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

Implementing copy protection on software is a difficult problem that has resisted a satisfactory solution for many years. This paper proposes a set of features that allows a machine to execute XOM code: code where neither the instructions or the data are visible to entities outside the running process. To support XOM code we use a machine that supports internal compartments, where a process in one compartment cannot read data from another compartment. All data that leaves the machine is encrypted, since we assume secure compartments cannot be guaranteed by anything outside the machine. 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. Our analysis indicates, while not cheap, 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, asymmetric decryption, private memory, fast symmetric ciphers, and traps on cache misses for efficient operation.

1. INTRODUCTION

The protection of intellectual property is an important issue in today's world. The Business Software Alliance, an international software piracy watchdog, has stated that piracy has cost the software industry 11 billion dollars in 1998 [1]. Implementing copy protection to combat software piracy is not a new problem, but it is one that has been difficult to solve. This has led to a widespread effort to find solutions to to intellectual property protection and models for secure computing [2, 5, 6]. This paper introduces a design that allows a machine to execute programs so that neither its instructions nor data is visible outside the running process. Our design uses cryptographic techniques to prevent the program code and the data values produced by the code from being read. Since this system creates code that the user can only execute, but cannot read, we call the the new architectural feature, XOM, for e**X**ecute **O**nly **M**emory [**?**].

To support a secure execution environment, we use the concept of a *compartment*, which is a logical "box" that provides isolation between the principals [12]. The compartment is built from a *session key*, used to encipher data. The key acts as the walls of the compartment: those who know the session key are inside the compartment and can decrypt data hidden by the key. In XOM, only one principal or program knows the session key for each compartment, and thus has access to data within it. There is a special compartment, referred to as the *unprotected* or *null* compartment, that has no key. All data in this compartment is unencrypted and is readable by all. As long as the notion of a compartment is not violated, then the privacy of all code and data is guaranteed.

The difficult part of creating compartments is securely sending the corresponding session keys to the appropriate processor for execution. Cryptography provides a solution to this problem in the form of asymmetric ciphers, also known as public key ciphers. Asymmetric key cryptography uses pairs of keys whose members are designated "private" and "public". The private key is kept secret by a principal, while the public key is freely distributed to the world. When one wishes to send a message to the owner of the private key, one encrypts the message with the public key. Only the private key can be used to decrypt the message, and since only the owner has the private key, the sender can be sure that only the intended recipient will be able to decrypt the message properly. In addition, neither key gives any information about its counterpart. Our design uses asymmetric cipher to transmit a symmetric session key used to create a XOM compartment. Using symmetric session ciphers has important performance implications because they are computationally less intensive than asymmetric ciphers. Our approach is related to that suggested by Gilmont [6].

A XOM machine has the private key of an asymmetric cipher as a unique identifier. A XOM program for this machine is encrypted with a symmetric key, and then this key is encrypted with the public key of the destination machine. Both the public key and XOM programs must be certified

*Compaq Systems Research Center
†SRI International

ASPLOS-IX 2000 Cambridge, Massachusetts USA

if authentication is also required. When the machine starts to execute the program,it decrypts the session key and uses this to create the compartment. The session key is protected by treating it as part of the data generated by the program, protected by the compartment.

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. There are many ways one can use this abstract machine to provide XOM functionality and we describe some of these. We first describe a very limited form of XOM where only small sections of the application code are "XOMed", and most of the application runs in the null compartment. The XOM sections form opaque functions that the programmer can use to secure the application. This functions have access to the session key, and can use it to encode or decode data it needs to use. The limitation of this approach is that the application design needs to consider which parts of the application to XOM, and what to include in these sections to ensure that the application is both copy and tamper resistant. To remove this restriction, we next describe a more general form where a possibly untrusted operating system can schedule and manage XOM processes, and full applications can run in XOM mode.

Section 3 looks at some of the security issues with XOM, addressing different methods 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 can control the execution of the XOM program. Providing guarantees in such a model is a particularly challenging problem.

