

# I<sup>2</sup>SEMS: Interconnects-Independent Security Enhanced Shared Memory Multiprocessor Systems

Manhee Lee   Minseon Ahn   Eun Jung Kim  
Department of Computer Science  
Texas A&M University  
College Station, TX 77843  
{manhee, msahn, ejkim}@cs.tamu.edu

## Abstract

*Protection and security are becoming essential requirements in commercial servers. In this paper, we present a fast and efficient method for providing secure memory and cache-to-cache communications in shared memory multiprocessor systems that are becoming enormously popular in designing servers for various applications. Since our scheme is independent of underlying interconnects and cache coherence protocols, we refer to it as Interconnects-Independent Security Enhanced Shared Memory Multiprocessor Systems (I<sup>2</sup>SEMS). The main challenge in designing I<sup>2</sup>SEMS is how to precompute keystreams in a timely manner, which is critical to minimize performance overhead. We achieve this goal by adopting a single system-wide Global Counter Controller (GCC) and three additional components for each processor: a keystream queue, a keystream cache, and a keystream pool. The GCC assigns a unique range of counters as a way to help processors precompute the counters' keystreams.*

*We have implemented I<sup>2</sup>SEMS using Simics with Wisconsin multifacet General Execution-driven Multiprocessor Simulator (GEMS). We tested our design with SPLASH-2 benchmarks on up to 16-processor shared memory multiprocessor systems. Simulation results show that the overall performance slowdown is 4% on average and the keystream hit rate is as high as 78%. The stable keystream hit rate shows that I<sup>2</sup>SEMS works well with both memory-read and memory-write dominant applications. Similar to the conventional cache, a large keystream pool size is beneficial to high hit rates.*

## 1 Introduction

Recent malicious attacks to many research/commercial servers have made protection and security essential requirements in all computer systems. Especially, server systems operated by institutes dealing with highly confidential data such as banks, military, or government agents need absolute security because one incident of information leakage could result in very serious problems. Therefore, those institutes should seek effective schemes to prevent/block any possible security

attacks on the systems. Considering that most of the server systems have multiple processors and share single or distributed memory, we believe that shared memory multiprocessor systems need more comprehensive approaches than uniprocessor systems. Moreover, since the system performance should be the last to be compromised in the server systems, any security schemes free of performance overhead are much desirable despite additional hardware costs.

There are two types of possible attacks: software and physical attacks. Software attacks exploit software vulnerabilities of the operating system (OS) and server applications such as web and database services. Buffer overflow attack is a good example [1]. Physical attacks capture or modify data from the system memory or system bus through external devices physically attached to the system. Although high-end servers are often assumed to be secure against such physical attacks, that assumption cannot be sustained any longer in many situations; it is possible that inside personnel turn against their organization to steal data and, moreover, there exist real snooping devices that are attachable to buses [4, 10]. To mitigate the physical attacks, memory encryption and authentication have been widely investigated for uniprocessor systems [5, 6, 12, 24, 26, 27, 31, 32]. However, the uniprocessor security models are not sufficient to design secure multiprocessor systems due to the absence of cache-to-cache communication protection.

When multiple processors are sharing data, cache coherence protocols guarantee data consistency among caches. Without protecting cache-to-cache communication, memory protection would be useless. A few studies raised this question and provided fast encryption and authentication techniques on bus-based shared memory multiprocessors [23, 33]. One common feature in the above studies is that a bus sequence number that counts every message on a shared bus is used as a counter to generate an encrypted counter, called *keystream*<sup>1</sup> [13]. Since all communications on the shared bus are broadcasted to all processors, they can predict the next bus sequence number and generate the same keystream in advance, thus turning complex encryption/decryption operations into a simple XOR operation. However, since the above schemes are dependent on

<sup>1</sup>Also known as one-time-pad (OTP) in previous literature.

the shared bus, they cannot be extended to other general multiprocessor systems having different underlying networks or distributed shared memory [8, 22].

A recent study by Rogers, *et al.* proposed a novel mechanism for protecting cache-to-cache communication in distributed shared memory (DSM) multiprocessors as a remedy for this problem [21]. However, since their design focused on the directory-based cache coherence protocol used for the point-to-point communication, their idea cannot be directly applied to the systems with other cache coherence protocols used for multicasting/broadcasting communication such as the token coherence protocol [2, 16]. For example, even in the directory-based cache coherence protocol, a multicast network is reported to improve system performance by up to 18% because the invalidation time is lessened by sending a multicast message to invalidate multiple cache blocks instead of sending multiple messages sequentially [9, 15, 20].

To provide diverse multiprocessor architectures with more general security models, we propose an interconnect independent and cache coherence protocol independent security model for shared memory multiprocessor systems, referred to as Interconnects-Independent Security Enhanced Shared Memory Multiprocessor System (I<sup>2</sup>SEMS). To our best knowledge, I<sup>2</sup>SEMS is the first attempt to support the protection of shared memory multiprocessor systems without any restrictions on communication types and cache coherence protocols.

Our main idea is that a global counter controller (GCC) assigns counters upon a request from a processor and broadcasts the assignment to the other processors. When receiving the counter assignment, processors generate the counters' keystreams and store them in a *keystream queue* or a *keystream pool* depending on whether the keystream will be used for encryption or decryption. To minimize the use of new counters, a *keystream cache* stores counters and keystreams after the keystream queue uses them to encrypt outgoing data blocks. We focus on providing timely availability of keystreams in the keystream queue to remove encryption delay and yielding a high hit rate in the keystream pool to minimize decryption delay.

We implemented I<sup>2</sup>SEMS using Simics along with Wisconsin multifacet General Execution-driven Multiprocessor Simulator (GEMS) in a hierarchical switched network. GEMS is a set of modules for Virtutech Simics to enable detailed cache, memory, and interconnection simulations [14, 17]. We use SPLASH-2 benchmarks and one SPEC OMP benchmark for workloads on up to 16-processor shared memory multiprocessors [25, 30]. Simulation results show that the overall performance slowdown is 4% on average, and the keystream pool hit rate is as high as 78%, meaning that 78% of incoming messages are decrypted without delay. When we investigate the relationship between cache coherence protocols and keystream pool hit rate, the directory-based cache coherence protocol shows a better keystream hit rate. More specifically, their relatively long latency to a directory controller conversely helps to improve the keystream pool hit rate by allowing a longer time to precompute keystreams.

