Architectural Support for Copy and Tamper Resistant Software

David Lie Chandramohan Thekkath* Mark Mitchell Patrick Lincoln†
Dan Boneh John Mitchell Mark Horowitz

Computer Systems Laboratory
Stanford University
Stanford CA 94305

ABSTRACT

Although there have been attempts to develop code transformations that yield tamper-resistant software, no reliable software-only methods are known. This paper studies the hardware implementation of a form of execute-only memory (XOM) that allows instructions stored in memory to be executed but not otherwise manipulated. To support XOM code we use a machine that supports internal compartments—a process in one compartment cannot read data from another compartment. All data that leaves the machine is encrypted, since we assume external memory is not secure. The design of this machine poses some interesting trade-offs between security, efficiency, and flexibility. We explore some of the potential security issues as one pushes the machine to become more efficient and flexible. Although security carries a performance penalty, our analysis indicates that it is possible to create a normal multi-tasking machine where nearly all applications can be run in XOM mode. While a virtual XOM machine is possible, the underlying hardware needs to support a unique private key, private memory, and traps on cache misses. For efficient operation, hardware assist to provide fast symmetric ciphers is also required.

1. INTRODUCTION

Software piracy is a significant economic problem. The Business Software Alliance, for example, estimates that piracy cost the software industry 11 billion dollars in 1998 [1]. In addition to legislative and law enforcement efforts, it appears useful to develop technical methods for combating software piracy. In principle, a software vendor might like to sell a single copy to a single user. However, it is generally easy to copy and distribute digital information. While software watermarking might aid in prosecuting pirates once they are detected, methods aimed at identifying copies will not prevent copying itself. We therefore investigate a method for preventing unauthorized *execution* of software, regardless of the number of copies made. Our method also prevents any software customer from examining the executable code itself, thereby protecting the algorithms and computational methods incorporated into the code.

The protection model studied in this paper is called "XOM", pronounced "zom", an acronym for eXecute-Only Memory. The mechanism, also described in [4] and related to concepts presented in [2, 9, 10, 11], is based on the idea that code stored on disk or other media can be marked "execute-only." Execute-only code, which is stored in an encrypted form, can only be decrypted by the instruction-loading path on the main processor chip, thereby preventing any user of the computer from examining the actual instructions. This simple idea requires additional mechanisms in order to preserve code security. In particular, data written during execution must be encrypted, in order to prevent an adversary from reverse-engineering the code. This is especially tricky for data stored in registers that are accessible to the operating system after an interrupt, since encrypting every write to a register and decrypting every read from a resister is prohibitively slow. In addition, merely encrypting data may not prevent sophisticated attacks that involve permuting the contents of encrypted data locations.

The aim of this paper is to establish that hardware implementation of a XOM-like execution model is feasible in practice. To support a secure execution environment, we use a form of *compartment* to isolate independent software applications running on the same processor [23]. Each compartment is built from a *session key*, used to encrypt the associated data. Regular unencrypted code can run in the *unprotected* or *null* compartment that has no key associated with it.

Protection is provided using a combination of public-key and symmetric-key cryptography [19]. In brief, each XOM chip contains the private decryption key of a public-key encryption pair. The corresponding encryption key is made public, so that anyone can encrypt code for this chip but only the chip itself contains the key required for decryption. However, if the entire instruction stream of a XOM application were encrypted using public-key cryptography (such as RSA [19]), instruction loading would be prohibitively slow. Therefore, the header block of each XOM applica-

*Compaq Systems Research Center
†SRI International

ASPLOS IX 11/00 Cambridge, MA, USA
 ISBN 1-58113-317-0/00/0011...$5.00

tion contains the encryption key of some faster, symmetric key encryption scheme (such as DES [20]) and the remainder of the application is encrypted with this key. Since each application will have a distinct symmetric key, the symmetric key embedded in an application can be used as the identifier for the compartment in which the code is executed.

The next section of the paper (Section 2) describes the abstract XOM machine architecture in more detail, explaining how the machine internally protects information in compartments. We begin with a description of a basic machine that does not use main memory and does not take interrupts. We then extend this model to support both of those.

Section 3 looks at some of the security issues with XOM, and addresses various ways that information might leak out of the compartments that we have constructed. Our security model does not trust external memory or the operating system that manages the execution of the XOM program. Providing guarantees in such a model is a particularly challenging problem.

Section 4 describes possible implementations of a XOM processor. We abstract many functions into a XOM Virtual Machine Monitor that could be implemented in microcode or software. Section 5 discusses hardware necessary to make the performance of XOM reasonable and Section 6 evaluates the overhead of this hardware. Finally, we close the paper with a summary of the XOM architecture in Section 7.

2. THE ABSTRACT MACHINE

The abstract XOM machine has three principal tasks: decoding the session key using an asymmetric cipher, decoding the instruction stream from external memory using the session key, and providing compartmentalized storage for the XOM code to use. The most interesting task is providing this storage, so we will begin with that problem.

Our basic approach to secure storage is to tag all data with a XOM identifier. This identifier is shorthand for a session key and is an index into a table, called the *session key table*, that maps XOM identifiers to a decrypted session key. Programs that run in the clear without encrypting their code belong, by default, to the null principal, and have a XOM identifier of zero.

The size of the session key table and tags depend on the number of concurrently executing principals that can have data in the machine. In the simplest machine, the identifier can be one bit and the table contains only two entries: the session key of the currently active XOM program, and the null session.

