

4.4.2 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. This is a problem, since a fork will appear exactly like a replay

```
# 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 4.3: Exiting and Entering a Compartment for a Library Call. To make a shared library call, the XOM program must move all subroutine arguments, as well as stack, frame and global pointers to the shared compartment. It then exists the compartment and jumps to the subroutine. On subroutine call return, the XOM program enters the compartment, moves the return value and pointers back, and continues with execution.

attack to the hardware since the same register state is being loaded twice.

As was mentioned earlier, 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 to use the old register ownership tag, while the child will use the new register ownership tag. Any data the parent wishes to pass to the child securely must be done through memory via the `secure store` and `secure load` instructions as `xom_fork()` is executed outside of the compartment.

Eventhough we have two different XOM ID's able to read the same memory values, this is not a security flaw. Both XOM ID's are allocated to programs with the same compartment key, so the only way an adversarial operating system might exploit this mechanism, is to copy state from one instance of the program to another. However, since a program can protect its memory from replay, it would be able to detect when its memory values has been overwritten with values from another instace.

4.4.3 User-defined Signal Handlers

Operating systems typically provide a mechanisms in which they may deliver interrupt-like events to a user process. In UNIX this mechanism is called a *signal*. Typically processes have default handlers defined that will be invoked when a signal arrives. Systems may also allow programs to define their own handlers which they register with the operating system to be called instead of the default handlers. These are termed *user-defined signal handlers*.

User-defined signal handlers are provided with a copy of the state of the interrupted process. It may access that state, as well as modify it and then restart the process with the new 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 4.2. In XOMOS, 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

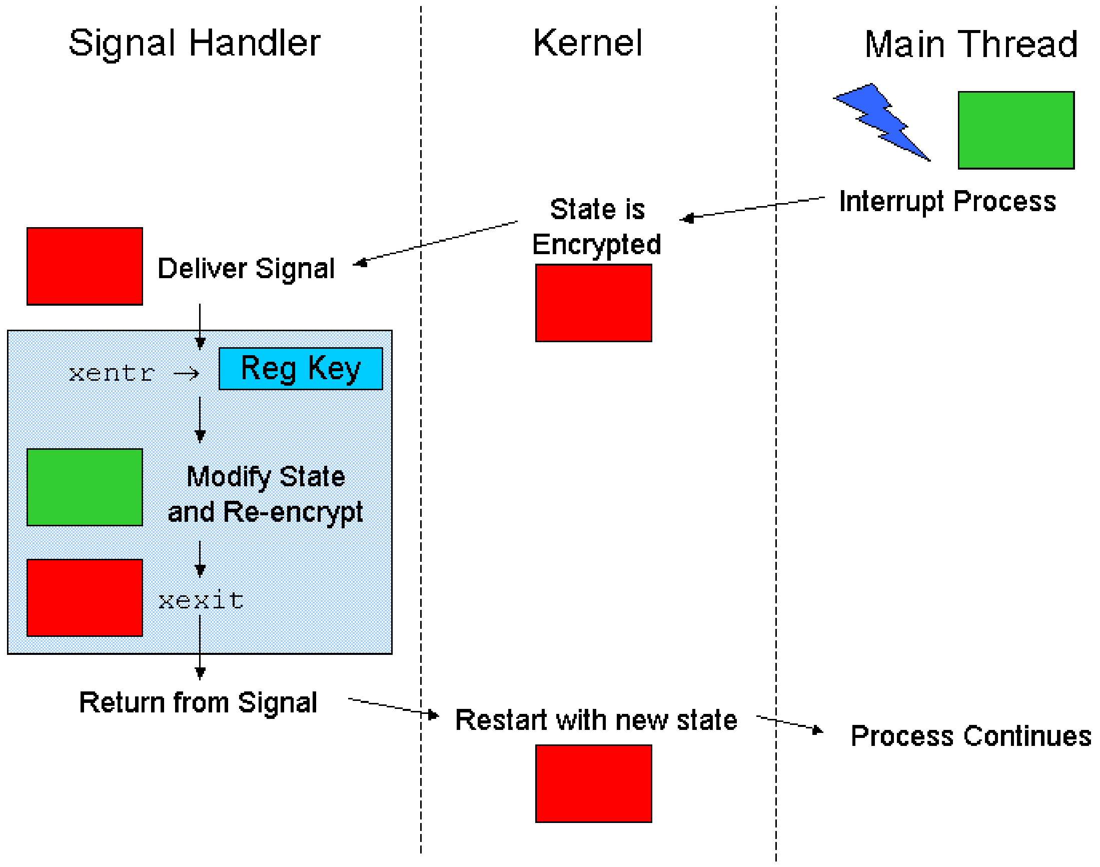

Figure 4.4: User-defined Signal Handler Support. The main thread of execution recieves a signal and the user-defined signal handler is invoked. The handler is atomically given the register key by the hardware when it enters the compartment. In this stage, the compartment code in the handler may alter the state of the main thread's compartment before restarting it.

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. This process is illustrated in Figure 4.4.

In fact, if the signal handler modifies any of the *sigcontext* registers, it should select a new register key and re-encrypt 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 since the contents in the *sigcontext* structure will only be accessible if they were encrypted and hashed properly.

We have described the process of porting XOMOS. In this process, the instructions in Table 3.3 were implemented in our simulator to execute XOMOS. Table 4.1 summarizes the functions that the instruction set extensions perform.

4.5 Costs of Implementing XOMOS

With an implementation, we can now evaluate the overheads of implementing XOMOS. There are two types of overheads in the porting XOMOS: the implementation effort in the amount of extra code that was added or changed in IRIX to make XOMOS, and the performance overhead in the slowdowns that the extra code adds to the execution of both the operating system and XOM programs. In this sections we will discuss each of these overheads in turn.

Instruction	**Description**
`xalloc $rt,offset($base)`	Privileged. `$rt` is the XOM ID of the allocated XOM Key Table Entry. `memory[$base + offset]` is the location of the encrypted compartment key.
`xentr $rt,$rd`	Enter XOM compartment with XOM ID `$rt`. The current register key is placed in `$rd`.
`xfree $rt`	Privileged. Mark the entry in the XOM Key Table indicated by `$rt` as invalid.
`xrclm $rt`	Privileged. Reclaim XOM ID `$rt` in the XOM Key Table.
`xexit $rt`	Exit XOM compartment and return to the shared compartment. `$rt` becomes the register key for the XOM ID.
`xsd $rt,offset($base)`	Stores `$rt` into `memory [$base + offset]`.
`xld $rt,offset($base)`	Loads `$rt` with `memory [$base + offset]`.
`xgetid $rt,$rd`	XOM ID tag value of `$rt` is placed in `$rd`.
`xenc $rt,$rd`	If so, encrypt the contents of `$rt` with the keys indicated by XOM ID `$rd`.
`xsave $rt,offset($base)`	Store contents of XOM coprocessor register `$rt` to `memory[$base + offset]`.
`xrstr $rt,offset($base)`	Load memory at `memory[$base + offset]` into XOM coprocessor register `$rt`.
`xdec $rt,$rd`	Decrypt the 256 bit value set by `xrstr`, validate the result and restore to register `$rt`. Set the XOM ID tag on `$rt`.
`xmvtn $rt`	Set the XOM ID tag of `$rt` to shared.
`xmvfn $rt`	Set the XOM ID tag of `$rt` to the XOM ID of the executing program.

Table 4.1: Description of Simulated Instructions. The instructions implemented in the simulator were implemented to support the port of XOMOS. Their functions are described here.

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 4.2: XOMOS Kernel Implementation Effort. This table gives number of lines and number of files that were changed in the IRIX Kkernel to make XOMOS.

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

Table 4.3: XOMOS User-code Implementation Effort. The number of lines changed in user level code in XOMOS.

4.5.1 XOMOS 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 4.2. 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 4.3.

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. As a result, much of the higher-level functionality of the operating system, such as the resource management policies, kernel architecture, file system and application binary interface, 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 should be possible to port other operating systems to run on the XOM architecture.

4.5.2 Operating System Performance Overhead

To try to estimate the performance of XOMOS, we ran simulated work loads on our simulator. Our cycle-accurate simulator models an in-order processor model as well as caches, memory and disks. Unless otherwise stated, we use the default simulated machine parameters given in Table 3.2.

The operating system modifications add overhead in several areas; Tables 4.4, 4.5 and 4.6 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 `xalloc` instruction takes 400,000 cycles to complete, during which time the CPU is completely stalled. 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 `xfree` 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 registers 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 our case the bandwidth increase of one eighth should result in only a modest performance decrease.

Reclaiming XOM Key Table entries also results in some operating system overhead, since it requires expensive flushing of the on-chip caches. However, note that each time a XOM Key Table entry is allocated, the XOM processor needs to perform an expensive public key operation. 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.

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

Table 4.4: Overhead Due to New System Calls in XOMOS. These are the overheads of the two system calls that XOMOS.

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. Tables 4.5 and 4.6 show 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 4.4.1, 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

Benchmark	**Total Instructions**			**Kernel Instructions**		
	IRIX	XOM	OV	IRIX	XOM	OV
System Call	3.8K	4.0K	5%	3.8K	3.8K	1%
Signal Handler	11.1K	14.6K	32%	11.0K	14.4K	31%
XOM_Fork	119K	123K	3%	118K	121K	3%

Table 4.5: Micro-benchmark Instruction Overhead of XOMOS vs. IRIX. We compare the instructions executed of three micro-benchmarks in XOMOS versus the original IRIX operating system. The overheads are small except for the Signal Handler benchmark performs more XOM operations and as a result incurs more overhead.

Benchmark	**Total Cycles**			**Cache Misses**		
	IRIX	XOM	OV	IRIX	XOM	OV
System Call	5.7K	6.3K	11%	4.2	5.6	33%
Signal Handler	31.2K	39.6K	27%	38.4	48.3	26%
XOM_Fork	13.9M	12.6M	-9%	1035	1058	2%

Table 4.6: Micro-benchmark overhead of XOMOS vs. IRIX. The extra instructions in the Signal Handling benchmark also affect the cache behavior adversley. This results in longer execution time.