The remainder of the paper is organized as follows. Sec-

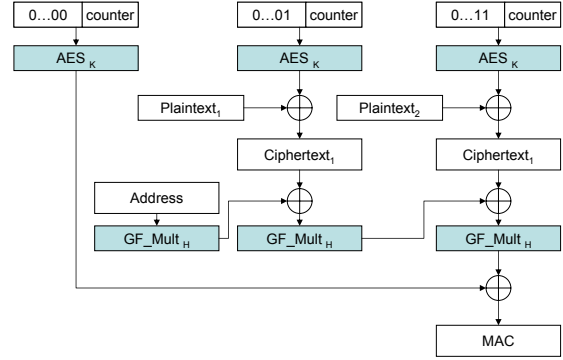


Figure 1. Galois/Counter mode.

tion 2 explains Galois/Counter mode and describes how the previous studies provided secure computing models in uni/multiprocessor systems. In Section 3, we explain our security model and its design in detail. Security analysis and simulation results are presented in Sections 4 and 5, followed by the concluding remarks in Section 6.

## 2 Secure Computing Models

In this section, we first explain Galois/Counter mode used in our study for encryption and authentication, and we describe our threat model. Then we explain how the previous works provided secure computing in uni/multiprocessor systems.

### 2.1 Galois/Counter Mode (GCM)

Galois/Counter Mode (GCM) is a block cipher mode of operation that encrypts and authenticates messages at the same time, and also called authenticated encryption [18]. Figure 1 shows how to encrypt a 32-byte plaintext and generate a message authentication code (MAC) using GCM. Decryption has a similar design.  $GF\_Mult_H$  denotes multiplication in  $GF(2^{128})$  using a hash key,  $H$ , which is the encrypted zero counter,  $AES(0^{128}, K)$ , with a secret key,  $K$ . Since the security strength of GCM is the same as the strength of its block cipher [18] and AES is considered to be secure without any serious weakness until now, the security of memory and cache-to-cache messages will be improved significantly. Assuming that the counter is predicted perfectly, after a plaintext and an address become available, the additional operations are two XORs below  $AES_K$  and three sets of  $GF\_Mult_H$  and XOR. Using a pipelined architecture of [3],  $GF\_Mult_H$  takes one clock cycle. Therefore, since the upper two simple XORs can be parallelized with the first  $GF\_Mult_H$ , the total delay will be six clock cycles.

Note that the counter does not need to be secret because, even though an attacker gets both a ciphertext and its counter, it is practically impossible to recover its original plaintext without the secret key. The biggest benefit of this scheme is that AES latency can be out of the critical path because it is possible to precompute keystreams even when plaintexts or ciphertexts are not available. However, it is imperative to use distinct

counters for different plaintexts to prevent security vulnerabilities.

## 2.2 Threat Model

Before enhancing security of a system, it is necessary to classify secure and insecure components in the system. Processing core, registers, caches, and control and data paths in a processor are considered to be secure. We assume that memory and I/O controllers are secure while other off-chip components including memory, memory bus, and I/O devices are insecure. As shown in Figure 2, we assume that the interconnection network that connects multiple processors and memory banks is insecure.

Through these insecure components the following physical attacks are possible. First, any wiretapping device attached to memory bus and interconnection network can get information from memory transactions and processor-to-processor communications. Since in many cases cables of the interconnection network are exposed to the outside of the system to connect multiple nodes packaged in one or several racks, they are vulnerable to eavesdropping attacks. To make servers resistant to this attack, it is necessary to provide the confidentiality service where all communications from processors are encrypted. Second, assuming that attackers are much more determined and well equipped, they can further inject or modify messages to the systems. This attack includes injecting new data messages, modifying data in messages in transit and replaying old messages. We would like to note that we assume this proactive attack is only related to data messages. Its limitation is explained in Section 4. To prevent this attack, the authentication service needs to make sure that all data messages are genuine, not spoofed, modified, or replayed by illegal devices. Third, in some cases attackers intend to undermine the availability of the systems by keeping injecting garbage messages or replaying old messages. These messages will be finally discarded since authentication information is not correct or cache blocks are not coherent, but they will have negative effects on the system performance because of possible congestion that would occur in cache and memory controllers and interconnection network. This is similar to the denial of service attack in the Internet. However, this attack does not seem much attractive to attackers not only because attackers get little benefit from the attack but also because a sudden performance drop or a system crash would lead to a thorough search for attached devices. Therefore, the availability service is not our focus. From now on, we will give an overview how the previous research provided the confidentiality and authentication services in uniprocessor and multiprocessor systems.

## 2.3 Uniprocessor Secure Model

Several uniprocessor memory authentication schemes were proposed in [5, 12, 27, 31]. XOM (eXecute Only Memory) uses MAC to verify each memory block's integrity [12]. MAC for each block cannot defend against replay attacks where a hacker gets a valid memory block and keeps resending it.

Gassend, *et al.* proposed a hash tree to guarantee the integrity of the whole contents of memory [5]. While it removes the replay attacks, a hash-tree has relatively high run-time overhead because it should check memory integrity for every memory access with a logarithmic number of integrity checks. To overcome this performance overhead, Suh, *et al.* suggested that MAC verifies a series of memory operations using multi-set hashing functions, resulting in only 5% performance slowdown [27]. Moreover, Yan, *et al.*, who first introduced GCM to this research area, minimized authentication overhead by authenticating all necessary levels of a hash tree in parallel [31].

For uniprocessor memory encryption, early studies used ECB mode [12, 26], but later Counter mode was adopted for performance reasons [24, 27, 28, 31, 32]. In the uniprocessor secure computing model, decryption speed is more critical than encryption speed because decryption should be performed quickly so that the processor can use it for execution as quickly as possible. In contrast, the encryption delay of evicted data is not time-critical because the data is first written in the write buffer and later stored back to memory.

