SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors

Youtao Zhang[†] Lan Gao[‡] Jun Yang[‡] Xiangyu Zhang[*] Rajiv Gupta[*]

[†]Computer Science Department
University of Texas at Dallas
Richardson, TX 75083

[‡]Computer Science and Engineering Department
University of California at Riverside
Riverside, CA 92521

[*]Computer Science Department
University of Arizona
Tucson, AZ 85721

ABSTRACT

With the increasing concern of the security on high performance multiprocessor enterprise servers, more and more effort is being invested into defending against various kinds of attacks. This paper proposes a security enhancement model called SENSS, that allows programs to run securely on a symmetric shared memory multiprocessor (SMP) environment. In SENSS, a program, including both code and data, is stored in the shared memory in encrypted form but is decrypted once it is fetched into any of the processors. In contrast to the traditional uniprocessor XOM model [10], the main challenge in developing SENSS lies in the necessity for guarding the clear text communication between processors in a multiprocessor environment.

In this paper we propose an inexpensive solution that can effectively protect the shared bus communication. The proposed schemes include both encryption and authentication for bus transactions. We develop a scheme that utilizes the Cipher Block Chaining mode of the advanced encryption standard (CBC-AES) to achieve ultra low latency for the shared bus encryption and decryption. In addition, CBC-AES can generate integrity checking code for the bus communication over time, achieving bus authentication. Further, we develop techniques to ensure the cryptographic computation throughput meets the high bandwidth of gigabyte buses. We performed full system simulation using Simics to measure the overhead of the security features on a SMP system with a snooping write invalidate cache coherence protocol. Overall, only a slight performance degradation of 2.03% on average was observed when the security is provided at the highest level.

1. Introduction

Today, computer systems built as symmetric shared memory multiprocessors (SMPs) are widely used as servers which run most popular but also vulnerable operating systems such as Windows, Linux, and Solaris. SMP servers provide superior performance for critical commercial and scientific applications (e.g., banking, airline ticketing, and web services). Unfortunately, many such servers are constantly the targets of both software and hardware security attacks. Often the OS is tampered with to enable meddling with the normal execution of a user program. Sensitive information can be stolen that may bring significant financial loss. Consequently, it is important to design a tamper-resistant environment for software that is executed on an SMP machine. Tamper-resistance of software encompasses two aspects: *confidentiality* which refers to maintaining the secrecy of the code and the data during execution, and *integrity* which refers to neither the code nor the data being changed illegitimately during execution.

Maintaining the confidentiality and the integrity of a program has been studied on uniprocessors intensively [7, 10, 24, 25, 29, 30]. The essential idea is to treat the processor as the only trusted entity but not other components such as the memory modules, buses, and the co-processors. The reason is that values stored in memory, or the transactions between the processor and the surrounding components could be monitored or tampered with at a modest cost by an adversary [9, 14]. The OS is also viewed as a potential adversary as it could be broken into and used as a tool to tamper with any user program execution.

It is natural to adopt the similar security model for SMPs since the off-chip components are also subject to similar attacks as in uniprocessors. In other words, we assume that the only trusted entities in an SMP are the processor nodes, and all other components are vulnerable especially the main memory and the buses. To preserve the program confidentiality, the program itself as well as its dynamic data are encrypted whenever they are sent off-chip, and decrypted whenever they are fetched on-chip as in [10, 24]. To preserve the program integrity, a one-way cryptographic hash value is computed to reflect every valid update to the memory. If the memory is corrupted, the new hash value from the memory will not match with the corresponding value stored in the processor, and thus any physical and software tampering that can alter the behavior of the program execution is detected [7, 24, 25].

However, directly applying memory encryption and integrity checking as in uniprocessors is not sufficient in SMPs. Besides the communication between processor caches and the memory, there is another dimension of communication — the cache-to-cache data exchange for maintaining the cache coherence — that is unique to SMPs. While the cache-to-memory traffic can be encrypted as before, the cache-to-cache traffic is still in clear text and cannot be encrypted in the same way (we will show how it could be easily broken). Not protecting the cache-to-cache traffic implies exposing data in memory indirectly, nullifying the memory encryption.

[*]Authors are supported by NSF grants CCF-0429986[†], CCF-0430021[‡] CCR-0324969[*], CCR-0220334[*], and CCR-0208756[*].

Therefore, the confidentiality and the integrity of the bus data transfers should be maintained just as the data and code stored in the memory.

In this paper, we propose a fast and inexpensive Security ENhancement to SMP Systems, or SENSS, for ensuring both bus *confidentiality* and *integrity*. We introduce a bus encryption scheme for securing cache-to-cache (bus) data transfers. This scheme inherits the merits of both one-time-pad (OTP) and cipher block chaining encryption of a block cipher such as AES (CBC-AES), which ensures low overhead of encryption latency for every bus transaction. In addition, the message authentication code (MAC[1]) of a sequence of bus transfers is also generated using the same algorithm with a different initial vector. The bus authentication is enforced periodically or on per-transaction basis by checking the consistency of MACs on all communication participating processors. Our authentication scheme has the advantage that under heavy bus load where the system cannot afford to authenticate on per-transaction basis, it is still guaranteed that *all* bus transfers are authenticated surpassing a scheme where randomly selected transactions are authenticated. We further develop a technique to match the AES unit throughput with the bus bandwidth at its peak transfer rate. Necessary architectural augmentations of the SENSS and its interactions to the cache-to-memory protection are also discussed.

We have implemented the proposed designs using a full system SMP simulator - Simics [11]. We measured the impact of having SENSS on the overall system performance as well as the additional bus traffic increase on the system bus. On average, the performance slowdown is less than 2.03% and the bus traffic increase is less than 34% when the security is provided at the highest level. For careful evaluation, we also integrated recently proposed techniques for uniprocessors [7, 10, 25, 29]. When integrated with `CHash` [7] and fast memory traffic encryption [25, 29], both slowdown and bus traffic only have small increases.

The rest of the paper is organized as follows. Section 2 briefly introduces the security model of uniprocessor systems. Section 3 motivates our work with real world examples of attacks. Section 4 elaborates the design of the SENSS system: encryption/authentication algorithms and how to match encryption with high bus bandwidth. Section 5 describes the architecture design in support of SENSS. Section 6 discusses how SENSS encompasses the cache-to-memory protection including memory encryption and integrity checking. Section 7 presents our experimental results. Section 8 discusses the related work and the last section concludes this paper.

2. Secure Uniprocessor Model

In this section, we discuss the uniprocessor security models in [10] and [24]. Since those models are quite comprehensive in nature, we will address only those issues that are relevant to the design of SENSS.

2.1. Encryption model

In a secure uniprocessor, the hardware requirement for security purpose is mainly a public-private key pair (K_u, K_p), where K_u is public and K_p is private, and a fast on-chip crypto engine that performs encryption and decryption whenever necessary. K_p is sealed inside the chip and cannot be seen by any outside parties such as the OS while K_u is known to public. All the information flowing between the memory and the processor is encrypted and stays encrypted in memory. The information includes the program itself and the data that is brought in and sent out of the chip during execution. The encryption is performed using a symmetric cipher with key K_s. This key is chosen by the program distributor to generate the encrypted program.

[1] A MAC function is a one-way keyed encryption function that can be easily produced but is difficult to reverse.

To communicate K_s to the processor, the distributor encrypts it using K_u and ships the ciphered key along with the program. Upon receiving and executing the program, the processor uses K_p to decrypt K_s and stores it in an on-chip private register. K_s is later on used to encrypt and decrypt program data transferred across chip boundary. This means that the instruction and the data in the internal caches are all plaintext. The data is only encrypted when it is written off-chip to the memory.

