Fast Secure Processor for Inhibiting Software Piracy and Tampering

Jun Yang
Computer Science and Engineering Department
University of California, Riverside
Riverside, CA 92521
{junyang,lgao}@cs.ucr.edu

Youtao Zhang*

Lan Gao
*Computer Science Department
University of Texas at Dallas
Richardson, TX 75083
zhangyt@utdallas.edu

Abstract

Due to the widespread software piracy and virus attacks, significant efforts have been made to improve security for computer systems. For stand-alone computers, a key observation is that other than the processor, any component is vulnerable to security attacks. Recently, an execution only memory (XOM) *architecture has been proposed to support copy and tamper resistant software [18, 17, 13]. In this design, the program and data are stored in encrypted format outside the CPU boundary. The decryption is carried after they are fetched from memory, and before they are used by the CPU. As a result, the lengthened critical path causes a serious performance degradation.*

In this paper, we present an innovative technique in which the cryptography computation is shifted off from the memory access critical path. We propose to use a different encryption scheme, namely "one-time pad" encryption, to produce the instructions and data ciphertext [1]. With some additional on-chip storage, cryptography computations are carried in parallel with memory accesses, minimizing performance penalty. We performed experiments to study the trade-off between storage size and performance penalty. Our technique improves the execution speed of the XOM architecture by 34.7% at maximum.

1. Introduction

Software copyright protection plays an important role in assuring the software market value and a fair return on their development investment. A study in 2001 done by the Business Software Alliance showed a 12 billion dollar loss in the software industry due to software piracy [2]. Preventing illicit duplication of software will have a large impact on economic development. Therefore, it is important to develop foolproof devices that disallow unauthorized execution of software.

---

[1]Ciphertext is the term assigned to encryption result. Likewise, plaintext is unencrypted instruction or data

Several techniques have been proposed to provide hardware support at micro-processor level against software piracy [19, 18, 17, 15, 13]. In those techniques, the only trusted hardware entity is the processor itself. Any other hardware components in the computer system are considered vulnerable to security attacks, particularly the co-processor and the main memory. This is because program privacy can be violated by tapping the communication channel such as the system bus. An adversary can easily tamper the execution of program once some knowledge of the code is obtained. Moreover, the operating system is also considered non-tamper resistant since it may be hijacked by the adversary to become malicious to the software running under its control.

The software is stored in the system storage in encrypted form. It can only be decrypted by the processor internally before execution. This prevents any user having the full control of the computer from examining the clear text instruction. More importantly, the data communicated between the processor and the memory are all encrypted to prohibit reverse-engineering the code. To protect from the potential malicious operating system that can access the register values on interrupts, register values need to be encrypted also on such events. The representative technique of the above model is called *execution only memory*, or XOM, meaning that software can be executed by the owner processor only but not copied (since it would not run on other processors) nor manipulated (since it would raise exceptions and then halt) by unauthorized entities [18, 17].

Though secure at a satisfactory level, one of the most important problems in the XOM-type architecture is its efficiency. As one may notice, every off-chip memory transaction including both instruction and data undergoes encryption and decryption. Even with the most optimistic assumption of finishing the crypto process in 48 cycles with fully pipelined hardware [18], performance loss can be as high as 34.7%, as our experiments indicate. The situation is even worse for applications that are memory bound or time critical. For this reason, the usefulness of the XOM architecture

is yet to be evaluated. Software users would find it very annoying every time the program runs significantly slower than the unprotected version, diminishing the attractiveness of copyright protection.

The purpose of this paper is to relieve the performance burden on XOM-type architecture. We propose to off-load the crypto computation from the critical path. In XOM architecture, instructions and data can not be used until they are fetched from the memory and decrypted afterwards. We propose to perform decryption in parallel with a memory access, overlapping crypto-computation time with memory latency.

Our technique strives to maintain the same level of security strength as the XOM architecture. Thus, our work is based on its proposed mechanisms in handling potential attacks. No attempts are made to enhance its security level. Our design also requires extra on-chip storage and we studied the trade-offs between storage size and performance improvement. Experimental results show this technique is able to lower the 16.7% average performance loss of XOM architecture to only 1.28% over the insecure baseline processor.

The remainder of the paper is organized as follows. We first describe briefly the XOM architecture in Section 2. Then we elaborate on the idea of off-loading the cryptography computation from critical path in Section 3. We illustrate the detailed architecture design in Section 4. In Section 5, we show the experimental results on performance gain with various hardware configurations. In Section 6, we give a brief description of the related work, and conclude the paper in Section 7.

2 XOM Architecture Overview

2.1 Software Encryption and Decryption

Background There are two major types of cryptography commonly used in information systems today: symmetric key ciphers and asymmetric key ciphers (see Figure 1). In symmetric key cryptography, communicating parties share a common private key in encryption and decryption. The advantage is that it runs as much as 1000 times faster than comparable asymmetric key ciphers [7]. The primary obstacle is the distribution of the private key to information exchange parties. Asymmetric key ciphers solve this problem by implementing encryption using a key pair: public key and private key. Information is encrypted using the publicly available public key at the sender, and decrypted using the private key which is kept secret by the receiver. Thus, the sender can send information securely without knowing the receiver's private key.

