Improving Memory Encryption Performance in Secure Processors

Jun Yang, *Member*, *IEEE*, Lan Gao, *Student Member*, *IEEE*, and Youtao Zhang, *Member*, *IEEE*

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. In this design, the program and data are stored in an encrypted format outside the CPU boundary. The decryption is carried out 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 using a different encryption scheme, namely, "pseudo-one-time pad" encryption, to produce the instructions and data ciphertext. With some additional on-chip storage, cryptography computations are carried in parallel with memory accesses, minimizing the performance penalty. We performed experiments to study the trade-off between storage size and performance penalty. Our technique reduces the performance overhead from 20.79 percent to 1.28 percent on average for reasonably sized (64KB) on-chip storage.

Index Terms—Memory design, hardware/software protection, security and protection.

1 Introduction

Software copyright protection plays an important role in assuring the software market value and a fair return on 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 [3]. 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.

Several techniques have been proposed to provide hardware support at the microprocessor level against software piracy [22], [21], [20], [18], [16]. 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 coprocessor and the main memory. This is because program confidentiality can be violated by tapping the communication channel such as the system bus. An adversary can easily tamper with the execution of the program once some knowledge of the code is obtained. Moreover, the operating system is also considered nontamper 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 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 a potentially malicious operating system that can access the register values on interrupts, register values also need to be encrypted 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) or manipulated (since it would raise exceptions and then halt) by unauthorized entities [21], [20].

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 [21], performance loss can be as high as 41.81 percent, 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 cannot be used until they are fetched from the memory and decrypted afterward. We propose performing decryption in parallel with a memory access, overlapping crypto-computation time with memory latency.

- *J. Yang and L. Gao are with the Department of Computer Science and Engineering, University of California at Riverside, Riverside, CA 92521. E-mail: {junyang, lgao}@cs.ucr.edu.*
- *Y. Zhang is with the Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083. E-mail: zhangyt@utdallas.edu.*

Manuscript received 9 Feb. 2004; revised 4 Oct. 2004; accepted 17 Dec. 2004; published online 16 Mar. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0044-0204.

Published by the IEEE Computer Society

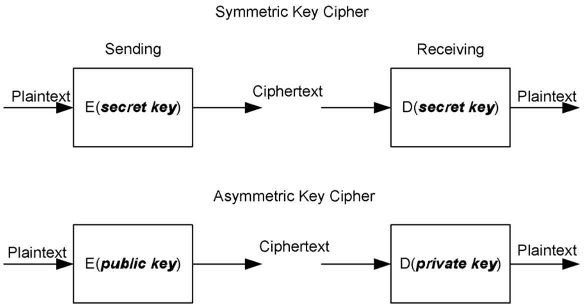


Fig. 1. Illustration of symmetric and asymmetric ciphers.

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 20.8 percent average performance loss of XOM architecture to only 1.3 percent over the insecure baseline processor.

The remainder of the paper is organized as follows: We first briefly describe the XOM architecture in Section 2. Then, we elaborate on the idea of off-loading the cryptography computation from the 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 Fig. 1). In symmetric key cryptography, communicating parties share a common secret key in encryption and decryption. The advantage is that it runs as much as 1,000 times faster than comparable asymmetric key ciphers [8]. The primary obstacle is the distribution of the secret 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 confidentiality of the software algorithm, 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 secret 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 $processor_1$ cannot run on $processor_2$ since they have distinct private keys.

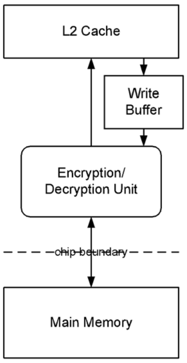


Fig. 2. The lengthened XOM memory path.

2.2 Interacting with External Memory

The XOM architecture adopts a complicated mechanism in protecting the program data confidentiality and providing memory integrity verification. Ensuring confidentiality means keeping 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.[1] 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. [12]. Thus, we do not address the issue of verification and concentrate on speeding up encryption and decryption process in this paper.

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. Fig. 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

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

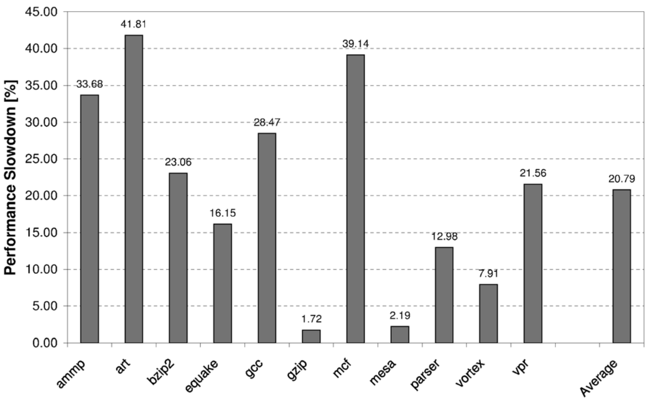


Fig. 3. Optimistic estimation on performance loss due to encryption/decryption.

memory is decrypted before it is stored in the L2 cache and used by the program.