As one can see, such an encryption model imposes significant performance overhead. Every memory write must be preceded by encryption and every memory read is followed by decryption before the data can be used. To reduce the impact of long latency crypto operations, a fast encryption algorithms were developed by Suh *et al.*[25] and Yang *et al.*[29]. The idea is to overlap encryption with memory accesses so that the ciphertext is not data dependent on the plain text or vice versa. In such a scheme, encryption is achieved by XORing the data value with a *pad* which is a cryptographic randomization of the address of the data. Therefore, the pads can be generated in parallel with the memory reads. Note that the pads for the data in the same address should be different every time the data is written into the memory. Otherwise the ciphertext would appear regular if the data is changed regularly over time. Therefore, an on-chip pad cache or buffer was used to remember the latest pad for a memory block. This improvement can reduce the performance slowdown from 17% to 1.3% with a 64KB on-chip pad cache [29]. Due to such performance advantage, we will use this encryption technique, referred to as "fast memory encryption" in later sections, in our SENSS for *cache-to-memory* transfers.

2.2. Integrity checking

The traditional method of memory integrity checking is to create and store MACs for memory blocks. However, such a method cannot defend the *replay* attack in which the adversary replaces a new memory block with an old one and its valid MAC, as pointed out by Gassend *et al.* [7]. To solve this problem, a tree of cryptographic hash values is created for the memory in which the leaves are the memory data, the internal nodes are hashes of their children and the root is the unique signature of the entire memory. The root is stored internally and can be updated only by the processor to be safe. The rest of the tree can reside partially in L2 cache depending on its available capacity. The part stored in the memory needs to be validated if it is acquired by the processor, similarly to that of any data. Once a node resides in L2, it is considered to be secure.

To check the integrity of a memory block (either leaf or internal node), its hash should be computed and compared against the one stored in its parent node. A mismatch indicates an integrity violation. If the parent is not in L2 cache, it should be fetched and checked for integrity. This procedure repeats until an internal node is found in L2.

Even though caching the hash tree is advantageous to the program performance, it is still reported to result in a 25% slowdown on average [7]. A lazy verification approach is later introduced by Suh *et al.* [25] to further reduce the overhead of integrity checking. Instead of checking each memory access strictly, they propose to cluster a sequence of memory accesses and then check them together at a later time. A multiset hashing function is used to support this checking in a small amount of trusted on-chip storage. The performance overhead is significantly reduced to 5% compared to 25% slowdown in [7]. In addition, the memory requirements for integrity checking are also reduced.

3. Vulnerabilities in SMP and Potential Attacks

Although the security models for uniprocessors have been extensively developed, they are not sufficient for SMPs. In this section, we will explain where the models would break in both confidentiality and integrity of a program executing on an SMP.

3.1. Uniprocessor encryption method will not work

One might ask: since the traffic between the processors and the memory is already encrypted as in the uniprocessor model, we could just use the same ciphertext for cache-to-cache traffic. In fact, such a simplified scheme is not as secure as before. To see this, let us analyze the situation where the fast memory encryption is used. This scheme is superior to direct encryption from the performance perspective as discussed in Section 2.1.

Assume that now the cache-to-cache traffic is encrypted using the same pad P as the cache-to-memory traffic for the same data D. Suppose that D is encrypted as $P \oplus D$ in the memory, and D is currently exclusively owned by a processor A. Thus, A can keep updating D in its local cache without the necessity of changing P. Now, if another processor B wants a most recent read copy of D, D would be sent as $P \oplus D$. Later on A modifies D to D' and B requests again, D' would be sent as $P \oplus D'$. An observer can simply XOR the two ciphertext to get $D \oplus D'$ which is obviously not good. Therefore, different encryption method should be used for bus (cache-to-cache) transactions.

3.2. Potential attacks on the shared bus

In addition to encrypting the bus transactions, we also need to develop schemes to authenticate them since they could be tampered with just as memory contents. There have been a number of techniques and devices targeting at tapping or tampering a bus to change program behavior [9, 14]. For example, Sony Playstation and Microsoft Xbox can be hacked by "modchips" which can be soldered to the buses on the motherboard and allow the console to play pirated games. A clean machine always checks the digital signature and media flags from a registered CD by executing a small code in BIOS. A modchip reads, measures and times the busses to block the code from the BIOS and injects spoofing code at the right time. The hacked machine now executes the new code that skips authorization checks. Such a chip is only around $60 and very easy to install [14].

Although attacking an SMP system is more involved than a game machine, it is reasonable to foresee that such tampering of a system is possible. We should thus develop authentication techniques to defend against as many imaginable attacks on the bus as possible to maintain the integrity of cache-to-cache transfers. The different types of attacks include:

- (Type 1) message dropping which means a message destined to a processor is blocked illegally.
- (Type 2) message reordering which means the messages are mis-ordered, e.g., swapped, on the bus.
- (Type 3) message spoofing which means a fake message is generated to fool some processors to receive them as legal messages. Examples are message insertion or replaying.

A normal authentication algorithm will not be able to catch all of the above attacks and thus needs to be augmented as we will explain later.

4. The Design of SENSS

In SMP systems, each processor contains its own local caches (typically L1 and L2) and all processors share a centralized main memory that is accessed through an external bus[2]. Both cache-to-cache and cache-to-memory transfers use this external bus, and the former occurs in executing certain cache coherence protocols. We will first give an overview of the SENSS model and then elaborate the security algorithms.

4.1. SENSS overview

Program dispatching. The procedure of preparing a program on an SMP is similar to that on uniprocessors, except that there are multiple processors now. Each processor is assigned a public secret key pair (s_i, t_i), $i = 0, 1, \cdots, n$. These key pairs should be distinct across all the processors to prevent cascading breakdowns if one processor's private key is compromised. The programs that are dispatched to such an SMP are encrypted by the distributor using a symmetric key cipher with key k. k is then encrypted using all processors public keys (s_i's) and bundled with the encrypted program and dispatched to the machine. This is illustrated in the "Program package" drawing in Figure 1.

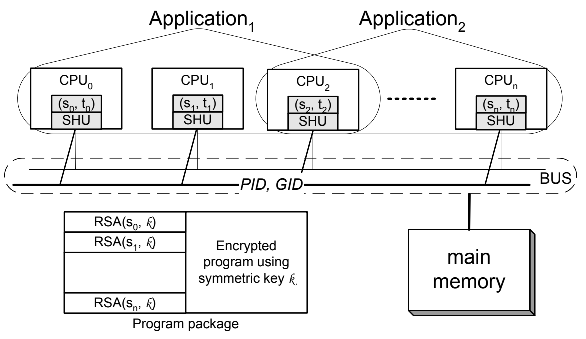


Figure 1: SENSS overview.

Processor grouping. The program distributor has the freedom to specify a subset of trusted processors on the target machine since it may have a good reason to suspect certain processors (e.g. those that are dedicated to processing network protocol stack). We term this subset a *group*. Thus, k can only be decrypted on group member processors, not on others. In Figure 1, we show examples of two applications 1 and 2 which use processors (0, 1, 2) and (2, 3, $\cdots$, n), where n is the maximum processor id (PID), as their group members respectively. Each group is assigned a unique group id (GID) which will be used in secure communication.

Message tagging. To protect communication of a group from being tampered with by another group we must meet the following requirements. The first requirement is that a message belonging to a group should not be read by another group. Thus, the GID should

[2] Distributed shared memory multiprocessor systems where directory based cache coherence protocols are used are not considered in this paper.

be used to tag each message so that each processor only picks up its own group messages. Second requirement is that it should not be possible for other messages to tamper with group messages. We design an encryption and authentication algorithm that incorporates the PID into each message. Hence, the PIDs need to be sent on the bus also. These activities are ensured by a security hardware unit (SHU) (Figure 1) which is solely controlled by hardware and cannot be accessed even by the OS. When a message is put on the bus, SHU automatically tags it with its GID and PID. When a message appears on the bus, the SHU only picks up the message for the GID that it maintains. With the message tagging, the original bus needs to be augmented with an additional bus that transfers PID and GID. We will describe additional hardware maintained by the SHU in Section 5.

