Hardware Support for Tamper-Resistant and Copy-Resistant Software

Dan Boneh, David Lie, Pat Lincoln, John Mitchell, Mark Mitchell

November 14, 1999

Abstract

Although there have been many attempts to develop code transformations that yield tamper-resistant software, no reliable software-only methods are known. Motivated by numerous potential applications, we investigate a prototype hardware mechanism that supports software tamper-resistance with an atomic decrypt-and-execute operation. Our hardware architecture uses a novel combination of standard architectural units. As usual, security has its costs. In this design, the most difficult security tradeoffs involve testability and performance.

1 Introduction

Tamper-resistant hardware is used in a number of applications where control over the execution of an algorithm or preservation of secrecy is required. A variety of smart cards, for example, provide reliable authentication by executing challenge-response protocols without revealing embedded secret keys. To give another example, E-Stamp [5] has developed a "small, but highly secure device" that allows postal users to buy stamps over the Internet and print them on envelopes. This is tamper-resistant hardware that is attached to ordinary computers in order to secure electronic transactions. Finally, hardware DVD players contain secret decryption keys that are used to access encrypted video files. While attempts have been made to produce reliable software DVD players, for example, the recent break of a software product [8] has left the entire movie industry open to unlawful distribution of decrypted content. In short, there are no reliable software mechanisms for simulating reliable hardware on standard computational devices. The basic limitation is this: no matter how software is encrypted or otherwise rearranged, the instruction processing algorithm must eventually deliver standard instructions to a standard processor. Therefore, running so-called obfuscated code under a debugger seems to allow many properties of the code to be determined.

In this paper, we propose a hardware mechanism that allows specially encrypted code to be securely executed on a modified form of standard hardware. The basic idea is to decrypt an encrypted instruction stream on chip, in a manner that does not reveal the decrypted stream to an observer. We call the mechanism "XOM", pronounced "zom"; this is an acronym for eXecute-Only Memory, since an encrypted block of instructions can be executed but cannot be meaningfully manipulated in any other way due to encryption. Our goal in designing XOM has been to identify the simplest possible machine-level primitive that would allow a variety of software protection mechanisms to be programmed. By analogy with the way that atomic test-and-set can be used to build a number of concurrency control mechanisms, we believe that the decrypt-and-execute instruction associated with XOM is a flexible enough programming primitive to justify serious investigation of the associated hardware requirements.

In one of several intended applications of XOM, a software vendor would encrypt code before sending it over the network to the owner of a XOM machine. The machine owner would not know the decryption key. However, the decryption key embedded in the XOM chip would allow the encrypted instructions to be executed. Since each chip contains a different decryption key, this basic mechanism makes it possible to produce software that will run only on one chip, providing a form of copy protection. To avoid paying a performance penalty when ordinary code is executed, decryption is performed along a separate instruction load path that operates only when the chip is in XOM mode. This load path also verifies the integrity of encrypted code so that if someone modifies a sequence of instructions, that instruction sequence will be rejected. Finally, as described in more detail below, interrupts are disabled during XOM execution and on-chip registers are flushed on exit from XOM mode to preserve the secrecy of completed calculations.

While striving to provide the same degree of security as smart cards and other special-purpose hardware, we also aim to minimize the overall impact on processor performance and design. To help realize the second goal, our proposal deviates as little as possible from current processor implementation and reuses many of the same units. As one might expect, however, security has its costs. In particular, security interferes with standard testing mechanisms that allow the state of the architecture be observed at any time. There are also obvious performance costs. In addition to the delay involved in decryption (which we minimize in several ways), the security requirements appear to be at odds with speculative execution and reordering. XOM therefore poses significant challenges to chip designers.

In Section 2, we review relevant cryptographic primitives and their use. Sections 3 and 4 present the XOM instruction set and architecture. Section 4.5 addresses some of the problems associated with verifying hardware and software operation under XOM. Potential applications of XOM and a brief comparison with competing security mechanisms appear in Sections 5 and 6, with concluding remarks in Section 7.

2 Cryptographic Issues

Execution of encrypted software uses a combination of public-key cryptography, symmetric-key cryptography, and message authentication mechanisms.