XOM Software Encryption The software that runs on the XOM architecture is encrypted by the vendor. The encryption not only protects the privacy of the software algorithm
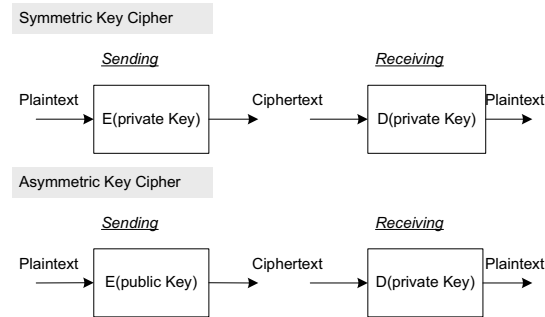
Figure 1. Illustration of symmetric and asymmetric ciphers

but also guarantees that it can only run on the target processor. To maximize security and performance, the software is encrypted using a combination of symmetric and asymmetric key cryptography. The vendor first encrypts the software using some fast symmetric key cipher with private key k_s. The decryption of the program using the same key is relatively fast. The XOM chip is installed with a private decryption key k_{xom} of a public-key encryption pair. The corresponding public key, k_p, is available to the public. To communicate the k_s to the processor, the vendor uses k_p to encrypt it and ships it along with the software. The execution of the protected software begins with computing k_s using k_{xom} which is carried only once but might take a relatively long time, and decrypting instructions using k_s which is much faster but is carried on every instruction fetched into the processor. In this way, software encrypted for $processor_1$ can not run on $processor_2$ since they have distinct private keys.

2.2 Interacting with External Memory

The XOM architecture adopts a complicated mechanism in protecting the program data privacy and providing memory integrity verification. Ensuring privacy means to keep data information hidden from anyone for whom it is not intended. This is achieved through data and instruction encryption. Memory integrity verification is to detect if the memory has been tampered with by an adversary. This is accomplished by creating a *hash (MAC)* value for each memory block [2]. A cryptographic hash function can take inputs of any length and produce a fixed length output. It is "one-way", meaning that it is computationally infeasible to find the original data given the hash value, and relatively easy to compute. Hashing is especially useful in the three types of attacks considered in XOM: *spoofing*, *splicing*, and *replay*. The first two attacks were handled satisfactorily in XOM. The *replay* attack is better developed for performance improvement by Gassend *et. al.* [11]. Thus,

---

[2]The block was chosen as an L2 cache line size.

we do not address the issue of verification and concentrate on speeding up encryption and decryption process in this paper.
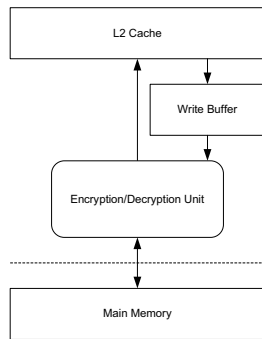

Figure 2. The lengthened XOM memory path.

To mitigate the performance impact, XOM pushes data encryption and decryption through the memory hierarchy so that it is only done when the data leaves the processor and enters insecure memory. Thus, all the on-chip caches are secure and store data and instructions in plaintext. Figure 2 illustrates the abstract model of the crypto procedure. A two-level cache structure is assumed in the processor. Writing to the memory is deferred through the write buffer. Every dirty L2 cache line is encrypted first and then sent down to memory. Likewise, every line read from the memory is decrypted before it is stored in L2 cache and used by the program.

2.3 Internal Protection for Multi-tasking

A major effort in designing XOM secure processor goes to protecting interactions among multiple active tasks. Each task is protected by a strict perimeter, termed "compartment". Each compartment has its own ID and a secrete key which was used for encrypting the program. The compartment ID is used in tagging data written into the registers and the caches. This tagging ensures no programs can access the data of another program.

New instructions are added to support security functionalities. They are used for handling start/termination of XOM mode, communication between programs, traps and interrupts, and storing and loading cryptographic data to and from memory traps and interrupts.

3 Offloading Crypto-Computation from Critical Path

In this section, we present a scheme that shifts the computation intensive crypto-process off from the critical path. First we analyze the performance degradation in XOM architecture.

3.1 Motivation

As we can see from Figure 2, the crypto hardware lies on the memory access critical path and therefore, the performance decrease is obvious. Developing fast crypto hardware has been the major focus recently to accelerate security applications [24, 3, 21, 23]. However, in spite of the effort in crafting the designs, the crypto-hardware here still inserts long latency on memory access due to the computation intensive nature.

Figure 3 shows the performance degradation due to the prolonged memory path in XOM architecture. We tested over 11 SPEC2000 [14] benchmarks with 32K separated L1 instruction and data cache and 256K L2 unified cache on an out-of-order 4-issue processor simulation using SimpleScalar [4]. We assumed a typical 100 cycle memory latency and a 50 cycle encryption/decryption delay similar to that in [18]. Such a fast hardware for widely used symmetric ciphers, e.g. DES [9], is possible with ASICS designs [10]. For stronger ciphers such as AES [1], a longer encryption/decryption latency would apply, which in turn results in longer average memory access latency. Thus, we give an optimistic estimation of the potential performance degradation in reality. On average, there is 16.7% slowdown posed to the programs. For memory intensive programs such as database applications, the delay will be more severe.
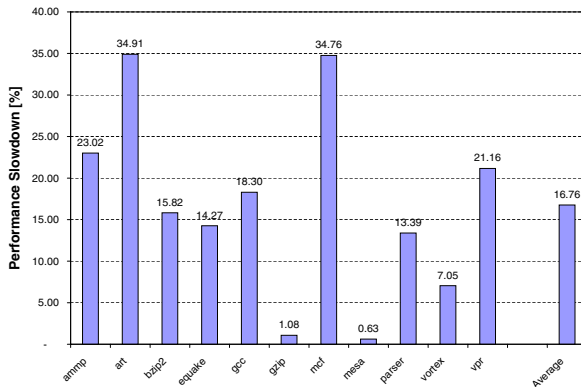

Figure 3. Optimistic estimation on performance loss due to encryption/decryption.

