Appears in the Proceedings of the 15th International Conference on
Parallel Architectures and Compilation Techniques (PACT), September 2006.

Efficient Data Protection for Distributed Shared Memory Multiprocessors

Brian Rogers[†], Milos Prvulovic[‡], and Yan Solihin[†]

[†]Dept. of Electrical and Computer Engineering
North Carolina State University
{bmrogers,solihin}@ece.ncsu.edu

[‡]College of Computing
Georgia Institute of Technology
milos@cc.gatech.edu

ABSTRACT

Data security in computer systems has recently become an increasing concern, and hardware-based attacks have emerged. As a result, researchers have investigated hardware encryption and authentication mechanisms as a means of addressing this security concern. Unfortunately, no such techniques have been investigated for Distributed Shared Memory (DSM) multiprocessors, and previously proposed techniques for uni-processor and Symmetric Multiprocessor (SMP) systems cannot be directly used for DSMs. This work is the first to examine the issues involved in protecting secrecy and integrity of data in DSM systems. We first derive security requirements for processor-processor communication in DSMs, and find that different types of coherence messages need different protection. Then we propose and evaluate techniques to provide efficient encryption and authentication of the data in DSM systems. Our simulation results using SPLASH-2 benchmarks show that the execution time overhead for our three proposed approaches is small and ranges from 6% to 8% on a 16-processor DSM system, relative to a similar DSM without support for data secrecy and integrity.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Parallel Architectures; K.6 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Design, Performance

Keywords

DSM Multiprocessor, Memory Encryption and Authentication, Data Security

1. INTRODUCTION

Security is an increasingly important factor in the design of today's computer systems, and researchers have recently begun to investigate tamper-resistant execution environments as a way to protect the secrecy and integrity of sensitive or copyrighted data in these systems. Providing this secure execution environment has become more challenging with the emergence of hardware attacks, such as snooping devices which can be attached to various buses [7, 8]. Due to the presence of such attacks, *software-only* approaches for security are no longer adequate since sensitive information used by security software, such as encryption keys themselves, can be compromised because they are kept as unencrypted program variables stored in the main memory and transmitted over the system bus. Since software-based security mechanisms are themselves vulnerable to hardware attacks, hardware-based security schemes are needed. This hardware support has commonly been proposed in the form of memory encryption to protect the secrecy of data and memory authentication to protect the integrity of data [5, 6, 12, 13, 16, 17, 18, 21, 23, 24, 25]. With this type of data protection, many important security issues in computing, such as Digital Rights Management violations, software piracy, and reverse engineering of code, can be addressed effectively.

One important class of systems that will require tamper-resistant designs for data secrecy and integrity are *Distributed Shared Memory* (DSM) Multiprocessors. This is especially evident when one considers the settings in which many DSM systems will likely be used in the future. For example, a growing use of large-scale DSM systems is in the context of *utility* or *on-demand computing* where a company owning large systems will ”lease” computational and storage resources of the system to customers who want to outsource their IT operations or who need more computational resources to run their applications. For example, DSM systems such as the HP Superdome are already being used to offer on-demand computing services [15] to a variety of users. Because large DSMs are powerful but expensive, customers often run critical applications which access and store secret corporate data (e.g. financial data, product information, client records, etc.) on them. As the utility computing model grows in popularity, a more diverse array of companies will adopt this model, and DSMs will host a wider range of applications using many types of sensitive data. In addition, DSM will be the predominant architecture of these systems since SMP cannot scale to large systems easily. Since these DSM systems are in a physically remote location relative to the customers, customers are often very concerned about the privacy and integrity of their computations, especially against hardware attacks that may be very hard to detect or trace. IndustryWeek pointed out that data privacy is one of the major concerns that prevents fast adoption of the on-demand computing model [4]. This concern may prompt customers to require on-demand com-

PACT'06, September 16–20, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-264-X/06/0009 ...$5.00.

puting providers to utilize tamper-resistant mechanisms in their DSMs.

We note that it is unlikely for the utility computing provider itself to be malicious, as this would create a poor business model. Instead, a large-scale DSM system owned by a corporation will likely be protected with relatively tight physical security that restricts system access to select employees. However, lessons from history have taught us that it is unlikely that this single layer of security would be failproof. For example, despite the relatively good physical security protection and limited access for Automatic Teller Machines, there have been repeated cases of ATM fraud by some *supposedly trusted* employees [2]. In one case, an employee inserted a PC into an ATM machine to monitor and steal customer accounts and PINs.