To speed up decryption, Yang, *et al.* suggested using an additional cache to store counters [27, 32]. While a processor is waiting for a reply after sending a request to memory, it can precompute a keystream using a stored counter. To alleviate this cache space overhead, a prediction scheme proposed by Shi, *et al.* precomputes incremental counters by utilizing the principle of locality in memory access patterns [24]. With an optimization scheme, they can predict keystreams with up to 99% accuracy. Yan, *et al.* combines encryption and authentication using GCM, resulting in 5% performance overhead [31].

## 2.4 Multiprocessor Secure Model

For multiprocessor shared-memory protection, it is possible to apply uniprocessor security schemes, but cache-to-cache communications need a different protection scheme. Unlike uniprocessor secure computing models, encryption and generation of MAC in multiprocessor systems become time-critical because a receiving processor stalls to wait for a reply. As for authentication of cache-to-cache communications, Shi, *et al.* proposed an *authentication speculation execution* to remove MAC latency from the critical path [23]. In this scheme, while the receiver verifies using MAC, it speculatively continues to execute using un-authenticated data. Those executions are committed only after all operands become authenticated. This scheme reduces performance overhead by overlapping authentication and CPU execution, but each processor needs a complex speculation circuit and this scheme is still vulnerable to replay attacks. Zhang, *et al.* used Cipher Block Chaining (CBC) mode in which the previous MAC is used to make the next MAC, preventing replay attacks [33].

Rogers, *et al.* pointed out the limitation of above schemes on DSM systems and proposed an efficient data protection design [21]. By focusing on point-to-point communications of the directory-based cache coherence protocol, they were able to utilize DSM systems' temporal locality of communications, which means a processor communicates with a relatively small

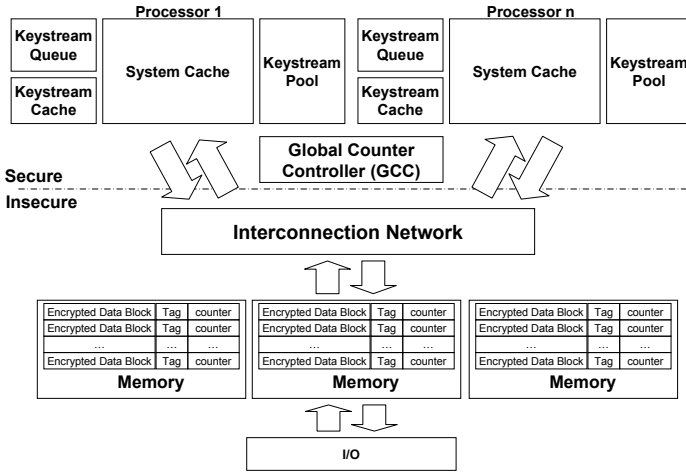


Figure 2. I²SEMS security model.

number of neighboring processors in a short period of time. Such locality makes it possible for each processor to have a small table to hold counters, resulting in good scalability.

Please note that in multiprocessor shared-memory protection, all processors and related components like the memory controller are assumed to share the same secret key. This can be done through the fabrication from factory or runtime distribution as described in [11]. Therefore, even if an ASIC or FPGA is hooked up to the system and pretends to be a peer processor in the multiprocessor systems, it cannot break the privacy and integrity of the system since it is practically impossible for an illegal device to have the same secret key.

### 3 Architectural Design of I²SEMS

In this section, we first discuss design considerations of I²SEMS. Then, we present the design overview of I²SEMS followed by detailed explanation on each component.

#### 3.1 Design Considerations for I²SEMS

In this section, we provide the rationale behind the current I²SEMS design. The first design consideration is to decide whether messages should carry counters. In previous studies on secure bus-based systems [23, 33], cache-to-cache messages do not carry counters because receiving processors can always correctly predict the next counter. However, such strict global synchronization is almost impossible in general shared memory systems. Therefore, although it will incur additional overhead, in our scheme each message carries its own counter. The overhead will be different depending on implementation of interconnection networks. For example, if a wide channel bus is used, an extra channel is necessary. In a network using serial links, the message serialization latency will be increased by the time to serialize the counter.

The second design consideration is to decide whether to maintain multiple local counters or one global counter. In the local counter management, the whole counter range is di-

vided into exclusive subranges and assigned to the processors so that each processor manages its own local counter without worrying about the duplicate use of the same counter. A disadvantage of this management is that adjacent blocks would have discontinuous counters when multiple processors share and update those blocks. This discontinuity will result in slow decryption since predicting the next counter using the current memory block's counter is often incorrect. In the global counter management, in contrast, a central counter controller assigns counters upon a counter request, resulting in relatively contiguous counters. To minimize the counter discontinuity and to improve prediction correctness, the global counter management is desirable and thus we propose GCC. Since a request to the GCC should be made early enough to finish generating new keystreams before their usage, a keystream queue is necessary to hold the precomputed keystreams. A completely different design alternative without using counters is possible by encrypting and authenticating all point-to-point links. However, this approach incurs additional encryption/authentication delay per hop, increasing the total network delay significantly in large networks. Power/energy consumption will also increase linearly depending on the number of hop counts. Therefore, this approach will not work in the multiprocessor environment.

The third design consideration is to decide whether to use counters persistently or compulsorily. In the persistent scheme, each processor always uses new counters to encrypt cache blocks regardless of their cache state. Most of previous studies consume counters in this way. However, we can make use of the fact that it is not necessary to use new counters all the time because a previously used counter can be re-used securely if a cache block has not been changed since the first use of the counter. The benefit of this compulsory scheme is that it will consume less number of counters than the persistent scheme, thus reducing the number of counter request messages. This is why I²SEMS stores used counters and keystreams in a keystream cache.

The last consideration is to decide when and how to start counter prediction; responsive or broadcast counter prediction. In the responsive prediction, a receiving processor starts to predict subsequent counters only after a message arrives. A similar approach was introduced in the uniprocessor prediction scheme [24]. The prediction correctness drops inevitably in data sharing intensive applications because it is hard to predict which counter will be used next. However, since we use the global counter management, a counter request to the GCC hints that the requesting processor is actively consuming new counters and likely to use the newly assigned counters shortly. In the broadcast counter prediction, the GCC broadcasts the global counter to all other processors and the processors precompute keystreams based on the global counter. Although there is no guarantee that messages carrying the new counters will arrive at all processors, if so, the prediction correctness will increase. Since the responsive and broadcast schemes are orthogonal to each other, we choose both of them to increase a prediction hit rate. To store these precomputed keystreams, we use keystream pools.