3.2 Proposed Solution

The difficulty in XOM lies in the fact that the crypto-computation is data dependent on memory accesses, i.e., without knowing the data to be written out or brought in the encryption or decryption can not begin. We propose to use a different encryption algorithm to generate the ciphertext in memory. The creation of the ciphertext must be data independent on the memory access so that it can be carried in parallel, not in serial with the memory operation. The designated ciphertext must also be related to the memory access so that each access has a unique ciphertext.

We propose to use an algorithm similar to "one-time pad" encryption [22] for both data and instructions in memory. In one-time pad encryption, the ciphertext is the exclusive-or of the plaintext and a true random key:

$$c \;=\; p \oplus random\ key \qquad (1)$$

where c is the ciphertext, p is the plaintext data value, and $random\ key$ is a true random number having the same bit width as p. In our model, we replace the $random\ key$ with an encrypted *seed*. The seed uniquely corresponds to the plaintext, and can be generated regardless of its value (see Section 3.4). Thus, with the "one-time pad" algorithm, the encryption and decryption of a plaintext data value can be expressed as the following:

$$c \;=\; p \oplus encrypt_k(seed) \qquad (2)$$
$$p \;=\; c \oplus encrypt_k(seed), \qquad (3)$$

where c is stored in insecure memory, p is the plaintext data value, and k is the private key shipped with the software. Operationally, when p is sent off chip, equation (2) is used; when p is read from memory, equation 3 is used. Calculating $encrypt_k(seed)$ is carried while the processor is waiting for the memory. Let us assume the memory access latency is 100 cycles and computing $encrypt_k(seed)$ is 50 cycles as before. When c is loaded from memory and arrives at the processor, $encrypt_k(seed)$ is already ready. With an additional one-cycle XOR, p can be obtained and sent to the processor for execution. Thus, instead of having 100+50 cycles delay, we now reduce it to 101 (i.e. MAX(100, 50)+1).

3.3 Encryption Strength

Using the one-time pad encryption (equation 2 and 3) achieves the same strength as a normal data encryption where $c = encrypt_k(p)$. This can be seen through the analogy of the proposed scheme and the *stream ciphers*[20]. The stream cipher is similar to one-time pad. The difference is that it uses pseudorandom number stream instead of a genuine random number stream. Many widely used encryption algorithms such as AES[1] and 3DES [8] are believed to do a good job in generating pseudo-random numbers. As a result, $p \oplus encrypt_k(seed)$ is as random as $encrypt_k(p)$ where the selection of seed is discussed next [3].

3.4 Seed Selection

The purpose of encrypting a seed instead of value itself is to do it in parallel with memory read operation. We do not consider penalty due to memory writes in this paper since most processors are equipped with write buffers which can

---

[3]The seed used here should not be confused with the seed that is used in random number generation functions supported by many higher level languages. As an example, the seed in C function sram() represent a starting point of in a chain of "so called" random numbers. This is not the case in our design. We treat the seed as an input to the encryption function.

steal idle bus cycles efficiently. Therefore, the seed must be available at the time the read command is issued to memory. It is also important to differentiate seeds for different encryption units, i.e. blocks, to de-correlate program data. Naturally, a seed *derived* from the location, e.g. address, of a value is a good candidate. Let us see why using addresses alone might be good in some cases and bad in the others.

Advantage: In the XOM model, every data value is encrypted directly and stored in its memory location. This implies that same data values at different locations have same ciphertexts. It is known that the memory contains a lot of repeated values [16, 25]. Thus even with encryption, the repetition pattern still preserves, creating potential security holes.

Using address of a data value as the seed in equation (2), each $encrypt_k(address)$ is different from others. Moreover, the property of an encryption function assures no patterns exist between sequential addresses, i.e. $encrypt_k(address)$ and $encrypt_k(address + 4)$ are completely unrelated, hence the neighboring memory ciphertext.

Disadvantage: However, for the same location, $encrypt_k(address)$ remains the same every time the value is written into memory. Thus, a series of data value 0, 1, 2,... generated at address $addr$ will have a series of ciphertexts $encrypt_k(addr)$, $1 \oplus encrypt_k(addr)$, $2 \oplus encrypt_k(addr)$ ... which amounts to

$$C, \; 1 \oplus C, \; 2 \oplus C \ldots$$

where C is a constant. With little effort, the ciphertexts stored in memory can be cracked by a skilled attacker. Therefore, the seed used for such a series of writes should not be a constant, i.e. it should vary. This is also pointed out in the XOM architecture for saving register values on OS interrupts. In such cases, a *mutating value* for varying the XOM ID is employed for encrypting register values on each interrupt event. To mutate the seeds in equation (2), we choose to adopt a sequence number associated with an address. The sequence number is updated every time it is used. Thus, the encryption becomes $encrypt_k(addr + sequence\ number)$. The details will be fully described shortly.

At this point, it is necessary to separate situations for encrypting instructions and data. The above analysis on the disadvantage of using address directly as seed applies to data writing only. For instructions, there are only read operations as they are only loaded from but never written back to memory. Therefore, a constant seed directly associated with instruction address can be used.