Section 4 describes hardware implementation issues for a XOM processor. We construct a XOM processor using a modest amount of hardware and Sections 5 and 6 evaluate performance implications of executing XOM code. 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 secure storage for the XOM code to store its results. The first two tasks have been discussed in the previous section and are conceptually straightforward, we therefore describe how the abstract machine provides secure storage.

Our basic approach to secure storage is to tag all data with a XOM identifier. This identifier is a 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. The null session always has the identifier zero. Programs that run in the clear without encrypting their code, by default, belong to the null principal.

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 one 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 discussed above.

MH - Why do you want to cause an exception? I think you also need to make sure the read only returns zero, since the running XOM process will not catch the exception, right?

DL - Chandu and I figured that even having the ability to write zeros at anytime still qualifies as tampering and may be cause undue leakage of information. I guess Dan and John are the ultimate authorities on this

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 is the starting address of the encrypted session key for the XOM code to follow. This 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, we set the active identifier to this entry and we start 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, and 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 was either created by itself or explicitly brought into its compartment. The simplicity of this compartment-based protection method appeals to us—it seems simple enough that it could be implemented correctly.

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.
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.

2.1 Simple Usage Model

The simple abstract machine described above can be used to execute copy-protected code, especially in a scenario where one expects XOM code to run significantly slower than ordinary code due to decryption overheads. In this scenario, propose a software model where most of the application to run unencrypted and only have certain sensitive sections of the code encrypted and run in XOM mode.

Since we are running on small sections of XOM code, a small (tagged) on-chip scratchpad could be used as temporary storage. All stores to main memory have an implicit *mv_to_null* instruction, i.e., no secure data is stored outside the machine.

There are several real-world examples of sensitive code that fit this scenario. For example, **XXX John, Dan: what are these?**.

Short Description of these examples

2.2 Full Usage Model

An important restriction of the simple abstract model presented so far is that the XOM tag is only applied to the machine structures that hold program data, such as registers and any on-chip memory such as caches. If all copy-protected data fit within these machine structures, the simple tagging scheme described so far would be sufficient. Once again, for the examples described above, this would not be an undue constraint.

However, for most programs access to external memory for data can be a serious issue if we expect entire applications to be run in XOM mode. Handling external data memory accessed by programs poses a problem with the current approach, since we assume that anything off-chip is subject to electrical probing and hence cannot be guaranteed to be secure.

Another restriction imposed by the simple abstract machine model is that it does not satisfactorily deal with interrupts and traps. The XOM code may not be restartable after the interrupt handler returns since the handler is unable to save away interrupted state without reading the registers. For some XOM applications, dealing with interrupts might not a big constraint. For instance, it might be possible to make the XOM code functions restartable and idempotent, so if there were interrupted they could simply be restarted and rerun. This may indeed be a feasible option if the code fragments are small as in the examples above, but we wanted to explore the possibility of running all code in XOM mode assuming the overhead of running XOM mode code could be made small enough.

The full XOM model enhances the previous model by extending compartments to include main memory and by allowing XOM code to be interrupted and resumed.

Conceptually, we extend compartments to external memory by using cryptography. That is, when tagged data leaves the confines of the machine, we encrypt the data using the appropriate session key and store it in external memory. This ensures that the guarantees of tagging is also available to data in external memory. Since external memory is untrusted we implicitly assume that it is owned by the null principal.

We provide 3 pairs of instructions to move data between external memory and the machine: *store*/*load*, *store_secure*/*load_secure*, and *save_secure*/*restore_secure*.

The *store* and *load* instructions take as arguments a register and a memory location and behave very much like ordinary store and load instructions in a standard CPU. The essential difference is that the *store* instruction takes a register tagged with the null identifier as the source; a tagged source register raises an exception. The *load* instruction loads the destination register with the contents of the memory location and tags it with the null identifier.

The *store_secure* and *load_secure* take a register and a memory location as their arguments. They are used by the currently executing XOM program to move data between memory and registers that are tagged with its own identifier. If the register named in a *store_secure* or *load_secure* is either tagged with the null identifier or an identifier other than the active one, the instruction raises an exception.

The *store_secure* instruction atomically encrypts the contents of a tagged register using the session key corresponding to the tag, creates a cryptographic hash that includes the memory location, and stores both into the external memory location.