2.3 Internal Protection for Multitasking

A major effort in designing the XOM secure processor goes to protecting interactions among multiple active tasks. Each task is protected within a strict perimeter, termed the "compartment." Each compartment has its own ID and a secret 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 the start/termination of XOM mode, communication between programs, traps and interrupts, and storing and loading cryptographic data to and from memory traps and interrupts. We assume that those features carry over from the original XOM model into this paper.

3 Offloading Crypto-Computation from the 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 Fig. 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 [29], [4], [25], [28]. However, in spite of the effort in crafting the designs, the crypto-hardware here still inserts long latency on memory access due to its computationally intensive nature.

Fig. 3 shows the performance degradation due to the prolonged memory path in XOM architecture. We tested over 11 SPEC2000 [17] 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 [5]. We assumed a typical 100 cycle memory latency and a 50 cycle encryption/decryption delay similar to that in [21]. Such a fast hardware for widely used symmetric ciphers, e.g., DES [10] and AES [1], is possible with ASICS designs [2], [11]. We also modified the SimpleScalar implementation to accommodate an 8-entry write buffer so that real write latency is also taken into consideration.

Across the 11 benchmarks tested, the performance overhead ranges from 1.72 percent for `gzip` to 41.81 percent for `art`. The arithmetic average slowdown of these benchmarks is 20.8 percent. For some benchmarks, e.g., `art` and `mcf`, their slowdowns are greater than some others such as `gzip` and `mesa`. This is because `art` and `mcf` have relatively more memory accesses than `gzip` and `mesa` and, thus, their performances are more sensitive to the memory path latency. We can project that, for memory intensive programs such as database applications, the delay will be more severe.

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 cannot begin. We propose using 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 using an algorithm similar to "one-time pad" encryption [26] 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:

$$c = p \oplus random\ key, \tag{1}$$

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 a modified one-time pad algorithm, the encryption and decryption of a plaintext data value can be expressed as the following:

$$c = p \oplus encrypt_k(seed), \tag{2}$$

$$p = c \oplus encrypt_k(seed), \tag{3}$$

where c is stored in insecure memory, p is the plaintext data value, and k is the secret key shipped with the software. Operationally, when p is sent off chip, (2) is used; when p is read from memory, (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 modified encryption ((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* [23]. The stream cipher is similar to one-time pad. The difference is that it uses a pseudorandom number stream instead of a genuinely random number stream. Many widely used encryption algorithms such as AES [1] and 3DES [9] are believed to do a good job in generating pseudorandom numbers. As a result, $p \oplus encrypt_k(seed)$ is as random as $encrypt_k(p)$. Thus, we call our scheme "pseudo-one-time pad" (POTP) encryption. Next, we discuss how to select the seeds.[2]

3.4 Seed Selection

The purpose of encrypting a seed instead of a value itself is to do it in parallel with memory read operation such that, when data fetched from memory is available, the encrypted seed is also available. For memory writes, we start encryption as soon as the evicted cache line enters write buffer. Therefore, when the time a packet spent in the write buffer is equal to or greater than the encryption latency, the encrypted seed will always be available at the time the cache line is sent to memory. It is also important to differentiate seeds for different encryption units, i.e., cache lines, to decorrelate 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 others.

Advantage. In the XOM model, every data value is encrypted directly and stored in its memory location. This implies that the same data values at different locations have the same ciphertexts. It is known that the memory contains a lot of repeated values [19], [30]. Thus, even with encryption, the repetition pattern still exists, creating potential security holes.

Using the address of a data value as the seed in (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 $encrypt_k(addr), 1 \oplus encrypt_k(addr), 2 \oplus encrypt_k(addr) \ldots,$ which amounts to

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

where C is a constant. This is vulnerable and it may potentially permit an attack that requires a less than brute-force effort. Therefore, the seed used for such a series of writes should not be a constant, i.e., it should vary. This is akin to the *mutating register* used in XOM. A mutating register is associated with an XOM compartment to protect normal register values on OS interrupts. It is included in calculating the hashes for the register values so that an adversary cannot perform a replay attack. Thus, the mutating register is updated each time the XOM process is interrupted. To mutate the seeds in (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 AES 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:

$$(I_1|I_2) \oplus AES_{KEY}(V0), \quad (I_3|I_4) \oplus AES_{KEY}(V0 + 8), \\ (I_5|I_6) \oplus AES_{KEY}(V0 + 16) \ldots,$$

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. Similarly to XOM, encryption and decryption are done on a 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:

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

$$seed_i = virtual\_line\_address(L) + seq\_num_i; \tag{5}$$

$$Ciphertext = Plaintext \oplus AES_{KEY}(seed_i), \tag{6}$$

[2] 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() represents a starting point 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.

where $seq\_num_0 = 0$. On a line L read:

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

$$Plaintext = Ciphertext \oplus AES_{KEY}(seed_i). \qquad (8)$$

From (6), we can see that it is possible for pad $AES_{KEY}(seed_i)$ generated for $Plaintext$ to repeat itself once the $seed_i$ starts to wrap around since it has a limited number of bits. This does not weaken the security level since, in the original XOM design, the $Ciphertext$ does not change as long as the $Plaintext$ stays the same (Section 3.4). Using POTP, this problem is solved by generating different pads each time. However, this additional level of randomness is not perfect since the pad may recycle itself periodically. It is not absolutely necessary to break this cycle as our POTP is no weaker than XOM. On the other hand, one can either prolong the repeating cycles by using a greater number of bits for the $seed$ or change the KEY used for encryption every time the $seed_i$ starts to repeat [27]. In the latter case, the processor needs to be stalled and the entire memory (used by the program) needs to be reencrypted using the new KEY. As we can see, there is a trade-off between the security strength we want to provide and the costs that are encountered.

It should be noted that, when applying (6) and (8) to an L2 cache line, the $Plaintext$ width might be longer than the AES output bit width—128 bits. In that case, the cache line should be divided into segments of 128 bits each. Each segment should use a different pseudopad. This can be achieved by simply using the virtual address of the segment instead of the entire line in (5) and (7). The immediate implication is that, now, for one L2 cache line, multiple pseudopads need to be generated, increasing the AES throughput requirement. In [13], the implementation of a 128-bit AES unit can achieve $30 \sim 70$ Gbit/s with $175 \sim 380$K gates using 0.18μm CMOS technology. Those throughputs translate to the generation of 128 bits in $2 \sim 4$ cycles on a 1GHz machine, which still leaves abundant time before a line is fetched from memory.

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. Fig. 4 illustrates the architecture of the abstract partial XOM model with our SNC.

The SNC should be accessed using the virtual address (VA) of an L2 cache line. This is because the 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 an open problem in XOM, we choose not to perform POTP 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.

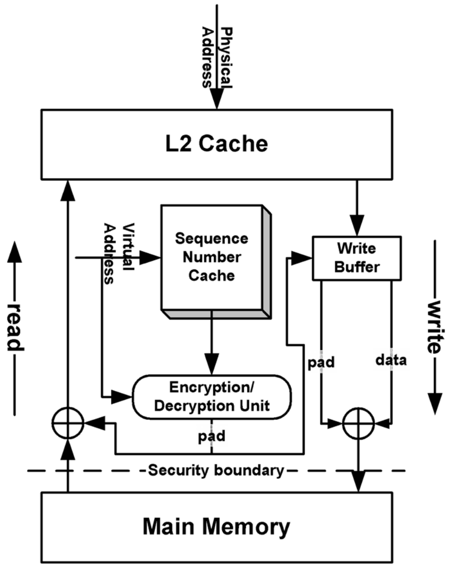


Fig. 4. Design of POTP encryption on data with sequence number cache.

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 then be used to address SNC on a cache write back. The storage incurred due to storing VAs in the 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 percent.

Ideally, the SNC should store all the sequence numbers of memory lines. Taking a 1GB memory and a 128B line size as an example, 8M (1GB/128B) sequence numbers need 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 occupies a larger chip area and takes a 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? A 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 cannot 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. This requires some changes in the memory layout to include the new metadata. Since the number of sequence numbers needed is a fixed value, i.e., $\frac{user\ memory\ size}{L2\ cache\ line\ size}$, we can allocate a fixed sized memory region starting at address A_{SN} for those values. The sequence numbers corresponding to memory blocks in a unit of L2 cache line size can be laid out linearly. Thus, the address of the sequence number of a memory address M is

$$\frac{M - MemAddr0}{L2\ cache\ line\ size} \times sequence\ number\ size + A_{SN},$$

where MemAddr0 is the starting address of user memory.

It is not preferred that the sequence numbers are encrypted using pseudo-one-time pad again since they themselves would need sequence numbers! It is also not necessary to encrypt sequence numbers directly since memory access time will be doubled in this way. In fact, those evicted sequence numbers need only be stored in plaintext. This is because the secret key used to encrypt the memory data is not revealed, even if the seeds can be derived. This way we can manage the replacement of sequence numbers economically.

The advantage of allowing replacement in SNC is to make POTP 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 a critical path, it imposes additional memory traffic and may compete with other memory requests that are critical. Thus, the number of replacements 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 POTP encryption is carried as long as there are vacant slots in the SNC. When the SNC is full, cache lines whose sequence numbers are not stored in the SNC will not be able to perform POTP 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 POTP 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 nonreplacement SNC.