while the application is in a compartment, the kernel must use the longer XOM save routines shown in Figure 4.2 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. This is an artifact of our implementation.

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.

4.5.3 End-to-end Application Performance 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 [48] 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 in these experiments. 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,

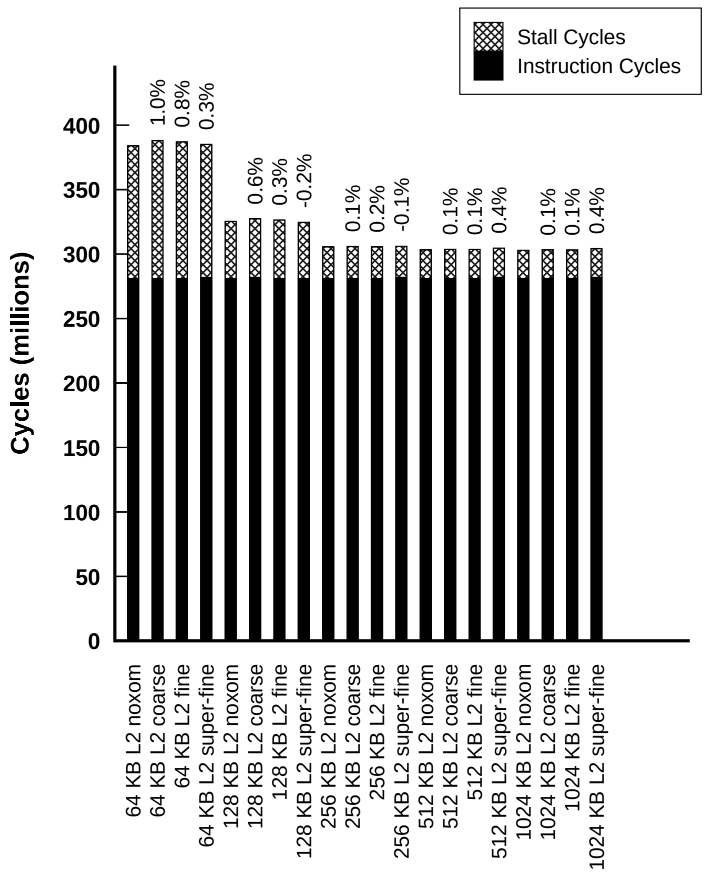

Figure 4.5: Performance of XOM-mpeg. Percentages above the bars show the increase relative to the non-XOM case for each cache size.

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

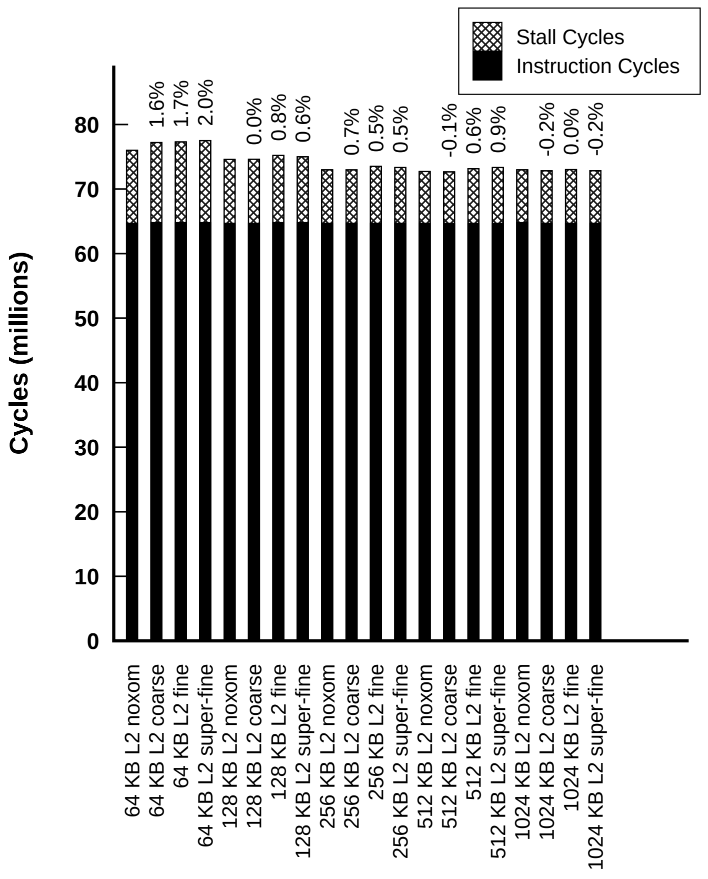

Figure 4.6: Performance of XOM-RSA. Percentages above the bars show the increase relative to the non-XOM case for each cache size.

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 3.1.4, 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 4.5 and Figure 4.6, 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 [42]. 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.

4.6 Summary

In this chapter, we have explored the implementation of XOMOS, an operating system for our example XOM architeture. XOMOS is implemented as a port of the IRIX operating system. This port involved adding two new system calls, `xom_alloc()` and `xom_dealloc()`, modifying the way the operating system handles user data, and updating various operating facilities such as shared libraries, fork, and signal handling. 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.

Our preliminary performance numbers look promising. The hardware overheads in our simulator 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. We found that in reality, end-to-end application overheads are often less than 5%, regardless of the granularity of the XOM compartments. The performance is dependent on the cache behavior, which is influenced by the number of transitions due to the instruction and data alignment transitions require. In the coarse compartment usage model, the only transitions are due to system calls and shared library calls. System calls are expensive in any case, so the additional cost of XOM appears smaller in cases where there are many transitions due to system calls. 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.

Chapter 5

Security Issues

To determine whether a system is secure against attacks, one must have a clear idea what attacks are possible. This chapter begins by outlining the goals of the adversary, as well as some general strategies the adversary can take in trying to achieve these goals. This chapter will also address various hardware or software based attacks that an adversary can take.

Verifying security is a problem that is comparable to the more general problem of system verification. As a result many of the approaches of used for general system verification are also applicable to the specific problem of security verification. This chapter examines the use of one of these *formal verification* method towards the verification of system security. It gives a formal specification of the XOM architecture, and shows using a model checker, that it is secure against an adversarial operating system.

With a level of security established by the formal verification, we address other issues that XOM raises, including key revocation and privacy.

5.1 Attack Model

An adversary who attacks a program in a XOM compartment, generally has the intent to observe or modify that program. This goal can be stated explicitly as three separate goals:

1. Copy a piece of software so that it runs on a processor for which it was not intended.

2. Obtain values of instructions, static data or dynamic data of the program.

3. Modify the execution of a program in its compartment without being detected.

The third goal can be achieved either by tampering with instructions that are executed in a compartment or by modifying data values that will be used by those instructions.

In trying to achieve her goals, the adversary may try several different strategies. These include:

1. Obtain master secret or private key of a processor.
2. Obtain compartment key of a program.
3. Read some instruction or data values inside a compartment.
4. Deterministically alter instruction or data values inside a compartment.
5. Randomly alter instruction or data values inside a compartment.

The first strategy is clearly fatal to a XOM system. The adversary can use this information to decrypt and totally compromise any secure software that was encrypted for that processor. In addition, the adversary can use this information to acquire and decrypt more software unless this attack is detected. Upon detection of a compromised master secret, the public key of the broken processor must be revoked. This process is described in Section 5.6.1.

The second strategy, if successful, is fatal to the particular program that is compromised. Knowledge of the compartment key allows the adversary to arbitrarily read or modify instructions or data in the program.

The last two strategies are possibly damaging, but may be only of limited use to an adversary. Being able to read some values gives the adversary partial visibility into the execution of a program, but perhaps not enough for the adversary to achieve her final goal. The ability to modify some values may alter the behavior of a program in a way that may leak information that is useful to the adversary, but this is not always the case. To exploit these weaknesses, the adversary must rely on some element of random chance, something she can mitigate by repeating the attack a numerous number of times. However, because computers are able to perform a large number of actions in a very short amount of time, it is often important to defeat even the last two strategies that an adversary may employ.

5.2 Hardware Based Attacks

Protecting against hardware attacks is a matter of cost. By spending more resources on equipment, time and XOM processors, the adversary can try to extract the private key from the processor. She can do this by trying to monitor the electric signals on the processor using techniques such as IBM's Pica system [49], power analysis techniques [11], differential fault analysis [4] or even trying to reverse engineer the chip by examination. Most of these attacks require expensive equipment on the part of the adversary and can be made expensive by increasing the costs of manufacture. For example, air sensitive layers can be added to the packaging to make it difficult for an adversary to gain access to the physical chip.

Another possible avenue of attack is through *scan*. Scan is normally used by hardware designers to perform silicon debug on a part. Scan consists of a set of *scan chains*, which link together certain registers on processor to form a "chain." For debugging, the chip can be placed in a special mode where the registers on this chain can be shifted out and new values can shifted in. Processor manufacturers typically leave the scan chains intact when they ship their products. In some cases, third party mother board manufacturers require that the boundary pins be scanned to debug their products. Scan poses a potential problem for XOM since a malicious adversary can use it to access state on the processor. Accordingly, internal scan chains should heed the access rules imposed by the compartment model or be disabled after packaging. In other words, architectural elements that are tagged with the XOM ID of a private compartment should not be readable or writable while using scan.

Finally, the adversary may try to read or modify values while they are outside of the chip, either by tampering with the memory bus, or by directly tampering with memory itself. The use of encryption combined with cryptographic MACs guarantees both the confidentiality and integrity of data stored outside of the processor.