DSMs used for on-demand computing are in a similar situation in that the main ingredients that are conducive for physical tampering are there. First, DSMs (like ATMs) store highly valuable information belonging to many customers. For DSMs, this information may include financial data, product information, and client records. Second, the financial motivation to perform an attack can be large because stolen information is valuable to other corporations (corporate espionage) or criminals (identity theft). Finally, there exist some forms of attacks that hardly leave any traces. For example, physically inserting a snooping device in a DSM would be quite easy due to the exposed interconnection at the back of server racks. USB drive-sized devices with multi-GB storage can likely be attached and removed in a matter of seconds without shutting down the system, and without leaving visible traces. Thus, many corporations will likely wish to add another, difficult to break, layer of protection for the security of their data in the form of tamper-resistant DSM systems.

Architectural support for data secrecy and integrity has been studied extensively by researchers for uniprocessor systems [5, 6, 12, 13, 17, 18, 21, 23, 24], and more recently for Symmetric Multi Processor (SMP) systems [16, 25]. Unfortunately, such support for DSM systems has not yet been studied in detail. Uniprocessor schemes provide data encryption and authentication only for *processor-memory* communication and the main memory but do not address data protection for *processor-processor* communication present in multiprocessor systems. Proposals for secure SMP systems include encryption and authentication mechanisms for processor-processor communication, but these mechanisms rely on the assumption that each processor can observe every coherence transaction in the system, which is satisfied due to the single shared bus in the system. This assumption cannot be made in DSMs, where communication is point-to-point rather than through broadcast mechanisms. As a result, new techniques for DSMs are needed.

Contributions. The first contribution of this paper is an analysis of the security requirements for protecting DSM systems against hardware attacks. The findings of this analysis are that passive/eavesdropping attacks are more likely to be attempted because they are non-intrusive and leave very few (if any) traces. Active attacks that modify coherence messages and alter the behavior of the DSMs are less likely to be attempted, especially if the system is augmented with the ability to detect them. Therefore, we seek to prevent passive attacks from succeeding, and we simply detect and report active attacks. To achieve this, we find that different coherence protocol messages and different parts of a message need to be protected differently: with both encryption and authentication, with authentication only, or with no protection.

One possible approach to create a secure DSM is to provide direct encryption and authentication, in which direct encryption (or decryption) and Message Authentication Code (MAC) generation (or verification) are performed for each coherence message sent (or received). However, this approach would directly add cryptographic latencies to the already problematic communication latencies in DSM systems. Therefore, our second contribution is a new combined counter-mode encryption/authentication scheme that hides much of the cryptographic latencies due to protecting processor-processor communication. Our scheme relies on two essential techniques. First, we observe that if communicating processors share the same communication counter, they can pre-generate one-time pads used for message encryption and decryption. Hence, to hide encryption/decryption latencies, we use per-processor pair communication counters that are incremented asynchronously after each message send/receive. Secondly, we also maintain data integrity through the use of GCM, a MAC-based authentication technique using a combined authenticated-encryption mode [3, 14] to reduce the MAC computation latency to only a few cycles after message ciphertext is available. Finally, we also show how our mechanisms can be seamlessly combined with previously proposed processor-memory data protection mechanisms to provide system-wide data protection for DSMs. Our results show a small performance overhead for our techniques: ranging from 6% to 8% on average across all SPLASH-2 [22] benchmarks when compared to a DSM system with no support for data protection. We also present a sensitivity analysis to show that overheads remain low across different system configurations such as L2 cache size and number of processors.

The remainder of this paper is organized as follows. Section 2 discusses related work. We discuss our assumptions and attack model in Section 3. Then, we present our contributions on security requirement analysis in Section 4 and our processor-processor communication protection scheme in Section 5. Section 6 details our evaluation setup. Section 7 presents our evaluation results and analysis. Finally, section 8 summarizes our findings and conclusions.