4.2 Algorithm

In this section, we discuss the SNC query (i.e., read) and update (i.e., write) operations in great detail. 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 (7). After the memory access returns, the plaintext value is then obtained by applying (8). Assuming a 100-cycle memory latency and 1-cycle XOR, the value is ready to the CPU at the 101st cycle. An update hit in SNC happens when an 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 (4). The seed is formed and the line is ciphered using (5) and (6), respectively. Note that the evicted lines should first go to the write buffer (Fig. 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 the write operation is not on the critical path and the write buffer greatly reduces the waiting time for the write operations to finish, the impact on overall performance is not a big concern and we will discuss it in the next section.

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 sequence number. For those that cannot fit in the SNC, they are stored in memory. As pointed out earlier, sequence numbers in memories are not encrypted. On an SNC query miss, a memory access is needed to fetch the target sequence number. Thus, each query miss incurs 100 cycles before the seed encryption can start, becoming the most expensive operation. As such, an update miss in SNC also needs to access memory for the sequence number. Since this is carried while the cache line is in the write buffer, the impact is less significant. Algorithm 1 gives the pseudocode for handling the SNC misses.

Algorithm 1. Pseudocode 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.     for each segment $va_{sg}$ in $L$'s virtual address $va$ do
4.         $E_{sg} \leftarrow Encrypt_{KEY}(va_{sg} + sn)$; /* executed in
           fully pipelined engine, in parallel with line 7 */
5.     end for
6.     $Cipher_L \leftarrow$ read $L$ from memory
7.     for each segment $C_{sg}$ in $Cipher_L$ do
```

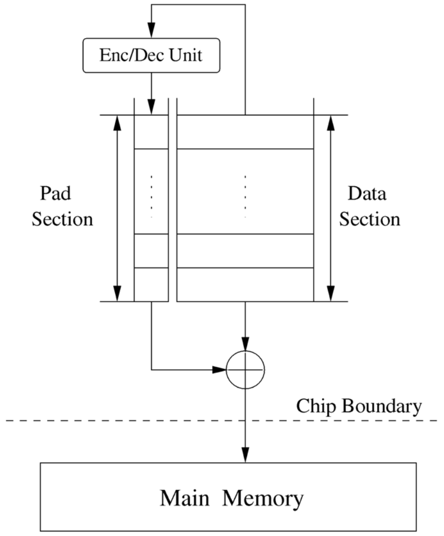


Fig. 5. POTP encryption on data writes.