At any time, only one principal is executing and thus, only one XOM identifier is active. We refer to this principal as the "active principal". The session key and the corresponding XOM identifier belonging to the active principal are called the "active key" and the "active XOM identifier". When data is produced by the program, the abstract machine automatically tags it with the active XOM identifier. When data is read, the tag on the data is compared with the active XOM identifier. If the comparison succeeds, the read is allowed, otherwise the read causes an exception. Thus, no principal can access the data of another principal—the tags create the compartments that provide the isolation required for security.

In addition to protecting the data, the abstract machine provides two instructions: *enter_xom* and *exit_xom*. XOM code is preceded by an *enter_xom* instruction, where the source register holds the starting memory address of the encrypted session key for this XOM code. The instruction indicates to the XOM machine that all following code belongs to a principal associated with the session key. The machine checks to see if the session key has already been decoded. If the session key is already in the table, it sets the active identifier to that entry and starts to fetch XOM code. If no entries in the session key table match, the machine chooses a free entry, sets the active identifier to this entry, runs the asymmetric decryption algorithm on the key and then enters the key pair in the table before fetching the first XOM instruction.

While in XOM mode, all instructions are decrypted using the session key before they are placed in the instruction stream for execution. Other than decrypting the instructions and tagging the data, the machine operates like a conventional machine. There are two kinds of events that will cause the active identifier to change. The normal event is the execution of an *exit_xom* instruction. This instruction changes the active identifier back to null, and the machine stops decrypting instructions. The "abnormal" event occurs when a trap or interrupt is taken. In this case, an implicit *exit_xom* instruction is executed before the instructions from the handler are executed.

To complete the abstract machine we need two additional instructions to allow communication between principals. This communication is provided by the *mv_to_null* and *mv_from_null* instructions. These instructions allow a controlled way to change the tags associated with a piece of data. The *mv_to_null* instruction takes data that is tagged with the active XOM identifier and changes the tag on it to the null principal. After this instruction is executed, access to the data by the original principal results in an exception. Executing this instruction on data that is not originally tagged with the currently executing principal also results in an exception. The *mv_from_null* instruction changes data tagged by the null principal to the active XOM tag. Once again data has to be originally tagged with null before this instruction can be executed.

These instructions and the semantics of tagging guarantee that when a principal reads data it will only get values that were either created by itself or explicitly brought into its compartment. The simplicity of this compartment-based protection method appeals to us—the basic tag approach has no special cases and does not depend on the specifics of the processor execution model.

In summary, the abstract machine described thus far provides the following four mechanisms.

1. A scheme to decrypt symmetric keys using the private half of a public key pair.
2. Facilities for decoding the program code using a decrypted session key.
3. Instructions for entering and exiting XOM mode, with traps and interrupts causing an implicit exit from XOM mode.
4. A data tagging scheme that prevents principals from accessing data belonging to other principals.

These mechanisms would be sufficient if either external memory was secure, or we did not need to use it. Since we acknowledge that memory can be probed, and that it will

be necessary to use external memory in most cases, we will need to deal with its insecurity. We again use cryptography, this time to extend the compartments to external memory.

2.1 External Memory

To extend compartments to external memory, we encrypt the tagged data using the appropriate session key when it leaves the confines of the abstract machine. Unfortunately, this is not sufficient to provide the same guarantees that the internal tags provide—we still need a mechanism to prevent access to data that was tampered with while it was in memory. To allow the machine to check the ownership as well as protect the data, a secure hash is associated with all values stored in memory. If an external agent tampers with the data, then the hash will not match and the instruction will cause an exception.

To support the needed functionality, a XOM machine provides two pairs of instructions to move data between external memory and the machine: *store_secure*/*load_secure* and *store_from_null*/*load_from_null*.

The *store_secure* and *load_secure* act like normal load and store instructions for the XOM process. They are used by the currently executing XOM program to move data between memory and registers that are in the active compartment. Thus, if the register named in a *store_secure* is tagged with an identifier other than the active one, the instruction raises an exception. Similarly, if the *load_secure* memory data hash does not match the data the instruction will cause an exception. The *store_secure* instruction encrypts data with the active session key and creates a hash to check for ownership. The *load_secure* instruction takes a destination register and memory location as arguments. It decrypts the contents of memory using the active session key, verifies the hash, loads the decrypted value into the register and changes the tag on the register to the active identifier. If there is a mismatch in the hash, the instruction will cause an exception.

The *store_from_null* and *load_from_null* instructions are additional instructions to make it possible for a XOM program to read and write data in the null compartment. These instructions neither encrypt or decrypt their data, nor have secure hashes associated with them. They behave just as load and store instruction do on a conventional processor. In a XOM machine these instructions do not change the ownership of the data, and so the data to be stored on a *store_from_null* instruction must be tagged with the null identifier, and the data left by a *load_from_null* is tagged with the null identifier. Again, a *store_from_null* on a register that is not tagged with null results in an exception.

As we will describe in the implementation section, to reduce both the time and bit overhead of memory operations, the encryption and hash generation are pushed through the memory hierarchy, so it is only done when the data leaves the processor and enters insecure memory. The on-chip caches use tags, just like the registers, to provide access control.