2. RELATED WORK

Architectural support for data secrecy and integrity has been studied extensively by researchers for uniprocessor systems [5, 6, 12, 13, 17, 18, 21, 23, 24]. These studies assume that on-chip storage is secure, while off-chip communication is not secure and needs to be protected against passive and active hardware attacks. They provide encryption and authentication for data in the *processor-memory* communication path through direct encryption [6, 12, 13] or through counter-mode encryption [17, 21, 23, 24]. In counter mode, instead of directly encrypting the data, encryption is applied to a *seed* to generate a *pad*. A seed typically consists of the memory block address and a counter. To encrypt or decrypt a data block, it is XORed with with the pad. When a block needs to be fetched from memory, if its counter is available on chip, pad generation can be overlapped with DRAM access latency. Counter mode encryption's security relies on the uniqueness of the pad/counter each time it is used for encryption (through incrementing the counter on each write back), hence it is often referred to as a one-time pad scheme. To provide data integrity, an authentication mechanism based on a Merkle Tree was proposed [5]. The Merkle Tree maintains an authentication tree whose leaf nodes are data blocks, and the root node is always stored securely on chip. Merkle Trees were proposed as a way to prevent *replay attacks* in which an attacker replays a previously observed data value and corresponding MAC.

Because uniprocessor protection mechanisms only apply to processor-memory communication, researchers have proposed protection schemes for *processor-processor* communication in bus-based Symmetric Multi-Processor (SMP) systems [16, 25]. The fundamental assumption used for such protection is that each processor can observe every coherence transaction in the system provided naturally through snooping the bus. In this system, each processor maintains a global encryption counter or global pad used for processor-processor communication. On each bus transaction, each processor updates its counter [16], or uses the snooped data to generate a new Cipher Block Chaining (CBC) encryption pad [25]. The pad is used for both encrypting and authenticating processor-processor communication.

Unfortunately, neither uniprocessor nor SMP protection schemes can be extended directly to protect DSM systems. Extending direct encryption/authentication for processor-processor communication would incur a very high performance overhead due to the added latencies at the sender side for encrypting data and generating MACs, and at the receiver side for decrypting data and verifying the MACs. With a recent hardware implementation showing an AES latency of 37ns and MD5 or SHA-1 over 300ns [11], this approach is either too costly or not feasible.

Alternatively, one may imagine an approach in which uniprocessor counter-mode encryption is directly extended to protect processor-processor communication by treating processor-to-processor data transfer similarly to a processor-to-memory writeback. However, this approach is problematic to support due to the need to keep the counters in both the sending and receiving processor coherent. For example, in response to an intervention to a dirty line, a processor flushes the line to the requester, and the flushed line would be encrypted by XORing it with a pad obtained by incrementing the current counter for the block. This increment would trigger invalidation of other cached copies of the same counter. In order for the receiving processor to decrypt the flushed line, it needs to obtain the new counter value for the block. It would do so by sending a read request for the cache block that contains the counter, which eventually appears as an intervention to the sender processor. Hence, the latency for processor-processor communication is effectively doubled (obtain data, then its counter). In addition, the counter communication needs to be protected against tampering as well, so it requires high-latency authentication. Similar difficulties exist with maintaining coherency among nodes in the Merkle tree.

It is also clear that SMP protection cannot be extended easily for protecting DSM systems. The requirement that each processor observes all coherence transactions would be costly to support in terms of ensuring a global ordering of all transactions as well as the large bandwidth requirement needed for broadcasting each transaction to all processors.

Our work in this paper differs from previous approaches in that it proposes architectural support for data secrecy and integrity in *DSM* multiprocessors. It does not rely on maintaining coherence for counters or the authentication tree, and does not require broadcasting of coherence transactions. Finally, while the use of GCM for processor-memory protection in uniprocessor system has been proposed in [23], this paper applies GCM in the different context of processor-processor communication protection. Hence, the input to GCM is very different than that for uniprocessor systems.

3. ASSUMPTIONS AND ATTACK MODEL

3.1 Architecture Assumptions

Our first assumption is that the DSM system uses a home-based directory protocol for maintaining cache coherence. Figure 1 illustrates processor-processor communication in such a DSM system. Suppose a processor P_R sends a read request to the home node P_H which keeps the directory for the requested address (Step 1). P_H finds that the line is currently owned (in a modified/dirty state) by another processor P_O. So it sends an intervention request to P_O (Step 2), which will respond by downgrading its line to a clean shared state, flushing its line to update the copy at home (Step 3a), and replying with the data to the requester (Step 3b). The figure shows that some processor-processor coherence messages only carry cache coherence protocol information (command, address, message source and destination), which we will refer to as *non-data messages*, while others also carry data of the application program, which we will refer to as *data messages*. In a DSM, data messages are either responses to non-data coherence messages, such as intervention or invalidation requests, or self-initiated write-backs to the home memory.