3.4.1 Encrypting Instructions

The instructions are encrypted by the vendor but are executed on the customer's processor. The vendor does not know the actual addresses when the program is loaded into the customer's memory space for execution. Therefore, it is easier for the vendor to use the virtual address starting from, for example, $V0$. Suppose the vendor chooses a symmetric key KEY, encryption function DES with block size of 64 bits. Each instruction is 32-bit. A sequence of instructions $I_1, I_2, I_3, I_4, I_5, I_6, \ldots$ will be encrypted as:

$$(I_1|I_2) \oplus DES_{KEY}(V0), \quad (I_3|I_4) \oplus DES_{KEY}(V0+8),$$

$$(I5|I6) \oplus DES_{KEY}(V0+16)\ldots$$

where "|" means concatenating two 32-bit instructions into a 64-bit data block. To decipher the program, the processor simply adds to $V0$ the offset of the current 64-bit instruction block to the first instruction block, obtaining the seed for the encryption. When the ciphered instruction is available from memory, plaintext instructions can be computed through XORing the encrypted seed in only one cycle.

3.4.2 Encrypting Memory Data

On-chip data are encrypted when they are evicted out due to cache conflicts. We assume a two-level cache structure as in most high-performance processors. Similar to XOM, encryption and decryption are done on per L2 cache line basis. Since we adopt sequence numbers on writes to the same memory location, each sequence number is maintained for each L2 cache line. The initial values of the seeds are the virtual cache line addresses. It is incorrect to use the physical line addresses since programs may be loaded to different physical memory spaces on context switches. As cache lines are transmitted across the chip boundary, the seeds are increased by the corresponding line's sequence number. Thus, on the ith $(i > 1)$ write to memory for a line L, the following steps are taken in sequence:

$$seq\_num_i = seq\_num_{i-1} + system\_timer; \quad (4)$$

$$seed_i = virtual\_line\_address(L) + seq\_num_i; \quad (5)$$

$$Ciphertext = Plaintext \oplus DES_{KEY}(seed_i) \quad (6)$$

where $seq\_num_1 = 0$. On a line L read:

$$seed_i = virtual\_line\_address(L) + seq\_num_i; \quad (7)$$

$$Plaintext = Ciphertext \oplus DES_{KEY}(seed_i) \quad (8)$$

Reading a cache line may happen long after it was written to the memory. To make sure it is available when a line is being fetched, we need to remember the sequence number that was previously assigned to the line. Next, we give the details of the design of a special on-chip cache that stores the sequence numbers. The sequence number cache should locate within the security boundary as in the XOM architecture.

4 Architecture Design

As clarified earlier, an on-chip sequence number cache (SNC) is needed in order to store sequence numbers for each cache line that goes off-chip. Thus, we place the SNC below the L2 cache and monitor the traffic between L2 and the memory. Figure 4 illustrates the architecture of the abstract partial XOM model with our SNC.
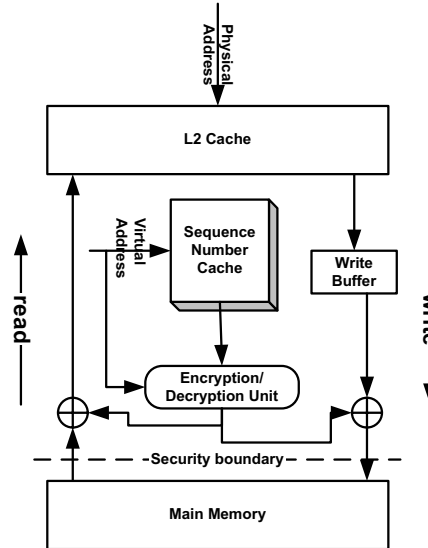
Figure 4. Design of one-time pad encryption on data with sequence number cache.

The SNC should be accessed using the virtual address (VA) of an L2 cache line. This is because physical address of a line may be changed after a context switch, losing encryption seed information. However, using VA to index the SNC may incur synonym problems in which two different VAs may map to the same physical address. The result is that two different sequence numbers may be generated for the same physical line. The synonym problem happens when either the OS and the user, or two users want to share a memory segment. The XOM architecture is very restrictive in sharing data among tasks, including the OS. The solution proposed by XOM is to share a key among tasks that have synonyms, which is considered vulnerable. Since this is still open problem in XOM, we choose not to perform one-time pad encryption on those shared data. In other words, SNC does not store the sequence numbers for memory segments that are aliased by two different virtual addresses.

In conventional cache design, the VA will not be available beyond L1 caches, and the L2 cache is physically addressed. Thus, the VA of each L2 cache line should be kept

within the L2 cache. The stored VA can be then used to address SNC on a cache write back. The storage incurred due to storing VA's in L2 cache is very modest. For example, in a 256KB L2 cache having 128B each line, 40 bits of a 48-bit VA (e.g. in Alpha architecture) need to be stored, enlarging L2 cache by 4.0%.

Ideally, the SNC should store all the sequence numbers of memory lines. Take a 1GB memory and a 128B line size as an example, 8M (1GB/128B) sequence numbers are necessary to be remembered. Having an 8M on-chip cache is unrealistic to ask. We therefore provide only a limited sized SNC which stores sequence numbers efficiently. To remove conflict misses as much as possible, a fully associative cache is desired. A fully associative cache normally provides the best hit rate but occupy larger chip area and take longer time to access. Normally, a highly associative cache, e.g. 32-way or 64-way, would perform equally well. We will present most of our experimental results using a fully associative SNC implementation in Section 5, and also show the results with a 32-way set associative SNC.

With a limited amount of SNC storage, not all sequence numbers can be stored on-chip. Thus, when the SNC is full, no further sequence numbers can be stored unless some stored contents are evicted out. If so, where will the evicted sequence numbers be stored? Complication arises as to whether a replacement policy should be employed, and what may happen with or without a replacement policy.