Public Key Cryptography uses separate encryption and decryption keys. The primary advantage of public-key cryptography is that an encryption key can be safely published without revealing useful information about the decryption key. XOM therefore starts with public-key cryptography: code is encrypted using the public encryption key associated with a secret decryption key stored on chip. A disadvantage of public-key cryptography is that it is computationally expensive and difficult to implement in hardware because it requires the exponentiation of large numbers.

Symmetric-Key Cryptography is much more efficient than public-key cryptography and easier to implement in hardware because it involves convolution and simple bit operations. Since the same key is used for both encryption and decryption, symmetric keys must be handled carefully. However, symmetric-key cryptography can be usefully combined with public-key cryptography as described below.

Message Authentication Codes or MACs, are used to guarantee the integrity of a message. If a low collision hash of a message is sent along with the message, the receiver can verify message integrity by checking the hash value. Because MACs are one-way hash functions, they do not reveal any information about the original message. Obviously, the larger the MAC, the more secure it is, since there is a lower probability of collision, and thus a lower probability that a modified message will hash to the same MAC. Digitally signed MACs allow the integrity and authorship of a message to be verified.

Cipher Block Chaining or CBC, may be used to guarantee the integrity of a stream of data. Essentially, each packet of data provides information that is used to decrypt the next packet of data. A simple implementation of this is to encrypt each packet with a key that can be derived from the plain text of the preceding packet.

Since public-key cryptography is approximately 1000 times slower than symmetric-key cryptography [10], it is important to use a hybrid approach. Specifically, a long message can be encrypted with a symmetric key that is transmitted using public-key encryption. This allows a receiver to perform only one public-key decryption, followed by a sequence of more efficient symmetric-key decryptions. For concreteness, we will describe XOM using RSA public encryption in combination with DESX symmetric encryption. Both are standard modern cryptography methods. To guarantee integrity of blocks of encrypted software, we use MACs on each unit of code and CBCs to maintain integrity for the entire code segment.

3 Instruction Set Additions

To secure the execution of XOM code the following must hold:

- XOM code must execute atomically – either the entire sequence completes or the entire sequence fails.
- No plain text can be left behind on exit from XOM mode that an adversary could potentially read.

To do this, a simple but powerful interface to the XOM hardware must be designed. Several instructions must be incorporated into the ISA. These will be an enter XOM instruction (XOM), and an exit XOM instruction (EXOM) and a XOM flavor of loads and stores.

Executing the enter XOM instruction acts like a "jump to subroutine" instruction and places the CPU in a mode called "XOM mode" which operates under a certain set of invariants much like the familiar "user" or "supervisor" modes. However, it will be orthogonal to those modes and will not give user code any privileges that it would not normally enjoy. The XOM instruction takes two arguments: the encryption of the symmetric key, and the location of the start of the XOMed block of code. The processor will decrypt the symmetric key and then jump to the head of the XOM code block. When the XOM mode bit is set, the instruction fetch path passes through the XOM Instruction Decryption Unit where the instructions are decrypted and their authenticity verified (see figure 1). It may be useful for a processor to also execute kernel instructions in XOM mode such that the operating system may perform secure operations.

A exit XOM instruction, EXOM, is also provided so that XOMed code may signal that it has completed its secure operations and the processor may return to its normal mode of operation. The EXOM instruction causes all architectural state to be secured and all pending writes to be completed. Finally, the XOM mode bit is switched off and the instruction fetch path returns to normal. The processor also performs an EXOM if it takes an interrupt while in XOM (i.e. EXOM becomes the