### 3.2 Design Overview of I<sup>2</sup>SEMS

Figure 2 shows the architecture of I<sup>2</sup>SEMS and its data flow. A system-wide *Global Counter Controller (GCC)* is located in the memory controller<sup>2</sup>. In addition to a unified L2 cache, called *System Cache*, each processor has a *keystream queue*, a *keystream cache*, and a *keystream pool*. The GCC assigns counters upon a request from a processor and broadcasts the assignment to the other processors. Keystreams in the keystream queue and the keystream cache are used to encrypt outgoing messages while those in the keystream pool to decrypt incoming messages.

When a processor transmits a data block to other caches or memory it needs to encrypt the block as shown in the left half of Figure 3. If the cache block is modified or exclusively owned by the processor, a new counter and its keystream are popped from the keystream queue to encrypt the cache block. In many cases, the cache state changes to *Owned* state, which means that this processor is in charge of replying requests that other processors send to get this block's copy. In this state, the previously used counter and keystream can be re-used because the data did not change. For this purpose, after using the new counter and its keystream, we store them in the keystream cache. Therefore, when a cache block to be transmitted is in the *Owned* state, the processor looks up the keystream cache. If not found, a new counter and its keystream need to be popped from the keystream queue.

When a processor receives a data block, it tries to find a keystream in the keystream pool as shown in the right half of Figure 3. Simultaneously it begins to generate  $p$  keystreams using the pipelining ability of AES logic. If found in the keystream pool, referred to as a *keystream hit*, the counter's keystream is used to decrypt the arrived message. If not found, called a *keystream miss*, the keystream will be available in an AES latency because the keystream generation already started regardless of keystream hit or miss. All generated keystreams except the first one are stored in the keystream pool, hoping that subsequent counters may be used in the near future.  $p$  is called *prediction depth* and set to five as default.

When the remaining number of keystreams in the keystream queue becomes less than *counter reserve (CR)*, it sends a request to the GCC to get new counters.  $CR$  is the number of counters that the GCC assigns at a time. More details on  $CR$  are described in Section 3.3. Upon arrival, the GCC assigns new counters and sends a reply to the requesting processor. The GCC also broadcasts the assignment to the other processors for them to precompute the counters' keystreams. When a GCC reply or broadcast arrives at a processor, the processor generates the  $CR$  number of keystreams and store them in the

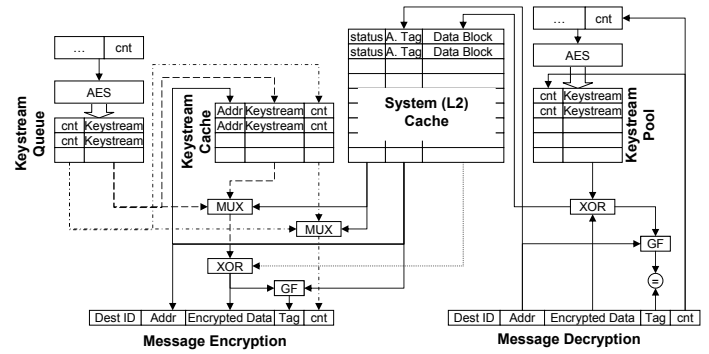


Figure 3. Keystream queue, keystream cache, and keystream pool architecture.

keystream queue or the keystream pool.

### 3.3 GCC and Keystream Queue

Since the GCC and the keystream queue work interactively to enable the efficient counter assignment, we explain two components together. Figure 3 shows the detail architecture of keystream queue, keystream cache, and keystream pool in I<sup>2</sup>SEMS. When a data block is to be transmitted, the cache state bits select one keystream from the keystream queue or the keystream cache. Since the system cache and the keystream cache are accessed in parallel, the access time to the keystream cache is not in the critical path. Note that I<sup>2</sup>SEMS only checks the cache block state, which means I<sup>2</sup>SEMS works independently of any cache coherence protocol as long as the protocol changes cache block state correctly.

To prevent a situation where the keystream queue has no available keystreams, it sends a request to the GCC early enough to guarantee the keystreams' availability all the time. Appropriate counter request timing is closely related to how many messages the processor can send in a round trip time to the GCC. We call this number *Counter Reserve (CR)*. Equation 1 defines  $CR$  formally where  $R$  is the round trip time to the GCC,  $O$  is the keystream generation delay,  $M$  is the cache block size, and  $B$  is the network bandwidth. If the remaining keystreams become less than  $CR$ , the keystream queue sends a request to the GCC. To decrease the number of requests, the GCC assigns a block of counters per request instead of one counter. We suggest that the block's size be also  $CR$  because the keystream queue can send only one request when the number of remaining counters becomes less than  $CR$ . Thus, the size of the keystream queue only needs to be large enough to hold  $2 * CR$  keystreams and counters.

$$\begin{aligned}
 R + O &\leq \frac{CR * M}{B} \\
 CR &\geq \frac{B}{M} * (R + O) \quad (1)
 \end{aligned}$$

Since the whole security mechanism depends on the counters assigned by the GCC, the protection of the GCC and its counter

<sup>2</sup>Depending on the location of the memory controller, the GCC can reside off or on chip. When a processor has an off-chip memory controller like most Intel processors, a multiprocessor system using such processors will have the GCC in its off-chip memory controller usually located in the north bridge (or Memory Controller Hub). However, some recent processors such as IBM's POWER5 and AMD's Athlon 64 and Opteron processors have the on-chip memory controller for faster memory access. In this case, by enabling one processor's GCC, a single GCC will be located in an integrated on-chip memory controller.

distribution is critical. Therefore, the GCC itself needs to be tamper-resistant so that its operation and secret keys will be protected. To remove any possibility of modifying or forging counters, we need to provide authentication on the counter distribution. To adopt GCM, the GCC and all keystream pools share an initial counter and the GCC and each keystream queue share a one-to-one counter in the system initialization stage.