The SHU also contains the en/decryption engine, the public-secret key pair, and a private memory similar to that in XOM [10]. In addition, it contains more hardware to maintain the group information in order to protect the group's communication on the bus.

4.2. Encryption scheme

Our goal is to design an encryption scheme that is both secure and fast. The proposed algorithm has the flavor of both one-time-pad (OTP) [22] encryption and Cipher Block Chaining mode of a symmetric block cipher such as AES (CBC-AES). We utilize OTP for its fast encryption speed, and CBC-AES [17] for its high security level and its capability in message authentication, especially the chaining feature that distinguishes it from a non-chained authentication algorithm. We will show later that using the CBC-AES MAC in SMP can defend attacks that would be transparent to a pure hash based authentication scheme. Another advantage is that for the same data transferred on the bus at different time, CBC-AES will generate different ciphertext strengthening the protection of data. In authentic OTP encryption, the ciphertext is the XOR of the data and a random key (termed pad). To decrypt a ciphertext, the same pad is used to XOR with the ciphertext and retrieve the plaintext. OTP based encryption can be done rapidly as the generation of the pads can be done off the critical path. Encryption and decryption involves only one XOR operation which is usually only one processor cycle. We use this property in conjunction with the CBC mode of AES.

		CBC-AES	**Bus encryption**
Encryption	1st	$c = Data \oplus m_{last}$	$c = Data \oplus m_{last}$
	2nd	$m = AES(k, c)$	send c
	3rd	send m	$m = AES(k, c)$
Decryption	1st	receive m	receive c
	2nd	$c = AES^{-1}(k, m)$	$Data = c \oplus m_{last}$
	3rd	$Data = c \oplus m_{last}$	$m = AES(k, c)$

Table 1: Adapting CBC-AES to the shared bus.

The algorithm. In the CBC-AES, the input to the AES at time t is the XOR of the data at time t and the cipher at time $t-1$, m_{last} (see the "CBC-AES" column in Table 1). The output of the AES then updates m_{last} and is sent out as the cipher of $Data$. On the decryption side, m is first decrypted into c, and c is XOR'ed with m_{last} to get $Data$. In SENSS, sending m onto the bus means that it cannot be produced until hundreds of cycles (due to AES delay) after $Data$ is ready. Similar reasoning applies to the receiving end for computing $Data$. Thus, we send c onto the bus since it can be computed just one cycle after $Data$ is ready (see the "Bus encryption" column). We then update m in the background to get ready for the next round of transfer. On receiving c, a processor simply XOR's it with the m_{last} to compute $Data$, and then updates m for the next transfer.

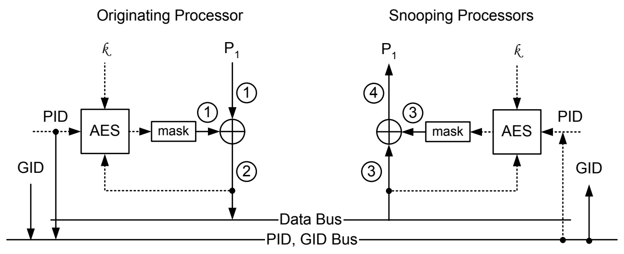


Figure 2: Bus encryption scheme.

This procedure is illustrated in Figure 2 where we use "mask" to represent the m's. The message originating from a sender processor is encrypted in step 1 and 2, and decrypted in step 3 and 4 on the receiver side. On both sides, the mask should be available from the last bus transfer. After that, the mask is updated from AES unit which takes inputs PID and c (the necessity of PID will be explained later). The update paths are shown by dotted lines in the figure.

Initialization. In the above bus encryption scheme, it is very important that all group members start with the same initial vector m_0 so that all processors are correctly synchronized. This can be carried before the application starts execution. A designated processor, e.g. the one with the smallest PID, is responsible for broadcasting a randomly generated initial vector to all group members (this action is performed in the SHU of the processor). The random vector can be obtained from the AES unit with an arbitrary input. The broadcasting process should be encrypted and authenticated using the secret key of the program. Since this is only required at the beginning of execution, spending some time in initialization is acceptable. Also, in every invocation of the same program, it is necessary to generate *different* m_0 so that each run produces different mask traces and it is more difficult for any opponent to analyze the program.

Maintaining the mask. There are multiple groups running in the SENSS and each group maintains its own mask across all group members. The masks are updated during the lifetime that the group is active even if no active thread from that group is on the processor. In this way, when a thread from an active group is scheduled to execute, the mask is ready for use. When an existing group is swapped out, all processes on all processors are stopped and the contexts are encrypted before being written out to the memory.

4.3. Authentication

The algorithm. Using the CBC mode of a block cipher in SENSS has an additional advantage: it can generate MAC to authenticate the bus transfers. This algorithm has been pervasively used in international and U.S. standards [6]. The principle is that if a message has t blocks, which is divided according to the underlying block cipher input width, then its MAC is the s-bit prefix ($16 < s < n$) of mac_t where

$$mac_t = AES_k(\cdots AES_k(block_1) \oplus block_2)\cdots \oplus block_t) \quad (1)$$

In SENSS, if each bus transfer is a block and we treat a fixed number of bus transfers as a big message, then the m_t can authenticate the whole sequence of transfers in one shot. In other words, m_t

reflects the entire history of messages up to time t. When using this scheme for every processor, the message history seen by every processor should be exactly the same. This implies that it can authenticate the broadcasting behavior of the bus while a non-chained authentication scheme may not be able to even though each individual message is authenticated. We will show an example of this argument.

There are some other benefits in doing so: 1) If the applications on an SMP generate heavy bus traffic, the system may not be able to afford authentication on every bus transfers. Checking on a sequence of transfers can greatly reduce the performance overhead yet not losing a single transfer that should be authenticated. 2) A typical crypto hash function can compute the hash only after the entire message is available, whereas CBC can compute the MAC *block by block as they are generated*. This attribute has promoted its usage even in real-time applications [16]. 3) The sequence length can be adjusted by the system. When it is set to one, the authentication is carried on every bus transfer providing maximum integrity protection. This feature allows the system to choose different authentication levels according to the varying security requirements and the application characteristics without changing the algorithm. The authentication procedure is the following.

All processors use the data block and its originating PID as inputs to the algorithm, i.e., the sender assembles the block and its own PID as the input, and the receiver takes them from the decryption path. Each processor sets a counter in the SHU as the authentication sequence length. This counter counts the number of bus transfers since the last authentication. All the group members have exactly the same view of its counter at any time. Once the counter saturates, an initiating processor starts an authentication transaction and sends the MAC onto the bus. Since the counter is updated synchronously, all member processors are expecting at this time the MAC on the bus. They compare the received MAC with their own copies. If a mismatch happens, a global alarm is raised indicating an authentication failure and the program is halted. Note that any tampering of masks during authentication will also result in failure since a mismatch would occur. The initiating processor should be decided in a round-robin fashion to avoid a single member failure in a group.

Defending Type 1 attacks. Type 1 attacks refer to data block droppings. In such a scenario, the sender sends out a block but the receiver did not get it. What's unique in SMP is that once one processor sends out a data block, *all* group members must receive it. Otherwise the bus transfer's integrity is violated. In a normal bus authentication scheme, a hash value is sent along with the block so that the receiver can verify the integrity of the received block [20]. This would be correct if the communication is point-to-point, but not in case of broadcasting in which all members should maintain consistent MACs at all times. Using a bus sequence number, hoping that the processors being blocked would have stale sequence numbers since the block did not reach them and so the sequence numbers did not increment, cannot solve this problem unfortunately.