The *load_secure* instruction takes a target register and memory location as arguments. It atomically decrypts the contents of memory using the active session key, checks the hash to make sure it matches, loads a 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 use of a hash in addition to encryption might appear unnecessary at first sight, but the need for it is explained in the next section. Also notice that the hash function implies that these instructions require multiple memory words unlike ordinary loads and stores.

The *save_secure* and *restore_secure* instructions are used by the currently executing XOM program to move data between memory and registers that are tagged with another non-null principal's identfier. A register tagged with the null identifier or its own identfier will cause an exception.

The *save_secure* instruction takes the same arguments as *store_secure*. It first encrypts the register contents and calculates a hash that includes the identity of the register. It uses the key of the register owner for this operation. Next it takes the result of the encrytion and hash and stores it in memory encrypting the result and hash a second time and calculating another hash that includes the memory location named in the instruction.

The *restore_secure* instruction is the inverse of the *save_secure* operation and it is used to restore the data stored in memory back to the same register when it was saved. The instruction takes a target register, a memory location, and a session identifier that is different from the active identifier. It first decrypts the contents of memory using the key of the active principal and verifies that the hash, which includes the memory location, matches. If it does, then it proceeds to decrypt the previous result, this time using the key of the supplied session identifier. It also verifies that the second hash, which includes the register name in it matches. If there is a match, the resulting data is written into the named register and the register is tagged with the session identifier.

We describe the implementation of these functions in more detail in the Section 4.

MH - Then what do normal load and store operations do? It seems like this sequence of instructions would be the same as a normal store? I think I see. Normal load and store instruction maintain the XOM state of the running process. I think this is a good idea, but is not clear. In fact you say the opposite 2 paragraphs up. Having programs use secure load and store won't work for two reasons – First programs don't load values into the same register that they used to store them, and second, the memory overhead for a hashing each register is too large. I really think you need to have different operations for an outside principle storing your data, and having you store your data.

DL - normal loads load stuff in and keep it as null (they don't decrypt). normal stores can't operate on owned data and secure stores operated on owned data only. I think we messed up above. The concern about hashing is noted, and explained below.

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. That is, on an interrupt, we must allow the untrusted operating system to save the register state of a XOM process, without actually being able to interpret or leak the contents of the registers. The *save_secure* and *restor_secure* instructions provide the necessary means to do this.

To summarize, the XOM model ensures that a copyrighted program will only run on a specific machine and requires that the copyright owner to know the public key of the machine.

A XOM program can take traps and interrupts between any two instructions. A supervisor program, such as an untrusted operating system, can field these exceptions and multiplex the abstract machine registers amongst multiple programs without compromising the secrets of the XOM program. However, such a supervisor program, if it is malicious, can mount a replay attack on the XOM program (as described below).

The full abstract machine model though functionally complete, requires careful treatment of the security issues that now arise. We next describe the guarantees and limitations of our model against typical cryptographic attacks.

3. SECURITY GUARANTEES AND LIMITATIONS

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 may be combined to achieve the same goal. Often an adversary will try to manipulate the target in such a way to leak information about the hidden secret. In this way, adversaries can constrain their 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. Here we discuss three potential attacks that can arise in this context: spoofing, splicing, and replay.

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 common cryptographic solution to a spoofing attack is to employ a Message Authentication Code (MAC) [13, 7, 8]. 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. 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 easily generate a valid MAC, spoofing values in memory is difficult.

Splicing attacks involve taking valid fragments of 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 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. For the case of data and instruction, we include the virtual address of the memory location. For 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.

Our architecture does not provide strong guarantees against a replay attack. This is an attack where the adversary records previous valid values and re-inputs them to the XOM machine. A possible remedy is to use a mutating session key that changes after every store/load pair. Unfortunately, for this to work in the general case, we would have to remember the current key for every data value that leaves the virtual machine. This is impractical and thus protection against a general replay attack in XOM is difficult. Similarly, a XOM program may be tricked into executing the same code twice. Programs must make separate provisions, independent of XOM mode, to prevent such attacks.