In addition to security problem of the GCC, its scalability could be a problem since I<sup>2</sup>SEMS has only one GCC. There are three possible problems; long latency between processors and the GCC, bottleneck of the GCC, and high traffic overhead of broadcasting of counters. The long latency problem can be solved by adjusting  $CR$ . In Equation 1, the increased  $R$  will change  $CR$  so that it will guarantee the availability of keystreams in the keystream queue. Since the GCC performs simple operations such as managing a counter and replying to processors, the GCC will not be a bottleneck in communications. If the underlying network does not support broadcasting or multicasting, the GCC should generate and send all messages one by one, possibly causing some delays in broadcasting. However, this delay will not be much critical to performance because the newly assigned counters are not needed until the requesting processor uses up all  $(CR-1)$  keystreams. Moreover, as we will show in Section 5.5, the total number of requests to the GCC and its broadcasting messages is very small compared to the total number of transmitted messages by processors.

Without loss of generality, we assume the underlying general interconnect allows out-of-order delivery but guarantees that there is no packet loss because many interconnection networks adopt per-hop or end-to-end checksum and message re-transmissions. This means that transmitted messages will not be lost in the networks regardless of whether they are modified or not. Since the keystream queue does not send additional requests to the GCC while it is waiting for a reply, there is no out-of-order delivery of counter requests and replies. This removes the possibility of replay attacks on counter assignments. Only broadcasting messages may be out-of-order. In this case, an older counter than the current maximum counter is discarded to prevent replay attacks. Therefore, our security scheme is independent of underlying interconnects allowing out-of-order delivery.

### 3.4 Keystream Pool and Keystream Cache

Both the keystream pool and the keystream cache have the cache architecture. The keystream pool size will affect the system performance. If its size is too small, the keystream pool hit rate will decrease due to capacity misses, but if the size is too big, the long keystream pool access time increases the total decryption time. To minimize performance slowdown and to get a high hit rate at the same time, we will use the largest possible cache architecture as long as its access time is hidden from the system cache access. According to CACTI model [29], one cache access consists of several circuit level delays. The following equation shows that a cache data access time can be divided into address decoding, wordline, bitline, and sense

amplifier time, which occur sequentially.

$$T_{data} = T_{decode} + T_{wline} + T_{bline} + T_{sense} \quad (2)$$

$T_{decode}$  is the address decoding time to access a specific cache set in the cache. This stage requires the address only, so we can utilize  $T_{decode}$  to parallelize accesses to the system cache and the keystream pool. In other words, if the keystream pool access time,  $T_{access(KeystreamPool)}$ , is equal to or smaller than the system cache's address decoding time,  $T_{decode(SystemCache)}$  as in Equation 3, we can hide the keystream pool access time. In our experiment, we used a 1M bytes 4-way system cache with 6ns access time. Since its  $T_{decode}$  is 3ns, we use a 512K bytes 4-way keystream pool with 3ns access time for the keystream pool, which meets the requirement of Equation 3. The effect of other associativity on the system performance is analyzed in Section 5. Note that the six clock cycle authentication delay is not considered here. That is because the authentication is not in the critical path for program execution. Therefore, I<sup>2</sup>SEMS issues a lazy authentication fail alert.

$$T_{access(KeystreamPool)} \leq T_{decode(SystemCache)} \quad (3)$$

The high hit rate of the keystream cache effectively reduces the counter usage. Thus, the design of the keystream cache is tightly related to the scalability of I<sup>2</sup>SEMS. Different from the keystream pool, the keystream cache can utilize the temporal locality of memory accesses. When a cache block is being shared by multiple processors actively, they are more likely to access the block within a short time than later. In our simulation, we observed that as the size of cache increases, the keystream cache hit rate also increases but the effect of larger caches is gradually diminishing. Therefore, we believe in most cases even a small keystream cache can yield a relatively high hit rate.

## 4 Security Analysis

### • Counter Wrap Around

If the counter wraps around, two different data blocks will use the same keystream. We estimate the expected wrap-around time by considering the maximum counter usage rate. Let's suppose a system has  $2^6$  processors using a 3.2G bytes/sec network and a 32 byte long message and counter is 64 bits long. When one half of the processors keep sending modified cache blocks at its full speed and the other half are receiving them, this system will consume  $2^{32}$  keystreams per second, thus taking about  $2^7$  years to wrap around the counter in the worst case. Considering that I<sup>2</sup>SEMS assigns new counters only when necessary, we expect the counter wrap around time will be long enough to support system lifetime.

### • Replay Attacks

As long as the keystream queue gets a set of counters from the GCC in advance and the keystream cache allows the same counter to be reused, it is hard to apply a simple preventive restriction such as the monotonic increment of counters. However, we found that all replay attacks on data messages will re-

**Table 1. Processor model parameters**

Parameters	Values
CPU	1 GHz
L1 I-Cache	128K bytes, 4-way, 2ns latency
L1 D-Cache	128K bytes, 4-way, 2ns latency
L2 Cache	1M bytes, 4-way, 6ns latency
Cache Block Size	32 bytes
Keystream Pool	512K bytes, 4-way, 3ns latency
Keystream Cache	32-entry fully associative, 2ns latency
Memory	2G bytes, 80ns
network link bandwidth	3.2G bytes/sec
AES latency	80ns
AES throughput	3.2G bytes/sec

sult in abnormal states that cache coherence protocols can detect. Therefore, replay attacks can be resolved without adopting a whole new scheme as described in the followings.

A replay attack can capture one of two messages sent to a processor or to the memory. Let us begin with to-processor communications. Assume that an owner processor of a cache block sends its authoritative copy to a requester that wants to share the cache block and an attacker captures the message. Later, if another processor modifies the cache block, the requester’s cache block will be invalidated. When the requester wants to access the cache block, it will find it invalidated. Then it sends a request message to get a new authoritative copy from a real owner. If the attacker knows this event anyhow and replays the captured message to the requester, the message will be successfully authenticated by the requester because the message carries an old but legitimate MAC. However, since we assume that any message will not be lost, the true owner of the cache block will finally receive the request message and send an authentic message to the requester. Once receiving this message, the requester will consider this abnormal since it should not receive another authoritative data block. Even when the genuine data block arrives first and the replayed data block does later, this will be detected, too. A replay attack on an invalidated block will be also detected since it is abnormal to receive a data block without requesting a block.