4.1 SNC Operation Policy

With Replacement If replacements are carried in the SNC, we need to solve where those evicted sequence numbers should be stored. It is clear to see that we can not discard them since otherwise, their corresponding memory lines would not be able to be deciphered. Then the only solution is to store them in the insecure memory. To protect the privacy of these sequence numbers, we choose to encrypt them just as normal program data. It should be noted that even without encryption, the on-chip one-time pad encryption remains secure since the private key is not revealed. However, it is not preferred that the sequence numbers are encrypted using one-time pad again since they themselves would need sequence numbers! Therefore, we choose to use encryption on the sequence numbers directly, just as the XOM solution.

The advantage of allowing replacement in SNC is to make one-time pad encryption available to as many memory lines as possible. If LRU replacement is adopted, the SNC will catch frequently used sequence numbers in the long run so as to reduce the SNC capacity misses. However, each replacement incurs another memory access plus the encryption latency of the contents. Although this does not necessarily happen on critical path, it imposes additional memory traffic and may compete with other memory requests

that are critical. Thus, the number of replacement should be small enough to overcome the above defect. Using LRU in this sense, helps reduce the SNC replacement frequency.

With No Replacement An alternative way is to disallow replacements. In such a situation, the one-time pad encryption is carried as long as there are vacant slots in the SNC. When SNC is full, cache lines whose sequence numbers are not stored in the SNC will not be able to perform one-time pad encryption. Consequently, they should be encrypted directly and sent to memory. The advantage of no replacement policy is its simplicity. The disadvantage is, however, only partial memory lines can employ one-time pad encryption, the rest are treated the same way as in XOM. We will show in Section 5 that using LRU is more advantageous than the non-replacement SNC.

4.2 Algorithm

In this section, we discuss the SNC query (i.e. read) and update (i.e. write) operations in great details. To be clear, we categorize various operations into query hits, update hits, query misses, and update misses. SNC is filled with update operations and looked up through query operations.

SNC Hits A query hit in SNC happens when a read miss occurs at L2 and the target line's sequence number is stored in SNC. A seed is then calculated using equation 7. After the memory access returns, the plaintext value is then obtained by applying equation 8. Assuming a 100-cycle memory latency and 1-cycle XOR, the value is ready to the CPU at the 101th cycle. An update hit in SNC happens when a L2 cache line is evicted down to the memory and this line's sequence number is stored in SNC. The sequence number is updated according to equation 4. Seed is formed and line is ciphered using equation 5 and 6 respectively. Note that the evicted line should first go to the write buffer (Figure 4) and are later flushed to the memory on certain conditions. Thus, the encryption can be done while they remain in the write buffer. With SNC, the delay is nearly the same as in XOM except that the XOR takes one more additional cycle. Since write operation is not on the critical path, its impact on overall performance is not a big concern.

SNC Misses Misses in SNC are more complex, especially in supporting LRU replacement. We will separate the no-replacement and LRU replacement designs for clarity. In no-replacement SNC, an update miss means no free entries are available. At this time, the cache line has to be encrypted directly like in XOM. A query miss means the corresponding L2 cache line's sequence number is not stored in SNC. As mentioned earlier, those lines were encrypted directly. Thus after the line is fetched from the memory, it should go through the decryption unit which is another 50 cycles on top of 100 cycles.

With LRU replacement, every L2 cache line has a se-

quence number. For those that can not fit in the SNC, they are stored in memory. As pointed earlier, sequence numbers in memories should also be encrypted (directly). On an SNC query miss, a memory access is needed to fetch the target sequence number, followed by the decryption. Thus, each query miss incurs 150 cycles before the seed encryption can start, becoming the most expensive operation. As such, an update miss in SNC also needs to access memory and decrypt the ciphered sequence number. Since this is carried while the cache line is in write buffer, impact is less significant. Algorithm 1 gives the pseudo-code for handling the SNC misses.

Algorithm 1 Pseudo-code for handling SNC misses employing LRU replacement

---

1: **if** SNC query miss on cache line L **then**
2:     $sn \leftarrow$ read memory for L's sequence number
3:     $D_{sn} \leftarrow Decrypt_{KEY}(sn)$;
4:     **for** each block va_{blk} in L's virtual address va **do**
5:        $E_{blk} \leftarrow Encrypt_{KEY}(va_{blk} + D_{sn})$; /* executed in fully pipelined engine, in parallel with line 7 */
6:     **end for**
7:     $Cipher_L \leftarrow$ read L from memory
8:     **for** each block C_{blk} in $Cipher_L$ **do**
9:        $P_{blk} \leftarrow E_{blk} \oplus C_{blk}$; /* P_{blk}'s form plaintext for L */
10:     **end for**
11:     replace a victim V_{sn} in SNC with D_{sn};
12:     push V_{sn} in write buffer; /* to be encrypted later */
13: **else if** SNC update miss on cache line L **then**
14:     $sn \leftarrow$ read memory for L's sequence number
15:     $D_{sn} \leftarrow Decrypt_{KEY}(sn)$;
16:     $D_{sn} + = system\_timer$;
17:     **for** each block va_{blk} in L's virtual address va **do**
18:        $E_{blk} \leftarrow Encrypt_{KEY}(va_{blk} + D_{sn})$; /* executed in fully pipelined engine */
19:     **end for**
20:     **for** each block P_{blk} in plaintext L **do**
21:        $C_{blk} \leftarrow E_{blk} \oplus P_{blk}$ /* C_{blk}'s compose ciphertext of L */
22:     **end for**
23:     write ciphertext of L into memory;
24:     replace a victim V_{sn} in SNC with D_{sn};
25:     push V_{sn} in write buffer; /* to be encrypted later */
26: **end if**

4.3 Other Security and Implementation Issues

Context switching One of the difficulties we realized is to handle situation in context switching. On context switches, XOM architecture employs expensive operations not to leak information to potential malicious OS and other users. The contents of our SNC should also be protected as the new user may use it for its own purpose. There are two ways of protecting the SNC: 1) flushing it to the memory with encryption; and 2) tag each entry with XOM ID. Each method encounters long latency either during context switching or after. Fortunately, context switching does not occur very often. The impact on the overall performance in multi-task systems is currently open.