5.3 Software Based Attacks

In contrast to hardware attacks, an adversary could mount software to try and circumvent the compartment architecture that XOM provides. We have already briefly outlined various software attack models in Section 3.1.4. Here, we will discuss those attacks in more detail. In a software based attack, the adversary gains control over the operating system and exploits its privilege to attack software running on the system. The tags in the system combined with encryption for off-chip values prevent even a malicious operating system from reading values to which it does not have permission to access. However, because the operating system's role is to manage resources, it does have the ability to overwrite or *tamper* with values in memory or on-chip. In general there are three types of software based attacks an adversary could attempt.

A *spoofing attack* is the simplest kind of attack. This is an attack where the adversary tries to alter program data by overwriting it with either a chosen or random value. The adversarial operating system can try to tamper with an on-chip value in a register or cache, as shown in Figure 5.1. However, the tag on the architectural element will reflect that the value written originated from the operating system. When the program under attack goes to access that element, it will get an exception since its XOM ID will not match that of the element. Tags are not used in memory so cryptographic techniques must be used to prevent spoofing attacks by the adversary on values in memory. XOM employs Message Authentication Codes or MACs [34] to check the integrity of data that is encrypted and store in memory. Because the MACs are keyed, the adversary may modify a cipher text stored in memory, but will not be able to forge a MAC to match that cipher text. When the XOM processor loads encrypted data from memory, it always verifies it against the MAC to check its integrity.

A slightly more complex attack is a *splicing attack*. Rather than trying to spoof valid cipher texts, the adversarial operating system copies valid cipher texts from one address in memory to another. The XOM processor defends against this tampering including the virtual address along with the data in the pre-image that is used to compute the MAC. When verifying the MAC on a load, the XOM processor checks it against the virtual address that the program is loading the value from.

The final type of tampering attack is known as a *replay attack*. In this attack, the adversary observes values and then reuses them at a later time. An adversarial operating system is capable of performing this attack on both register and memory values. To replay a register value, the adversarial operating system interrupts a running process and saves the register state using the `save register` instruction. The adversary than restores the

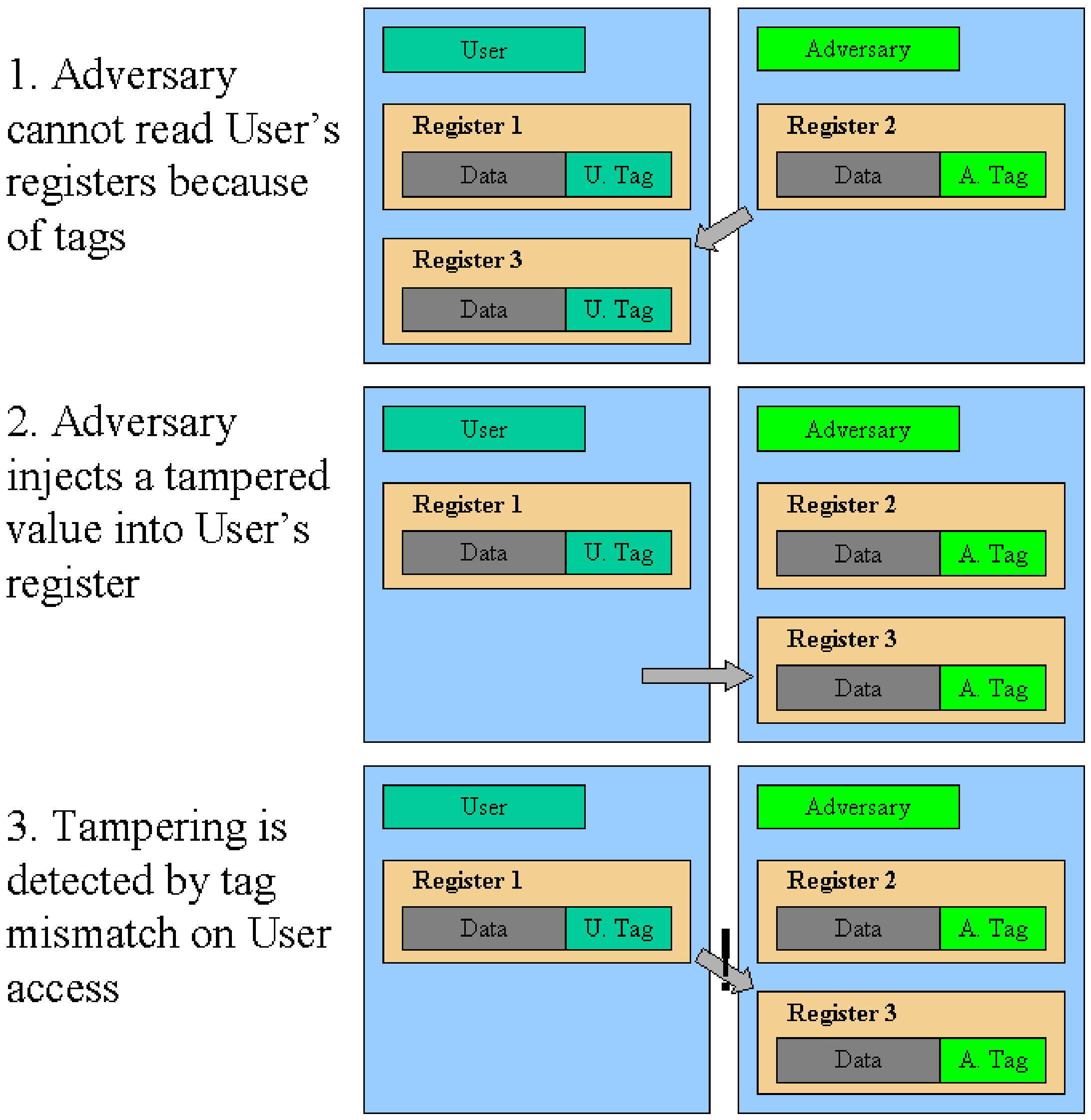
1. Adversary cannot read User's registers because of tags
User
Register 1
Data
U. Tag
Register 3
Adversary
Register 2
A. Tag
2. Adversary injects a tampered value into User's register
3. Tampering is detected by tag mismatch on User access
!

Figure 5.1: A Register Spoofing Attack.

process state and restarts the process. At a later time, the adversarial operating system interrupts the process again, but instead of restoring the register values from the second interruption, it restores the values from the first interruption. When the process restarts, it will be using the replayed register values. However, the XOM processor defends against such an attack. When encrypting and decrypting registers with the `save register` and `restore register` instructions, the XOM processor uses a *register key*, which is regenerated every time a particular XOM compartment is interrupted. As a result, the register key that is used to save the register at the time of the first interrupt, will have been destroyed and regenerated when the adversary tries to restore the register at the second interrupt. As a result, trying to restore the stale value will result in an exception.

Instead of trying to replay values in registers, the operating system may try to replay data in memory. To do this, the operating system simply records values and MACs in memory and then overwrites values at a later time with the stale values and MACs. To defeat this attack the application can keep a hash for a region of memory in one of the registers. To replay this region, the adversary must also be able to replay the hash kept in the register. However, the regenerating register key will protect the register from replay, thus defeating the memory replay attack. The problem with this approach is that every time a value in the region changes, the hash kept in the register must be updated. If the region of memory is large, or if the values in this region change frequently, this results in a large overhead as the entire region must be read to update the hash. The performance impact can be mitigated with additional hardware support by using a Merkle trees to perform memory authentication [20]. A Merkle tree is a hierarchical hash structure that allows efficient update of a hash that protects a set of elements, in this case data in memory. Figure 5.2 shows an example tree. The data elements to be protected make up the leaves of the tree and each node in the tree contains the hash of all of its children. This structure is replicated until a single root is reached. To protect the tree from tampering, all that is necessary is to ensure the integrity of the root, which can be stored in a register. The reader should note that it is non-trivial to combine Merkle trees with a cache. The reason for this will be made clear in the next section.

5.4 Formal Specification and Verification

A formal specification was constructed and checked with the Murφ finite-state model checker [14]. In specifying this model several assumptions were made about the types of errors the checker is trying to catch. First, the model uses a "black box" model for cryptographic functions [15]. This means that encrypted data cannot by decrypted by the

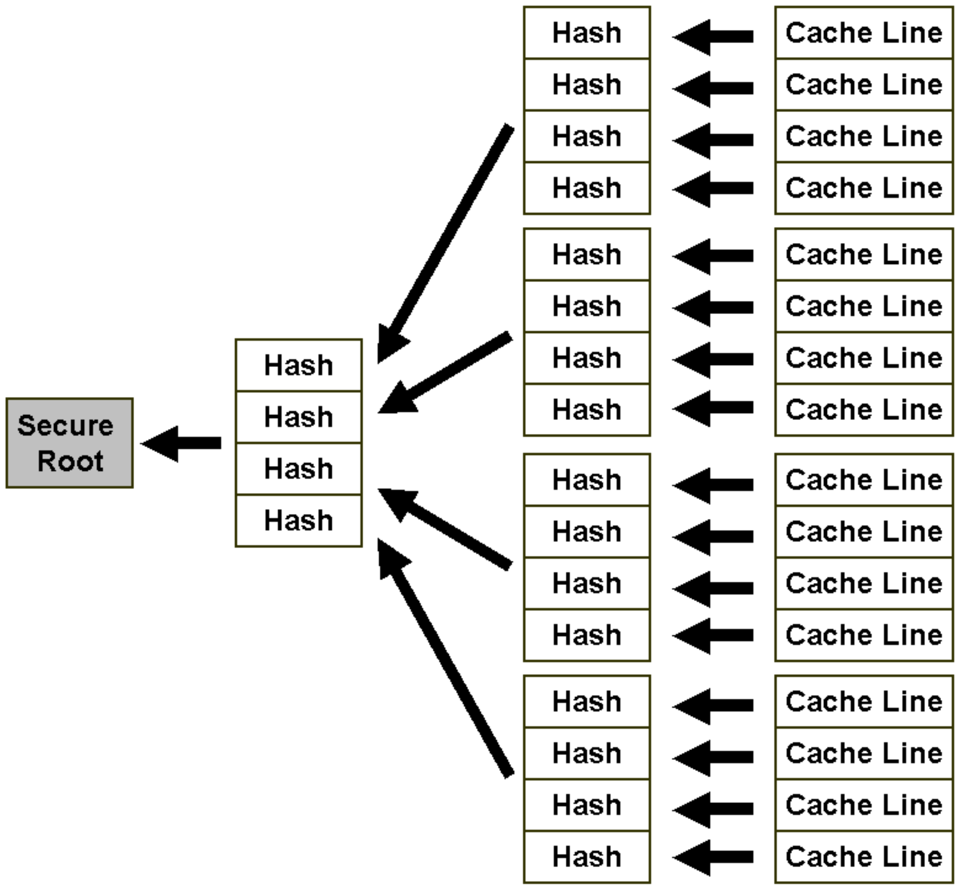

Figure 5.2: A Merkle Tree. This structure uses a hierarchical hashing scheme to protect a large number of elements with a single hash. Update to the hash is efficient as only hashes of nodes along the path between the changed cache line and the root need to be re-calculated.