Similarly, a replay attack on memory store messages can be taken care of by cache coherence protocols since the replay attack will finally result in an inconsistency. For example, let us assume an attacker captures a write-back message and tries to replay the message. The target memory block will be one of two states: owned state or not. When a replayed write-back message is delivered to an owned state block, the replay attack will be instantly detected because no processor other than the memory owns the data block. On the other hand, when a memory block is invalid and an attacker replays a previous write-back message to the memory, it will be authenticated and stored in the memory. However, later the true owner processor will write back the block because the cache block should be evicted in the end. When the memory receives the write-back message, it will raise a replay attack alert.

Note that the above method relying on the original ability of cache coherence protocols has limitations as long as control messages are not authenticated. For example, if the at-

tacker deliberately creates an invalidation message to the current owner processor as well as sends a replayed message to the memory, it will satisfy the consistency of cache coherence protocol, raising no replay alert. This limitation comes from the fact that I<sup>2</sup>SEMS encrypts and authenticates only data messages. We are currently investigating to solve this problem.

## 5 Performance Analysis

### 5.1 Simulation Framework

**Simulator:** We evaluate the performance of I<sup>2</sup>SEMS using the Simics full-system multiprocessor simulator developed by Virtutech AB [14]. To simulate general shared memory systems and various cache coherence protocols, we use General Execution-driven Multiprocessor Simulator (GEMS) as an extension of Simics [17]. We developed I<sup>2</sup>SEMS in four cache coherence protocols: broadcasting, directory, hammer, and token. Broadcasting and directory coherence protocols are traditional protocols used in current shared memory multiprocessor systems. Hammer cache coherence protocol is an approximation of AMD’s Hammer protocol used in AMD multiprocessor systems [19]. Token coherence protocol recently proposed by Martin, *et al.* is a low latency cache coherence protocol enabled by decoupling performance and correctness [16].

**Configuration:** We configured Simics as a Sun Fire server with UltraSPARCIII+ processors and Solaris 9 OS using the parameters shown in Table 1. Since the decoding time of the unified L2 cache is approximately 3ns, we use 512K bytes keystream pools with 3ns access time to parallelize a keystream pool access with an L2 cache access. The keystream cache is 32-entry fully associative cache with 2ns hit latency. Due to the recent development of AES implementation, a 128 bit AES unit can produce 30~70 Gbps using 0.18μm CMOS technology. In our experiment, default AES latency is 80ns and throughput is 3.2G bytes/sec [7, 24, 33]. CR is calculated to be 32. We configured the network using hierarchical switches with fanout degree of 4, which is default in GEMS [17]. Since in our simulation the network latency to transmit a 32 bytes cache block is 10ns, the six clock cycle delay for GF multiplications discussed in Section 2.1 can be overlapped without causing an additional delay.

**Benchmarks:** We used four SPLASH-2 benchmarks; FFT, LU, RADIX, and OCEAN with their typical settings as described in [30] and one SpecOMP2001 benchmark, APPLU.

### 5.2 Overall Performance Slowdown

We ran five benchmark programs using two secure configurations and one non-secure configuration; one secure configuration is I<sup>2</sup>SEMS and the other is *prediction only* scheme to compare with I<sup>2</sup>SEMS. Similar to Counter mode proposed by Shi, *et al.* [24], the prediction only scheme has a keystream generator and a small number of keystream buffers. Regardless of a keystream pool hit or miss, the keystream generator keeps generating  $p$  keystreams using the next counters. We as-

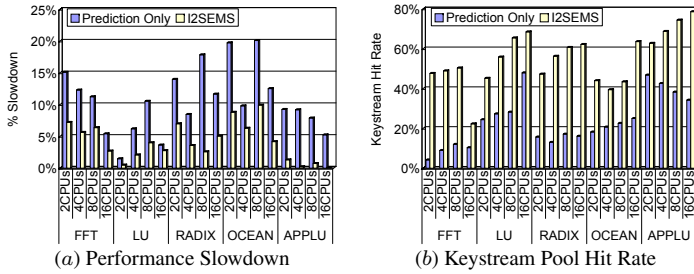


Figure 4. Overall performance.

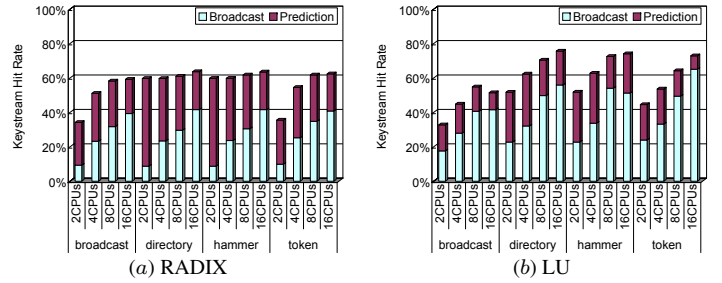


Figure 6. Keystream origination.

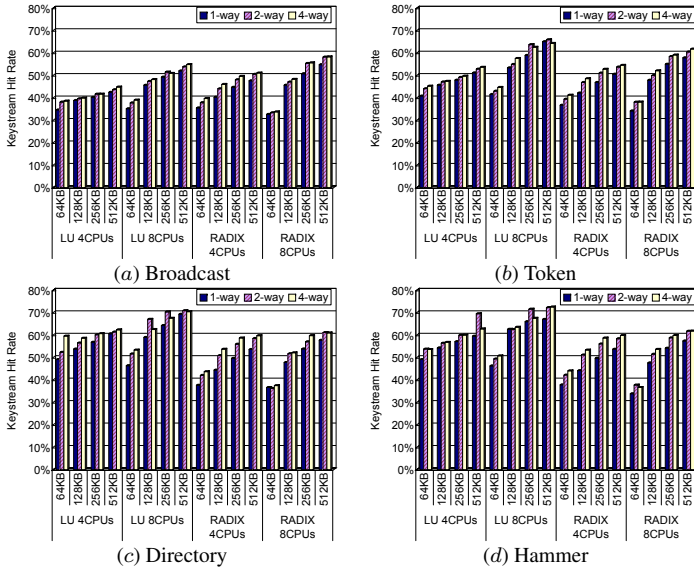


Figure 5. Hit rate vs. keystream pool size and associativity.

sume that in the prediction only scheme each cache controller knows its own exclusive range of counters.