2.2 Supporting Interrupts

To allow XOM code to be interrupted and restarted we need to remove any dependency between the operating system's resource management responsibilities and compartment security. On an interrupt, an untrusted operating system must be allowed to save the register state of a XOM process without being able to interpret or leak the contents of the registers. The current machine does not have this capability, since a process can access protected registers only if the contents of the register is in its compartment. We need to add two more instructions—the *save_secure* and *restore_secure* instructions—to provide this ability.

The *save_secure* and *restore_secure* instructions are used by the currently executing program to move data that it does not own and does not belong to the null principal. The *save_secure* instruction takes the contents of a register and creates an encapsulated version of this data that another principal can move but cannot manipulate. It first encrypts the register contents with the key of the register owner, and then calculates a hash that includes the identity of the register. It places the encrypted data, hash, and the XOM identifier into a set of special registers, which are now owned by the currently active session. This data can then be stored to memory if needed.

The *restore_secure* instruction is the inverse of the *save_secure* operation and it is used to restore the data back to the same register. The instruction uses the special registers that hold the encrypted data, hash, and destination key identifier. It uses the destination key to decrypt the data and check the hash. If the hash matches both the decrypted data and destination register, the decrypted data is written into the destination register and the tag is set to the destination key identifier. 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 *save_secure* and *restore_secure* instructions package up protected data, allowing another principal to move the data around without allowing them to tamper with it. This provides an operating system the means to schedule XOM processes without violating the security of our compartments. We describe the implementation of these functions in more detail in Section 4.

3. SECURITY ISSUES

Any system may be the target of a wide range of security attacks. While the ultimate attack is one that directly causes the secrets to be revealed, it is more often the case that several weaker attacks are combined to achieve the same goal. To this end, an adversary may try to manipulate the target in such a way as to leak information about the hidden secret. With this additional information, an adversary can constrain her search space and eventually mount an exhaustive search. Since we expect our model to work in the presence of untrusted external memory, we must assume that an adversary will tamper with the values stored in memory. We first discuss three potential attacks that can arise in this context: spoofing, splicing, and replay, and then look at other ways a XOM machine can leak information.

A spoofing attack is where an adversary generates data and tries to pass it off as valid data. Such an attack against XOM would involve replacing values in memory, including instructions or data values, with spurious ciphertext.[1] If the XOM machine blindly accepted these spurious values and operated on them, it may alter the behavior of the XOM program in such a way that information about the copy-protected code is revealed. The usual cryptographic solution to a spoofing attack is to employ a Message Authentication

[1]Ciphertext is the term assigned to encrypted data. Likewise, plaintext is any data that is unencrypted.

Code (MAC) [24, 13, 15]. A MAC is a keyed, one-way hash of the message. The hash is easily reproduced to check for authenticity, but it is difficult to find another message that hashes to the same value. Thus, the *store_secure* instruction generates a MAC of the encrypted data and saves it along with the data in external memory as mentioned earlier. When the value is read back in, via the *load_secure* instruction, the data is checked with its accompanying MAC for authenticity. Execution is halted if the MAC cannot be verified. Since the adversary cannot generate a valid MAC, spoofing values in memory is impossible.

Splicing attacks involve taking valid fragments of valid ciphertext (in our case, portions of XOM code or data) and reordering or duplicating them at different locations. The intended goal of this type of attack is also to trick the machine into executing the modified XOM program in the hope that it will reveal some secret about the original XOM program.

To prevent this type of attack, the MAC used in the abstract machine includes a position dependent attribute within it. In the case of data stored in memory and instructions, we include the virtual address of the memory location. Likewise, in the case of register data, we include the register number. During both instruction and data fetch from external memory, the MAC of the fetched data is checked to ensure that the data has not been tampered with. If the MAC does not match, the machine will take an exception and the XOM application will halt.

The XOM architecture also addresses replay attacks where the adversary records previous valid ciphertexts and re-inputs them to the XOM machine at a later time. Since the code is static, this approach only attacks the data values. The ability to interrupt the XOM process gives an adversary access to many valid register ciphertexts. Likewise, the ability to watch the memory traffic allows the adversary to replay memory values. To remedy this problem, we associate a mutating register with each XOM identifier and place this in the session key table. This register is updated each time the XOM process is interrupted. Including this value in the hash used for the *save_secure* and *restore_secure* commands prevents the register values from being replayed, since values from a previous interrupt will not match the current mutating register value.[2] Safe register values can then be used to protect critical memory data.

Aside from guarding against spoofing, splicing and replay attacks, our architecture also guarantees that XOM code intended for one machine may not be executed on another machine with a different key. Thus, the software is copy-resistant. We accomplish this without trusting any other entity other than the XOM machine itself. In particular, we do not rely on the security of the operating system or the memory.

However, there are also some limitations of this execution model. Since the abstract machine is trusted, a buggy, malicious, or a compromised abstract machine can reveal the secrets of a XOM program, allowing it to be copied or modified. We believe that the simplicity of the internal tagging mechanism will reduce the probability of errors, but it is still an issue. Furthermore, we assume that the data on-chip is secure, although there are techniques that allow one to probe a chip using internal scan chains, power analysis [6, 14] or various other more exotic techniques [21]. In a XOM processor this can be solved by disabling internal scan chains and other test hardware, or forcing them obey the same tag access rules as the rest of the hardware, and packaging chips in a way that makes probing impossible.

If two applications wish to share data, they must negotiate a common shared key through standard cryptographic methods. Two XOM programs may not share a session key. Sharing a key would enable an adversary to splice instructions from the two programs in unauthorized ways.