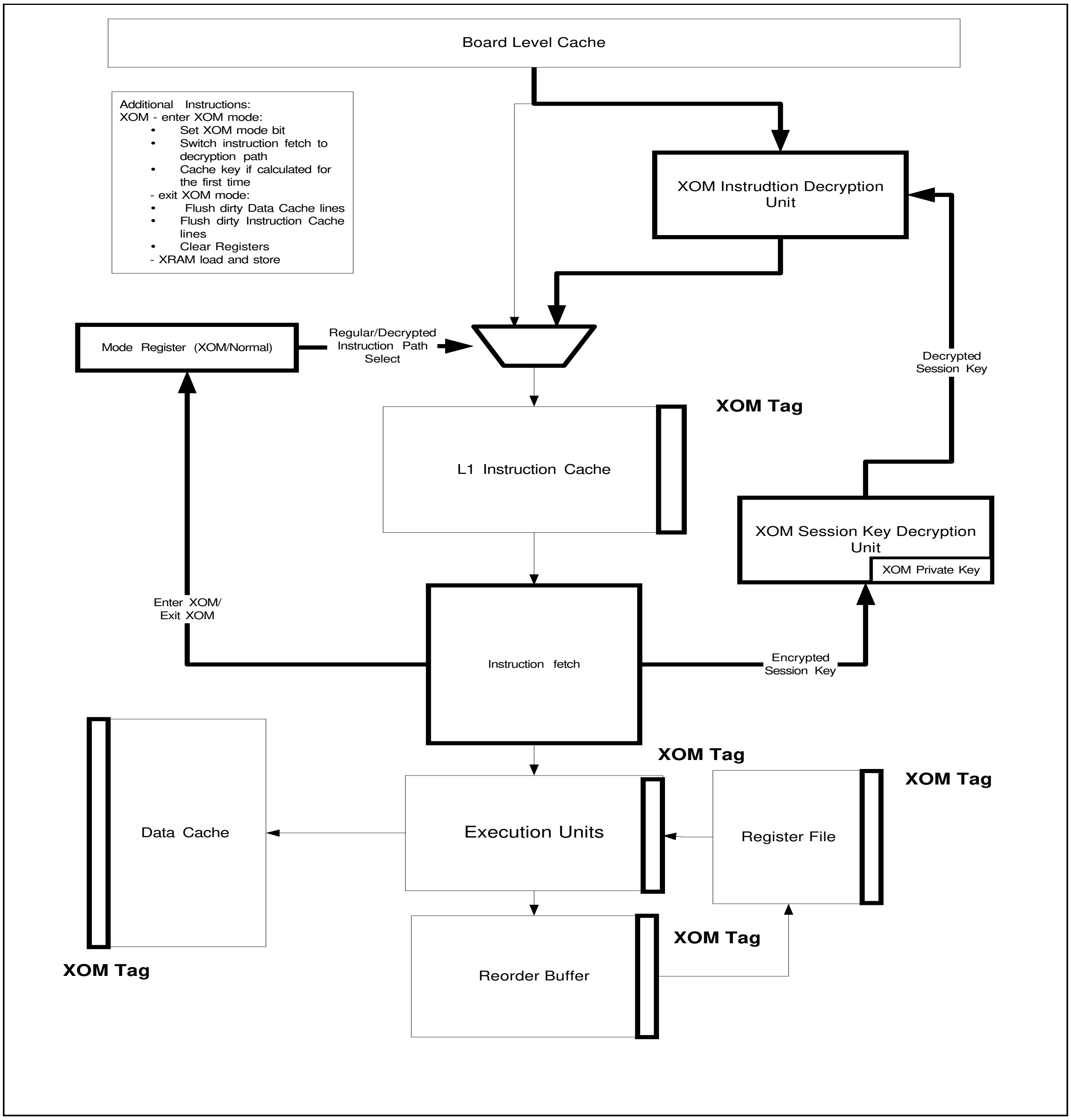

Figure 1: The XOM architecture

interrupt handler while in XOM mode) – this fulfills the second requirement above. The interrupt handler may not run in XOM mode since it would have access to the state of the XOM code it interrupted. When the XOM code restarts after the interrupt, it must start all over again since all its previous state was destroyed. It is the responsibility of the programmer to ensure that if the XOM code needs to know that exit status of the previous invocation of XOM, it will be able to deduce that via data values written out during XOM. To aid the programmer, write reordering is prohibited during XOM mode. While this causes performance to suffer, it is required to ensure security. There is also a possibility of live lock since a piece of XOM code may cause the eviction of a page containing a piece of data it requires. This problem can be alleviated by enlisting OS support. Before entering XOM code, the application declares which pages it would like to be in memory to the OS, and the OS ensures that these pages will remain in memory for some time until the exit of XOM. This does not give away information since it is not the operation we are protecting but the implementation.

Finally, the XOM code may need a secure scratch pad, called XRAM, to store temporary values. XRAM will have its data rendered unreadable after leaving XOM mode. This scratch pad can be accessed through XOM flavored loads and stores. The XOM, EXOM and XOM flavor memory operations represent the minimal modifications required to the ISA to provide XOM functionality.

4 Implementing XOM

Since most applications will use XOM in only a few places, the performance of the processor while in XOM mode was not a paramount design consideration. However, we made every effort to minimize the impact of XOM mode on ordinary, non-XOM application code, following the principle that one should only pay for the features one uses.