Shared library and program inputs. If the software package contains shared library code, e.g. .dll, they are meant for usage by multiple users. Therefore, those library codes should be provided in plaintext. Similarly, program inputs are also provided in plaintext since they are brought in from I/O devices. As a result, memory spaces taken by them do not need sequence numbers in SNC.

5 Experiment Evaluation

We implemented the two schemes in order to compare the one-time pad encryption scheme with XOM. We used SimpleScalar Tool Set [4] to run 11 SPEC2000 [14] benchmarks, and compared performances for various algorithms and configurations. The benchmarks are fast forwarded by 10 billion instructions to warm up the pipeline as well as L1 and L2 caches, and then continued to execute for another 10 billion instructions so that they finish within reasonable amount of time. Our baseline is a 4-issue out-of-order execution processor with 32KB, 4-way, L1 separate instruction and data caches, plus a 256KB, 4-way, 128B per line, L2 unified cache. We set the memory access latency as a typical 100 cycles, the encryption/decryption delay as 50 cycles as before. All other parameters are set as default values provided by SimpleScalar.

5.1 Performance Comparison

The first set of experiments measures the performance loss due to security operations. We compare the XOM architecture with one-time pad encryption having an SNC. As described in Section 4, the SNC can either allow or disallow sequence number replacements. We plot the result for both schemes as shown in Figure 5. Here the SNC is set to 64KB, each sequence number taking 2 bytes. Thus, there are 32K numbers stored in the SNC, covering 32K L2 cache lines, or 4MB memory data space. It is clearly shown from the graph that our scheme drastically reduces performance loss—from 16.7% to 4.59% for no replacement SNC and 1.28% for LRU SNC. We can draw two conclusions from these results:

1. Using one-time pad encryption is an excellent solution to minimize performance degradation of secure processors. The 1.28% slowdown from the LRU SNC design is not noticeable to the user and thus increases the practicability of a secure processor.

2. The difference between no-replacement and LRU

proves that using the latter is beneficial in the long run since it will catch relatively frequently accessed cache lines. For example, the benchmark `gcc` shows a big difference between the two, sequence numbers filled into the SNC initially are hardly used later.
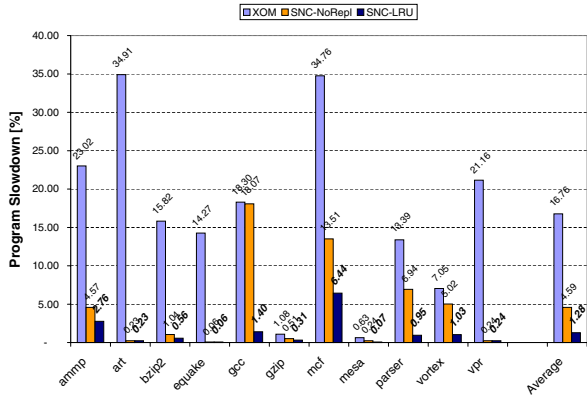

Figure 5. Performance comparison for XOM, SNC with LRU and no cache replacement.

5.2 SNC of 32KB, 64KB, and 128KB

The second set of experiments intends to answer how our scheme is sensitive to SNC size. To see this, we tested 32KB, 64KB, and 128KB SNC with LRU. Figure 6 shows the execution slowdown in percentage of the baseline. We can see that with smaller SNC, the scheme under-performs the larger SNCs. Since a 128KB on-chip cache may be a high requirement for processors, we conclude that the 64KB is a better choice among the three.
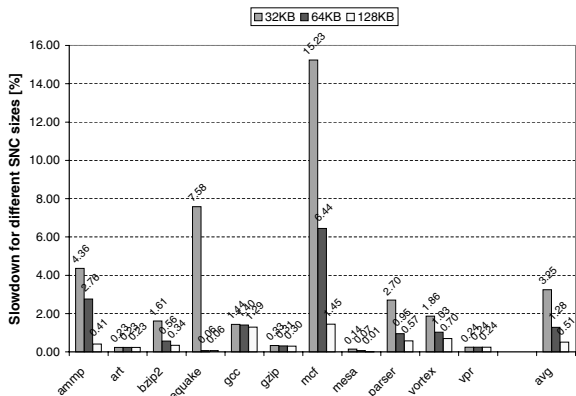

Figure 6. Performance comparison for different sized SNC. LRU replacement is used.

5.3 SNC of Different Associativity

The third set of experiments is to see if a fully associative SNC is really necessary. Implementing a 64KB cache with full set associativity might be expensive. We therefore ran the benchmarks with a 32-way, 64KB SNC, and compare it with the fully associative, 64KB SNC. Figure 7 plots the results. Apart from one benchmark `ammp` (which increases the slowdown from 2.8% to 9.6%), all the rest programs show an equivalence of using the two caches. Sometimes, 32-way is even slightly better. Therefore, in most cases, a 32-way SNC serves as good as a fully associative SNC.
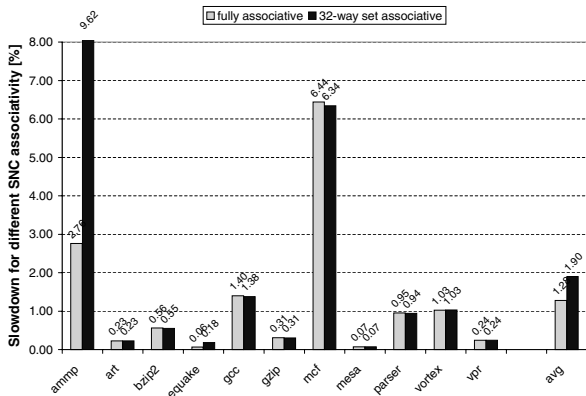

Figure 7. Performance comparison between fully associative and 32-way set associative SNC's.