Finally, our model also leaks information at the external memory interface. An adversary can watch the memory traffic and determine an address trace of the XOM program. The coarseness of this address trace will vary with the amount of caching used in an implementation of the abstract machine. Whether this information leakage can be effectively exploited is currently an open question.

4. IMPLEMENTATION OF XOM

There are many hardware and software tradeoffs in implementing a processor capable of executing XOM code. This section initially describes a modest amount of hardware in conjunction with a virtual machine monitor [12] that can be used to run simple XOM code. By simple XOM code, we mean code that cannot be interrupted and does not store trusted data in external memory. Next, we augment this hardware to implement a machine that efficiently supports the full XOM execution model, which handles external memory and interrupts.

4.1 A Simple XOM Machine

A simplified abstract machine that does not support external memory or interrupts can be used to create copy-protected code. Since the XOM code in this model is quite restricted, we propose a software model where most of the application runs unencrypted and only certain sensitive sections of the code are encrypted and run in XOM mode. In this scenario, the XOM sections form opaque functions that the programmer uses to secure the application. These functions have access to the session key, which is used to encrypt or decrypt required data and instructions. Both for security and simplicity, the XOM code segments are not restartable if interrupted. If an exception does occur, all the data computed in the XOM code segment is lost, and there is no mechanism to undo any actions the XOM code may have performed. Temporary data storage for the XOM code could provided by a tagged on-chip memory.

There are several possibilities for selecting which parts of the application can be made into these opaque functions. The ideal candidate for XOM is a piece of code that is short, idempotent, but critical to the usefulness or operation of the overall code. For example, in the case of a streaming media decoder, short, periodic sections of the media stream could be encrypted, and the XOM functions would decrypt the data while maintaining the secrecy of the key.

4.2 Creating a Simple XOM Machine

This simple XOM machine can be implemented almost completely by running a special XOM Virtual Machine Monitor (called XVMM) on a slightly modified CPU. The main

[2] A side effect of this is that the XOM machine cannot differentiate between a fork, which is a legitimate case where memory values must be duplicated, and a real replay attack. Our current research is looking at methods that allow applications to fork, while still preventing replay attacks.

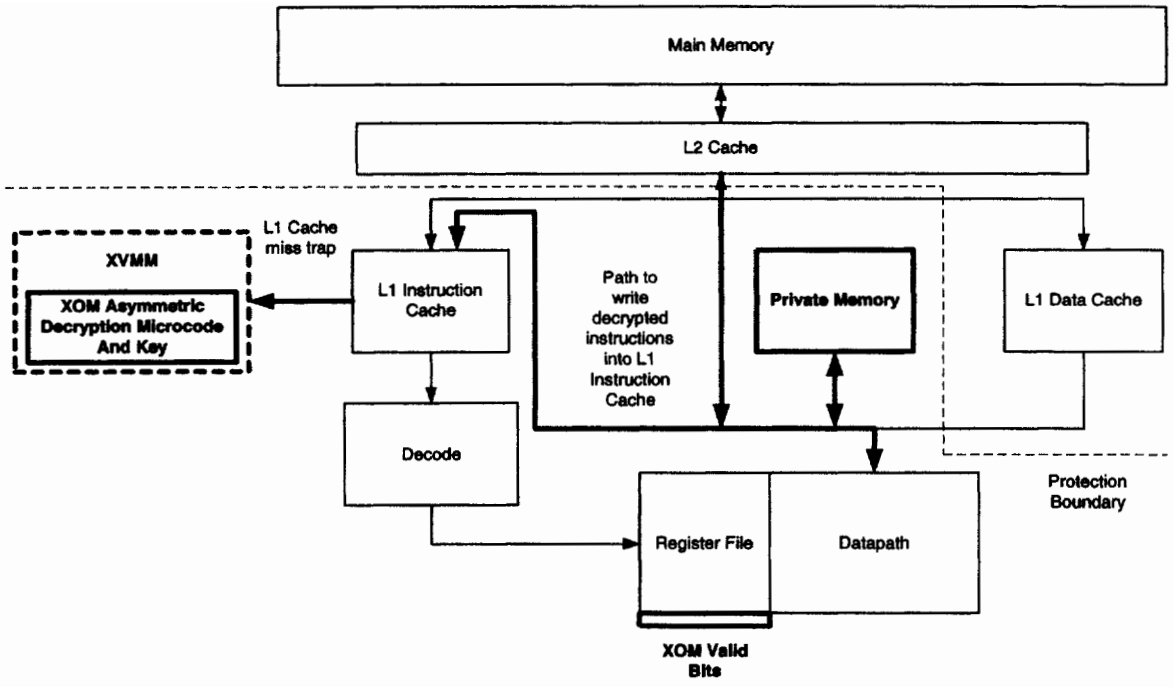

Figure 1: The Simple XOM Architecture

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.

A block diagram illustrating the additional architectural blocks required to implement the simple XVMM is shown in Figure 1. The modifications required on a standard processor are outlined in bold.

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 the literature [2, 26, 17]. 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.

For the simple execution model, the XVMM must support: decryption of the instruction stream, tagged data within the machine, and the four instructions *enter_xom*, *exit_xom*, *mv_to_null* and *mv_from_null*.