Imagine processor A intends to send data D_{AB} to B in the i^{th} bus transaction and C intends to send data D_{CD} to D in the $(i+1)^{th}$. Both transactions should be seen by all processors, so the final sequence numbers on them should be $i+2$. However, if the i^{th} data block was dropped from C and D and the $(i+1)^{th}$ was dropped from A and B. All sequence numbers are incremented to $i+1$, and most of all none of the processors can detect this attack since each data block they receive is a valid one. In our scheme we can detect such a split within group members since the MACs take the data block and its originator as the inputs. Thus, A and B would update their MACs using D_{AB} and PID_A, and C and D would take D_{CD} and PID_C. Even though the following bus transactions are not messed up, the i^{th} transaction has been different between $\{A, B\}$, and $\{C, D\}$. And this inconsistency will propagate until the next authentication since the MACs are all chained up from the very beginning.

Defending Type 2 attacks. Equation (1) for authentication must take a different initial vector (IV) than the one used in encryption so that the mac_i's are different from the masks. In the original algorithm (1), a zero IV is used to eliminate the need of IV initialization on every message. We will use a different IV every time the program is invoked, and the initialization can be carried with the encryption's initial mask m_0. Using different IV's for encryption and authentication is necessary since using the masks directly cannot defend the Type 2 attacks. To see this, let us assume a swapping attack in which the adversary swaps the first and second bus transfers. Defending other misordering attacks can be derived in a similar manner.

Assume that the sender processor sends out $m_1 = m_0 \oplus block_1$, $m_2 = AES_k(m_1) \oplus block_2$, and $m_3 = AES_k(m_2) \oplus block_3$ onto the bus, but the receiver received m_2, m_1, and m_3, out of the original order. Using the masks shown in Figure 2 will result in a mask sequence of $AES_k(m_1)$, $AES_k(m_2)$, and $AES_k(m_3)$ on the sending processor, but $AES_k(m_2)$, $AES_k(m_1)$, and $AES_k(m_3)$ on the receiving processors. And we can see that starting from m_3 the masks will be consistent again, i.e., the algorithm has recovered from an attack which is not desired. By using the algorithm in equation (1), the sequences of the MACs are:

$$\begin{aligned}
sender: MAC_1 &= AES_k(block_1),\\
MAC_2 &= AES_k(MAC_1 \oplus block_2),\\
MAC_3 &= AES_k(MAC_2 \oplus block_3),\\
receiver: MAC'_1 &= AES_k(block'_1) = AES_k(m_0 \oplus m_2),\\
MAC'_2 &= AES_k(MAC'_1 \oplus block'_2)\\
&= AES_k(MAC'_1 \oplus m_1 \oplus AES_k(m_2)),\\
MAC'_3 &= AES_k(MAC'_2 \oplus block'_3)\\
&= AES_k(MAC'_2 \oplus m_3 \oplus AES_k(m_1))
\end{aligned}$$

It's easy to see that the MACs at two ends are different, and the future MACs at the receivers will continue to differ from the senders'.

Implications. Using a different IV for authentication implies that the AES needs to be invoked twice for every data block. For a highly pipelined (i.e. high throughput) AES unit nowadays, the authentication can follow the encryption in the immediate next AES pipeline stage. Therefore, the latency due to authentication can be mostly overlapped with the encryption. There are also newly developed algorithms that can provide encryption and fast MACs calculation involving only one invoking of AES such as the GCM [13] algorithm. In that case, the MACs are calculated using Galois Field $GF(2^{128})$ multiplication that takes the outputs of the counter mode of AES as inputs.

Defending Type 3 attacks. The use of the PID in the encryption, as shown in Figure 2, is to let the encryption include the PID and pass it on to the authentication path. This is to detect the Type 3 attack (message spoofing) as listed in Section 3.2.

Assume that the adversary is so powerful that a spoofed message with valid GID g and PID p can be injected onto the bus. Then processor p will identify this spoofed message immediately and raise an alarm since it tries to snoop every message that belongs to g but not with PID p (p should not receive its own message from the bus). Let us further assume that the adversary is intelligent enough to single out p with a message tagged with g and p' where p' is also a valid member of g. At this time, no member in g can recognize this spoofed message since everybody gets valid GID and PID. However, the MAC updated by p is different from the rest since it uses PID p' instead of p in AES! And this inconsistency will propagate till the next bus authentication, detecting the spoofing attack.

4.4. Rapid bus encryption

One mask for unidirectional traffic. Although the lengthy crypto computation is moved off from the critical path in our encryption scheme, the mask generation can still be a bottleneck under high volume of bus traffic. To see this, let us assume the bus is transferring data at its peak rate. If the flow of data is unidirectional (i.e. receiving or sending, but not interleaved), the AES unit has to generate new masks fast enough so that the incoming data from the bus is absorbed in time or the outgoing data is sent out timely without delay. This would require that the throughput of AES be the same as the bandwidth of the bus. Fortunately, modern AES's throughput can be designed to match with the gigabyte buses [8, 26].

Two masks for bidirectional traffic. Now consider a bidirectional flow of data. Suppose a processor snoops a message at time t, and decrypts it at time $t + 1$. At this time it wants to send an outgoing message using the updated mask. However, the AES starts to generate new mask at time t but will not complete until $t + c$ where c is the encryption latency (∼80 CPU cycles). Thus, the outgoing message is blocked waiting for the mask calculation. This situation can happen quite often as the incoming and outgoing data are usually interleaved on an SMP.

To solve this problem, we use a pair of masks to decouple the mask dependency between back-to-back messages of opposite directions. One mask is used solely for the odd numbered messages (m_o) and the other for even numbered messages (m_e). Here we utilize the fact that there is a total order of the entire set of messages generated from within a single group during its lifetime. Since every message is snooped by every member, the total order is explicitly known to every member. Hence, every group member can keep such a tuple (m_o, m_e). In this way, messages in a row will be encrypted using m_o and m_e alternately avoiding the wait in updating the mask.

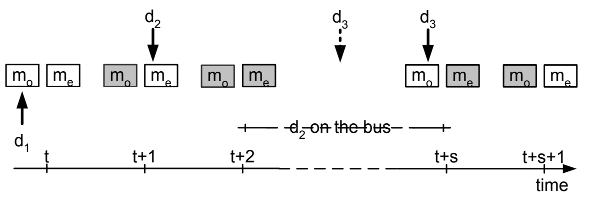


Figure 3: Using a pair of mask to avoid waiting the latency of mask update.

Figure 3 illustrates our mask pair scheme. Suppose both masks are available initially. At time t, an incoming data d_1 is received and thus the m_o is used for decryption. At $t+1$, m_o's update begins and thus it is not available for some time (shaded in the figure). Meanwhile, an outgoing data d_2 arrives at this processor. Since m_o is not available, m_e is used to encrypt d_2. (In fact, since d_2 is an even numbered message, m_e should be used by default even if m_o is available.) At $t + 2$, both masks are undergoing updates, and d_2 is being transmitted on to the bus. Suppose that the bus cannot start next transaction until $t + s$, i.e., its cycle time is s, and AES latency is also s, then m_o will become available at $t + s$. If there are any outgoing data, such as d_3, that arrive between $t + 2$ and $t + s$, they should be delayed until $t + s$ even in course of normal SMP operation. At $t + s$, since m_o has come back to service, d_3 can be encrypted with m_o quickly. At $t + s + 1$, m_o is off again but m_e is now ready. As we can see, without the mask pair, d_2 would be delayed to $t + s$.