Our proposal for implementing XOM entails modifications to five basic functional units of a typical processor. Figure 1 provides a high level block diagram showing these modifications and additions, with the affected portions in bold type.

4.1 XOM Mode bit and XOM Tags

One additional bit in the status register is required in order to indicate whether or not the processor is in XOM mode. Entry to XOM mode will set this bit; exiting XOM mode will clear the bit. These mechanisms are the only ones that will allow modification of the XOM bit.

When the processor exits XOM mode, it is essential that all state in registers and cache be unreadable by any subsequently executing code. Otherwise, users could wait until XOM mode is exited, and then examine the register state, and thereby obtain information about the workings of the XOMed code. In principle, the XOM code could itself explicitly clear registers and the cache, but doing so would be expensive, and require additional care on the part of the software engineer. Therefore, we suggest a mechanism by which the processor can provide this functionality itself.

Two tags must be present on all potentially sensitive data items. The first (a "valid" tag) indicates whether or not the data stored in that location is up-to-date; for example a cache-line for memory that has been written to by another processor in a multi-processor system could be marked as "invalid" so that data in that line would be reread when the line is next accessed. A write (from the CPU in which those data items are stored) to one of these locations will set the valid tag. We will take advantage of this tag, which already exists in most caches, for the purposes of XOM.

We propose adding a XOM tag to these items, indicating whether or not the item was last written to in XOM mode. On exit from XOM mode, all data items with the XOM tag set are marked invalid. In this way, we prevent other code from accessing these values. Alternatively, the processor could just zero the values marked invalid; that might or might not be more efficient depending on the processor in question.

While in XOM mode, the processor must disallow the use of scan modes for testing. Otherwise, an adversary could simply scan the processor state to learn the values kept in registers, etc. Even when not in XOM mode, it must be impossible for users to alter the XOM and valid tags on registers or the cache, as doing so might allow programs to access data that should be unavailable.

4.2 Instruction Fetch Unit

Instructions executed in XOM mode are encrypted while in main memory, but must be unencrypted when passed to the usual instruction decoding logic present on the processor. We suggest using the instruction fetch unit to perform the decryption. A diagram of the modified instruction fetch unit is available in figure 2.

Since changing from XOM mode to ordinary mode, or vice versa, critically affects the way in which the processor operates, the processor must not speculatively execute instructions past a XOM or EXOM. This limitation, however, is precisely analogous to the speculation barriers caused by trap, or software interrupt, instructions that change the processor state from kernel mode to user mode, or vice versa. Like those instructions, the XOM and EXOM instructions will be executed infrequently, so the speculation barrier should not carry a particularly severe penalty.

When the XOM instruction is encountered, the processor will load the session key (the first argument to the XOM instruction) via the instruction fetch path. This key will be provided to the XOM Session Key Decryption unit. (Alternatively, the XOM arguments could be placed in registers, and routed from there to the decryption unit.) We suggest using a 120 bit DESX key, for the session key, although other symmetric key algorithms might also be appropriate.

The XOM Session Key Decryption Unit will decrypt the session key, stalling the instruction fetch unit until the decryption is complete. When the XOM Session Key Unit has completed its decryption and loaded the plain text session key into the XOM Instruction Decryption Unit (see figure 1), it signals the fetch unit to begin operating in the usual fashion. At this point, the fetch unit loads the second argument to the XOM instruction (the destination PC value, called the XPC) into the Fetch PC, and begins fetching from that point. The XPC is simply another input (like a branch target or predicted branch target) to the Fetch PC.

4.3 XOM Session Key Decryption Unit

While even 512 bit keys are considered secure today, we feel a 1024 bit key is probably a better choice, in order to provide greater resistance to brute force methods on ever more powerful hardware. Estimates of the size of the decryption units range from 33,000 to 400,000 transistors with the majority falling around 100,000 transistors[10]. Development in implementations is active and we anticipate the implementations to grow faster and more efficient as time goes on.