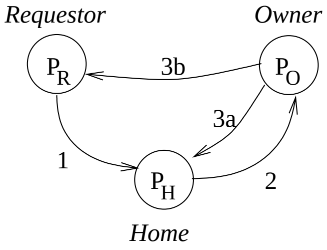


Figure 1: DSM processor-processor communication.

In addition, we assume a DSM system in which a node's directory controller is integrated in the processor chip, similar to the configuration used in the IBM Power4 system [9] and AMD Opteron [1]. In this case, any accesses to the home memory consist of two steps: communication between the home node's directory controller and the requesting processor, and between the home node's directory controller and its local memory. For example, a reply to a remote read (to a clean line) first causes the line to be fetched from the home's local memory into the home processor chip, and then forwarded to other processor chips through an interconnection network. This assumption enables us to employ two separate protection mechanisms: processor-memory communication protection can be handled using well-studied uniprocessor memory protection techniques, while processor-processor communication needs a new protection mechanism.

3.2 Security Assumptions and Attack Model

As mentioned earlier, our goal is to protect DSM systems against hardware attacks in the context of on-demand computing. We assume that the system has relatively strong physical security, but is not immune to attacks by a select few employees or other parties who have physical access to the machine. Since it is likely that only a few people have physical access to the machine, any attacks that leave *traces* may easily provide sufficient information that can lead to the attacker. We define a trace as a detectable anomaly of the system behavior. Hence, the fundamental assumption is that *the goal of an attacker is to perform traceless attacks* in order to steal sensitive data that belongs to the application.

We broadly categorize hardware attacks into three categories. The first category is *sabotage* attacks in which the attacker's goal is to crash the application or even damage the system. Our scheme does not seek to protect against sabotage attacks, including application or system crashes, since it is extremely difficult to protect the system against such sabotage when the attacker has physical access to the machine. On the other hand, the attacker lacks the incentive to do so because the attack can be easily traced back to him/her, and there is probably little financial reward for sabotage attacks.

Another category is *passive* attacks in which the attacker's goal is to eavesdrop on processor-processor or processor-memory communication, as illustrated in Figure 2. An example of this attack is the physical insertion of a snooping device onto the exposed interconnect at the back of server racks. A small USB drive-sized device with multi-GB storage can likely be attached and removed in a matter of seconds without shutting down the system if the system can recover from temporary link failures. Cable clutter may also hide the device from cursory visual inspections.

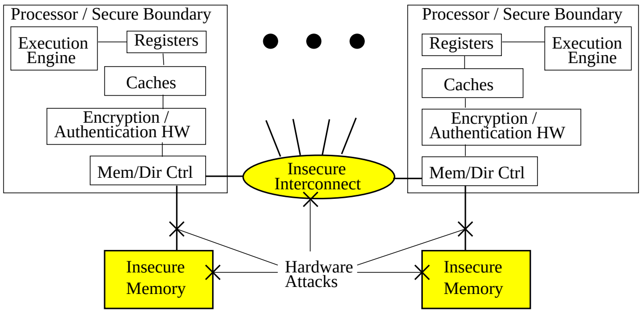


Figure 2: Attack model, secure boundary, and location of our encryption and authentication hardware.

Finally, in *active* attacks, the goal of the attacker is to steal sensitive information by modifying coherence messages communicated between processors, or data in a node's local memory or on the memory bus. Although active attacks are certainly more difficult to perform than passive attacks, we cannot rule out the possibility of an attacker attempting them, especially if passive attacks are no longer fruitful due to the system encrypting all off-chip communication, and if the attack does not result in any traces. A coherence message typically contains message type, memory block address, routing information (source and destination processors), and, for data messages, user data. We do not make any assumptions as to the specific abilities of attackers to modify signals, so we assume the worst case in which the attacker is able to modify any parts of the message. We distinguish between attacks that modify application data as *data spoofing* versus ones that modify other information as *non-data spoofing*. The attacker may also be able to *replay* an old message. Finally, the attacker may also modify the coherence protocol directory information stored at each node.

4. SECURITY ANALYSIS FOR DSM PROTECTION

This section analyzes the requirements for secure data protection in DSM systems. We note that the security requirement for protecting against passive attacks on data confidentiality is straightforward: application data in data messages must be securely encrypted during communication. Thus we focus our discussion on the requirements for protection against active attacks. At first glance, it may seem that this requirement is also obvious: simply authenticate all parts of *every* coherence message through the use of a MAC. However, we note that this requirement would introduce excessive performance overhead, and it is actually overly strong for DSM systems and can be relaxed.