Decryption of Instructions

Decrypting the instruction stream is straightforward to achieve if the CPU vectors I-cache misses to the XVMM, which can then decrypt the code and insert the decrypted instructions into the cache. I-cache miss handling in software is not typically available on modern processors, but does not require much additional hardware. In addition to the input to the exception logic, all the machine needs is a mechanism that allows the processor to write data into the I-cache. The I-miss trap handler would fetch the requested cache line, decrypt it and check the hash, and then write this data into the I-cache. Depending on the desired performance, decryption of the instructions can be done entirely in software by the XVMM or with special-purpose hardware.

Tagging Data

Data in the machine (I-cache and registers) can be tagged with minimal changes to the hardware. The obvious solution is to directly add a hardware tag to each unit of hardware storage that requires one. However, this additional hardware is not strictly needed for the simple XOM machine. Instead of explicitly adding hardware, the XVMM could simply remove the tagged data from the machine on every (implicit or explicit) *exit_xom* instruction. For the simple machine both the machine registers and the I-cache would be flushed, since no XOM state is needed after the XOM code finishes. In this model, a shadow register file is needed to hold public data required for the special XOM instructions such as *mv_to_null* and *mv_from_null*.

The limitation of this model is that the machine will not cause an exception if the data has the wrong XOM tag—it only prevents the protected data from being read. If XOM code cannot be interrupted, this level of protection is sufficient, because interrupts cause an implicit *exit_xom*. To ensure that protection violations are trapped requires that register state be extended to include a valid bit. If the valid bit is cleared, the contents of the register are invalid, and a read access to a register causes a trap. Register reads proceed as normal if the valid bit is set. A write to a register always succeeds and sets the valid bit. The valid bit can be explicitly tested as well as cleared by the XVMM, and it indicates that the requested data is owned by the active XOM process.

In addition to the registers, the machine needs to contain private memory for the XVMM to use. The XVMM memory

needs to be protected so it can only be accessed by the monitor, while any storage for the private data of XOM applications can be cleared on an *exit_xom*.

Private Key

The last addition to the hardware is a private key that could be contained in microcode of the processor. This code implements the asymmetric cipher and outputs the symmetric key needed for XOM execution. Because the private key exists as microcode, it may be updated by a secure XOM program. In this way, an authorized party could create several chips with the same private key. This would be required for processor upgrades or in the implementation of a shared memory multiprocessor.

The XOM Virtual Machine Monitor (XVMM)

The XVMM implements the special XOM instructions and provides data tagging using the facilities of the hardware described above.

If the registers of the machine are not explicitly tagged, it organizes a portion of the private memory as a set of tagged registers. For each general purpose register in the CPU, private memory has a corresponding shadow register of the same size and an associated XOM identifier tag of a suitable length. The basic idea is that the combination of a CPU register with its single valid bit, the corresponding shadow register and tag implemented in software is functionally equivalent to having CPU registers with long tags.

A separate region of the private memory holds a session key table containing decrypted session keys for the various XOM identifiers. The XOM tags used in the shadow registers are indices into this key table. The XVMM keeps track of the index of the currently executing XOM session. Index zero refers to the null tag, which refers to the untrusted null principal.

The XVMM implements the four special XOM mode instructions as follows.

enter_xom: The XVMM loads the session key of the XOM code into the session key table if not already present. The XVMM maintains a 128 bit cryptographic hash of the encrypted session key along with its decrypted form. The presence check is performed using this hash value. A failed check entails an asymmetric key decryption operation to generate the session key. Shadow registers whose tags match that of the XOM session are copied into their corresponding CPU registers, which are then marked valid. All other CPU registers are marked invalid.

The XVMM registers a handler for cache miss events so that I-cache misses incurred during the execution of XOM code will be correctly vectored to it. Similarly, it also revectors all CPU exceptions and interrupts to itself so that it can do an implicit *exit_xom* instruction whenever there is an interrupt or exception.

exit_xom: The XVMM unregisters the handler for cache miss faults and restores handlers for all CPU interrupts and exceptions. It copies all shadow registers whose tags are null into the corresponding CPU register. All other CPU registers are marked invalid. If this *exit_xom* is a result of an interrupt, the mutating register associated with the interrupted principal is updated.

mv_to_null: The XVMM checks that the CPU register has the valid bit set. If not, it raises an exception. Otherwise, it moves the contents of the CPU register into the corresponding shadow register, tags the shadow register as null, and marks the CPU register as invalid.

mv_from_null: The XVMM checks to see if the CPU register is valid. If it is, then it raises an exception. Otherwise it moves the contents of the shadow register into the CPU register and sets the valid bit.

4.3 Full XOM Machine

To extend this implementation to support the full XOM machine requires support for secure loads and stores to the external memory, as well as saves and restores of protected registers. The hardware consists of tag bits in the on-chip cache to store the XOM identifier for XOM data. We first describe what is needed to support context switches, and then look at the more complex problem of support secure external memory.