Unfortunately, public-key decryption operations are still relatively expensive. The time to decrypt a message with a 1024 bit RSA key is on the order of one to two million cycles, while a 512 bit RSA key would require roughly two hundred thousand cycles. (These issues have been explored in the context

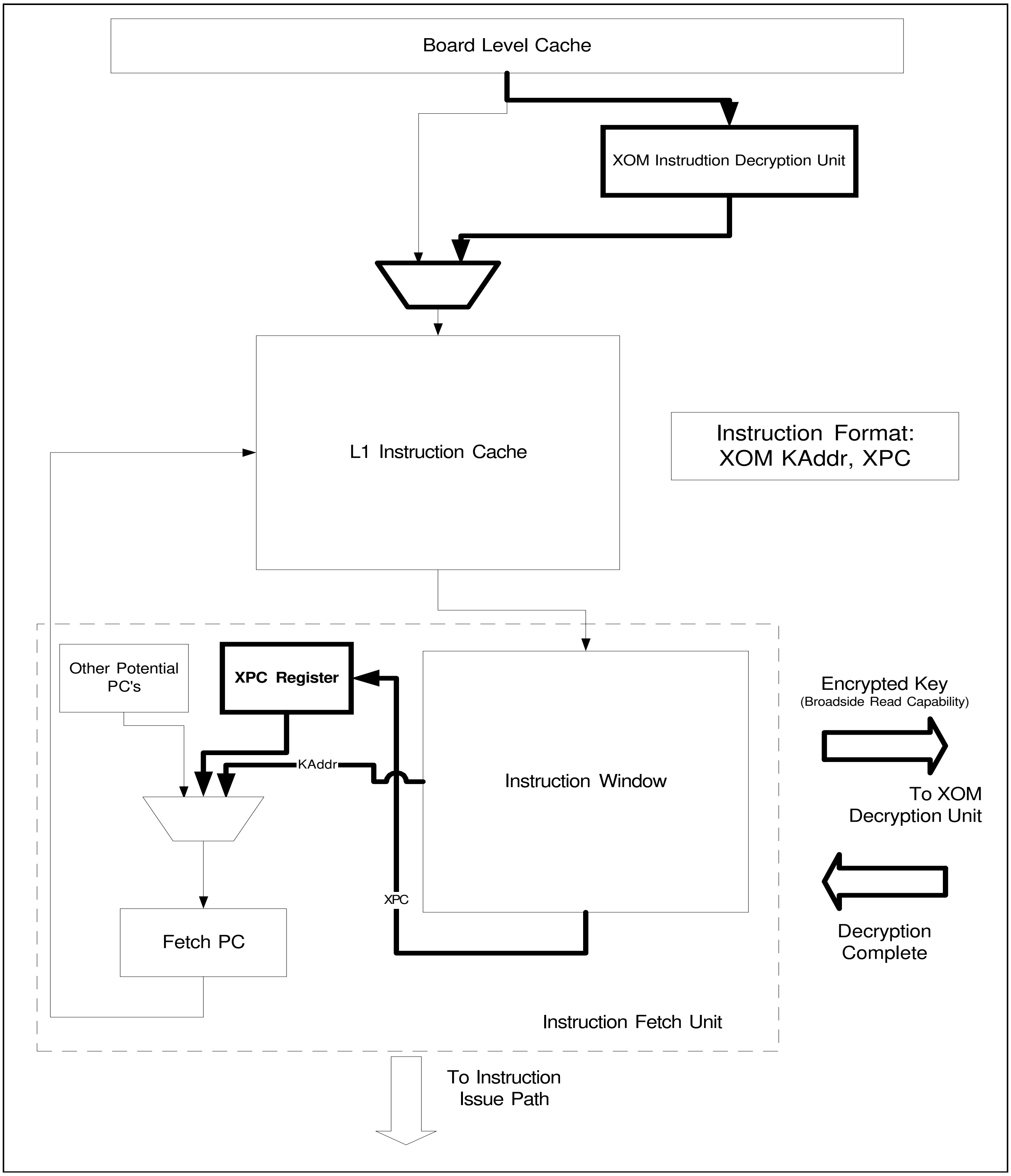

Figure 2: The XOM Instruction Fetch Unit

of implementing RSA signature verification in smart-cards; see for example [7] or [4].)

Because of the extremely long latency of the decryption, we suggest caching the result of the decryption. Since a single application will typically use only one session key, even if run in multiple processes, the probability of getting a hit in such a cache will be very high. Essentially, then, there will be significant latency when the application first enters XOM mode; from that point on the decryption cost will be minimal.