Given above, it is possible that two processors try to send out data to the bus simultaneously both thinking their data is an odd (or even) numbered message, i.e., a race condition arises. However, since only one message can be sent on the bus at a time, the competing processors are ordered with respect to each other from the bus arbiter's standpoint. Thus, we require that every sender first obtain the bus from the arbiter and then XOR its data with the right mask. Now that the competing processor will first receive the bus data, it will decrypt it using m_o and then send out its data encrypted using m_e.

Note that we assume that the AES has a latency comparable to the bus cycle time. Under this condition, two masks in a pair are enough to remove all the possible delay in transferring messages. Also remember that with the mask pair implementation, the authentication now needs to be performed for both of them.

Multiple masks for higher bus speed. In the example shown in Figure 3, if the bus cycle latency was shorter than the AES latency, m_o would not be ready at $t + s$ and d_3 would be delayed. If more masks were provided, e.g., a third mask was there, d_3 could be encrypted in time. At the peak traffic volume of high throughput buses, multiple requests may be sent to the bus back-to-back. Thus, a mask is consumed every bus cycle and a new mask is needed after each bus cycle. While the AES throughput can achieve such a rate, i.e., one mask being produced in every bus cycle, the mask will not be available in many cycles. Thus, we need an array of masks to saturate the AES throughput. The number of masks necessary is $\left\lceil \frac{AES\ latency}{bus\ cycle\ time} \right\rceil$.

5. Hardware Requirements of the SHU

In the previous section, we elaborated on our bus encryption and authentication algorithms. All the techniques are designed with security and speed as the paramount priority. Besides the effort in crafting the algorithms, we must provide hardware that is also secure and fast as part of the design.

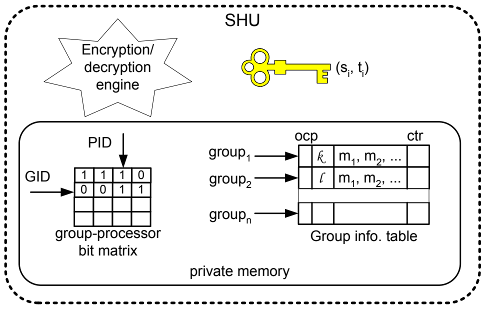


Figure 4: Architecture of the SHU.

5.1. Group processor bit matrix

Recall that each data message on the bus is tagged with its GID and PID. The SHU has to decide very quickly if it should read the message. It should read only if the message belongs to a group maintained on the host. Since the group information is dynamic as programs start and exit, we use a "group-processor bit matrix" shown in Figure 4 to maintain the information and perform lookup in $O(1)$. It is indexed by the GID horizontally and PID vertically. A bit at (g,p) being set means that processor p belongs to group g, and unset otherwise. Thus, a row of the matrix indicates all the member processors of a group and a column indicates all the groups maintained on a particular processor. An exception is that if a processor does not maintain group g, the g^{th} row in the matrix should contain all zeroes. This means that a processor should not know the information about a group which it does not belong to.

Thus, a processor snoops the GID and PID of the bus messages and uses them to index the matrix. If the bit is set, the corresponding key and masks are read out from the group information table (introduced next), and used to decrypt the message. Otherwise, the message is discarded.

5.2. Group information table

The other table, "group info. table" in Figure 4, is used to keep the secret information for each group. The group entry contains three fields: an "occupied" bit, the symmetric key k, and the masks. When an application is first loaded into the SENSS, the OS must assign it with a valid group ID (GID) which is used and non changeable throughout the life time of this application. This GID is obtained by inquiring the group information table for a free entry. The "occupied" bit indicates whether the current entry has been allocated. If all entries are occupied, the application is put into a queue waiting for the next available GID which is reclaimed upon completion of a program.

Once the GID is set up, the application passes its encrypted key information to every member in the group. Each processor then decrypts the key using its own private key and then stores the plaintext key k in the entry indexed by GID. The next field in the group information table is the mask(s) field. Each group uses *distinct* masks that are generated from the communication among the group members only. This is necessary because if different groups share a common mask pair, it is very easy to leak information to other groups as all bus transfers are encrypted using their masks. The last field, "ctr", stores the authentication interval for each program.

Once a GID is selected for a program, the corresponding table entry in *all* processors must set their "occupied" bits *even for non-group members*. This is to prevent the same GID from being used between non-trusted applications. Since every message sent on the bus is tagged with GID by the SHU, bus data with different GID should be picked up by different groups. The difference is that non-group members do not have the key and mask information as the group members do. It is also consistent with its corresponding null entry in the group processor bit matrix.

6. Combining With Cache-to-Memory Protection

We now discuss the integration of SENSS and memory encryption/authentication. We will use the fast memory encryption [29, 25] and the hash tree technique for memory integrity check [7, 25] since they are by far the most efficient techniques. Special considerations have to be taken as we are adapting them to an SMP architecture. The main concern is the consistency of the pads and hash trees stored on-chip in each processors. Next, we discuss these problems and how they can be solved in different cache coherence protocols.

6.1. Memory encryption

When each processor performs fast memory encryption on its own, the pads in their local caches may become inconsistent since different processors have different memory access pattern (pads are updated only on *memory* writes). For example, suppose initially both processor A and B hold data D in their caches and D's pad is consistent in their pad caches. Later on, if A pushes D down to the memory and updates D's pad, B cannot use the local old pad for D anymore.

As we can see that this problem is similar to traditional cache coherence problem. Thus, we can either use a "write invalidate" or a "write update" protocol to handle the changing of a pad. In "write invalidate", if a pad is changed in one cache, an invalidate message is sent on the bus invalidating the pad's other cached copies. Later on if the pad needs to be used by other processor, a request message is sent on the bus to acquire the latest copy. In "write update", every time a pad is changed in one cache, an update message is generated on the bus to all other caches so that other copies are always synchronized with the latest change.

6.2. Memory integrity check

When memory authentication is performed, its hash tree is partially cached locally and each processor maintains its local part of the tree. It is easy to see that tree node inconsistency among different processors could happen, just as the pads explained before.

Using similar strategy, we can adopt either "write invalidate" or "write update" protocol for newly modified data and its hash. The complication lies in that multiple invalidate or update messages may be invoked. Recall that the hash values of cache blocks are stored in their parent nodes. When a cache line is evicted to memory, its parent node in hash tree may not be in the cache for hash updating. If the parent node is in another cache, using either protocol can achieve a coherent update of the hash, and the process stops here. If the parent node is in the memory, we need to fetch it as well as initiate a second access to the grandparent node to authenticate the parent node and change its hash in the grandparent node. When the grandparent node is modified, its consistency needs to be handled again. This procedure repeats until a node is hit in local cache in which no further parent nodes need to be updated.

Since most of the SMPs adopt the "write invalidate" protocol for its better performance, we also use this protocol for pad and hash coherence in the SENSS.

7. Experimental Evaluation

7.1. Hardware overhead

Table sizes. There are two tables maintained in the SHU: group information table and group processor bit matrix. The matrix needs only 1024 entries $\times$5 bits per entry = 640 bytes, assuming the maximum number of processors is 32. For each entry in the group information table, the "occupied" field is 1 bit; the session key is 128 bits. The counter field can be from 0 to 32 bits. We chose 8 bits in our experiment. The number of masks we store for each group is 8 for encryption and for authentication. These masks can perform as good as using infinite number of masks from our experiments. The total number of bits per entry is 1161 bits, or 148.6KB for 1024 entries.