```
        $P_{sg} \leftarrow E_{sg} \oplus C_{sg}$; /* $P_{sg}$'s form plaintext for $L$ */
    end for
    replace a victim $V_{sn}$ in SNC with $sn$;
    then write $V_{sn}$ into memory if updated;
else if SNC update miss on cache line $L$ then
    $sn \leftarrow$ read memory for $L$'s sequence number
    $sn += system\_timer$;
    for each segment $va_{sg}$ in $L$'s virtual address $va$ do
        $E_{sg} \leftarrow Encrypt_{KEY}(va_{sg} + sn)$; /* executed in
        fully pipelined engine */
    end for
    for each segment $P_{sg}$ in plaintext $L$ do
        $C_{sg} \leftarrow E_{sg} \oplus P_{sg}$ /* $C_{sg}$'s compose ciphertext of $L$
        */
    end for
    write ciphertext of $L$ into memory;
    replace a victim $V_{sn}$ in SNC with $sn$;
    write $V_{sn}$ into memory if updated;
end if
```

4.3 Dealing with Memory Writes

The write buffer plays an important role when dealing with memory writes and most processors are equipped with a write buffer to hide the write latency from the next level memory and reduce write traffic [24]. Therefore, we use a realistic memory access model with a limited write buffer to investigate the impact on performance in our SNC design. Fig. 5 shows the write buffer structure. The cache blocks evicted from L2 are stored in the data section as usual. In addition, we allocate a pad section to store the pads corresponding to each block in the data section. The generation of pads and the operations on pushing the blocks into the memory will be clear shortly. First, let us look at when the write buffer retires an entry.

Retirement policy. As we know, the write buffer holds the evicted L2 cache dirty blocks temporarily and retires them later at an appropriate time. Similarly to the scheme proposed by Skadron and Clark ([24]), entries in the write buffer are retired in a FIFO manner. Also, we adopt the occupancy-based "lazy retirement" policy that has a "high water mark," or threshold, as in [24]. We start to retire entries when the following two conditions are both satisfied: 1) the number of occupied entries exceeds a threshold and 2) the bus is currently free. Since its fill speed may surpass its drain speed, the write buffer may become full, upon which the memory bus must give the highest priority to the write buffer over other components. Instead of flushing the whole write buffer, only the oldest entry is written into memory. This is similar to the Alpha 21064 and 21164 processors, which retire the oldest entry if two or more entries are valid (pointed out in [24]). With this simple retirement policy, we provide more opportunities for L2 cache read misses to hit in the write buffer and the CPU stalls are greatly reduced.

Preparing pads for write buffer data blocks. Since data blocks stay in the write buffer for some time, it is clearly beneficial to prepare their pads during this waiting time. The purpose is to get the pads ready before the block is retired. In our retirement policy, a block is written into the memory only when it reaches the queue head, the buffer occupancy is above threshold, and the bus is free. Thus, there is plenty of time between the time the block is enqueued and the time the block is dequeued. However, excessively long pad generation time, i.e., time to encrypt the block's seed, may impose pressure on the write buffer. In that case, even if the block moves to the queue head and the bus is free, it is still unable to retire since the pad is not ready. Fortunately, the encryption latency is shorter than the memory latency. This means that, as long as a block is not at the head, there is abundant time for pad generation since the earlier blocks will all spend a longer time in writing to the memory. The case where a block is pushed into the empty write buffer and the occupancy quickly grows to exceed the threshold within the encryption latency only happens in the write buffer cold start stage. Thus, there is little overall pressure from the pad generation on the write buffer.

Some modern processors adopt a write combining (WC) technology for enhancing graphics performance. The WC buffer is similar to a write buffer, but differs in that the entire block can retire the moment the last word of the block arrives. With such a retirement policy, our POTP can still gain noticeable performance improvement since the generation of a pad starts immediately after an entry is allocated in the WC buffer. By the time the last word arrives at the WC buffer, the pad may have been produced, whereas, in the original XOM, the encryption for the block only starts at that time. Stalls may happen only when the WC entry becomes ready before the pads are produced. Still, many encryption cycles have been hidden and the performance gain is evident.