Specifically, we can derive a new security requirement for authentication based on the assumption that the goal of an attacker is to perform traceless attacks (Section 3.2). This assumption implies that the protection requirement can be met by two components: (1) Ensuring that every attempt at an active attack results in a *trace* of the attack, and (2) Traces prompt analysis and preventative/corrective actions to block the active attack attempts from succeeding. We define a trace of an attack as a detectable anomaly of system behavior. More specifically, in the context of DSMs, we can define a trace as *incorrect/anomalous coherence protocol behavior* or *cryptographic errors* such as a failed message authentication. Interestingly, many active attack attempts will naturally result in detectable coherence protocol anomalies. In this paper, we assume that mechanisms are in place to detect such anomalies, and the discussion of these mechanisms is outside the scope of this paper. Hence, we only need to ensure that attacks that do not result in coherence protocol anomalies can still be detected as cryptographic errors.

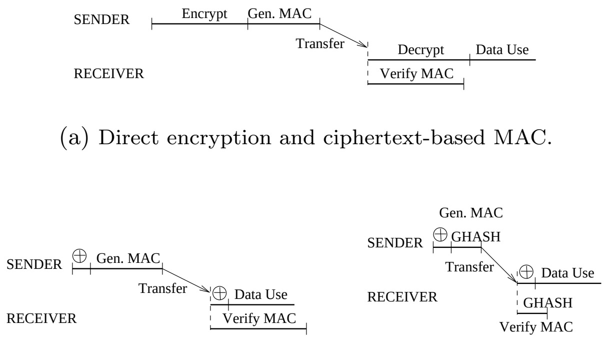
We first analyze this requirement for non-data messages. These messages contain coherence message type, address, and routing information (source and destination processor IDs). If a message's type is modified by an attacker, a protocol anomaly will result because the receiving processor can check whether the message is allowed based on the stored coherence state for that address. Examples of anomalies would be receiving an invalidation acknowledgment without a matching invalidation request, or not receiving acknowledgment or negative acknowledgement as expected, etc. Similarly, if a message's address or source/destination processor IDs are changed, a protocol anomaly will likely result. There may be special cases in which no anomaly results because the cache state of a line allows such an operation. However, we note that the attacker in this case can only either force a data write back, force data in the cache to be sent across the interconnect (and data secrecy is not compromised as long as data messages are protected), or cause denial of service (which we do not seek to protect against because its success is itself an easily detected trace). An attacker may also attempt to delete messages rather than tamper with them. This can achieve an effect similar to a replay attack because the attacker can, by not delivering invalidation messages, force a processor to use stale data from its cache. In this case, we assume that the protocol logs an anomaly if, after a certain period of time, it has not received a response to its invalidation or intervention message. Finally, some protocol anomalies can be expected to occur naturally due to protocol races. For example, an invalidation may be received while a line is being written back to the home, so an invalidation message to a line that is not in the cache may not signal an active attack attempt. In this case, we rely on the protocol to detect unusual patterns such as an excessive number of anomaly cases. Again, hardware extension for recording protocol anomalies is beyond the scope of this paper.

Now we consider data messages which carry sensitive data that belongs to the application program. Since application data is not used directly in the coherence protocol, it may be modified by an attacker without raising any protocol anomalies. Therefore, application data needs to be protected by authentication codes so that tampering with this data is detected. Finally, attacks such as routing application data to another processor, or faking the address of the data may achieve a similar effect to data tampering (i.e. causing a processor to use the wrong data value). Therefore, for data messages, we also need to authenticate *all parts of the message* including the message header information. In addition, an attacker may attempt to *replay* an old message together with its authentication code, so a separate mechanism must be used to detect such replays.

In summary, the security requirement for message integrity is met in the following way. We do not encrypt or authenticate non-data messages, but assume that the coherence protocol tracks protocol anomalies and raises an alarm if anomalies are detected. We encrypt application data and authenticate all parts of data messages so that data tampering is completely avoided, and any attempt at data tampering results in cryptographic errors, which also raises an alarm. Finally, we provide a separate mechanism to detect message replay attacks.