Encryption unit. We will need a fast encryption unit that can provide sufficient throughput compared to the bus bandwidth, and at reasonable latency. AES speed has been constantly improved by recent research and industrial effort: ASICS.ws introduced an AES IP core that needs 22-26 cycles at 266Mhz [3]; Schaumont *et al.* [19] introduced an AES prototype that requires 14 cycles at 154 Mhz. In accordance with these works, we model an AES implementation that requires 80 cycles for 1GHz processor. The throughput, however, needs to match the peak bus bandwidth which is 3.2 GB/s in our experiments. Throughput is generally improved by pipelining and logic duplication. In [8], the implementation of 128-bit AES unit can achieve 30∼70 Gbit/s with 175∼380K gates using 0.18μm CMOS technology. Thus, it is easy to match AES throughput with the bus bandwidth.

Bus designs. To distinguish different messages on the shared bus, the bus arbitrator needs to generate different message types for each transaction. This may require extra lines on the command bus. In SENSS, the following three additional types of transactions are distinguished:

- Type "00" indicates a *bus authentication* message.
- Type "01" indicates a *pad invalidate* message.
- Type "10" indicates a *pad request* message.

Hash invalidation and request do not need extra signals since the hashes are stored in L2 cache, which follows normal coherence protocol.

Besides the message type, PID and GID are sent along with each cache-to-cache transfer message. For the machine we modeled (SUN E6000 [26]), the PID is already there as "source ID" for each message. Thus no extra lines are necessary. GID is 10 bits in our design assuming the maximum groups simultaneously exist in SENSS is less than 1024.

These bits increase the bus lines in the system by an amount acceptable for modern SMP servers that adopt separate address and data buses. SUN gigaplane bus used in E6000 [21], for example, has a total of 378 bus lines (256 data lines, 41 address lines, and many other control lines). SENSS model adds 11 extra bus lines: 2 bit for message type, and 10 bits for GID. This is a modest increase of 3.1%.

Each bus message is XOR'ed with the mask before sending and after receiving the message. It takes only one cycle at the sender side since the mask is ready to use while it requires two cycles at the receiver side: one cycle is to find the GID and its current mask and the other cycle is to perform the XOR operation. Nevertheless, it adds 3 cycles to the bus delay.

7.2. Simulation framework

We modeled SENSS using Simics [11], a functional full-system multiprocessor simulator developed by Virtutech AB. We configured Simics with SPARC V9 and Solaris 9 OS and the system parameters are similar to Sun E6000 (Figure 5). The MESI cache coherence protocol is adopted. We selected the typical memory and bus latency similar to [4]: when the DRAM access time is 80 ns, the memory access latency is about 180ns due to the extra control delay. The cache-to-cache transfer bus latency is 120ns when there is no contention. The bus throughput is 3.2 GB/s which is comparable to a high speed Gigaplane bus [21].

Architectural Parameter	Value
Processor clock frequency	1 Ghz
Separated L1 I- and D-cache	64KB, 2-way, 32B line
L1 hit latency	2 cycle
Integrated L2 Cache	4-way, 64B line
L2 hit latency	10 cycle
Hashing throughput	3.2 GB/s
Hashing latency	160 cycles
Cache-to-cache latency	120 cycles (uncontended)
Cache-to-memory latency	180 cycles
Shared bus	3.2 GB/s, 100MHz, 32B line
AES latency	80 cycle
AES throughput	3.2 GB/s

Figure 5: Architectural parameters.

The benchmarks we used are unmodified binary codes for SUN Solaris. We chose programs from SPLASH2 benchmark suite [27]. We ran them with their typical setting as indicated in [1, 27] and used the same method as in [4] to collect statistics. The metrics we experimented are performance degradation, typical mask number necessary for a program, bus traffic increase, performance variation with different authentication intervals, and the performance changes when we integrate SENSS with cache-to-memory encryption and authentication.

7.3. Performance slowdown

To study the performance impact of SENSS, we compare it to a normal SMP machine with no security features. A "naive" implementation of bus encryption and authentication (direct encryption and MAC authentication) is of less interest because of its performance penalty. Here we do not include the overhead of cache-to-memory protection to see the net effect of securing the bus only.

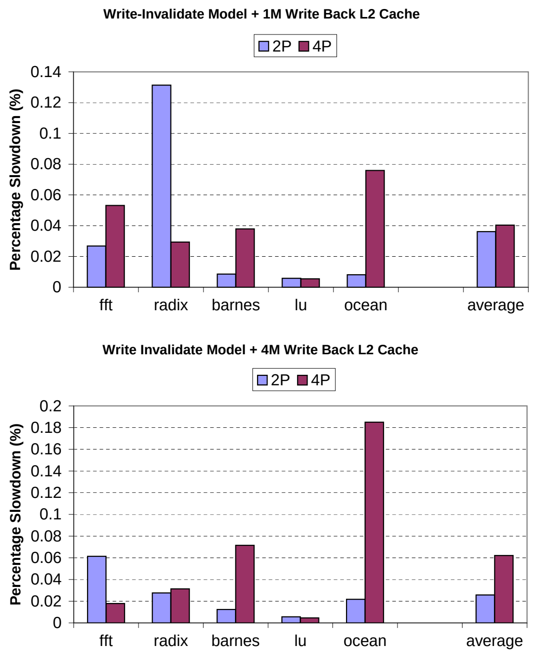


Figure 6: Performance comparison.

Figure 6 shows the performance slowdown using a 1MB and 4MB write back L2 cache on 2, and 4 SMPs. In the experiment, we set the authentication interval to 100 cache-to-cache bus transactions. That is, a mask consistency check is performed on every 100 bus transactions.

From the graph we can see that the performance degradation in general increases with the number of processors, and using a larger L2 cache incurs relative larger slowdowns. This is not because using more processors and larger caches are not good, but because the number of cache-to-cache transfers also increases. In fact, using either 4 processors or 4M L2 machines does speed up the programs. The SENSS impacts more on configurations that introduce more cache-to-cache transfers. Sometimes the results are counter-intuitive, e.g. the performance of `radix` in 2-processor is worse than in 4-processor with 1M L2. This is due to the *variability* in a full system simulation. This phenomenon happens throughout the experiments we have performed. In section 7.8, we will explain in details why such a situation can occur.

The overhead in SENSS comes from two sources: the encryption/decryption delay and the extra authentication messages. The former has a flat impact on the system and the latter is not on the critical path. When the bus is not saturated (in our case), it does not degrade the performance much. As we can see, the maximum slowdown is only 0.18% which is very minimal.

7.4. Number of masks needed

The previous experiment assumed a perfect supply of masks, i.e., a mask is always ready for use when it is needed. As we discussed in Section 4.4, the maximum number of masks required is a mask update latency (AES latency) divided by bus cycle time. A bus cycle indicates how frequently a message can be loaded onto the bus. We are modeling CPUs at 1GHz and the bus at 100MHz. Thus, one bus cycle is 10 CPU cycles. Consequently, the maximum mask number for this configuration is $\lceil 80/10 \rceil = 8$.

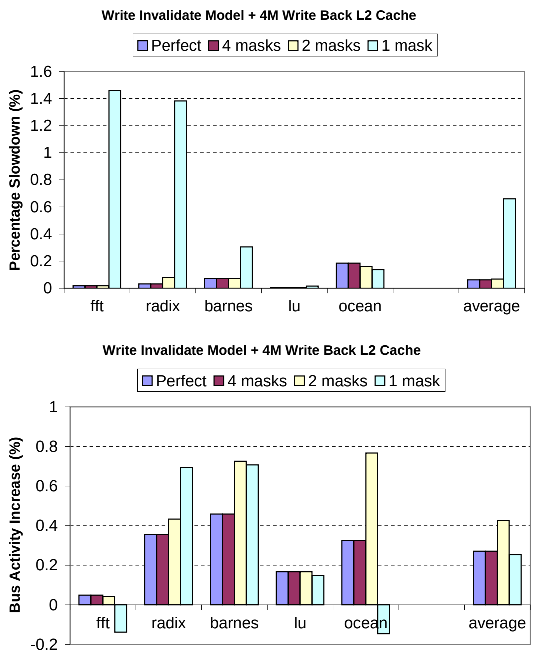


Figure 7: Impact with different number of masks.