4.4 XOM Instruction Decryption Unit

Once the session key has been decrypted, much faster symmetric key cryptography can be used. DESX encryption has very efficient hardware implementations; its central primitive is bit-permutation rather than the modular exponentiation found in RSA. Figure 3 illustrates one “round” of DES. DESX consists of sixteen of these rounds plus one initial permutation and one final permutation. Each round is 64 bits wide and each block of data is split into an upper and lower 32 bit value. The expansion, P-box and S-box permutations in each stage are fixed and can be performed by wire routing. Since each permutation is independent, the process can easily be pipelined. In addition, since there are no registers involved, it is conceivable that each stage can be made to take one cycle or less; thus the entire structure could perhaps be wave pipelined.

In addition, at the end of the 16 rounds, there must be a small amount of logic to test that the hash of the message matches the current MAC so that the authenticity of the line is verified. A 64 bit MAC that will occupy the last two instructions in a cache line is used. There is some logic to convert the last two instructions of a line into NOPs so that the instruction PCs are not altered.

Finally to implement CBC, the key for the next block is a function of the MAC and the current key. Thus we estimate the latency for the decryption of a 64 bit word or two instructions could be as great as 19 cycles (one cycle for each round, one for each of the initial and final permutations and one more for the MAC check and computation of the next key). While significant, we expect that our estimate may be overly conservative; sufficient pipelining might well reduce this penalty significantly.

A significant drawback of the use of CBC is that the proper decryption of a particular cache line is dependent on the decryption of all previous cache lines. Thus forward branches in XOM code are extremely costly since they require the fetch of all intermediate addresses. Programmers and compilers should endeavor to avoid such branches whenever possible.

4.5 Hardware Debugging

The goal of secure computation is somewhat at odds with the goal of easy debugging and verification of processor hardware. The harder we work to prevent attacks, the harder we make it to verify the correct operation of the hardware. For example, ordinary JTAG port scans would allow an adversary to halt the chip and read sensitive data. In addition to disabling scanning during XOM mode, we could also have two separate scan chains; one for sensitive areas of the chip and one for less sensitive ones. After fabrication testing, we could remove the path to the sensitive scan chain by not bonding the pad to a pin in the package or by laser etching out the wire on the die.

There is also the possibility of removing scan all together from the extremely sensitive areas, such as the storage for the chip specific private key. Inductive tests could be used as an alternative to scanning. For example, dynamically generated vectors containing encryptions of randomly chosen symmetric keys could be fed to the decryption unit; then, verifying that the unit produced the correct