If XOM code is interruptible, an adversary may mount a spoofing attack, by changing values while the code is interrupted. To guard against this, the XOM machine must raise an exception if a principal reads data from a different XOM session. This would detect the case where an adversary has overwritten XOM data with spoofed data. Thus, if a full XOM tag is not used, registers need to be protected minimally by a valid bit. In addition a mutating key is associated with each session key to protect against replay attacks. The XVMM must change this key each time the XOM process is interrupted. The only other support needed is for XVMM code to implement the *save_secure* and *restore_secure* instructions.

save_secure: This instruction takes one source register and uses a set of special registers as destinations. First, the register value is encrypted using the session key corresponding to the register tag and is placed in the first destination register. Next, a 128 bit hash is calculated based on the register contents, the mutating key, and the register number. We assume that our machine has 64 bit registers and so the hash is placed in the next two registers. Finally, the XOM register tag is stored in the fourth register. Thus, we need four special registers to support this operation. The destination registers are all tagged with the identifier of the principal that called *save_secure*. The calling principal is now free to save these values as it would any other data it owns.

restore_secure: This instruction is simply the reverse of *save_secure*. It takes a destination register as an argument, and reads the contents of the four special registers used in *save_secure* to determine the value to place in the destination register. The XOM identifier in the fourth register is used to decrypt the data in the first register. This data, along with the register number of the destination register, is then used to regenerate the hash, which is compared with the contents of the second and third registers. If the hash matches, the decoded data is placed in the destination register, and its XOM tag is set to the value in the fourth source register.

Supporting secure memory will require the encryption and decryption of data that is loaded or stored. As Section 6 will show, even with additional hardware this operation is not cheap. For performance reasons we want to cache the results of these operations as much as possible, which drives us to move the on-chip caches into the protected XOM machine. With this arrangement the machine only enciphers data that leaves the chip. Thus, the protection boundary is expanded to include the caches as shown in Figure 2.

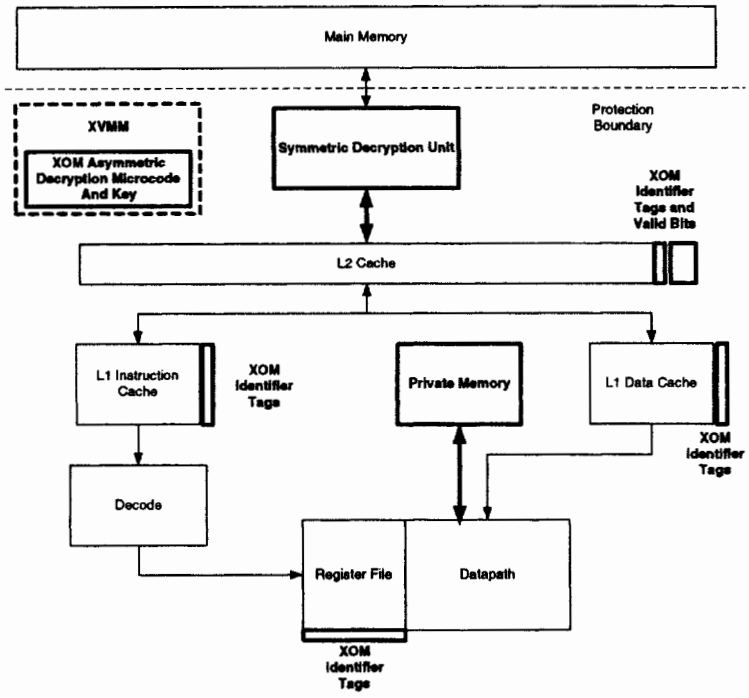

Figure 2: The Full XOM Architecture

Adding the caches to the protected XOM machine does cause some complications since we do not want the overheads of adding XOM tags per word, or generating a hash for each memory word. Instead, we want to tag and hash larger blocks of data.

We protect data in the caches by adding a tag for a XOM identifier to each cache line, and adding a valid bit per word. When a line is fetched as a result of a secure operation, the contents of that line and its hash are decrypted and compared. If the hash check succeeds, the tag for that line is updated to the active XOM identifier, the data is written into the line, and all the valid bits are set. The more interesting case is what happens when the hash check fails. In this case we still set the line's tag to the active XOM identifier, but we clear all the valid bits. An exception occurs either if a *load_secure* reads an invalid word, or if it reads a line whose tag does not match the active identifier. Note that even though there is a valid bit per word, data is encrypted or decrypted on a per cache line basis.

The use of valid bits allows a XOM process to perform a *store_secure* into a line that currently holds old data from another compartment. The store will first cause the line to be fetched into the cache. The hash will not match, since it is for a different XOM session, and all the data will be marked invalid. The store will set the valid bit for the word written—so as long as the processor only reads data it has already written, no exception will occur. If the requested line for the store is already in the cache, but the tags do not match, a similar operation occurs. Writing into a cache line always updates the tag to the active XOM identifier. If this changes the tag on the line, all the valid bits are cleared, except for the data word that was updated. Note that the clearing of valid bits is conservative—while it might lose valid data if two XOM principals share a cache line, it will never allow valid XOM data to leak out of its compartment. To prevent data loss, the operating system must prohibit two XOM applications from writing data into the same physical page.

The XOM tag and valid bits allow us to push the encryption/decryption operations to operate only on second-level cache misses, and guarantees that each cache line has only one active XOM identifier. Thus, the MAC can be done on the whole cache line rather than each word, which greatly reduces the memory overhead needed for storing the MAC. For example, a 128 byte cache line could easily accommodate a 128 bit hash using the extra bits normally allocated for ECC. The hash will still be able to catch almost all memory errors, but will not be able correct for single bit errors. The added complexity is that the valid bits must be stored along with the data to allow the cache to flush a partially valid line. When a cache line is loaded, the hash is checked and if the hash is correct, the valid bits are restored.