adversary. However, if two plain text values are equal, they will have the same cipher text values. The model also models hashes as collision-free, and assumes that programs have been properly written, so that all data sharing occurring in the shared compartment is intentional.

The adversary in the model is assumed to be an adversarial operating system. Thus the abilities of the adversary are that which privileged code would have on a XOM processor. The code can arbitrarily interrupt the target process and read, write and copy data in registers, caches or memory according to the rules of the XOM architecture. Despite this, model was verified to have two properties. The XOM processor is able to prevent the observation of program code or data, as well as prevent modification of program code or data by halting the process upon detection. The general approach taken by this verification will be to define two models — an *actual* model with an adversary and a *idealized* model that does not have an adversary. The verification will check that the program states in the two models are always consistent, otherwise, the model with the adversary has a fault, as the modeled adversary was able to tamper with the program state.

This section will begin by briefly describing the Murφ model checker. It will then describe an abstracted instruction set that will be used to model the instruction set of a XOM processor. The next sections describe the modeling of the XOM hardware in both the actual and idealized models. Finally, it will describe how to simultaneously check both models for consistency using a model checker, and give an example of an attack that was found.

5.4.1 The Murφ Model Checker

This thesis used the Murφ [14] model checker to verify the XOM model specification. Murφ uses explicit enumeration to check the state space of a model. A model describes the system to be checked as a state machine by providing an initial state and a set of next-state functions. The next-state functions are specified by a set of "rules", which have a precondition guard, and a set of actions that modify the current state to produce a new state. A precondition is a boolean statement based on the current state of the machine. Murφ performs the state exploration by starting with the initial state and exhaustively searching for all successor states. Murφ finds and executes rules whose precondition is satisfied by the current state to identify successor states. Murφ verifies the correctness of each new state against a set of safety criteria to determine if any of the states are illegal. Safety criteria in Murφ are specified as a set of invariants, which are boolean statements that are evaluated every time a new state is found. When Murφ detects an error, it outputs a counter example that indicates the states it traversed to reach the error state. Murφ has been successfully used in other work to verify both security protocols [44, 57] and computer hardware [38, 60].

Model checkers, in general, have some limitations. First, they verify models of systems, not the systems themselves. Models abstract details of the system to make the size of the state space tractable for the model checker. This is often done by simplifying functionality and by scaling down the models. Second, model checkers can only explore a finite number of states, and may miss states to save memory. For example, rather than explicitly remembering the states it finds, Murφ saves a smaller, randomized low-collision hash of them. There is some probability that a collision will result in missed states, but because the hash is randomized, successive verifications of the same model reduce this probability.

5.4.2 Abstracting the Instruction Set

To reduce the state space of the models, the specification contains a simplified version of a real XOM machine instruction set. The instruction set is reduced to just the operations that affect the flow of data and information. For example, generic register operations are amalgamated under the `def` and `use` operations, while control flow operations such as branches and jumps are left out. Similar simplifications were performed to analyze Java in [19]. The instructions available to the user are summarized in Table 5.1.

An adversarial operating system can execute both user instructions and privileged kernel instructions. The additional privileged instructions available to the adversary are summarized in Table 5.2. Note that the `prefetch`, `write_cache`, `invalidate` and `flush` instructions are not defined in the abstract XOM machine, nor were they implemented in the XOM simulation system. However, they are part of the specification to be verified here as they are reasonable future additions that could be made to the XOM architecture. Another simplification that was made is that the model assumes that there is only

Instruction	Description
i1. `def $rt,immediate`	Generic register definition where `immediate` is written into register `$rt`. This models any non-memory operation that writes to a register.
i2. `use $rt`	Generic use of register `$rt`. This models any non-memory instruction that reads a register.
i3. `xsd $rt,addr`	This stores the value of register `$rt` to the memory location at `addr`. The store sets the XOM cache tag to the value of the program that executed the store.
i4. `xld $rt,addr`	This loads the register `$rt` with the memory value at `addr`. If the load hits in the cache, the XOM cache tag of the cache data is checked against the program's XOM cache tag. If the data is in memory, the data is decrypted with the program's compartment key and the hash verified before loading into the register.

Table 5.1: User Instructions. List of simplified instructions available to the user in our models.

one process per program. This simplification allows the model to make XOM ID tag and XOM cache tag values equal, as there would be a one to one relationship between them.

5.4.3 The Actual Model

The model of the physical hardware in a XOM machine, called the *actual* model contains three homogeneous arrays. Each array represents one of the storage levels in the machine: registers, cache, and memory. The records stored in the arrays have different properties. For example, they all have a data value property, as they can all be used to store data. However, not all have XOM ID tag, hash, or key properties.

A state in the model S_{actual} contains the three arrays, as well as a single bit that indicates whether the *user* or the *adversary* is executing on the machine. Programs run as *users* when the mode is *user_mode*, while the operating system is adversarial and runs when the mode

Instruction	**Description**
i5. `xsave $rt,$rs`	The operating system uses this instruction to store a program register. The XOM processor encrypts and hashes the program register `$rs` with the *register key* and stores it to register `$rt`. The register `$rt` is tagged with the operating system's XOM ID.
i6. `xrstr $rt`	The operating system uses this instruction to restore registers saved with `xsave`. The XOM processor decrypts the data, verifies the hash, and returns the data to its original register. The *register key* is regenerated each time the XOM processor traps to the operating system.
i7. `prefetch addr`	The operating system can move a value from memory into cache, even if its XOM ID does not match the key the data in memory is encrypted with. The data is tagged in the cache, not with the XOM ID of the operating system, but with the XOM ID of the key used to decrypt it from memory.
i8. `write_cache addr`	The operating system can change the value of a cache location. The new data is tagged with the operating system's XOM ID.
i9. `invalidate addr`	The operating system can invalidate a cache location causing the data to be destroyed. The data is not flushed to memory.
i10. `flush addr`	The operating system can flush a cache location with `addr` from the cache to memory. The value is encrypted with the key indicated by the XOM cache tag, hashed and the placed in memory.
i11. `trap`	The operating system can cause the processor to trap to the operating system at any time by sending an interrupt. However, each time this happens, the *register key* used to encrypt and decrypt user registers from any previous traps is regenerated.
i12. `return`	The operating system can return use of the processor to the user at any time.

Table 5.2: Privileged Instructions. Additional instructions also available to a malicious operating system, the adversary in our verification.

is *adversary_mode*.

$$\begin{aligned}\boldsymbol{S_{actual}} = <& r_0, r_1 \ldots r_x;\\ & c_0, c_1 \ldots c_y;\\ & m_0, m_1 \ldots m_z;\\ & mode >\end{aligned}$$

The state has 3 storage classes, each representing a storage level. A register r_i, has a data value d, a XOM ID tag t, and if the value is encrypted, a key k, and a register hash h, associated with it. Similarly, a cache line c_i, has a data value d, XOM ID tag t, and a memory address a. Note that each cache line only holds one register word. As noted in Section 3.3, a XOM machine will likely have several words per cache line, requiring the addition of valid bits. In this case, we have consciously removed that case to allow for a simpler model. Finally, memory locations m_i, have data values d, keys k, and address hashes h. The $\varnothing$ value means that the parameter is not defined. For example, register values can be in plain text, meaning they have no key or hash value associated with them. Similarly, an unused cache location has a $\varnothing$ address value.

$$\begin{aligned} r_i &: \text{record } \{d : \boldsymbol{D};\quad t : \boldsymbol{P};\quad k : [\boldsymbol{P}, \varnothing];\quad h : [\boldsymbol{R}, \varnothing]\}\\ c_i &: \text{record } \{d : \boldsymbol{D};\quad a : [\boldsymbol{M}, \varnothing];\quad t : \boldsymbol{P}\}\\ m_i &: \text{record } \{d : \boldsymbol{D};\quad k : \boldsymbol{P};\quad h : [\boldsymbol{M}, \varnothing]\}\\ mode &: [adversary\_mode, user\_mode]\end{aligned}$$

The model has four basic data types, to which members of the records belong. $\boldsymbol{D}$ is a set of distinct data values, $\boldsymbol{P}$ is a set of principals and can either be the user or the adversary, $\boldsymbol{R}$ is a set of register hashes, one for each register in the model, and $\boldsymbol{M}$ is a set of memory addresses in the model.

$$\begin{aligned}\boldsymbol{D} &= [0 \ldots w, \alpha]\\ \boldsymbol{P} &= [user, adversary]\\ \boldsymbol{R} &= [0 \ldots x]\\ \boldsymbol{M} &= [0 \ldots z]\end{aligned}$$

The α in $\boldsymbol{D}$ represents a value that was created by the adversary, which is used to model data injection attacks. The XOM machine uses tags as proxies for keys, so tags and keys are type-equivalent. Similarly, the pre-image for a memory hash is an address so they are also type-equivalent. Initially, the model has $\varnothing$ in all its storage locations and the *mode* is set to *user* mode. The maximum number of data values w, number of registers x, number of cache lines y, and number of memory locations z parameterize the model.