Once a pad is generated, it needs to be stored so that it will be available for XOR upon the block retirement. We store all such pads in the pad section, as shown in Fig. 5. An alternative solution is to XOR the data block with the pad and use it to replace the plaintext one in the write buffer. This solution appears to save the space for storing the pads. However, remember that L2 read misses will need to check the write buffer for a potential hit. If the block has been

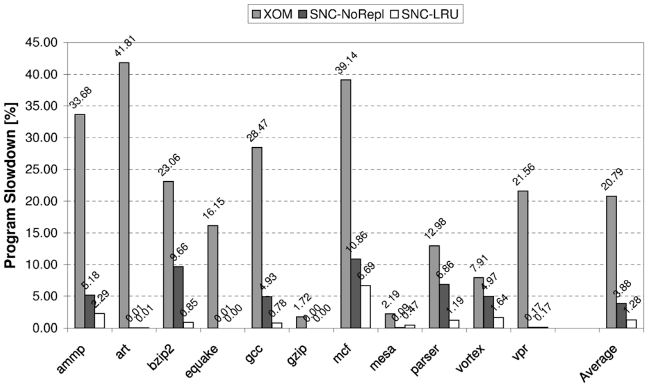


Fig. 6. Performance comparison for XOM, SNC with LRU, and no cache replacement.

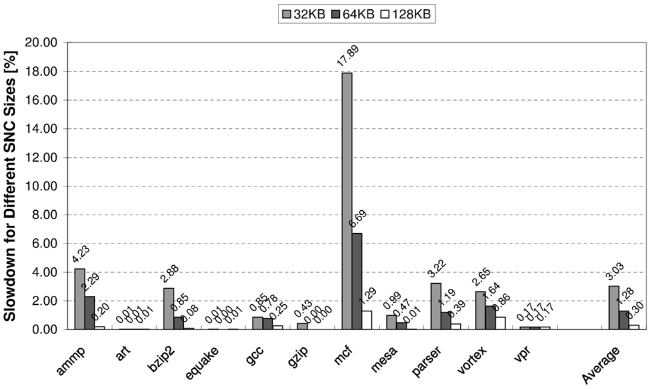


Fig. 7. Performance comparison for different sized SNC. LRU replacement is used.

encrypted using POTP, the L2 read cannot immediately return with the target value in plaintext. Thus, we choose to save the pads along with their data blocks. This will double the size of the write buffer since every pad is of equal length to the line.

4.4 Other Security and Implementation Issues

Context switching. One of the difficulties we realized is to handle a situation in context switching. On context switches, XOM architecture employs expensive operations to avoid leaking 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) tagging 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 multitask 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 POTP encryption scheme with XOM. We used the SimpleScalar Tool Set [5] to run 11 `SPEC2000` [17] 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 continue to execute for another 10 billion instructions so that they finish within a 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. Instead of assuming an unlimited write buffer, as previously presented in [31], we modified the SimpleScalar implementation and added a write buffer with a realistic size and a realistic memory bus model. The write buffer is set to eight entries for all the configurations. 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 POTP 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 Fig. 6. 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—the arithmetic average drops from 20.79 percent to 3.88 percent for no replacement SNC and 1.28 percent for LRU SNC. The highest reduction is from 41.81 percent slowdown to 0.01 percent for program `art`. We can draw two conclusions from these results:

1. Using POTP encryption is an excellent solution to minimize performance degradation of secure processors. The 1.28 percent 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 `bzip2` shows a big difference between the two, sequence numbers filled into the SNC initially are hardly used later.

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. Fig. 7 shows the execution slowdown in percentage of the baseline. We can see that, with smaller SNC, the scheme underperforms 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.

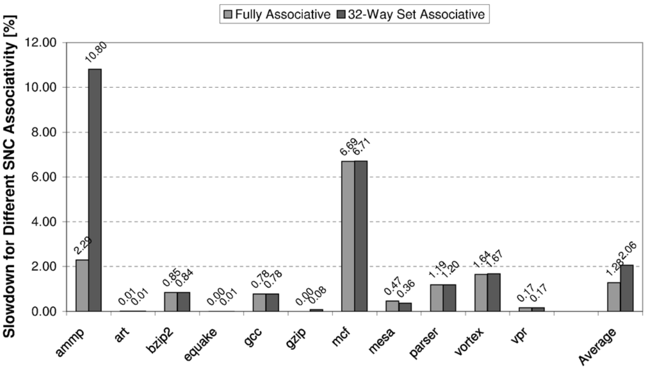


Fig. 8. Performance comparison between fully associative and 32-way set associative SNCs.

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 compared it with the fully associative, 64KB SNC. Fig. 8 plots the results. Apart from one benchmark, ammp (which increases the slowdown from 2.29 percent to 10.8 percent), all the rest of the 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.

5.4 Larger L2 versus L2+SNC

The fourth set of experiments we conducted is to illustrate that the added on-chip SNC storage is indeed very effective. We show this by comparing the execution time for LRU SNC with an 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 [6] 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. Fig. 9 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 POTP encryption scheme still outperforms XOM on average (2 percent versus 12 percent slowdown). Program gcc and vortex show a speedup of 2 percent and 7 percent in execution time compared to the baseline. This is because, with a 50 percent capacity increase in L2, almost everything in the two programs fit into L2 and, thus, the need to go off-chip is greatly reduced. This experiment shows that, in general, having a larger L2 cache cannot mitigate the performance impact of XOM when using the POTP encryption is satisfactory.

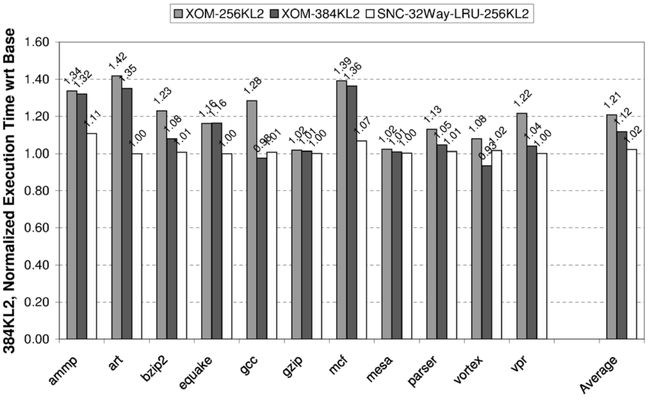


Fig. 9. Impact of a larger L2 cache.

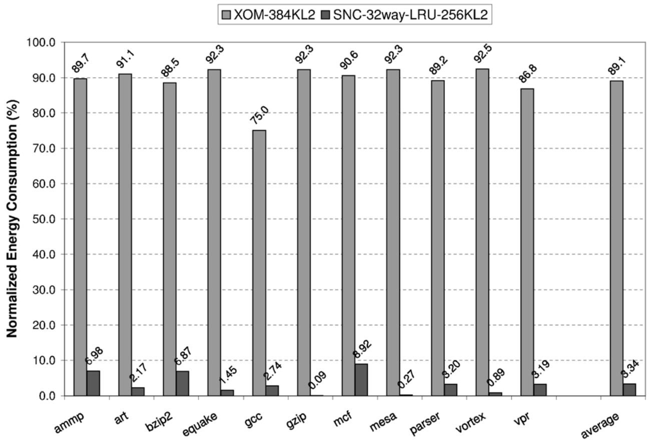


Fig. 10. Energy consumption comparisons.

With the same on-chip cache area budget, using SNC is more energy efficient than enlarging the L2 cache. We measured the total energy consumption for the 384K L2 cache and a 256K L2 with 64K 32-way SNC cache. We found that the latter increases the XOM L2 cache energy by 3.34 percent on average, whereas using the 384K L2 increases the energy by 89.1 percent! We used XCacti [14] to measure the cache energy and, for the SNC, we chose to use a *phased cache* design since it is more suitable for a highly associative cache. The detailed results are presented in Fig. 10. Combining with the results in Fig. 9, we conclude that the SNC design is an effective way of restraining performance loss without much increase in energy expenditure.