Aside from guarding against spoofing and splicing attacks,

[1]Cipher text is the term assigned to encrypted data. Likewise, plain text is any data that is unencrypted.

our architecture also provides some other guarantees. Because the private key is kept securely in hardware, we can guarantee that XOM code intended for one machine may not be executed on another machine with a different key. Thus software is copy-resistant. We can also guarantee that XOM code executing on the machine may not have its contents observed or altered. In this way, it is tamper-resistant. Finally, we accomplish this without trusting any other enitity other than the XOM machine itself. In other words, we do not rely on the security of the operating system or the memory.

Our abstract machine of course has some limitations. Since the abstract machine is trusted, a buggy, malicious, or a compromised abstract machine can reveal the secrets of a XOM program. Our implementation proposal in the next section shows one way to mitigate, but not eliminate, the danger from this vulnerability, by building the trust into the CPU hardware.

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. Our implementation proposal limits traces to be fairly coarse by aligning external memory addresses to a secondary cache line boundary (a 128 byte boundary is typical).

Two XOM programs may not share a common symmetric key. Sharing a key would enable an adversary to splice instructions from the two programs in unauthorized ways.

4. IMPLEMENTATION OF XOM

There is 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 XOM virtual machine monitor [?] 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 doesn't have these restrictions. A subsequent section describes the performance implications of these implementation choices.

4.1 Running Simple XOM Code

We propose a special XOM virtual machine monitor (called XVMM) running on a CPU augmented with a set of hardware changes, to provide a suitable execution environment for simple XOM code.

XVMM could be implemented in software or in microcode. Software implementations must be authenticated by a secure booting mechanism such as those described in the literature [2, ?, ?]. Either way, XVMM executes as a trusted, authorized, and privileged program. It has unique capabilities that require additional hardware support from a standard processor.

Recall that in order to run simple XOM code it is sufficient if XVMM supports the following functionality: 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 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 it would be no more difficult to provide than software TLB miss handlers. Depending on the speed of the processor, decryption of the instructions can be done entirely in software by XVMM or with special-purpose hardware.

MH - Note that this ICache miss trap is not strictly needed. With a secure on-chip memory, you could decode the instructions, place them in this memory and then have the processor jump to it. Think about machine translation engines. The same techniques would work.

DH - This is true. We thought about this and realized it causes other complications. How does the XOM process address the scratch pad? We have to memory map it. This could be kind of confusing, what if it is self modifying code? The VM has to keep track of what it has mapped into its secure scratch pad. The scratch pad also needs base and bounds registers now. The VM may need to do some jump remapping and address remapping so if it control jumps to an unencrypted part it basically passes to the VM. Don't know if we want to explain all this?

Tagging Data

MH - I don't think any data needs to be tagged in this simple model. The basic idea is simple, and you have it already – a shadow regfile for each XOM context including the null context. When a XOM process runs, all the registers have values only for it, the other regisers store zero – Oh I see why you have the tags – they are needed for the exceptions. Huh, I guess I need to think about it some more. Note if you don't want exceptions, move to null just store in the null context, etc.

DL - I think we're saying the same thing. We don't need tags in icache if we blow it away on exception

Data in the I-cache can be effectively tagged with minimal changes in the hardware. We know of two ways of doing this. The simplest way is extend the cache lines to include XOM identifiers. Another way is to add no hardware, but to flush the I-cache on every (implicit and explicit) *exit_xom* instruction.

Unlike tagging the I-cache, adding several tag bits to each CPU register can be more problematic because registers are often multiported for speed. Instead, we use a combination of simple hardware and the XVMM to support multiple tag bits.

Having multiple tag bits is generally a benefit because the session key table addressed by these bits can be larger. A larger session key table allows more principals to share the machine concurrently. It also means that the key table slots do not have to be multiplexed as often amongst principals. An expensive public key decryption of a session key is involved each time a session key table slot is reused.

We extend CPU registers to have a valid bit. If the valid bit is clear, 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 tested as well as cleared by XVMM.

With this level of support from hardware, the virtual machine monitor can support tags of longer length as we describe below.