If the model detects tampering, it prevents the user from continuing execution by causing a *reset* action. The reset action is a special action that sets the entire machine state to a legal state. In the XOM model, the reset condition sets the state back to the initial state of the model. The user is able to perform the operations specified in Table 5.1. The actions that produce the next state for each instruction are:

```
i1. Define data value $a \in [0 \ldots w]$ in register $i$:
    $r_i = \{d = a, t = user, k = \varnothing, h = \varnothing\}$.

i2. Use register $i$:
    if $r_i.t \neq user$
    then reset.

i3. Store value at register $i$ into address $j$:
    if $r_i.t \neq user$
    then reset
    else if $j$ is in cache such that $\exists\ c_l.a = j$
        then $c_l = \{d = r_i.d, a = j, t = r_i.t\}$
        else pick an $l \in [0 \ldots y] : c_l.a = \varnothing \rightarrow$
         $c_l = \{d = r_i.d, a = j, t = r_i.t\}$.
```

```
i4. Load memory address $j$ into register $i$:
    if $j$ is in the cache such that $\exists\ c_l.a = j$
    then if $c_l.t \neq user$
        then *reset*
        else load from cache $r_i = \{d = c_l.d, t = user, k = \varnothing, h = \varnothing\}$
    else load from memory: if $m_j.k \neq user \lor\ m_j.h \neq j$
        then *reset*
        else pick an $l \in [0 \ldots y] : c_l.a = \varnothing \rightarrow$
        $c_l = \{d = m_j.d, a = j, t = user\}$,
        $r_i = \{d = m_j.d, t = user, k = \varnothing, h = \varnothing\}$ .[1]
```

The model checks that the user's data has not been tampered with. As explained in Section 3.3, a hash validates two components — the integrity of the encrypted data and the validity of the address it is being loaded from. These two checks are modeled separately. Preventing the adversary from creating any data that is encrypted with the user's key simulates the integrity check of the data. On the other hand, the validity of the address is modeled by keeping a shadow copy of the original address of the data in h.

Here is a section of the model in the Murφ description language. This particular section describes instruction i3:

```
Rule "User secure store"
  !isundefined(reg_i.data) &
  mode = user_mode
==>
Var
  cache_l : cache_range;
Begin
  if (reg_i.tag != user) then
    reset();
  endif;
  -- find the data in the cache
  cache_l = find_data(addr_j);
```

[1]Note that this definition prevents the user from reading uninitialized memory values. Such a read is considered a programmer error, which is not covered in this verification.

```
    if (!isundefined(cache_l)) then
      -- hit in the cache,
      -- write data to cache
      cache_l.data := reg_i.data;
      cache_l.addr := addr_j;
      cache_l.tag := reg_i.tag;
    else
      -- cache miss (code not shown)
    endif;
  endrule;
```

A “`--`” indicates the following text on that line is a comment. The precondition is the boolean expression before the “`==>`,” which checks that the register has data in it, and that the processor is not in adversary mode. The body of the function, after the `Begin` statement checks the permissions on the register, checks if the data is in the cache, and if so, writes it to the cache. The actions for servicing a cache miss are left out for brevity.

This model has some safety conditions that are checked:

1. Each cache location must have a different address tag. This ensures the cache is implemented correctly.

2. User data (data that is not α) is either tagged with the user’s XOM ID or encrypted with the user’s key. This ensures the access control guarantees are correct.

5.4.4 The Idealized Model

The *idealized* model is a very simple version of a XOM architecture. It has no caches since they are invisible to the user, and does not contain an adversary. The model has a two storage levels: registers and memory locations. The state of the model can be expressed as:

$$\boldsymbol{S_{ideal}} =< r_0 \ldots r_x; \quad m_0 \ldots m_z >$$
$$r_i : [0 \ldots w, \varnothing]$$
$$m_i : [0 \ldots w, \varnothing]$$

The parameters of the model are w, x and z which are the number of data values, registers, and memory locations respectively. The model is much simpler and each storage class only has a data value property. As in the actual model, the initial state has all locations initialized to $\emptyset$. There is only one user in this model and the user can perform the following actions:

i1. Define data value $a \in [0 \ldots w]$ in register i:
$r_i = a$.

i2. Use register i:
this action is a null action.

i3. Store value at register i into memory location j:
if $r_i \neq \emptyset$
then $m_j = r_i$.

i4. Load an initialized memory location j into register i:
if $m_j \neq \emptyset$
then $r_i = m_j$.

The same register store action in the actual model is written as the following in the idealized model:

```
Rule "User secure store"
  !isundefined(reg_i.data)
==>
Begin
  mem_j.data := reg_i.data;
endrule;
```

The model has no safety conditions since the absence of an adversary means it cannot be tampered with.

5.4.5 The Adversary

The actual model also includes an adversary that can perform actions to modify the state of the machine. In the model checker, the adversary is given a set of primitive actions that it can perform. The model checker will exhaustively try all combinations of these actions in an attempt to break the XOM machine. It is important to ensure that the adversary is adequately powerful to model all attacks, but constrained so as not to be capable of things not possible in reality. Based on the "black box" model for cryptography, and assuming reasonable countermeasures against physical tampering of the hardware, the adversary is not able to:

1. Access registers not tagged with its ID.
2. Decrypt values for which it does not have the key.
3. Access keys stored on the XOM machine.
4. Forge hashes.

The actions that the adversary can perform are the basic user operations and the kernel mode operations detailed in Table 5.2. Aside from the 12 basic instructions available to the adversary, two composite instructions are added. These instructions are composed from several basic instructions, but do not require a free a register or cache line.

i1. The adversary can define data in a register i:
$r_i = \{d = \alpha, t = adversary, k = \varnothing, h = \varnothing\}$.

i2. The adversary can use a register i:
```
if $r_i.t \neq adversary$
then *reset*.
```

i3. The adversary can store a register i to address j:
```
if $r_i.k = \varnothing$
then if $r_i.t = adversary$
    then if $j$ is in the cache such that $\exists\ c_l.a = j$
        then $c_j = \{d = r_i.d, a = j, t = adversary\}$
```

else pick an $l \in [0 \ldots y] : c_l.a = \varnothing \rightarrow$
$c_l = \{d = r_i.d, a = j, t = adversary\}$

else *reset*.

i4. The adversary can load a cache location i to register j:
if $c_i.t = adversary$
then $r_j = \{d = c_i.d, t = adversary, k = \varnothing, h = \varnothing\}$
else *reset*.

i5. The adversary can save register i to register j:
if $r_i.k = \varnothing$
then $r_j = \{d = r_i.d, t = adversary, k = r_i.t, h = i\}$.

i6. The adversary can restore register i to register j:
if $r_i.h = j$
then $r_j = \{d = r_i.d, t = r_i.k, k = \varnothing, h = \varnothing\}$
else *reset*.

i7. The adversary can prefetch memory location i into cache location j:
if $m_i.h = i$
then $c_j = \{d = m_i.d, a = i, t = m_i.k\}$
else *reset*.

i8. The adversary can write cache location i:
$c_i = \{d = \alpha, a = c_i.a, t = adversary\}$.

i9. The adversary can invalidate a cache location i:
$c_i = \{d = \alpha, a = \varnothing, t = adversary\}$.

i10. The adversary can flush cache location i:
given $c_i.a = j \rightarrow m_j = \{d = c_i.d, a = c_i.k, h = j\}$.

i11. The adversary can trap to adversary mode. When doing so, the register key is revoked, so all encrypted registers are cleared:
if $mode = user$

then $mode = adversary$,
$\forall\, r_i \in [r_0 \ldots r_x]$: if $r_i.k \neq \varnothing$
  then $r_i = \{d = \alpha, t = adversary, k = \varnothing, h = \varnothing\}$.

i12. The adversary can return to user mode and allow the user to execute:
if $mode = adversary$
then $mode = user$.

c13. The adversary can copy a memory location i to another memory location j:
$m_i = m_j$.

c14. The adversary can copy a register i to register j:
if $r_j.t = adversary$
then $r_i = r_j$
else *reset*.

In modeling the adversary, two simplifying assumptions are made. First, in primitive i5, the adversary is not allowed to encrypt an already encrypted register. The same is true for an adversary trying to decrypt a value that has not been encrypted. While a real adversary could encrypt or decrypt register contents an arbitrary number of times by executing the appropriate instruction over and over again, the model is finite so they cannot model this. The model can easily be extended to allow an arbitrarily long chain of these instructions, but this would create a larger state space. Thus, for efficiency, the model specification restricts these operations to be performed only once on any value.

Second, the model does not allow the adversary to store encrypted register values to memory in primitive i3. The only operation that could be performed on an encrypted value is primitive i6, which only operates on values in registers. As a result, storing the values to memory means at some point, the adversary will have to load that encrypted value back from memory to another register to do anything with it. Since, this is already modeled in action c14, modeling this functionality again is unnecessary.

In Section 5.3, attacks were classified into three categories: spoofing, splicing, and replay. The primitive actions above allow an adversary to at least implement all three of these attacks. While, the adversary does not know the user's key and thus cannot insert chosen text into the user's compartment, she can still attempt to randomly write values there. A spoofing attack can be performed by actions i1, i8 and i10. A splicing attack is one where the adversary tries to copy valid, user encrypted values from one location to another. Actions c13, c14, and i7 allow an adversary splice registers, memory locations and caches respectively[2]. Finally, a replay attack involves an adversary who actively records data, waits for the user to overwrite that location with different data, and then inserts the old, stale data. To replay a register, the adversary executes c14 and then i11 to copy and return control to the user. When the user overwrites the old register value, the adversary executes i11 again, and uses c14 to copy the saved data back. To replay a memory location, the adversary can either use c13 instead of c14, or use i9 to prevent a new value in the cache from reaching memory.

5.4.6 Combining the Models

The primary goal is to verify two properties of the XOM architecture: that adversaries cannot read user data and that adversaries cannot modify user data without being undetected.