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 Fig. 11. We can see that the effect of SNC replacement is negligible in terms of memory traffic increase. For quite a few benchmarks, art, equake, gzip, vpr, the increase is almost zero. On average, there is only 0.2 percent of the L2 memory traffic posed onto the system bus. This also explains why SNC with LRU performs best even though replacements are expensive.

5.6 Sensitivity to Encryption Latency

Our POTP 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 $\mathrm{MAX}(mem\_lat, crypto\_lat) + 1$. Therefore, we

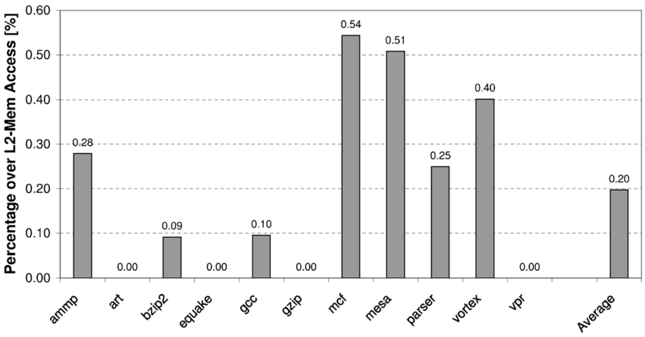


Fig. 11. SNC induced additional memory traffic (64KB SNC).

performed experiments that use a different encryption/decryption latency, 102 cycles [15].

Results are shown in Fig. 12. It is clearly seen that the XOM degrades greatly with the prolonged encryption latency: from 20.79 percent to 42.33 percent slowdown. This is because the 102-cycle roughly doubles the original 50-cycle latency, while, in our design with LRU replacement, the performance is almost unchanged: from 1.28 percent to 1.29 percent. 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 POTP encryption.

6 Related Work

The research closely related to us is the fast hashing mechanism for memory integrity verification [12]. 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 speed up verification of the untrusted memory. In their later improvement [27], they employed a more efficient way to reduce the speed and the space overhead of integrity verification. In handling memory encryption, a buffer-based "one-time pad" encryption was independently developed also. Since we adopted an on-chip cache as the sequence number storage and performed detailed analysis on various cache configurations, our POTP scheme achieves better performance with reasonable hardware cost under realistic machine model.

One of the early hardware techniques in protecting software copyright is to use a tamper-resistant plug-in model—a "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, the bus-encryption microprocessor, has been used for almost a decade in 8-bit microcontrollers such as the Dallas Semiconductor DS5000 series. Its application ranges from credit card termination, ATM to pay-TV access control devices and communication encryption modules [18]. In such microprocessors, software is stored in encrypted form outside the CPU and decrypted only when it is read into the chip. Both the data and address bus values are encrypted in order to send data to external memory. Bus-encryption microprocessors target single application environments in which the code size is usually very small.

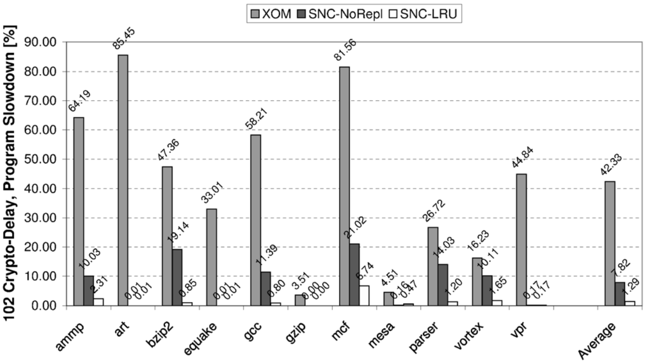


Fig. 12. Performance comparison using a longer delay for encryption/decryption unit.

Fast cryptographic coprocessors have been developed to support security applications for Internet communication and E-commerce [4], [29], [25], [28]. Such a coprocessor can support multiple ciphers at competitive speed simultaneously. The model we are using in this paper is fundamentally different from those coprocessors 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 a 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 the copyright notice in the software code to allow the owners of the software to assert their intellectual property rights [7]. These software techniques discourage software theft, can trace piracy, and prove ownership, but cannot *prevent* copying itself.

7 Conclusion

We proposed using a fast cryptography method—POTP 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. To make our new scheme efficient, we use an on-chip cache to store sequence numbers required in POTP. Our experiments show that allocating a certain chip area to the sequence number cache instead of the regular L2 cache is advantageous in achieving both good performance and energy efficiency. The new cache benefits from the adoption of the LRU replacement policy and the memory traffic increase due to the replacement is within 0.6 percent over the benchmarks we tested. The POTP can reduce the performance overhead from 20.79 percent for critical path cryptography to 1.28 percent on average. The maximum reduction reaches 41.8 percent using a 64KB SNC with LRU replacement policy.

References

[1] "Advanced Encryption Standard (AES) Development Effort," US Government, http://csrc.nist.gov/encryption/aes/, 2001.

[2] http://www.asics.ws/doc/aes_brief.pdf, ASICS.ws technical report, 2003.

[3] Int'l Planning and Research Corp., "Sixth Annual BSA Global Software Piracy Study,"http://www.bsa.org/resources/2001-05-21.55.pdf, 2001.

[4] J. Burke, J. McDonald, and T. Austin, "Architectural Support for Fast Symmetric-Key Cryptography," *Proc. ACM Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX),* Nov. 2000.