Additional Hardware Support

In addition these changes in hardware, we need to provide a privilege bit in the process status word to denote whether the process is in XOM mode or not. This bit is only writable by XVMM.

The CPU needs to provide a small amount of on-chip private memory that is only accessible to XVMM. It uses this memory to store structures such as the XOM key table and active XOM identifier. In addition, intermediate results for the asymmetric and symmetric decryption could be placed here as well.

The private half of the public key pair is also implemented within the hardware.

The XOM Virtual Machine Monitor (XVMM)

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

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. In some CPU architectures, it may be feasible to implement long tags in the CPU registers themselves, which would simplify the virtual machine monitor.

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. XVMM keeps track of the index of the currently executing XOM session. Index 0 refers to the null tag, i.e., to the untrusted null principal.

XVMM implements the four special XOM mode instructions as follows.

enter_xom: XVMM loads the session key of the XOM code into the session key table if not already present. 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. Non-present keys entail an asymmetric key decryption operation to generate the session key. Shadow registers whose tag match that of the XOM session are copied into their corresponding CPU register, which are marked valid. All other CPU registers are marked invalid.

XVMM registers a handler for cache miss faults 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: XVMM unregisters the handler for cache miss faults and restores handlers for all CPU interrupts and exceptions. It copies all shadow registers whose tag is null into the corresponding CPU register. All other CPU registers are marked invalid.

mv_to_null: 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: 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.

On an I-cache miss, XVMM first locates the active XOM session key in the session key table. It uses this key to decrypt data from external memory and fill the I-cache.

4.2 Running XOM Code with Interrupts and External Data

Next, we describe how to support full semantics where the XOM code may be interrupted and is also allowed access to external memory to store its data. Our strategy requires some additional hardware support than what is described earlier.

The basic idea is to add XOM tags to all levels of the memory hierarchy that are on-chip and use these tags to encrypt and decrypt data that goes off-chip. This extends the boundary of the XOM machine to encompass the on-chip caches. Given the architecture of modern CPUs, this implies putting tags on each L1 and L2 cache line.

Given tags, data in the L1 and L2 can be kept unencrypted without violating the compartment model of security. Data that is flushed from L1 to L2 retains its tag, which constrains the granularity of ownership to a L2 cache line, because each L2 cache line can have only one tag. Write-backs from L2 to memory are encrypted with the owner's key and combined with a MAC before going to memory. L2 cache fills will cause memory to be decrypted using the active session ID and the line to be tagged.

Recall that to support full XOM mode execution, XVMM must implement secure stores and loads.

store_secure: This instruction specifies a source register and a destination memory location like a conventional store. If the tag matches that of the active principal, then XVMM moves the contents of the CPU register into the L1 cache line. The tag on the L1 cache line is set to the register tag. Subsequently the tag and data will be propagated to the L2 cache, and then sent to external memory after encryption and hashing during a write-back operation. The key used for encryption will be that of the principal that executed the *store_secure* instruction. The hash value will include the virtual address of the destination memory location.

save_secure: This instruction also specifies a source register and a destination memory location like a conventional store. It first ensure that the register tag does not match that of the active principal, and that it is not null. Next, a position dependent hash is calculated based on the register and its contents. This is then encrypted using the session key corresponding to the register tag. The result is stored in the L1 cache with the tag of the active principal. As in the case of *store_secure*, a subsequent write-back will result in a second encryption and hashing using the active principal's key.

Let r be the register number, c the contents in register r, va the memory address location specified in the instruction. Let key_r and key_a be the keys corresponding to the register tag and the active key respectively. Let Hash1 and Hash2 be two suitable hash functions.

Then, the *store_secure* instruction will store into memory location va the value $[c, Hash1(va)]^{key_a}$. The *save_secure* will store the value $[[c, Hash2(r)]^{key_r}], Hash1(va)]^{key_a}$. Here exponentition by a key denotes encryption under that key and comma "," denotes concatenation.

In practice, we expect *save_secure* to be used most commonly by the operating system running as the null principal to save registers belonging to a XOM program. In this case, the second set of encryption and hashing is avoided.

load_secure: This instruction specifies a memory location, and destination register. The result of doing a *load_secure* instruction is to load the register with the contents of the memory location and to tag with with the active session key.