However, maintaining 8 masks in the group information table is expensive in space. The table will grow to 148.6KB, larger than a normal L1 cache. In fact, 8 masks are only necessary when bursts of messages appear frequently. For the benchmarks we tested, 2∼4 masks are enough to handle back-to-back messages. Figure 7 shows this result. Using the same configuration as the previous experiment, we limited the mask supplies to the benchmarks. The results show that using 2 masks is generally satisfactory and using 4 masks is as good as the perfect case. Again, the odd behavior of some benchmarks such as `fft` will be explained in 7.8.

7.5. Bus traffi c increase

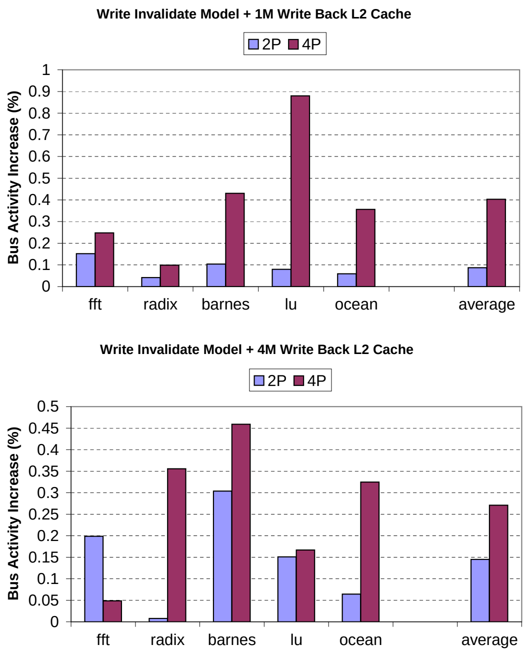


Figure 8: Bus traffic increase.

The periodically performed bus authentication adds additional traffic on the bus. We measure the percentage of the bus traffic increase in Figure 8. Here the authentication interval is still 100 transactions. We can see that the increase in the bus traffic is also very small: 0.46% at maximum. The total bus transactions include both cache-to-cache and cache-to-memory transfers. The authentication is added only to the cache-to-cache transfers. That is why nearly all the results are well below 1%.

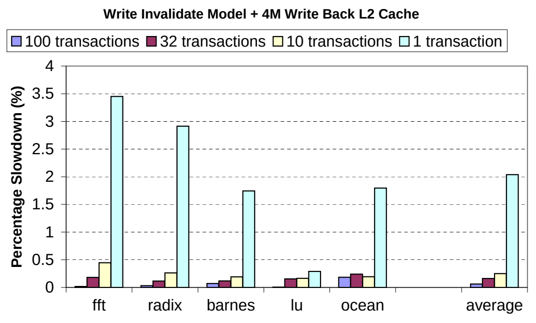
7.6. Varying authentication interval

The frequency with which we perform the authentication has some impact on both performance and bus traffic. We varied the interval from 1, 10, 32, to 100 bus transactions and measured the performance degradation and the bus traffic increase in Figure 9. Setting the interval to 1 transaction means that every cache-to-cache transfer is authenticated; thus, providing the maximum security level. Here we used 4-processor configuration and each processor is configured with a 4M L2 cache.

We found that even for short authentication interval, the performance degradation is still modest, the maximum is 3.4% when the interval is 1 bus transaction. Increasing the interval can restrain the performance loss but we feel the difference is not worth the decrease in security level. Short authentication interval does increase the bus traffic as we can see the maximum increase is 46% for 1-transaction interval. The numbers are actually the proportion of the cache-to-cache transactions within the total bus activity which also includes the cache-to-memory transactions.

7.7. Integrated system

As discussed in Section 6, extra messages are necessary when we integrate cache-to-memory protection into SENSS. Encrypting



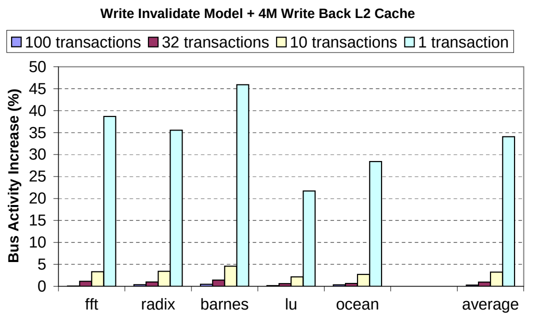
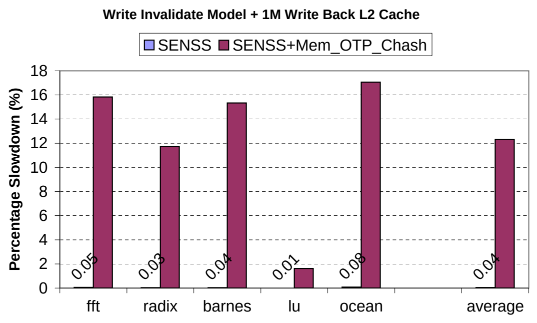


Figure 9: Impact with different authentication intervals.



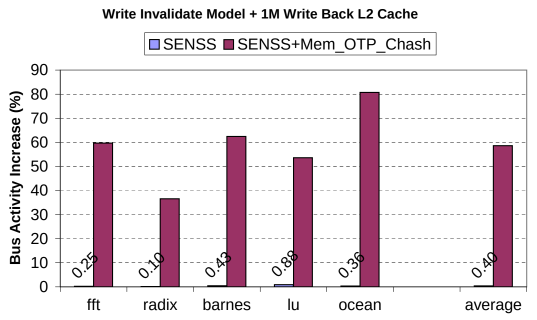


Figure 10: Comparison of the integrated system.

memory data uses the same encryption logic while additional hashing hardware is required for integrity check.

We integrated the fast memory encryption in [25, 29] and CHash memory authentication [7] algorithm. The memory encryption is relatively independent from our work, so we used a perfect sequence number cache (SNC) for simplicity since the difference between a perfect SNC and large SNC is small [29]. The resulting performance slowdown and bus traffic increase are shown in Figure 10. Here we used a 1M bytes L2 cache. On average, we see a 12% degradation in performance, lower than what was reported in [7] (∼25%) because of different benchmarks and experimental platform we are using. The performance penalty mainly comes from the polluted L2 cache due to the hash tree and the increased bus contention. The bus traffic is increased by 58% on average. The major reason is due to the multiple hash tree fetching requests for a newly fetched memory block and the hash tree coherence maintenance among different processors. The LHash algorithm presented in [25] gave much better performance than the CHash algorithm and thus will also be very effective in SENSS.

7.8. Simulation variability

An important note about the simulation results presented here is that they are only close estimates. As pointed out in [4], full system architectural simulations of multithreaded workloads incur both time and space variability. Small timing variations could cause runs starting from the same initial state to follow different execution paths, leading to performance variation. Thus multiprocessor simulations are not deterministic when system parameters are changed.

In our experiments, every bus transaction delay is increased by at least 3 cycles. This small increase affects the number of instructions executed on different processors in multiple runs of the same workload. It also changes the order of memory accesses, causing the same cache block to be accessed at different times. Cache coherence accesses in a multiprocessor environment complicates the problem. We use an example to illustrate how such a situation happens in Figure 11. The left timing diagram shows an access order in the base system, the right graph shows the same accesses with different order in the SENSS model. This example is extracted from a real program trace observed on Simics.