[5] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report 1342, Computer Science Dept., Univ. of Wisconsin-Madison, 1997.

[6] CACTI3.2, HP-Compaq Western Research Lab, http://research.compaq.com/wrl/people/jouppi/CACTI.html, 2001.

[7] C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation—Tools for Software Protection," *IEEE Trans. Software Eng.,* vol. 28, no. 8, Aug. 2002.

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

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

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

[11] H. Eberle and C. Thacker, "A 1Gbit/second GaAs DES chip," *Proc. IEEE Custom Integrated Circuits Conf.,* pp. 19.7.1-19.7.4, May 1992.

[12] B. Gassend, G.E. Suh, D. Clarke, M.v. Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," *Proc. Ninth Int'l Symp. High Performance Computer Architecture (HPCA9),* Feb. 2003.

[13] A. Hodjat and I. Verbauwhede, "Minimum Area Cost for a 30 to 70 Gbits/s AES Processor," *Proc. IEEE CS Ann. Symp. VLSI,* pp. 83-88, Feb. 2004.

[14] M. Huang, J. Renau, S.M. Yoo, and J. Torrellas, "L1 Data Cache Decomposition for Energy Efficiency," *Proc. IEEE/ACM Int'l Symp. Low Power Electronics and Design (ISLPED),* pp. 10-15, 2001.

[15] "Sandia Researchers Develop World's Fastest Encryptor," http://www.sandia.gov/media/NewsRel/NR1999/encrypt.htm, 1999.

[16] T. Gilmont, J.-D. Legat, and J.-J. Quisquater, "Enhancing the Security in the Memory Management Unit," *Proc. 25th EuroMicro Conf.,* pp. 449-456, Sept. 1999.

[17] http://www.specbench.org/osg/cpu2000, 2000.

[18] M. Kuhn, "The TrustNo1 Cryptoprocessor Concept," technical report, Purdue Univ., Apr. 1997.

[19] K.M. Lepak, G.B. Bell, and M.H. Lipasti, "Silent Stores and Store Value Locality," *IEEE Trans. Computers,* vol. 50, no. 11, Nov. 2001.

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

[21] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horwitz, "Architectural Support for Copy and Tamper Resistant Software," *Proc. ACM Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX),* pp. 168-177, Nov. 2000.

[22] T. Maude and D. Maude, "Hardware Protection against Software Piracy," *Comm. ACM,* vol. 27, no. 9, pp. 950-959, Sept. 1984.

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

[24] K. Skadron and D.W. Clark, "Design Issues and Tradeoffs for Write Buffers," *Proc. Third Int'l Symp. High-Performance Computer Architecture,* pp. 144-155, Feb. 1997.

[25] S.W. Smith, E.R. Palmer, and S. Weingart, "Using a Higher Performance, Programmable Secure Coprocessor," *Financial Cryptography,* pp. 73-89, Feb. 1998.

[26] W. Stallings, *Cryptography and Network Security, Principles and Practice,* third ed. Prentice Hall, 2003.

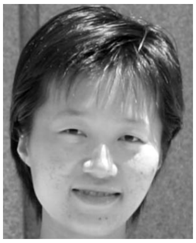
[27] G.E. Suh, D. Clarke, B. Gassend, M. vanDijk, and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," *Proc. 36th Intl Symp. Microarchitecture,* pp. 339-350, Dec. 2003.

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

[29] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: A Fast Flexible Architecture for Secure Communication," *Proc. ACM 28th Int'l Symp. Computer Architecture (ISCA '01),* June 2001.

[30] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 150-159, Nov. 2000.

[31] Y. Zhang, J. Yang, and L. Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," *Proc. 36th Int'l Symp. Microarchitecture,* pp. 351-360, Dec. 2003.



Jun Yang received the BS degree in computer science from Nanjing University, China, in 1995, the MA degree in mathematical sciences from Worcester Polytechnic Institute, Massachusetts, in 1997, and the PhD degree in computer science from the University of Arizona in 2002. She is an assistant professor in computer science and engineering at the University of California at Riverside. Her research interests are in the areas of secure program execution, temperature-aware microarchitecture designs, and network processor designs. She is a member of the ACM and IEEE.

Lan Gao is a PhD candidate in the Computer Science and Engineering Department at the University of California, Riverside.She received the BS and ME degrees from Huazhong University of Science and Technology, China. Her research interests include architectural support for security and trusted computing. She is a student member of the IEEE.

Youtao Zhang received the PhD degree in computer science from the University of Arizona in 2002. He is an assistant professor of computer science at the University of Texas at Dallas. His research interests are in the areas of computer architecture, program profiling, program analysis and optimization, and data compression. He was the recipient of a US National Science Foundation Career Award in 2005, the distinguished paper award of the IEEE/ACM International Conference on Software Engineering (ICSE) conference in 2003, and the most original paper award of the International Conference on Parallel Processing (ICPP) conference in 2003. He is a member of the IEEE and the IEEE Computer Society.