The XVMM decrypts the contents of the memory location using the session key of the active principal, verifies the hash (Hash1), and stores the contents in the L2 and L1 cache. The caches lines are tagged with the active principal's tag. These actions are just the inverses of the *store_secure* steps. The data is then written into the register and its shadow and the tag is updated in the shadow.

Let r be the destination register number, va the memory address location specified in the instruction, and m the contents of the memory location. Let key_r be the key specified in the instruction, and let key_a be the active key. Then, the contents of the cache line is $[m]^{key_a{}^{-1}}$, where exponentiation with the inverse of a key denotes decryption with that key. Next, $[m]^{key_a{}^{-1}}$ is written into the register r.

restore_secure: This operation takes a memory location, a destination register, and an additional session key as its arguments. XVMM first verifies that the session key is different from the active key. Then, it executes the same decryption as outlined in the *load_secure* case. Next, XVMM decrypts the contents of the L1 cache, verifies the hash (Hash2), and writes the data into the register and tags the register with session identifier corresponding to the session key specified in the instruction. In this case $[[m]^{key_a{}^{-1}}]^{key_r{}^{-1}}$ is written into the register r.

5. PERFORMANCE AND SPECIALIZED HARDWARE

We note that some XOM operations happen rarely while others may occur more often. For example, the asymmetric operations, as well as register saves and restores by the operating system are relatively rare events. Howeverr, the encryption, decryption, and MAC computation of cache lines appears on the critical path for many operations. Supporting the full XOM model may become very expensive without hardware acceleration of these operations.

Since write-backs and cache misses from the L2 cache to memory may happen quite often, the overhead of performing the encryption in software may make the design extremely slow. As an example, the best software implementation of DES [9], a common symmetric block cipher, has a throughput of approximately 9 Mbit/s on a 133 MHz Pentium/MMX processor**Dan, is there a citation for this?**. This translates into approximately 1 byte every 14 cycles. To decrypt a 128 byte L2 cache line this would require 1792 cycles which is two high an overhead to pay for every L2 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 L2 cache to memory interface.

As an example, we consider the cost of typical processor running a typical encryption algorithm. The next generation x86 CPU, Willamette, [**?**] will have a peak memory bandwidth of 3.2 GB/s and a clock speed of 1.5 GHz. With a 64 bit data bus, there is data coming on the bus every 3.75 cycles. This requires the cryptographic unit to capable of encrypting or decrypting a 64 bit block every 3 cycles. We select Triple DES [11, 15, 9] as an example of a symmetric cipher. All symmetric ciphers are DES-like, consisting of an algorithm which iterates the text through several rounds performing linear and non-linear transformations on the text at each round, so our analysis would apply equal well to any symmetric cipher.

Triple DES, takes a 64 bit block and performs 48 rounds of transformations on it. Each round consists of two XOR operations, two permutations, and a table lookup. We believe that it is possible build a DES implementation that can compute two rounds per cycle, but we conservatively assume that only one round can be computed per cycle [**?**, **?**]. 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. Thus, after the initial block, the DES unit is capable of producing a block every cycle while the memory busy is only capable for inserting a block every 3.75 cycles. As a result, we only need a DES unit with 16 pipeline stages with each stage performing three rounds itteratively. With this method, overhead of the encryption or decryption is an additional latency of 48 cycles.

MH - David this is WAY too much hardware since you need to create 48 copies of the hardware. You would only really build 16 DES sections and still be faster than the bus.

DL - agreed. I don't know what I was thinking.

Several other ciphers also exist that may be used. Among the most promising are the AES [16] candidates Rijndael [4] and Serpent [3]. Rijndael is a block cipher that only has 10 rounds of calculation while Serpent has 32 rounds. Both have rounds which are simple enough that they it should be possible to implement them in a single cycle. In addition, like Triple DES, they should be straightforward to pipeline.