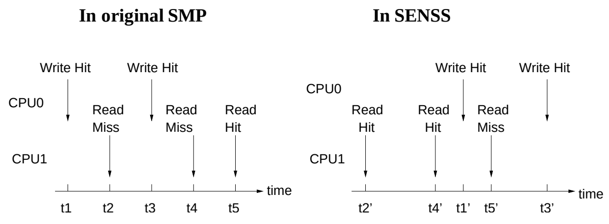


Figure 11: Reordered memory accesses in SENSS.

Suppose CPU0 and CPU1 need to access the same cache block B, and the initial state of B is shared between these two processors. In SENSS, if CPU0 spent more time waiting for some event such as the readiness of the masks, its first write hit may be delayed till even after CPU1 executes its first two reads. In such a case, not only the order of the accesses to the same block but also their hit/miss status have been changed. As a result, this program in SENSS might be faster than in the base case since it has fewer misses now. Also, the cache-to-cache transfers are reduced from 2 to 1. This scenario happens when CPU0 and CPU1 are false sharing block B, i.e., they access different words in B. Thus, reordering the write and read from different processors does not affect the correctness of the program but it does affect the timing.

Therefore, one might notice that some of the programs run faster and have less buss traffic than the base case. Since we are only trying to prove that the SENSS model has relatively little performance impact, we feel that our conclusions still hold.

8. Related Work

Section 2 discussed the background as well as the research that is most closely related to ours [7, 10, 25, 29]. In this section, we discuss some additional related work.

Shi *et al.* have proposed a security design for a multiprocessor environment [20]. In the proposed scheme, the bus was encrypted using OTP incorporating a bus sequence number, and authenticated using a hash function SHA256. The MAC is sent along with the data on each bus transaction. Unlike SENSS, the MACs generated every time were not chained as CBC MACs. Moreover, the data originator (PID) was not included in each bus transaction. Therefore, the aforementioned message dropping and spoofing (Type 1 and 3) attacks could not be detected by their scheme. The detailed analysis can be found in section 4.3.

Hardware supported protection can provide security at different levels. Fast secure co-processors [5, 28] have been proposed for efficiently supporting various cryptographic functions. Techniques developed for securing smart card [23] employ multi-phase encryption for small application, such as protecting credit card, etc. However they are not applicable to large applications and general purpose processors.

Securing network communication through an insecure channel has similarity to cache-to-cache data transfers in SMP systems. Both have to ensure confidentiality through encryption and data integrity through validation. However, network security focuses on securing long streams of data and is sensitive to encryption throughput. The design goal is to match the increasing line speed. In SENSS for SMP systems, the data access pattern and transfers are relatively random. The design goal thus is to speed up the processing of each single request. In the design, we take advantage of the shared snooping bus which does not generally exist in a large network. On the other hand, network security is more general and covers topics such as *denial of service* attacks that are not addressed in our secure computation model.

9. Conclusions

In this paper, we propose SENSS model - a secure computation model for symmetric shared memory multiprocessors. For securing cache-to-cache data transfers, we take advantage of the snooping bus and propose an efficient encryption scheme. We discuss its design and implementation in an SMP system. Our preliminary experiments show that the design is fast, low cost and results in modest overhead for the system.

10. REFERENCES

[1] A. R. Alameldeen, M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill and D.A.Wood, "Simulating a $2M commercial server on a $2K PC," *IEEE Computer*, Vol. 36, No. 2, pp.50-57, 2003.

[2] AlphaServer 4000/4100 systems.

[3] http://www.asics.ws/doc/aes_brief.pdf, ASICS.ws Technical Report.

[4] A.R.Alameldeen and D.A.Wood, "Variability in architectural simulations of multi-threaded workloads," *the 9th International Symposium on High Performance Computer Architecture*, pp. 7-18, 2003.

[5] J. Burke, J. McDonald, and T. Austin, "Architectural support for fast symmetric-key cryptography," *the 9th Architectural Support for Programming Languages and Operating Systems*, pp. 178-189, 2000.

[6] "FIPS PUB 113," *Federal Information Processing Standards Publication*, May 30, 1985.

[7] B. Gassend, E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and merkle trees for effi cient memory authentication," *the 9th International Symposium on High Performance Computer Architecture*, pp. 295-306, 2003.

[8] A. Hodjat, I. Verbauwhede, "Minimum area cost for a 30 to 70 Gbits/s AES processor," *IEEE Computer Society Annual Symposium on VLSI*, pp. 83-88, 2004.

[9] M.G.Kuhn, "Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP," *IEEE Transactions on Computers*, Vol. 47, No. 10, pp. 1153-1157, 1998.

[10] D. Lie, C. Thekkath, P. Lincoln, M. Mitchell, D. Boneh, J. Mitchell, M. Horowitz, "Architectural support for copy and tamper resistant software," *the 9th Architectural Support for Programming Languages and Operating Systems*, pp. 168-177, 2000.

[11] P.S.Magnusson, *et al.*, "Simics: A full system simulation platform," *IEEE Computer*, Vol. 35, No. 2, pp. 50-58, 2002.

[12] M.K. Martin, D.J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood, "Timestamp snooping: an approach for extending SMPs," *the 9th Architectural Support for Programming Languages and Operating Systems* , pp. 25-36, 2000.

[13] D. A. McGrew, J. Viega, "The Balois/Counter Mode of Operation," *http://www.csrc.nist.gov/CryptoToolkit/modes/proposedmodes/*

[14] http://www.modchip.com/.

[15] D.S. Nikolopoulos, and C.D. Polychronopoulos, "Adaptive scheduling under memory pressure on multiprogrammed clusters," *the 2nd International Symposium on Cluster Computing and the Grid*, pp. 22-29, May 2002.

[16] E. Petrank, C. Rackoff, "CBC MAC for Real-Time Data Sources," *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, pp. 315-338, 2000.

[17] N.I. of Science and Technology. FIPS PUB 197: Advanced Encryption Standard (AES) , November 2001.

[18] R.L. Rivest, A. Shamir, and L.A. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, Vol. 21, No. 2, pp. 120-126, 1978.

[19] P.R.Schaumount, H.Kuo, and I.M. Verbauwhede, "Unlocking the design secrets of a 2.29 gb/s rijndel processor," *Design Automation Conference*, 2002.

[20] W. Shi, H. S. Lee, M. Ghosh, and C. Lu, "Architectural support for high speed protection of memory integrity and confi dentiality in multiprocessor systems," *the International Conference on Parallel Architecture and Compilation Techniques*, pp.123-134, 2004.

[21] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres, "Gigaplane: A high performance bus for large SMPs," *IEEE Hot Interconnects IV*, pp. 41-52, 1996.

[22] W. Stallings, "Cryptography and network security, principles and practices" 3rd Ed., *Prentice Hall*, 2003.

[23] ISO/IEC 7816-3 Identifi cation cards - Integrated circuit(s) cards with contacts, 1st Ed., September 15, 1989.

[24] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architectures for tamper-evident and tamper-resistant processing," *the 17th International Conference on Supercomputing*, pp. 160-171, 2003.

[25] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Effi cient memory integrity verifi cation and encryption for secure processors," *the 36th International Symposium on Microarchitecture*, pp. 339-350, 2003.

[26] SUN Enterprise 6000 Series. http://www.sun.com/.

[27] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," *the 22nd International Symposium on Computer Architecture*, pp. 24-36, 1995.

[28] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: A fast flexible architecture for secure communication," *the 28th International Symposium on Computer Architecture*, pp. 110-119, 2001.

[29] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," *the 36th International Symposium on Microarchitecture*, pp. 351-360, 2003.

[30] X. Zhuang, T. Zhang, S. Pande, "HIDE, an infrastructure for effi ciently protecting information leakage on the address bus." *the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 72-84,2004.