5.4 Larger L2 vs. L2+SNC

The fourth set of experiments we conducted is to justify the added on-chip SNC storage is indeed very effective. We show this by comparing the execution time for LRU SNC with a XOM architecture that has a larger L2 cache size. A fair comparison requires that the enlarged L2 occupies the same amount of chip area as the original L2 plus SNC since the increase in cache area is not linear to its capacity. We used CACTI 3.2 [5] to obtain the area estimation. We found that a 64KB 32-way set associative SNC on top of a 4-way 256KB L2 cache occupies chip area between that of a 5-way 320KB and a 6-way 384KB L2 cache. We therefore compare our configuration with XOM having a 6-way 384KB L2 cache. Figure 8 plots the normalized execution time *w.r.t.* the baseline having 4-way 256KB L2 cache. With the same amount of on-chip area, our one-time pad encryption scheme still outperforms XOM on average (2% vs. 9% slowdown). Program `gcc`, `mesa` and `vortex`, show a speedup of 4%, 1% and 7% in execution time compared to the baseline. This is because with 50% of capacity increase in L2, almost everything in the two programs fit into L2 and thus the need of going off chip is greatly reduced. This experiments show that in general, having a larger L2 cache can not mitigate the performance impact of XOM while using the one-time pad encryption is satisfactory.
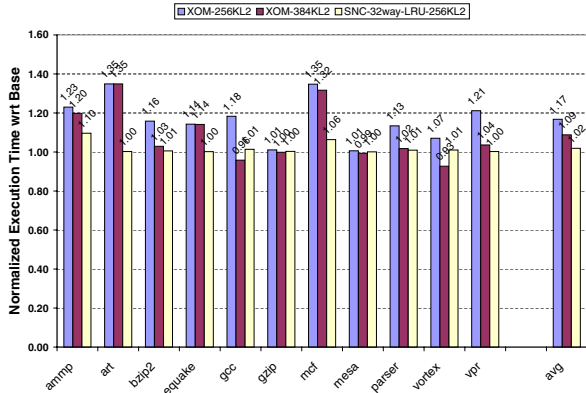
Figure 8. Impact of a larger L2 cache.

5.5 SNC Induced Memory Traffic

The fifth set of experiments is designed to show the induced memory traffic due to SNC LRU replacements. The results are measured in percentages of L2 cache memory traffic. See Figure 9. We can see that the effect of SNC replacement is negligible in terms of memory traffic increase. For quite a few benchmarks, `art`, `equake`, `vpr`, the increase is almost zero. On average, there is only 0.31% of the L2 memory traffic posed on to the system bus. This also explains why SNC with LRU performs best even though replacements are expensive.
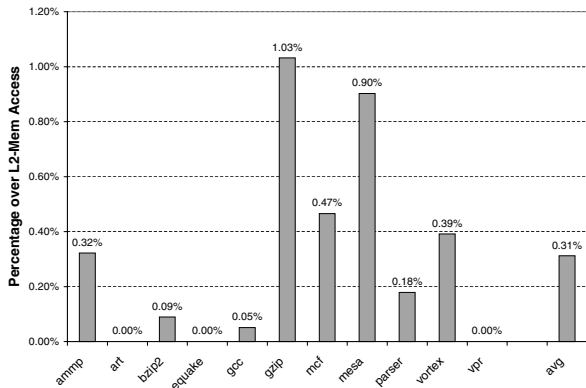

Figure 9. SNC induced additional memory traffic (64KB SNC).

5.6 Sensitivity to Encryption Latency

Our one-time pad encryption has an advantage in that it is insensitive to the cryptography latency. Compared with the XOM memory latency, $mem\_lat + crypto\_lat$, the new memory latency on cache read misses (which is critical to speed) is now $MAX(mem\_lat, crypto\_lat) + 1$. Therefore, we performed experiments that use a different encryption/decryption latency, 102 cycles [12]. Results are shown in Figure 10. It is clearly seen that the XOM degrades greatly with the prolonged encryption latency: from 16.7%

to 34.2% slowdown. This is because 102-cycle roughly doubles the original 100-cycle latency. While in our design with LRU replacement, the performance is almost unchanged: from 1.28% to 1.3%. The difference between the no-replacement policy and the LRU also proves that the latter is much more effective than the former. This result enhances the usability and attractiveness of our proposed one-time pad encryption.
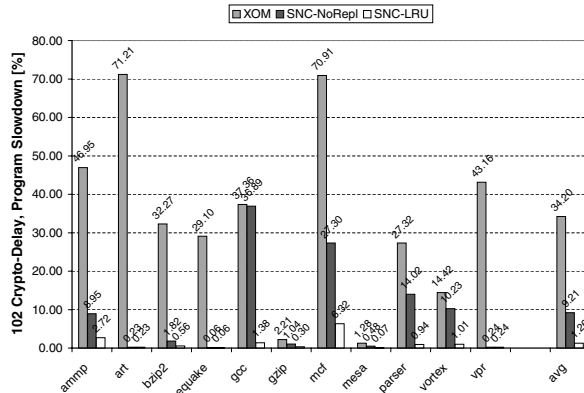

Figure 10. Performance comparison using a longer delay for encryption/decryption unit.