5. PROCESSOR-PROCESSOR DATA PROTECTION FOR DSM

As we discussed in Section 4, our goal is to provide protection against hardware attacks on data messages in DSM systems. To accomplish this goal it is necessary to have mechanisms to encrypt and authenticate data during processor-processor data transfers across the interconnection network. Section 2 has discussed that direct encryption and authentication of data messages incurs a high overhead, while direct extension of uniprocessor or SMP protection is either costly or infeasible. To make DSM protection practical, our mechanism should significantly reduce or hide encryption, decryption, and MAC generation/verification latencies, while meeting the security requirements discussed in Section 4.



(a) Direct encryption and ciphertext-based MAC.

(b) Pre-generated pads for encryption.

(c) Pre-gen. pads for enc. and auth.

Figure 3: Processor-processor communication latencies using various mechanisms for enc. and auth.

We achieve this goal through *counter-mode encryption* and *GCM authentication*. Let us discuss encryption issues first. First, it is well known that if two communicating processors agree on a common stream of counter values used for encryption, they can pre-generate encryption pads before data is ready to be sent or received. To encrypt or decrypt data, it only needs to be XORed with the pre-generated pad. Figure 3(b) shows the new reduced encryption/decryption latencies compared to direct encryption (Figure 3(a)). However, the latency of authentication is still not hidden. To hide authentication latency, we use a combined counter-mode encryption/authentication scheme, Galois Counter Mode [3, 14]. GCM offers many benefits. First, its security strength has been thoroughly studied and proven to be as strong as the underlying AES encryption algorithm [3, 14]. Second, GCM utilizes the existing AES hardware already used for encryption. Finally, since GCM relies on the use of counters, if a counter value is known, the authentication pad can be pre-generated, hiding most of the authentication latency. The only exposed part of the latency is GHASH computation, which is a short chain of Galois Field Multiplications and XORs, each of which can be performed in one cycle [3]. Figure 3(c) illustrates this reduced authentication delay and compares it to direct MAC generation (Figures 3(a) and 3(b)).

Figure 4 shows our processor-processor encryption mechanism in the upper part and authentication in the lower part. The encryption pad is obtained through AES encryption of the *encryption seed*. To ensure that pads are not reused, which is the security requirement of counter-mode schemes, the seed must be unique for each message in the system. Thus, we choose the concatenation of the communication counter (Ctr), processor ID of the sender ($ID(S)$), processor ID of the receiver ($ID(R)$), and an arbitrary Encryption Initialization Vector (EIV). Once a data transfer needs to be made for a certain cache line, the plaintext of the data $Ptext$ is XORed with the pre-generated pad to produce the ciphertext $Ctext$. The seed selection is such that a pad is always unique for any processor pair, and even for a processor pair it is unique if the role of the sender and receiver are switched. The counter is incremented after each message is sent or received, hence the pad is also unique across messages communicated by a processor pair. Therefore, each pad is globally unique across all messages, so the attacker cannot discover application data through passive/eavesdropping attacks. Finally, note that the seed we choose still allows pads to be pre-generated because the seed input is independent of application data.

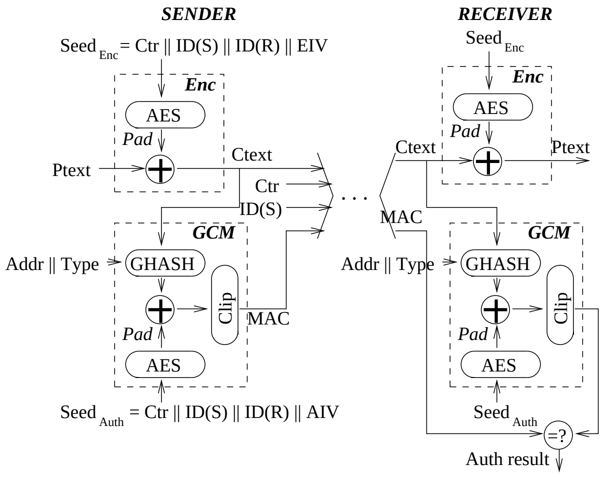


Figure 4: Our mechanism for processor-processor secure data transfer in DSM systems.