We also note that all values must have a MAC and position dependent hash added for integrity. As an example, we will consider an HMAC [8] implementation using MD5 [7]. MD5 is a one way hash function which takes 64 rounds, each performing a non-linear operations followed by four bitwise additions and a barrel shift. MD5 produces a 128 bit hash which, for a 128 byte L2 cache line results in a storage overhead of exactly one bit per effective byte. As it happens, this is the same storage overhead of ECC so we may use ECC RAM to store the MAC. A benchmark of the OpenSSL [10] software implementation of HMAC with MD5 has a throughput of 23 MB/s on a 350 MHz Pentium II system. Thus, each byte takes approximately 15 cycles to produce and a 128 byte cache line would require in the vicinity of 243 cycles. This is a fairly large cost to pay for each memory access. Again we may build a specialized hardware implementation to decrease the cost somewhat. However, the problem is that because the entire cache line is required to compute the hash, there is no benefit to pipelining the implementation. Thus if we again assume that each round takes a cycle, the operation requires 64 cycles.

This is still a large overhead to pay. Fortunately, 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 is difficult to reverse. Since the cache lines are encrypted, we are free to use a reversible hash for redundancy. Since the adversary does not know the session key, she can not 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. Regardless of the hashing technique, the hash must be of sufficient length so that the adversary cannot generate a message with a valid accompanying hash by random trials. A hash length of 128 bits as in MD5, is generally considered to be effective. A 128 bit CRC hash may be generated in 1**anyone know what this number really is??** cycle but requires that the extra 128 bits also be encrypted or decrypted with the above symmetric key. Thus the overhead for generated the hash is actually 9 cycles. The total overhead including the 48 cycles of Triple DES is thus 57 cycles per L2 cache operation.

As a final note, we observe that even though the hash calculation requires the entire cache line, we may still support critical word first in the memory system. We see that for the Willamette example above, there is an additional 57 cycles from the time the first 64 bit block is decrypted to the time that the last block is decrypted. Thus we may view the machine as speculatively executing for that 57 cycles. All operations that allow information to leak out of the machine such as stores or loads cause the machine to stall. If the hash verifies incorrectly the thread of execution is killed and all state destroyed since it is unsafe to try to restart when data has been tampered.

We use simulation to study the effect of various options on the end-to-end performance of applications.

6. SIMULATION RESULTS

We used the SimOS simulation system [?] to evaluate performance impact of the Full XOM model. We simulated these as if the entire application was running in XOM mode since we do not have concrete data on how the partially XOM'ed applications that would be used for the Simple XOM model would behave. Thus in these simulations every load from memory and write back to memory incurs the XOM overhead. BusUMA, the bus based memory model provided by SimOS was modified to include an extra encryption stage on all write-backs and a decryption stage for data returning from a read to memory. Reads which hit in the write buffer do not incur the decryption overhead. The encryption and decryption overheads are the same for symmetric ciphers and in our case, also include the time to calculate and if necessary verify the hash. Note that write backs are not in the critical path and thus do not directly contribute to the latency of the runs. Rather, they add occupency to the bus and thus may indirectly add latency.

Since by varying whether the cipher and hash are implemented in hardware or software, as well as varying which algorithm is used affects the latency of the operations, several different overheads were simulated. We used four different workloads. The first is *FFT* a scientific code which performs a Fast Fourier Transform. The second is *make* which simply builds the Irix 6.4 kernel. Third is *informix* a database benchmark and the last is *apache* a web server benchmark.

Simulations are still running: Charts and Discussion to follow. So far it seems the cost is small – less than 5% increase in overall exec time and less than 1% increase in overall CPI for overheads less than 50% of memory cycle time, when boundary is placed at L2 cache. L2 miss rates for this FFT are about 8% (misses/refs). misses/inst are small though about 0.0001% – the commercial apps should have more

7. SUMMARY

To be written

Acknowledgments

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] 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`.

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

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

[7] B. Kaliski Jr. and M. Robshaw. Message authentication with MD5. *CryptoBytes*, 1995.

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

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

[10] OpenSSL, 2000. `http://www.openssl.org/`.

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

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

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

[14] J. G. Spooner. Intel brands willamette as pentium 4, 2000. Available at `http://www.zdnet.com/eweek/stories/general/0,11011,2595817,`

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

[16] 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`.