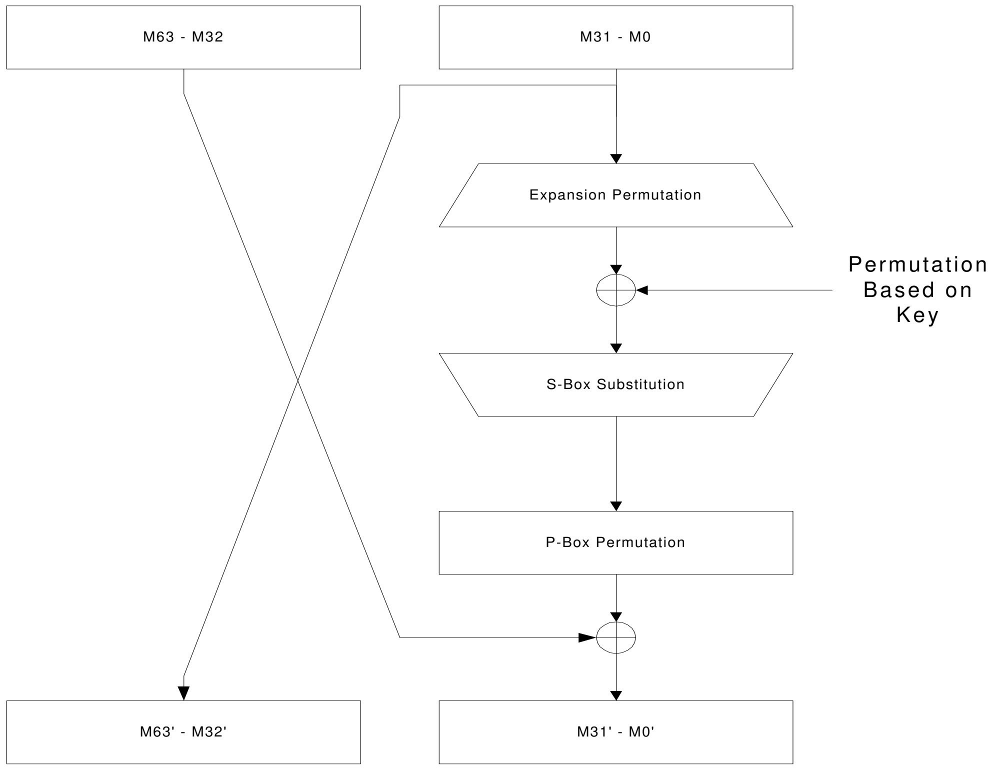

Figure 3: One Round of DES

decrypted values would confirm the integrity of that circuitry.

4.6 Software Issues

¿From a practical perspective, the ability to debug an application is of vital import. Traditional debugger technology relies heavily on being able to inspect the contents of registers and memory, and on being able to dynamically modify machine code. In order to set a breakpoint (i.e., a location in the program at which the debugger will gain control), the debugger would likely replace a machine instruction in the target program with a "trap" instruction. The effect of such an instruction would be to transfer control to the debugger via an interrupt. When debugging XOMed code, however, these techniques are unavailable.

To some extent, it is reasonable to expect users to debug their code before XOMing it. This point of view is analogous to current practice with optimized code. Typically, a user will debug code in an unoptimized form, and enable optimization only once it seems to be working correctly. Unfortunately, optimization all too often introduces bugs, due to problems either in the optimizer itself or in the underlying user code.

Therefore, most modern development environments allow users to debug optimized code as well, although control flow is necessarily altered, and it may be more difficult to examine source-level variables. All debuggers at least allow users to step machine instruction by machine instruction through the compiled output, thereby providing rudimentary debugging capability for the most highly optimized code. With XOM, even this would seem impossible.

Since it is possible that the process of XOMification could introduce subtle bugs, either in the XOMed code, or in the surrounding program, it is desirable to provide the capability to debug XOMed code.

Ordinarily, the software vendor receives a signed public key for a particular processor. The signature allows the vendor to verify that the public key corresponds to a legitimate chip, and is therefore unknown to any person; the vendor can then encrypt the software for that particular chip.

However, if the vendor omits the signature check, then any public key could be used. Equivalently, a single certified key could be provided which does not correspond to a real physical machine. The vendor could then simulate a chip with this XOM key, and provide debugging using the simulated machine. In other words, we can facilitate debugging by permitting precisely the situation which we normally prevent by the use of certified public keys. The debugger could therefore operate normally until reaching XOMed code, simulate the XOMed code, and return to normal operation when execution of the XOMed code is complete.

5 Software Applications

XOM allows software to be shipped securely to a designated user in a manner that prevents tampering or reverse engineering. More specifically, suppose a customer wishes to purchase a software application. We assume that she has not only a public key corresponding to the private key embedded in her XOM processor, but also a publicly verifiable digital signature that allows the validity of this key to be verified. She begins by providing her public-key certificate to the software distributer. The distributer verifies the certificate and sends encrypted software to the customer. As described in Section 2, this code is then sent under a combination of public-key and symmetric key encryption, with message authentication.

It is worth pointing out that without certification of keys, the user could generate a new encryption key for which he has the decryption key, and decrypt the application received, thereby thwarting our entire purpose.

Although it is possible in principle to ship large software applications under encryption, the execution constraints on XOM make it unlikely that this would be considered feasible for most customers. Instead, it seems more feasible to ship small software modules under XOM (such as the decryption unit of a software DVD player). Although full discussion is beyond the scope of this short paper, we believe that it is possible to secure many large applications by securing small critical sections. For example, large applications might be structured so that computation regularly returns to a jump table that determines the future course of computation. If a small jump table is encrypted, this might prevent various forms of modification to the entire application. If structuring applications in this way can prevent the transfer of code from one customer to another, then XOM can effectively reduce software piracy. Since piracy is estimated to have cost the software industry 11 billion dollars in 1998 [1], this potential application is significant.

There may be some resistance to having unique keys on each processor as some fear this may compromise the privacy of computer users. Group signatures with revocation [2] provide one technique for ensuring privacy, while still allowing authentication.