For authentication, recall that the security requirement requires us to authenticate all parts of a *data* message. Therefore, the MAC must encompass the counter, processor IDs of the sender and receiver, address, and type of the coherence message. This ensures that tampering with any part of a data message will be detected as a cryptographic error (Section 4). The authentication seed is chosen as the concatenation of Ctr, $ID(S)$, $ID(R)$, and an arbitrary Authentication Initialization Vector (AIV). Such a seed ensures that each authentication pad is unique across all messages in the system, and is different from encryption pads. We note that in GCM, *additionally authenticated data* (i.e. data that is authenticated but not encrypted) can be supplied along with the data ciphertext into the GHASH function to generate the MAC. We leverage this feature in order to protect the message header information such as address and message type. The counter along with sender and receiver processor IDs are accounted for in the MAC since they are part of the seed used to pre-generate the authentication pad. When encryption has produced the ciphertext of a cache line to be transferred, the ciphertext and additionally authenticated data is input into the GHASH function. The output of the GHASH function is XORed with the authentication pad, and the result may be clipped to the desired MAC size [14]. Therefore, the GHASH and XOR latencies are the only part of the MAC generation latency that cannot be hidden. Fortunately, with GCM the GHASH latency is only a few cycles [3].

Finally, the coherence message sent must now contain the ciphertext of the application data and the MAC. Additional fields need to be sent also, including the counter and the processor ID of the sender. The counter is needed because the counters kept by the sender and by the receiver may occasionally be out of synch. The counter sent along with the message allows the receiver to verify that its pre-generated pad corresponds to the correct counter value. The need for sending the processor ID of the sender along with the message will be described in the following subsections.

In order for our mechanisms to operate efficiently, it is clear that the key issue is how to manage the communication counters and their pre-generated pads so that the counters are synchronized at the sender and receiver side most of the time. If counters and pads are not available, or the sender and receiver get out of sync, then we will suffer the full pad generation latency before data can be encrypted or decrypted and authenticated. Thus, a counter management scheme should ideally incur a low *performance overhead*, low *storage overhead*, and at the same time have *no scalability restrictions* which prevent the DSM from scaling to a large number of processors. We propose three counter management techniques which we describe in the following subsections. They differ in how they meet the criteria we just outlined.

5.1 Private Counter Stream (*Private*)

Our first scheme is a straightforward approach in which a processor uses a separate counter stream for sending data to each of the other processors. In this scheme, each processor pair maps one-to-one to a counter for each communication direction. If P denotes the total number of processors in the system, each processor contains a *send table* with $P - 1$ entries for encrypting and sending data messages, and a *receive table* with $P - 1$ entries for receiving and decrypting data messages. Separate send and receive tables are necessary because when the two processors switch roles as sender and receiver, the order of concatenation of their processor IDs is different, and thus the pre-generated pads are different. Between a specific sender and a receiver, counters are kept synchronized by having each processor increment the corresponding counter each time time that counter's pad is used to encrypt or decrypt a data transfer. Each send or receive table entry contains a 64-bit *counter value*, a 512-bit *pre-generated encryption pad* (assuming 64-byte cache block size), a 128-bit *pre-generated authentication pad* (assuming 128-bit MACs), and a *valid bit* indicating that the pad has been pre-generated but has not been *consumed* (used for encryption/decryption and authentication). The total storage per entry is $1 + 64 + 512 + 128 = 705$ bits. The total table overhead is small except for very large DSMs. For example, for a 64-processor DSM, the table overhead is $(2 \times 64 \times 705)/8 = 11,280$ bytes per processor.

To send a data data message, a processor first looks up its send table to find the entry that corresponds to the receiving processor. If it finds the entry with the valid bit set (i.e., *a pad hit*), the pads can be immediately used for encrypting and authenticating the data. After the pads are consumed, the counter value is incremented, the valid bit is cleared, and new pads based on the new counter value are generated. If the sender finds the entry with the valid bit cleared (i.e., *a pad half-miss*), it waits until the pads that are currently being generated become available. Note that a pad half-miss is relatively rare: it only occurs when two data requests to/from the same processor are separated in time by less than the AES unit's latency.

Upon receiving the message, the receiver locates the sender's entry in its receive table, and decrypts and authenticates the message immediately if it has valid pads and the counter value of the entry matches the counter value in the message. Otherwise, it waits until it finishes computing the correct pads. If messages are delivered in-order through the interconnection, then decryption and authentication latencies will always be fully or partially hidden. However, if messages from a single sender can arrive out-of-order, performance of the receiver could be degraded because the receiver must generate a pad that corresponds to the counter value in the message. Since in general out-of-order message delivery is quite rare, the extra pad generation latency also occurs rarely, so we choose to tolerate it. An alternative solution would keep a few counter values per processor and their pads in the receive table, at the expense of having a larger receive table.