6 Related Work

The research closely related to us is the fast hashing mechanism for memory integrity verification [11]. Defense of the replay attacks for XOM type of architecture is addressed. The solution is to build hash trees and combine them into the on-chip caches to speedup verification of the untrusted memory. Yet dealing with the data privacy and its computation overhead is not considered.

One of the early hardware techniques in protecting software copyright is to use a tamper resistant plug-in model—"dongle". Software is sold together with a dongle. It periodically queries the dongle based on an authorization protocol. If the dongle does not respond the software will halt. However, a skilled programmer can easily analyze the machine code and disable the software protection functions.

Another type of secure processor, bus-encryption microprocessor, has been used for almost a decade in 8-bit microcontrollers such as Dallas Semiconductor DS5000 series. Its application ranges from credit card termination, ATM to pay-TV access control devices and communication encryption modules [15]. In such microprocessors, software is stored in encrypted form outside CPU and decrypted only when it is read into the chip. Both the data and address buses values are encrypted in order to send data to external memory. Bus-encryption microprocessors target for single application environments in which the code size is usually very small.

Fast cryptographic co-processors have been developed to support security applications for Internet communication and E-commerce [3, 24, 21, 23]. Such a co-processor can support multiple ciphers at competitive speed simultaneously. The model we are using in this paper is fundamentally different from those co-processor since ciphers are directly implemented on the main processor and we do not trust any components other than the main CPU.

Many software techniques have been proposed in providing certain level of copyright and intellectual property protection. *Obfuscation* attempts to transform the code into a form that is harder to reverse engineer. *Tamper-proofing* causes a program to malfunction when it detects that it has been modified. *Software watermarking* embeds copyright notice in the software code to allow the owners of the software to assert their intellectual property rights [6]. The software techniques discourage software theft, can trace piracy, prove ownership, but can not *prevent* copying itself.

7 Conclusion

We proposed to use a fast cryptography method—one-time pad cryptography, to speed up the execution of a secure processor. In our design, the cryptography computation is off loaded from the processor's critical path and is carried in parallel with memory access. We developed the new cryptography scheme and its hardware support. Experiments show that our technique reduces the performance overhead from 16.7% for critical path cryptography to 1.28% for one-time pad cryptography.

References

[1] "Advanced Encryption Standard(AES) Development Effort," US Government, *http://csrc.nist.gov/encryption/aes/*.

[2] International Planning and Research Corporation, "Sixth Annual BSA Global Software Piracy Study," *http://www.bsa.org/resources/2001-05-21.55.pdf,* 2001.

[3] J. Burke, J. McDonald, and T. Austin, "Architectural Support for Fast Symmetric-Key Cryptography," *ACM 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX),* November 2000.

[4] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," *Technical Report 1342, Univ. of Wisconsin-Madison, Comp. Sci. Dept.*, 1997.

[5] CACTI3.2, HP-Compaq Western Research Lab, *http://research.compaq.com/wrl/people/jouppi/CACTI.html.*

[6] C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation — Tools for Software Protection," *IEEE Transactions on Software Engineering,* Vol. 28, No. 8, August 2002.

[7] "An Introduction to Cryptography," Network Associates, Inc., *http://www.pgpi.org/doc/pgpintro,* 1999.

[8] D. W. Davies and W. L. Price, "Security for Computer Networks," Wiley, 1989.

[9] "Data Encryption Standard (DES)," *Federal Information Processing Standards Publication 46-2,* December, 1993.

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

[11] B. Gassend, G. E. Suh, D. Clarke, M. v. Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," *The 9th International Symposium on High Performance Computer Architecture (HPCA9),* pages, February 2003.

[12] "Sandia researchers develop world's fastest encryptor," *http://www.sandia.gov/media/NewsRel/NR1999/encrypt.htm.*

[13] T. Gilmont, J.-D. Legat, and J.-J. Quisquater, "Enhancing the Security in the Memory Management Unit," *Proceedings of the 25th EuroMicro Conference,* pages 449–456, September 1999.

[14] *http://www.specbench.org/osg/cpu2000.*

[15] M. Kuhn, "The TrustNo1 Cryptoprocessor Concept," *Technical Report,* Purdue University, April 1997.

[16] K. M. Lepak, G. B. Bell, and M. H. Lipasti, "Silent Stores and Store Value Locality," *IEEE Transactions on Computers,* Vol. 50, No. 11, 2001.

[17] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horwitz, "Specifying and Verifying Hardware for Tamper-Resistant Software," *IEEE Symposium on Security and Privacy*, 2003.

[18] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horwitz, "Architectural Support for Copy and Tamper Resistant Software," *ACM 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX),* pages 168–177, November 2000.

[19] T. Maude and D. Maude, "Hardware Protection Against Software Piracy," *Communication of the ACM,* Volume 27, Number 9, pages 950–959, September 1984.

[20] M. J. B. Robshaw, "Stream Ciphers," *Technical Report TR-701,* version 2.0, RSA Laboratories, 1995.

[21] S. W. Smith, E. R. Palmer, and S. Weingart, "Using a Higher Performance, Programmable Secure Coprocessor," *Financial Cryptography,* pages 73–89, February 1998.

[22] W. Stallings, "Cryptography and Network Security, Principles and Practice," *Prentice Hall, 3rd ed.* 2003.

[23] J. Tygar and B. Yee, "Dyad: A system for Using Physically Secure Coprocessors," *Technical Report CMU-CS-91-140R,* Carnegie Mellon University, May 1991.

[24] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: A Fast Flexible Architecture for Secure communication," *ACM 28th International Symposium on computer Architecture (ISCA01),* June 2001.

[25] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design," *International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 150–159, November 2000.