Since the XOM machine only permits principals to read registers that have been tagged with the correct XOM ID, the first property is verified by checking that data created by the user is never tagged with the adversary's XOM ID. This is actually the second safety condition in the actual model given in Section 5.4.3. However verifying the actual model alone does not ensure that the adversary has not modified user data. The difficulty is that there is no condition to check the user's data against in the actual model. The key observation is that there can be no tampering by the adversary on the idealized model by virtue of the fact that there is no adversary. Thus, the idealized model can be used as a "golden" model against which the checker can compare the state of the actual model. For this, assume the existence of function f that checks whether a certain state in the actual model matches a

[2]Note that actions i5 and i6 do not constitute splicing attacks since the copied values are inaccessible to the user due to the XOM ID tags on the registers.

certain idealized model state:

$$\begin{aligned} f : & f(Actual\ Model\ State, Idealized\ Model\ State) \\ & = \{true, false\} \end{aligned}$$

Tamper resistance is verified by exploring both the idealized and actual model states simultaneously. This involves concatenating the idealized model state with the actual model to create the "joint" state. The model labels every action in the actual model as either a user action or an adversary action. User actions are ones that would be performed by a user, and as a result, these actions have analogies in the idealized model. All other actions are considered adversary actions and have no analog in the idealized model. User actions affect the state of the idealized and actual portions of the model state, while adversary actions only affect the actual portion of the joint state. The model checker applies f to each new joint state created to verify consistency. If this check above holds for all the states found, then the adversary is not able to make the actual state inconsistent from the idealized state, which leads to the conclusion that the adversary was not able to tamper with the user's data.

The merging of the models must be done in a way that does not restrict the state exploration of either model, otherwise this may result in the Murφ failing to find states where the two models might have been inconsistent. Murφ rules have a guard condition that states when a particular rule can be applied. The idealized model is what the user should think she is running on, so user actions in the joint model derive their guards from the idealized model. The one caveat is all user actions can only execute when the actual model is in *user_mode*, so this check is added to all user guards. The bodies of those rules are a combination of the actions that modify the idealized model and the actual model in parallel.

Adversary actions have no corresponding actions in the idealized model and so they are guarded by elements from the actual state. Naturally, the adversary actions only modify the actual state of the model. Because of this, adversary actions change the actual state but leave the idealized state unchanged. Below is a section of the Murφ description that combines the same user action from the actual and idealized models given in Sections 5.4.3 and 5.4.4. Because the states of the models are now combined into the same name space, elements from the idealized model are prefixed with an "i", while elements in the actual model are prefixed with an “a”:

```
Rule "User secure store"
  -- only guarded by idealized state
  !isundefined(ireg_i.data) &
  -- must be in user mode
  mode = user_mode
==>
Var
  acache_l : cache_range;
Begin
  -- actual model part
  if (areg_i.tag != user) then
    reset();
  endif;
  acache_l = find_data(addr_j);
  if (!isundefined(acache_l)) then
    acache_l.data := areg_i.data;
    acache_l.addr := addr_k;
    acache_l.tag := areg_i.tag;
  else
    -- cache miss (code not shown)
  endif;
  -- idealized part
  imem_j.data := ireg_i.data;
endrule;
```

Now that a method for combining the two models has been established – all that is left is to defined f. The reference XOM CPU is based on a RISC-like load/store architecture. As a result, assembler instructions either move data between registers and memory, or they perform logical or arithmetic operations on register values. As such, from the point of view of a running program, the only state that needs to be consistent is the register state since that is what is used to do any computation that could produce output. Because of this property, the function f turns out to be very simple:

$$\begin{aligned}
&f : (S_{actual}, S_{idealized}) \rightarrow \{true, false\},\\
&\text{if } \forall\ S_{actual}.r_i.id = user \wedge\\
&\qquad S_{actual}.r_i.data = S_{idealized}.r_i.data\\
&\text{then } f = true\\
&\text{else } f = false
\end{aligned}$$

The XOM architecture only allows programs to read registers that are tagged with their XOM ID. The above f is true if registers tagged with the user's ID, and thus are accessible to the user, have the same data in both the actual and idealized model. If a XOM machine was built on top of a CISC machine, where arithmetic or logical operations may be performed directly on memory values, then values in memory and cache must always be consistent as well. This does not preclude a definition for f, but would make the definition more complex.

One of the limitations is that the model must approximate the real machine by scaling down the number of elements in the model to limit the state space. The verification was performed on a scaled down model with 3 memory locations, 3 cache locations, 3 register elements, 2 user data values and 1 adversary data value. All three attacks detailed in Section 5.4.5 require a minimum of two user data values, in the case of a splicing or replay attack, or a user value and an adversary value in the case of a spoofing attack. Thus, the largest number of different data values we need to support in our models is three. As a result, the verification requires three elements in each storage level so that the adversary can move all possible values around without overwriting some. Slightly larger models were checked before the machine on which the model checker was run ran out of memory. None of these turned up any errors that were not found in the 3 element models. With more computational resources, larger models could be checked.

5.4.7 Verification Results

In performing our verification, we found a way for an adversary to perform replay attacks on memory values. With our tool, we were able to verify that our solution to the error is correct. We then searched the XOM architecture for extraneous actions and were able to find one. Finally we checked for liveness guarantees and were able to show that when certain constraints were placed on the operating system, the user could be guaranteed forward progress. The models ran for approximately 4-6 hours on a Sun Workstation with 8GB of memory and 1Ghz Ultra SPARC 3 processors.

We were able to find an exact sequence of events that allow adversary to replay values in memory. This existence of this attack was also suggested in [20, 39, 56]. Our method also helped us find and implement a safe solution to the memory replay problem.

We start by noting that Section 5.3 indicates that a hash of a memory region can be used to protect that region from replay. We model this hash by creating a second memory array that shadows the memory in the actual model. Because the hash is meant to be kept in a replay-proof register, we make this shadow memory inaccessible to any of the adversary actions. A *calculate hash* function models the calculation of the hash by copying the contents of memory in the actual model into the shadow. A *verify hash* function then checks the contents of the shadow memory against the contents of memory in the actual model and an exception is thrown if the two do not match.

However, it is not clear when the *calculate hash* and *verify hash* functions should be invoked. Since the hash updates would be expensive in reality, we decided to try to reduce the frequency of the updates. We take advantage of the fact that cache locations are on chip and thus cannot be modified by the adversary without detection. We implemented a model where the hash is only updated when a cache line is flushed to memory and verified when a memory location was read into the cache. While this appears to be all right at first glance, Murφ was able to find an attack where the adversary would invalidate cache lines before they were flushed to memory so that the old value in memory became the current value. It is possible to exploit this vulnerability as shown in Table 5.3.

Even though the hash matches the memory value A, the last value written was B, so the adversary has successfully performed a replay attack. This problem arose because the

Action	**$**	**M**	**H**
1. User writes A to cache.	A	$\varnothing$	$\{\varnothing\}$
2. Cache is flushed to memory.	$\varnothing$	A	$\{h(A)\}$
3. User writes B to cache.	B	A	$\{h(A)\}$
4. Adversary invalidates cache.	$\varnothing$	A	$\{h(A)\}$

Table 5.3: Vulnerability from Caching MACs. Updates of the hash cannot be delayed as shown here with **$** is the contents of the cache, **M** is the contents of memory, and **H** is the value of the hash.

write to an address, which occurs when the value is written into the cache, is not atomic with the update of the hash, which occurs when the value is flushed to memory. With Murφ we are able to show that against an adversary who cannot invalidate cache lines, delaying the update of the hash in this way is safe. However, since many architectures support his operation, this optimization is generally unsafe. As a result, we must be sure the hash is updated whenever a value is written to the cache.

Since reducing the frequency of hash operations failed, we try instead to reduce the cost of each hash calculation by using an incremental hash. An incremental hash uses a function to add and remove elements from a hash efficiently. An example of an incremental hash that uses the exclusive-or function is given in [7] and an implementation appears in [20]. Such a hash would make updates to the hash more efficient as we would not need to read all memory locations to recalculate the hash. Instead, we simply remove the old value from the hash, and add the new one. Again, we were able to find a weakness. Essentially, Murφ exploited the fact that when we read in the old value to remove it, we do not actually verify that the old value is the correct value. A clever adversary can insert a different value at this point to create havoc. For example, the adversary could insert a value that will cancel out the value that the user is about to write, thus leaving the hash unchanged. The adversary is then free to replay an old value, since the hash was not updated as shown in Table 5.4.

Again, though the last value written is actually A, the hash for B validates correctly. It seems that the only way to defeat this is to verify that the value being removed is the correct value, which requires reading in all of memory when updating the hash. Unfortunately, this negates the benefit of the incremental hash.

From these two failed hash implementations, we were able to create a successful hash

Action	**$**	**M**	**H**
1. User writes A to cache.	A	$\varnothing$	$\{h(A)\}$
2. Cache is flushed to memory.	$\varnothing$	A	$\{h(A)\}$
3. User writes B to cache.	B	A	$\{h(A) - h(A) + h(B)\} = \{h(B)\}$
4. Cache is flushed to memory.	$\varnothing$	B	$\{h(B)\}$
5. Adversary replays A in memory.	$\varnothing$	A	$\{h(B)\}$
6. User writes A to cache.	A	A	$\{h(B)\}$
7. Cache is flushed to memory.	$\varnothing$	A	$\{h(B) - h(A) + h(A)\} = \{h(B)\}$
8. Adversary replays B in memory.	$\varnothing$	B	$\{h(B)\}$

Table 5.4: Vulnerability from Incremental Hashes. An incremental hash cannot be used as shown here with **$** is the contents of the cache, **M** is the contents of memory, and **H** is the value of the hash.