Figure 4 (a) shows the overall performance slowdown normalized to the execution time of the non-secure configuration. Compared to the baseline configuration with no security measures, I<sup>2</sup>SEMS incurs around 4% performance slowdown on average. It is clear that I<sup>2</sup>SEMS causes less performance slowdown than the prediction only scheme. However, it is hard to find a common trend in performance slowdown among varying number of processors. That is possibly because the execution time can be easily affected by other factors. Since we use the non-deterministic full-system architectural simulator and multithread workloads, the execution path of each benchmark program is not always the same. Thus, a small difference in the selection of OS scheduling may cause large variability in the execution time. The similar effect was discussed by Zhang, *et al.* [33]. Instead, we use the keystream pool hit rate for performance comparison not only because it is less variable than execution time but also because the high hit rate is positively correlated with performance.

Figure 4 (b) shows keystream pool hit rates of five benchmark programs. The hit rates vary from 22% to 78%, but note

that in all benchmarks I<sup>2</sup>SEMS has higher hit rates than the prediction only scheme. This is because the prediction and broadcast schemes adaptively yield a high keystream pool hit rate in both memory-read and memory-write dominant applications.

We would like to emphasize that the counter prediction of a uniprocessor system is completely different from that of multiprocessor systems. According to [24], uniprocessor’s prediction scheme showed 82% keystream hit rate and optimization techniques can increase the rate even up to 99%, but the prediction only scheme shows a lower keystream pool hit rate in our simulation than in a uniprocessor system. That is because the counter in the uniprocessor system will be highly contiguous, so the sequential prediction is very likely to be correct. However, in the prediction only scheme the exclusive counters would result in counters’ discontinuity in adjacent cache blocks when those blocks are modified by multiple processors having different counter ranges. Therefore, it is unavoidable to have a high miss rate, especially in data sharing intensive applications. In contrast, our scheme will show a high prediction hit rate in both high and low data sharing situations because the high hit rate of I<sup>2</sup>SEMS comes from not only the counter prediction but also the counter broadcast.

### 5.3 Keystream Pool Size

To investigate the effect of the keystream pool size and associativity on the keystream pool hit rate, we varied the keystream pool size from 64K bytes to 512K bytes with direct-mapped, 2-way, and 4-way set associativities. In Figure 5, as the size increases, keystream pool hit rates also increase. It is intuitive that a large cache can hold more keystreams and consequently yield a higher keystream pool hit rate. Note that even after the keystream pool size is increased exponentially, the keystream pool hit rate does not go up dramatically, but only around by 5%~20%. The reason is that the recently assigned or predicted keystreams are more likely to be hit. Cache set associativity does not have a substantial effect on the keystream pool hit rate. A high set associative cache is useful only when many blocks are mapped to the same set. However, counters are monotonically increasing, so cache contentions do not occur often. Therefore, we conclude that in the presence of area and power constraints, the direct-mapped keystream pool is more desirable.

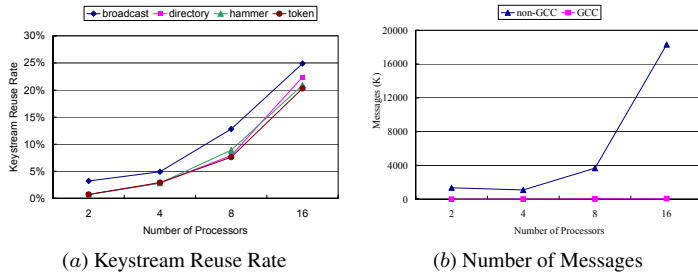


Figure 7. Scalability of GCC.

## 5.4 Cache Coherence Protocol

We investigate the relationship between the keystream pool hit rate and cache coherence protocols. In Figure 6, to illustrate individual contributions, we show the breakdown of keystream hit rate according to their origination: Broadcast and Prediction. We observe that, in both graphs, the proportion of hits on broadcasted counters is increasing as the number of processors is increasing. Therefore, we can conclude that a memory-writing dominant application will have a high keystream pool hit rate. However, even in a memory-reading dominant application, its keystream pool hit rate will not drop sharply because read-only memory accesses usually show high locality of memory accesses, and consequently the prediction scheme will contribute to a high keystream pool hit rate more than the broadcast scheme. Therefore, I<sup>2</sup>SEMS will show good performance irrespective of memory access patterns in large multiprocessor systems.

## 5.5 Scalability of Global Counter Controller

Figure 7 shows the simulation results of RADIX benchmark up to 16 processors. Figure 7 (a) illustrates how much a 32-entry keystream cache reduces the counter usage. When the number of processors is two, less than 3% of outgoing data blocks reuse keystreams, but in the 16 processors the rate reaches around 25%. Considering the rate is going up rapidly as the number of processors increases, it is expected that a significant portion of outgoing messages will reuse keystreams. In Figure 7 (b), while the number of normal messages increases dramatically at 16 processors, the number of the GCC messages does not change proportionally but even appears static. This is because our scheme successfully reduces the number of new counters through the keystream reuse using the keystream cache and the block assignment of new counters by the GCC. Although simulation results are only available up to 16 processors, the trend hints that the GCC will be scalable to larger shared memory multiprocessor systems.

## 6 Conclusions

In this paper, we proposed an Interconnects-Independent Security Enhanced Shared Memory Multiprocessor Systems (I<sup>2</sup>SEMS) guaranteeing confidentiality and integrity of shared

memory and cache-to-cache communication in multiprocessor systems by incorporating a small amount of additional hardware components: *Global Counter Controller (GCC)*, *keystream queue*, *keystream cache*, and *keystream pool*. The GCC assigns globally unique counters for memory/cache-to-cache communication security. Keystream queues and keystream caches minimize encryption delay while reducing the counter usage rate. For fast decryption, keystream pools precompute/store keystreams at a counter broadcast by the GCC and at a message arrival. In addition, by separating security implementation and cache coherence verification, I<sup>2</sup>SEMS can work on diverse cache coherence protocols.