The full XOM model requires slightly more extra hardware than the simple model, but it is still relatively modest (except for the needed symmetric cipher acceleration that is described in the next section). It consists of adding tag and valid bits to the on-chip caches, the extra control logic needed to deal with these bits, and any cryptographic hardware. The need and cost of this hardware is described next.

5. SPECIALIZED HARDWARE

We note that most XOM operations are not performance critical. Asymmetric operations only occur when a new XOM program is started. Similarly, register saves and restores are infrequent, and only occur on interrupts and context switches. For instance, a study [18] shows that commercial applications have a mean execution length between

10,000 and 49,000 instructions between each context switch. With hardware support for the XOM identifier, the main performance issue is the implementation of secure external memory. The encryption, decryption, and MAC computation of cache lines appears on the critical path for each second-level cache miss. Supporting the full XOM model is very expensive without hardware acceleration of these operations.

The most commonly used symmetric cipher today is DES [20]. DES works by iterating the plaintext through 16 rounds of computation consisting of two XOR operations, a table lookup and bit permutations. Because of its short key length, it is usually run three times on a data block to create the Triple DES cipher [22, 25]. However, there are efforts underway to replace DES with newer algorithms that are both more efficient and more secure. Among the most promising are the AES [27] candidates Rijndael [7] and Serpent [3]. Rijndael is a block cipher that only has 10 rounds of calculation while Serpent has 32 rounds. A recent study found that when run on a slightly optimized 1 GHz processor, Rijndael offers the best performance, encrypting 92.6 MB/s [5]. This translates into approximately 1 byte every 11 cycles. To decrypt a 128 byte second-level cache line would require 1408 cycles, which is too high an overhead to pay for every second-level cache miss or write-back.

We can mitigate this cost by adding special hardware to perform the symmetric cryptography. The maximum rate at which this hardware must be able to decrypt and encrypt data is dictated by the peak bandwidth of the second-level cache to memory interface. As an example, the next generation x86 CPU, the Intel "Willamette" [16], will have a peak memory bandwidth of 3.2 GB/s and a clock speed ranging from 1.2 GHz to over 1.6 GHz. With a 64 bit memory bus this corresponds to data being placed on the bus every 3-4 processor cycles, and would require a cryptographic unit capable of keeping up with this rate.

We will use Triple DES as an example of a symmetric block cipher that can be implemented as specialized hardware, even though faster ciphers are available. Triple DES takes a 64 bit block and performs 48 rounds of transformations on it. We believe that it is possible to build a DES implementation that can compute two rounds per cycle, but we conservatively assume that only one round can be computed [8]. Thus, it takes 48 cycles to decrypt a 64 bit block. Because each round is essentially identical to every other round, DES can easily be pipelined. A fully pipelined unit would require 48 DES stages and could produce a 64 bit output each cycle. Since this is 3-4 times the required rate, one only needs a DES unit with 16 pipeline stages with each stage performing three rounds iteratively. Because each stage consists only of XOR gates and bit swizzles, the area of such a unit is dominated by the size of the tables. The tables are 6 bit lookups producing a fixed 4 bit output, and each pipeline stage requires 16 such tables. Thus, in total 256 such tables are required. Since these tables are simply ROM's they can be implemented in a very dense fashion.

We also need to create a MAC for each cache line. Unfortunately these secure hash functions are also expensive to compute. As an example, we will consider an HMAC [15] implementation using MD5 [13]. MD5 is a one way hash function that takes 64 rounds, each performing a non-linear operations followed by four bitwise additions and a barrel shift and produces a 128 bit hash. This computation is comparable to encrypting the data. While we could add more custom hardware for the hash (which would be smaller than the symmetric cryptography hardware since it does not need to be pipelined) there is a simpler solution.

We can exploit the fact that a MAC provides much more functionality then we require. A MAC is able to provide authentication for messages that are not encrypted, by using a hash that is difficult to reverse. Since the cache lines are encrypted, we are free to use a reversible hash for redundancy. Because the adversary does not know the session key, she cannot generate a valid hash of any message she creates. Thus, we may pick a much faster hash (such as CRC) and append that to the cache line before encrypting with the session key. A CRC hash may be generated in parallel with the decryption, and thus has no additional cost.

The final issue deals with the latency for verifying the hash value is correct. Since the hash depends on the entire line, a simple implementation would delay returning any data to the processor until the entire line was fetch onto the processor. For the Triple DES example this would double the overhead from 48 cycles for the first word, to over 100 cycles, since it takes 3*18 cycles to fetch all the data in a 128 byte + 128 bit cache line. To eliminate this additional overhead we can return the requested word first, and speculatively start the processor when the requested word is decoded. If the hash does not verify the XOM thread will abort, so we do not need to worry about being able to redo the load. All we need 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.