implementation. The first optimization failed because hash calculations are not atomic with updates to memory. An adversary may perform a replay attack by invalidating values in memory. One solution is to limit the ability of the adversary to invalidate cache lines by only supporting a cache flush function during regular operation. The other solution, is to call the *calculate hash* function every time the user writes values to the cache. Similarly, the *verify hash* function must be executed every time a value is read into the cache from memory and before we recalculate a new hash as shown by the second failed optimization. Unfortunately, this method may be inefficient because the entire memory region must be read each time the hash is calculated or verified.

To detect extraneous actions, we removed actions from the model to see if they were necessary for security. With Murφ we found one action in the model that appeared to be extraneous. When the user loads data from memory, it is not necessary to check that the data is actually encrypted with the user's key. It is sufficient to simply tag the register that the data is stored to with the key that the data was encrypted with. In other words we can change the user action to:

```
4. Read memory address $j$ into register $i$:
   if $j$ is in the cache such that $\exists\ c_l.a = j$
   then load from cache $r_i = \{d = c_l.d, t = c_l.t, k = \varnothing, h = \varnothing\}$
   else load from memory if $m_j.h \neq j$
        then reset
```

else pick an $l \in [0 \ldots y] : c_l.a = \varnothing \rightarrow$
$c_l = \{d = m_j.d, a = j, \boldsymbol{t = m_j.k}\},$
$r_i = \{d = m_j.d, \boldsymbol{t = m_j.k}, k = \varnothing, h = \varnothing\}.$

Later, when the user tries to use the data with a `use` operation in Table 5.1, the machine will check the tag anyway. Though unnecessary for security, specifying the key in the instruction does make the hardware more efficient. The XOM architecture allows encrypted programs to select whether data stored to memory should be encrypted or in plain text. Having the program specify the whether the data it loads is encrypted or not, saves the hardware from having to maintain that information.

5.5 Attacks not Covered by XOM

There are some attacks that XOM has no defense against. While these attacks provide the adversary with some information, it us unclear how they could be used to by the adversary to achieve one of the goals outlined in Section 5.1.

Because the XOM machine allows an untrusted operating system to manage resources, programs running in the XOM machine are vulnerable to a malicious operating system mounting a denial of service attack. Since the operating system has full control over the allocation of resources, it can prevent any or all programs from making forward progress by simply denying them resources such as memory or CPU time. It is fundamentally impossible to have a malicious operating system and still guarantee forward progress for programs who's resources are managed by that operating system.

A malicious operating system can also gather limited information about an application. Programs running on a XOM processor are vulnerable to frequency analysis. An adversary who watches cipher text values in memory can learn how often a particular value is stored to a particular address location. This attack can be defrayed to a certain extent by using address and even time dependent salts to randomize the cipher texts in memory, but the full implications of using this strategy are beyond the scope of this thesis.

Finally, a malicious operating system can obtain a full address trace of every memory access. The operating system simply locks all pages in the TLB to force every access to memory to be caught by a TLB miss. There exists mechanisms to implement "Oblivious

RAM" [25], which adds extraneous loads and stores to the instruction stream that are indistinguishable from the actual memory accesses. However, these loads and stores add a non-trivial amount of overhead to the program.

5.6 Additional Security Issues

There are two other security issues that we must address. The first deals with key revocation for XOM. A system that is "brittle" will fail of any one its components is compromised. Here we discuss methods to minimize the damage done if a XOM processor's master secret is ever revealed. The goal is to provide *forward security* so that software distributed after the compromise to other XOM processors, will remain safe.

The other issue is one of privacy for the owner of the XOM processor. Because each processor has a unique private/public key pair, the public key could be used as an identifier to track the purchases the owner makes.

5.6.1 Key Revocation

No matter how much effort is made to make a system secure, there is always the possibility of a compromise. If compromised, the XOM architecture requires a way of isolating faults so that they don't cause more damage. If a XOM processor is ever compromised in such a way that it's master secret is revealed, then all software that was ever encrypted for that processor is immediately compromised. However, to ensure *forward security* — that is that the key exposure doesn't harm future pieces of software, we must have a way of *revoking* the compromised key so that distributors will no longer encrypt compartment keys for the compromised processor. There are several requirements for this to occur.

First, authorities must be aware what processor has been compromised. If the attacker who compromises the processor broadcasts the master secret, then authorities will immediately know which key to revoke. If on the other hand, the attacker does not broadcast the key, but instead simply posts the decrypted compartment keys or decrypted binaries, then authorities will have a harder time detecting which processor is compromised. Watermarking keys or binaries specifically for each processor would help identify the compromised processor, but would interfere software merchants being able to distribute a single encrypted binary as mentioned in Section 3.1.3. The problem of identifying a compromised processor remains an area for future research.

Second, once the compromised processor has been identified, a *key revocation scheme* must be in place to inform software distributors not to encrypt any more compartment keys for that processor. To do this, authorities will create a message containing a list of keys being revoked and then sign list that with their own key so that the revocation message cannot be forged. There exist a variety of key revocation schemes that can used to augment this operation [3, 26, 33, 46, 47].

5.6.2 Privacy

The XOM system for software distribution poses a potential problem for privacy since the public key of a processor can be used as an identifier for the owner of that processor. Thus, if software distributors collude, they can figure out what purchases a particular individual is making. To solve this problem, XOM processor can be enhanced with the ability to create ephemeral keys and then sign them with a common master signing key. Thus, every purchase a customer makes with the same XOM processor will be with a different key, making it impossible to track the customer's purchases. The ability to create an ephemeral key can be embedded in a trusted piece of software that is protected from tampering by the XOM system. However, the ability to create ephemeral keys prevents authorities from identifying a compromised processor because all processors must share a common signing key. This makes the system brittle since a single compromised processor will result in a large number of compromised processors. This number can be reduced by having a set of ephemeral keys thus reducing the number of processors sharing the same key.

5.7 Related work

There has been much work on the verification of both security systems and hardware systems. Theorem proving is a formal verification alternative to model checking. The design of IBM 4758 Secure Coprocessor [58] used theorem proving techniques to verify its security. While theorem proving is not restricted to a finite number of states, it significantly more difficult and time consuming to use.

The idea of performing verification by checking for consistency between a higher level model and a lower level model has been detailed in work on refinement maps [2]. Our technique differs from refinement maps in that we ensure that every transition in the idealized model has an existing transition in the actual model, while a refinement map specifies the converse. There has also been some work that verified security by asserting an equivalence between an idealized model and a model with certain actions available to the adversary. One specification method using equivalence between a realistic model and an idealized attack-impervious model is outlined in [40], with related ideas presented earlier in [1]. Prior work on CSP and security protocols, also uses process calculus and security specifications in the form of equivalence or related approximation orderings on processes [52, 54].

Lastly, none of the techniques presented in this thesis are Murφ specific. A variety of other model checkers such as SPIN, TLA+, or SMV could have been used [29, 36, 43].

5.8 Summary

This chapter illustrates the techniques we envision the adversary to circumvent the security provided by XOM. Naturally, one cannot think of every possible attack so we turned to a formal verification method to *automatically* generate attacks to verify the system. In using a model checker, we found that an effective way to check for tampering is to define two models: one with an adversary and one without. We detect tampering by checking for inconsistency in user data between the two models. We found that it is difficult to optimize the protection of memory from replay attacks in XOM. Two optimizations we tried failed, though one would be possible if the operating system is not allowed to invalidate data in the caches, but only to cause it to be flushed to memory.

Chapter 6

Conclusions and Future Work

With the wide spread use and adoption of the Internet, more and more media and intellectual property is being distributed digitally. As a result, there is a growing interest and need for copy and tamper-resistant software to enable the safe distribution of digital content. Fundamentally, these systems rely on hiding secrets from observation and protecting them from unauthorized modification by an adversary. On-chip circuitry is both difficult to observe and difficult to modify, making it inherently more tamper-resistant than software. Successful systems work by hiding at least one secret in a chip, exploiting the tamper-resistant properties of hardware. However, it is possible to observe the data that passes through the pins of a chip, and to modify the data that is stored in the memory chips in a computer. This means that we need to rely on cryptographic algorithms to protect data that leaves the chip.

This thesis presents XOM, an eXecute Only Machine that uses on-chip tags and cryptographic mechanisms for off-chip data to provide compartments that applications can use to execute protected code. While the cryptographic operations are not cheap, since they are only used for data that is flowing off-chip the added overhead is small. Off-chip communication is sufficiently slow (hundreds of processor cycles) that the tens of cycles needed for the cryptographic operation is a small, 10% change in current operation cost. Thus, through a combination of cryptographic mechanisms, such as ciphers and MACs, and architectural mechanisms, such as tags and caches, we are able to provide both performance and security.

We used a model checker to explore possible exploits that may violate the security of the XOM containers. The most difficult attack to protect against is a memory replay attack. In this attack, an adversary records values of memory, and reuses them at a later point in time, causing the victim to use an old, stale value. While protecting against this attack is possible, it is difficult to do so in a an efficient manner.

Creating XOM processors also raises a number of other issues, since in such a machine, the applications do not trust the operating system with its code or data, but still need it to manage the machine's resources. We were able to show that is not hard to separate the trust component from the other aspects of an operating system. With a small number of modifications, we were able to port IRIX to create XOMOS, an untrusted operating system that manages resources on the XOM hardware. A malicious version of XOMOS cannot tamper or observe the execution of any of its user processes. In developing XOMOS from IRIX, we found most of the modifications occurred in the low-level portion of the operating system, and does not affect the application binary interface with the exception of two new system calls. This leads us to believe that a variety of operating systems could be ported to XOM architectures with the same ease.

While the hardware overheads of memory encryption are on the order of 10%, and our operating system overheads appear to be substantial, our simulations show that the end-to-end application overhead is actually quite modest. This is because the overheads introduced by XOM occur on events which may not be very common. The hardware overhead of memory encryption is incurred on cache misses and the operating system overheads are incurred mainly on system calls. On the applications studied, we found the overheads to be within 5% of the execution time.

This dissertation demonstrates that through a combination of architectural and cryptographic methods, it is possible to design a system, with reasonable performance, that supports copy and tamper-resistant software that is secure in the face of an adversarial operating system.