The important conclusions of this work are the following: First, we provided confidentiality and integrity of shared memory and cache-to-cache communication in multiprocessor systems with low performance overhead. We used Galois/Counter mode with AES for better security and performance. The performance overhead of I<sup>2</sup>SEMS was 4% on average although execution time was too variable to find a common trend. The keystream pool hit rate was as high as 78%, meaning 78% of incoming messages were instantly decrypted and authenticated upon arrival. Second, simulation results showed that I<sup>2</sup>SEMS will have good performance in large scale shared memory multiprocessor systems. Even though we tested up to 16-processors due to the current status of our simulator, its trend looks obvious. In addition, I<sup>2</sup>SEMS works well with any applications. If an application is memory-read dominant, the prediction scheme where subsequent counters are predicted will contribute significantly to a high keystream pool hit rate because of the high locality of memory access. In memory-write dominant applications, the broadcast scheme where newly assigned counters are broadcasted will increase the keystream pool hit rate. Therefore, we conclude that I<sup>2</sup>SEMS can support large scale shared memory multiprocessors with diverse memory access patterns. Note that due to the limitation of the simulator we could not simulate distributed shared memory systems. Nevertheless, we believe general trends will be similar because the number of GCC-related messages will be still very small and because both the prediction scheme and counter broadcast scheme will result in high keystream pool hit rates. Third, we found that relatively small keystream pools can support a large system although the larger keystream pools are beneficial to the high keystream pool hit rate. Set associativity does not have much impact on the keystream pool hit rate. Therefore, a simple but moderate sized keystream pool is desirable in I<sup>2</sup>SEMS.

We are currently examining a number of possible expansions to this work. First, we were unable to analyze the realistic web-based or database servers. In-depth experiments with those server applications should fortify the I<sup>2</sup>SEMS design. Next, we would like to expand our research to much larger multiprocessor systems with distributed shared memory (DSM) and to new multiprocessor architectures such as Chip Multiprocessor (CMP) systems.

## References

- [1] "Aleph One". Smashing The Stack For Fun And Profit. *Phrack*, 7(49), 1996.
- [2] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast snooping: A new coherence method using a multicast address network. In *26th Annual International Symposium on Computer Architecture (ISCA'99)*, pages 294–304, 1999.
- [3] R. K. Bo Yang, Sambit Mishra. A high speed architecture for galois/counter mode of operation (gcm). Cryptology ePrint Archive, Report 2005/146, 2005.
- [4] A. "bunnie" Huang. The trusted pc: Skin-deep security. *Computer*, 35(10):103–105, 2002.
- [5] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.
- [6] T. Gilmont, J. Legat, and J. Quisquater. Enhancing the Security in the Memory Management Unit. In *The 25th EuroMicro Conference*, volume 1, pages 449–456, 1999.
- [7] A. Hodjat and I. Verbauwhede. Minimum Area Cost for a 30 to 70 Gbits/s AES Processor. In *IEEE Computer Society Annual Symposium on VLSI*, pages 83–88, 2004.
- [8] HP Superdome. <http://www.hp.com/products1/servers/scalableservers/superdome/index.html>.
- [9] H.-C. Hsiao and C.-T. King. An application-driven study of multicast communication for write invalidation. *The Journal of Supercomputing*, 18(3):279–304, 2001.
- [10] A. B. Huang. *Hacking the Xbox: An Introduction to Reverse Engineering*. No Starch Press, San Francisco, CA, USA, 2003.
- [11] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *32nd Annual International Symposium on Computer Architecture (ISCA'05)*, pages 2–13, 2005.
- [12] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, 2000.
- [13] H. Lipmaa, P. Rogaway, and D. Wagner. CTR-Mode Encryption. In *NIST Workshop Symmetric Key Block Cipher Modes of Operation*, 2000.
- [14] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.
- [15] M. P. Malumbres, J. Duato, and J. Torrellas. An efficient implementation of tree-based multicast routing for distributed shared-memory multiprocessors. In *SPDP '96: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, page 186, Washington, DC, USA, 1996. IEEE Computer Society.
- [16] M. M. Martin, M. D. Hill, and D. A. Wood. Token Coherence: a new framework for shared-memory multiprocessors. *IEEE Micro*, 23(6):108–116, 2003.
- [17] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News (CAN)*, 2005.
- [18] D. McGrew and J. Viega. The Galois/Counter Mode of Operation (GCM). Submission to NIST Modes of Operation Process, 2004.
- [19] Milo M.K. Martin and et. al. Protocol specifications and tables for four comparable MOESI coherence protocols: Token coherence, directory, snooping, and hammer. [http://www.cs.wisc.edu/multifacet/theses/milo\\_martin\\_phd/](http://www.cs.wisc.edu/multifacet/theses/milo_martin_phd/). 2003.
- [20] D. K. Panda, S. Singal, and P. Prabhakaran. Multidestination message passing mechanism conforming to base wormhole routing scheme. In *PCRCW '94: Proceedings of the First International Workshop on Parallel Computer Routing and Communication*, pages 131–145, London, UK, 1994. Springer-Verlag.
- [21] B. Rogers, M. Prvulovic, and Y. Solihin. Efficient data protection for distributed shared memory multiprocessors. In *15th International Conference on Parallel Architecture and Compilation Techniques (PACT 2006)*. ACM, 2006.
- [22] SGI Origin 3000. <http://www.sgi.com/products/servers/origin/3000/>.
- [23] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu. Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, 2004.
- [24] W. Shi, H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2005.
- [25] SpecOMP2001 Benchmark Suite. <http://www.spec.org/omp/>.
- [26] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, 2003.
- [27] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, 2003.
- [28] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2005.
- [29] S. Wilton and N. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.
- [30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.
- [31] C. Yan, B. Rogers, D. Englander, Y. Solihin, and M. Prvulovic. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *33rd Annual International Symposium on Computer Architecture (ISCA'06)*, 2006.
- [32] J. Yang, Y. Zhang, and L. Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th International Symposium on Microarchitecture*, 2003.
- [33] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta. SENSS: Security enhancement to symmetric shared memory multiprocessors. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, 2005.