6. PERFORMANCE IMPLICATIONS

We see from the previous section that most of the performance cost of XOM will show up as an additional stage in the memory pipeline. We can model this cost as increased memory access time. There is increased latency because the XOM stage is serialized with the bus, and the additional encryption or decryption takes extra cycles to complete. It is unlikely that an out-of-order processor can hide this latency since the memory access time is already high. To the first order, we can compute the slow down factor as follows:

$$slow\ down = \frac{comp + L2\_miss \times (XOM\_lat + mem\_lat)}{comp + L2\_miss \times mem\_lat}$$
$$= 1 + \frac{XOM\_lat}{mem\_lat} \times \%\ of\ time\ stalled\ on\ memory$$

Where *comp* is the amount of time the CPU is not stalled on a memory access, *L2_miss* is the number of second-level misses, *XOM_lat* is the latency of a XOM symmetric unit and *mem_lat* is the latency of a memory access. As we can see, the two factors that affect the performance is the average proportion of time the CPU is stalled on memory per instruction and the ratio of the latency of the XOM symmetric units to the memory access time. The memory stall time is dependent on the number of second-level misses, which depends on cache size, associativity, and application. The XOM overhead is clearly most important to applications that are already dominated by memory latency (high second-level miss rates). For these applications the slow down for running in a XOM application will simply be one plus the ratio of the XOM latency to the memory latency. This is encouraging, since the XOM delay should scale with

processor performance, while the memory access time is scaling more slowly. For the "Willamette" processor example, the memory delay is over 100 cycles, so for a completely memory bound application the slow down will be less than 50% for a 48 cycle Triple DES implementation.

7. SUMMARY

Supporting code that can be executed, but not copied, read, or changed is a challenging problem when one considers that an adversary may modify the operating system running the code, the data stored in memory, or the machine the code runs on. This paper examines the implementation of XOM, a system that protects an application's code and data using cryptographic techniques, and restricts the machine on which this code is allowed to run. From a security standpoint, the critical issue is to prevent information from leaking out of a XOM application. We accomplish this by placing each application in its own compartment implementing a policy that causes the process to halt if it tries to read data that is not in its compartment.

Implementing the compartments for on-chip data is relatively straight-forward. We tag the data storage locations with the compartment that created it, and cause an exception if the data being read does not match the active compartment. Protecting off-chip data is more complex. Encryption alone only prevents the data from being read, it does not prevent it from being changed. Providing a hash with each piece of data allows us to guard against spoofing, splicing, and even replay attacks. By associating the hash with large cache lines, we can accomplish this protection with small memory overhead—using the bits currently allocated for ECC.

The performance and hardware cost for this protection is surprisingly modest. The essential pieces of hardware to be added are the tags on the on-chip storage, the logic associated with the tags to check for access violations, and the hardware support for the symmetric encryption of the memory traffic. The rest of the XOM functionality is not performance critical and can easily be implemented in software, either in a virtual machine monitor, or in microcode. If a program could be secured by using only small XOM functions, then very little new hardware is needed at all.

XOM allows one to create a machine that prevents users from copying or modifying code, but raises a number of security and privacy issues. These issues continue to be interesting areas of research.

Acknowledgments

We would like to thank Robert Bosch and Kinshuk Govil for their help in setting up the simulation environment which we used to help us better understand the performance issues with XOM. This research was supported in part by DARPA contract MDA904-98-C-A933.

8. REFERENCES

[1] Business Software Alliance, 2000. `http://www.bsa.org`.

[2] The Trusted Computing Platform Allicance, 2000. `http://www.trustedpc.com`.

[3] R. Anderson, E. Biham, and L. Knudsen. Serpent: A proposal for the advanced encryption standard. Technical report, National Institute of Standards and Technology (NIST), March 2000. Available at `http://csrc.nist.gov/encryption/aes/round2/r2algs.htm`.

[4] D. Boneh, D. Lie, P. Lincoln, J. Mitchell, and M. Mitchell. Hardware support for tamper-resistant and copy-resistant software. Technical Report CS-TN-00-97, Stanford University Computer Science, 2000.

[5] J. Burke, J. McDonald, and T. Austin. Architectural support for fast symmetric-key cryptography. In *Proceedings of the 9th International Conference Architectural Support for Programming Languages and Operating Systems*, 2000.

[6] 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, 1999.

[7] 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`.

[8] H. Eberle and C. Thacker. A 1Gbit/second GaAs DES chip. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 19.7.1–19.7.4, May 1992.

[9] Wave Corporation Embassy Technology, 2000. `http://www.wave.com`.

[10] T. Gilmont, J.-D. Legat, and J.-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, November 1998.

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

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

[13] B. Kaliski Jr. and M. Robshaw. Message authentication with MD5. *CryptoBytes*, 1(1):5–8, 1995.

[14] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of CRYPTO'99: 19th Annual International Cryptology Conference*, volume 1666, pages 388–397, 1999.

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

[16] K. Krewell. Quicktake: Willamette revealed. Technical report, Calmers Microprocessor, February 2000. Available at `www.MPRonline.com`.

[17] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authenticaton in distributed systems: Theory and practice. In *Proccedings of the 13th ACM Symposium on Operating Systems*, volume 10, pages 265–310, 1992.

[18] A. Maynard, C. Donnelly, and B. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In

Proceedings of the 6th International Conference Architectural Support for Programming Languages and Operating Systems, pages 145–156, 1994.

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

[20] National Bureau of Standards. NBS FIPS PUB 46, "Data Encryption Standard". National Bureau of Standards, U.S. Department of Commerce, January 1977.

[21] 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, 2000.

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

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

[24] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 2nd edition, 1996.

[25] W. Tuchman. Hellman presents no shortcut solutions to DES. *IEEE Spectrum*, 16(7):40–41, July 1979.

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

[27] B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke. Hardware performance simulations of round 2 advanced encryption standard algorithms. Technical report, National Security Agency, August 2000. Available at `http://csrc.nist.gov/encryption/aes/round2/r2anlsys.htm`.