6.1 Future Work

Our work has shown that building a machine where trust is maintained in the hardware is not difficult, and these machines can export nearly the same operating system API that programmers are familiar with. There are really two large remaining questions: can a XOM processors really be built, and how will application programmers take advantage of these features.

This dissertation lays out extensions that can be added to the ISA of a processor. However, these extensions have implications on the processor architecture that will benefit from further study. The interactions XOM would have on out-of-order architectures, or on simultaneously multi-threaded architectures for example, are not well understood. Because the work was done in simulation, micro-architectural issues that may arise in a silicon implementation would not be apparent. Further study in this area would reveal both the complexity of the hardware implementation as well as more detail on what the hardware overheads are.

In this work, we have made a start at identifying how programmers should use the features of XOM to reduce the performance impact on applications that are ported to XOM. However, this dissertation does not explore the possible new applications that may arise with facilities such as those provided by XOM. In the past, there did not exist any trusted hardware platforms. Given that XOM and other platforms now exist, a study into techniques of effectively using the features they provide would be of interest.

Finally, this thesis examines the XOM architecture from a technical stand point. However, with the recent announcements of industrial initiatives in trusted computing, such as TCPA and Palladium, we realize that such systems must be integrated into a complex commercial setting. Tamper and copy-resistant systems try to address what was for the most part a legal issue, with a technical solution. However, flexibility for different licensing agreements, providing techniques for upgrading or recovering XOM systems, limiting damage from compromised systems, as well as privacy and ease of use will be issues that will help determine the viability of trusted computing. Ultimately, even if as researchers we can convince the computing industry that trusted computing is feasible, it remains to be seen whether that industry can convince consumers that trusted computing is beneficial to them.

Bibliography

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 143:1–70, 1999. Expanded version available as SRC Research Report 149 (January 1998). 5.7

[2] M. Abadi and L.Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. 5.7

[3] W. Aiello, S. Lodha, and R. Ostrovsky. Fast digital identity revocation. In *Proceedings of CRYPTO'98*, 1998. 5.6.1

[4] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *IWSP: International Workshop on Security Protocols, LNCS*, 1997. 5.2

[5] ANSI X9.17 (Revised). American national standard for financial institution key management (wholesale). American Bankers Association, 1985. 2.2

[6] 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, 2001. 1, 3.6.2

[7] M. Bellare, R. Guerin, and P. Rogaway. XOR MAC's: New methods for message authentication using finite pseudorandom functions. *CRYPTO'95, Lecture Notes in Computer Science*, 963, 1995. 5.4.7

[8] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of The 16th ACM Symposium on Operating Systems Principles*, Oct. 1997. 3.2

[9] Business Software Alliance, 2003. `http://www.bsa.org`. 1

[10] CERT/CC. Overview incident and vulnerability trends. Technical report, CERT Coordination Center, Apr. 2002. 3.1.2

[11] S. Chari, C. Jutla, J. Rao, and P. Rohatgi. Towards sound approaches to counteract power analysis attacks. In *Proceedings of CRYPTO'99: 19th Annual International Cryptology Conference*, volume 1666, pages 398–412, Aug. 1999. 5.2

[12] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, July 1997. 3.6.2

[13] J. Daemen and V. Rijmen. AES proposal: Rijndael. Technical report, National Institute of Standards and Technology (NIST), March 2000. Available at `http://csrc.nist.gov/encryption/aes/round2/r2algs.htm`. 2.2, 3.3

[14] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–5, 1992. 5.4, 5.4.1

[15] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983. 5.4

[16] T. ElGamal. A public-key cryptosystem and signature scheme based on discrete logarithms. In *Advances in Cryptography: Proceedings of CRYPTO 84*, pages 10–18, 1985. 2.3

[17] P. England, J. DeTreville, and B. Lampson. Digital rights management operating system. U.S. Patent 6,330,670, Dec. 2001. 3.6.3

[18] P. England, J. DeTreville, and B. Lampson. Loading and identifying a digital rights management operating system. U.S. Patent 6,327,652. Dec. 2001. 3.6.3

[19] S. Freund and J. Mitchell. A type system for object initialization in the java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, Nov. 1999. 5.4.2

[20] 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, 2003. 5.3, 5.4.7, 5.4.7

[21] 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. 1, 3.6.1

[22] T. Gilmont, J. Legat, and J. Quisquater. Hardware security for software privacy support. *Electronics Letters*, 35(24):2096–2097, Nov. 1999. 1, 3.6.1

[23] T. Gilmont, J.-D. Legat, and J.-J. Quisquater. Enhancing the security in the memory management unit. In *Proceedings of the 25th EuroMicro Conference*, volume 1, pages 449–456, Sept. 1999. 3.6.1

[24] R. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):35–45, June 1974. 3.2

[25] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 43(3):431–473, May 1996. 5.5

[26] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and communitative hashing. In *Proceedings of DARPA DISCEX II*, June 2001. 5.6.1

[27] J. Heinrich. *MIPS R10000 Microprocessor User's Manual*, 2.0 edition, 1996. 3, 3.4

[28] S. A. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, Feb. 1998. 3, 3.4

[29] G. Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. 5.7

[30] M. Horowitz, M. Martonoisi, T. C. Mowry, and M. D. Smith. Informing memory operations: Memory performance feedback mechanisms and their applications. *ACM Transactions on Computer Systems*, 16(2):170–205, May 1998. 3.2

[31] A. Huang. Keeping secrets in hardware: The microsoft XBox case study. Technical Report 2002–008, Massachusetts Institute of Technology, May 2002. http://web.mit.edu/bunnie/www/proj/anatak/AIM-2002-008.pdf. 3.1.1

[32] IBM Corporation. *IBM PCI Cryptographic Coprocessor: General Information Manual*. 3.6.1

[33] P. Kocher. On certificate revocation and validation. In *Proceedings of the International Conference on Financial Cryptography*, volume 1465 of *Lecture Notes in Computer Science*, 1998. 5.6.1

[34] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. `http://www.ietf.org/rfc/rfc2104.txt`, Feb. 1997. 2.4, 5.3

[35] M. Kuhn. The TrustNo1 cryptoprocessor concept. Technical Report CS555, Purdue University, Apr. 1997. 1, 3.6.1

[36] L. Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2002. 5.7

[37] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 10(4):265–310, 1992. 3.2

[38] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001. 5.4.1

[39] 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. 1, 5.4.7

[40] P. Lincoln, M. Mitchell, J. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In M. Reiter, editor, *Proc. 5-th ACM Conference on Computer and Communications Security*, pages 112–121, San Francisco, California, 1998. ACM Press. 5.7

[41] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES-modes of operations: CTR-mode encryption. In *Symmetric Key Block Cipher Modes of Operation Workshop*, Baltimore, Maryland, USA, 2000. 3.3

[42] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec. 1995. 4.5.3

[43] K. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991. 5.7

[44] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murphi. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–153, 1997. 5.4.1

[45] S. Morioka and A. Satoh. A 10 gbps full-AES crypto design with a twisted-BDD S-Box architecture. In *Proceedings of the 2002 International Conference on Computer Design*, pages 98–103, Sept. 2002. 3.3

[46] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 internet PKI online certificate status protocol - OCSP. IETF RFC 2560, June 1999. 5.6.1

[47] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings 7th USENIX Security Symposium*, Jan. 1998. 5.6.1

[48] OpenSSL, 2000. `http://www.openssl.org`. 3.3, 4.5.3

[49] S. Polonsky, D. Knebel, P. Sanda, M. McManus, W. Huott, A. Pelella, D. Manzer, S. Steen, S. Wilson, and Y.Chan. Non-invasive timing analysis of IBM G6 microprocessor L1 cache using backside time-resolved hot electron luminescence. In *Proceedings of the IEEE International Solid-state Circuits Conference*, pages 222–224, Feb. 2000. 3.1.1, 5.2

[50] 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, 1978. 2.3

[51] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. A block-cipher mode of operation for efficient authenticated encryption. In *Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS-8)*, pages 196–205, 2001. 3.3

[52] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *CSFW VIII*, page 98. IEEE Computer Soc Press, 1995. 5.7

[53] J. Saltzer and M. Schroeder. The protection of information in computer systems. *IEEE*, 63(9):1278–1308, Sept. 1975. 2, 2.1

[54] S. Schneider. Security properties and CSP. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996. 5.7

[55] SGI IRIX 6.5: Home Page, May 2003. `http://www.sgi.com/software/irix6.5`. 4

[56] W. Shapiro and R. Vingralek. How to manage persistent state in DRM systems. In *Digital Rights Management Workshop*, pages 176–191, 2001. 5.4.7

[57] V. Shmatikov and J. Mitchell. Analysis of a fair exchange protocol. In *Seventh Annual Symposium on Network and Distributed System Security*, pages 119–128, 2000. 5.4.1

[58] S. Smith, R. Perez, S. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *Proceedings of the22nd National Information Systems Security Conference*, Oct. 1999. 5.7

[59] S. W. Smith, E. R. Palmer, and S. Weingart. Using a high-performance, programmable secure coprocessor. In *Financial Cryptography*, pages 73–89, Feb. 1998. 1

[60] U. Stern and D. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995. 5.4.1

[61] The Trusted Computing Platform Alliance, 2003. `http://www.trustedpc.com`. 3.2, 3.6.3

[62] J. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. Technical Report CMU–CS–91–140R, Carnegie Mellon University, May 1991. 1, 3.2, 3.6.1

[63] VMWare, Inc., 2003. `http://www.vmware.com`. 3.2

[64] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of ACM SIGMETRICS'96: Measurement and Modeling of Computer Systems*, pages 68–79, 1996. 3.2

[65] B. S. Yee. A sanctuary for mobile agents. Technical Report CS97-537, University of California at San Diego, Apr. 1997. 3.6.1