6 Comparison with competing mechanisms

Dongles prevent copying by ensuring that the user has a piece of hardware that typically plugs into the parallel port. Since dongles are hard to copy, they supposedly prevent software piracy. However, when a dongle is used, a clever hacker can disable the "dongle presence" test in the application thus allowing the application to run without the dongle.

Cryptographic coprocessors offload many of the expensive cryptographic instructions onto another processor[12]. Essentially the coprocessor is equivalent to the main CPU while it is in XOM mode. Unfortunately, secure coprocessors are typically much slower than the main CPU and they are expensive. Typically, the problem of tamper resistance is solved by encasing the package in a box that is radiation, heat, impact and electrically sensitive.

7 Summary

In this paper, we have only considered attacks of two forms: malicious code executing on the processor and malicious individuals using debugging facilities of the hardware. Of course, as any cryptographer knows well, there are always more possible attacks!

For example, a recent attack based on power analysis does exist [6]. The technique extracts the key by measuring the chip's power consumption while the secret key is being used. It is most effective against the simple, low power processors used in smart-cards. We do not explicitly account for this since the power consumption signatures will be considerably less obvious when embedded in the power consumption patterns of a complex microprocessor. In addition, much work is already in progress towards a solution[3].

In a similar vein, the PICA system[11] being developed at IBM would allow the adversary to view the switching of any transistor on the chip. If the manufacturer were to mount the chip in such a way that PICA could be used, that system could be used to read the private key.

Space limitations have prevented us from fully explaining the software techniques we envision applications utilizing in order to maximize the power of XOM. Similarly, we have refrained from discussing interesting software systems issues, such as whether or not XOM should be a privileged instruction. (It is reasonable to expect that operating systems might wish to prevent applications from using XOM in some circumstances, such as during periods of intense CPU utilization, or while processing real-time data.)

While these details will be explored elsewhere, we believe that the XOM architecture provides a comprehensive solution for both copy protection and secure execution in a hostile environment. XOM allows software developers to produce software which cannot be altered, reverse-engineered, or observed while executing. It allows software developers to provide cryptographically strong copy protection. We believe the reduction in software piracy resulting from the adoption of XOM could lead to lower software prices for legitimate users, thereby offsetting the cost of developing XOM hardware.

In the future, we hope to implement or simulate the XOM architecture, thereby gathering information on the implications of the architectural decisions made in this paper. A concrete implementation will allow us to experiment with various software protection schemes while measuring the performance impact of XOM.

Because the XOM architecture leverages the functional units found in most modern processors, our design reduces the additional cost required to implement XOM. Although it is unlikely that code running in XOM mode will achieve peak performance, the impact on non-XOM is minimal. Because XOM code runs infrequently and only for a short time, the overall change in processor performance will be relatively small.

References

[1] Business Software Alliance (BSA), `http://www.bsa.org`

[2] D. Boneh, M. Franklin, "Anonymous authentication with subset queries", in proc. of 6th ACM Conference on Computer and Communications Security.

[3] S. Chari, C. Jutla, J. Rao, P. Rohatgi, "Towards sound approaches to counteract power analysis attacks", in proc. of CRYPTO '99, pp. 398–412, 1999.

[4] J. Dhem, D. Veithen, J. Quisquater, "SCALPS: Smart Card for Limited Payment Systems", IEEE Micro, June 1996.

[5] E-Stamp description, `http://www.estamp.com/our_service/s_overview.html`.

[6] P. Kocher, J. Jaffe, B. Jun, "Differential Power Analysis: Leaking Secrets", in proc. of CRYPTO '99, pp. 388–397, 1999.

[7] D. Naccache, D M'Raïhi, "Cryptographic Smart Cards", IEEE Micro, June 1996.

[8] A. Patrizio, "Why the DVD Hack Was a Cinch," *Wired*, Nov 2, 1999, `http://www.wired.com/news/technology/0,1282,32263,00.html`.

[9] D. Patterson, J. Hennessy, "Computer Architecture – A Quantitative Approach", 2nd ed, Morgan Kaufmann Publishers, Inc., 1996.

[10] B. Schneier, "Applied Cryptography", 2nd ed. Katherine Schowalter, 1996.

[11] D. Vallett, J. Kash, J. Tsang, "Watching Chips Work", http://www.chips.ibm.com/micronews/vol4_no1/vallett.html

[12] http://www.ibm.com/security/cryptocards/html/pr_fips.html