5.2 Shared Counter Stream (*Shared*)

The *Private* scheme is simple and has almost perfect pad hit rates, but its counter storage overhead can be non-trivial for very large DSMs (e.g. 180KB for 1024-processor DSM). The second table organization scheme which we refer to as *Shared* seeks to reduce the storage overhead by a factor of two. To achieve this, we replace the send table of a processor with a single counter and pad for sending data messages to any processor. This shared counter is incremented after each sent message, so pad uniqueness is still guaranteed. In order to pre-generate a sender's pad that is usable for sending a message to any receiver, the encryption and authentication seeds used for pad computation do not include the receiver processor ID (i.e., $ID(R)$). Pads are still unique because a processor updates its sending counter and pad after each sent message. To meet the security requirement for authentication, $ID(R)$ is now concatenated with the block address and message type as input to the GHASH function.

As in *Private*, upon receiving a data message, the receiving processor accesses the entry corresponding to the sending processor and checks whether a pre-generated pad is available for that sender. However, since the sending processor uses the same counter to send messages to all processors, it is less likely for a receiving processor to see back-to-back messages with contiguous counter values. Non-contiguous counters occur when a processor receives a message from a particular sender, while the sender's previous message was sent to different processor. As a result, the receiver will suffer from a higher pad miss rate and full decryption and MAC generation latencies. The ability of the receive tables to pre-generate the correct pad more often could be enhanced through prediction of sharing patterns, but this is left as future work.

5.3 Cached Counter Stream (*Cached*)

Recall that *Private* achieves high performance due to a low pad miss rate, but needs larger storage overhead, while *Shared* sacrifices performance for lower storage overhead. However, both *Private* and *Shared* are not scalable in the sense that the tables are designed to support only a fixed number of processors. Unless the tables are very large, they prevent DSMs from scaling to larger numbers of processors. In this section we introduce the *Cached* table configuration to address the scalability drawback of previous schemes.

Our *Cached* scheme can scale to an arbitrary number of processors with fixed send and receive table sizes, while still providing good performance. The intuition behind its design is that processors in a DSM system often communicate with a set of neighbors that is much smaller than the total number of processors in the system. Therefore we can limit the table size of each processor's send and receive table to some number of entries that is a fraction of the number of processors in the system. This table can operate similarly to a cache, where a send/receive to/from a processor that does not have an entry in the table will create a new entry for this processor in the table, replacing the entry that has been unused for the longest time. However, unlike a regular cache, displaced entries are simply discarded, instead of written back to other storage. By simply discarding displaced entries, we avoid the need to allocate off-chip storage for them, which would need to be protected against attacks with additional security mechanisms.

This *Cached* scheme raises the question of which counter value should be used to generate the encryption/decryption pad if no table entry is found. For receiving, a straightforward solution is to use the counter that is sent with the data. For sending, we must select a counter that has not been used before, in order to prevent pad reuse. To achieve that, we keep track of the *maximum* counter value that has been used by a given sender to generate a pad across all receivers ($maxCtr$). Furthermore, as an optimization, we always keep a pre-generated pad for a counter $maxCtr + 1$. If the sender finds a table entry corresponding to a receiver, it simply uses the pads in that entry. If it does not find an entry, it creates a new entry by replacing the LRU entry in the send table, then immediately uses $maxCtr + 1$ and its pre-generated pads to encrypt and authenticate its message, increments $maxCtr$, and generates its next pads. This optimization avoids pad-miss stalls at the sender in most cases. Compared to *Shared*, *Cached* will tend to have more consecutive counter values at the receiver, so it will have fewer pad misses and better performance than *Shared*. However, it is still expected to perform worse than *Private*.

Finally, in order for the $maxCtr + 1$'s pads to be usable for any receiver, the encryption/authentication seed used for the $maxCtr+1$'s pad computation does not include the receiver processor ID ($ID(R)$). However, pads stored in the table still include the receiver processor ID in the seed. For the receiver to handle this correctly, each message is augmented with a bit to tell the receiving processor which seed to use for pad generation.

5.4 Detecting Replay Attacks

Section 4 states that data messages need to be fully authenticated, and a special mechanism is needed to detect replay attacks. While we have satisfied the former requirement (Figure 4), this section presents the mechanism to deal with the latter. Note that due to the full protection of data messages, the attacker cannot modify any part of a message without causing failed authentication. Therefore, the only replay attack an attacker can do is to send an old copy of a message together with its valid old MAC.

One way to detect such replay attempts is to note that a counter's value is monotonically increasing. A replayed old message is detected when a received message's counter is smaller than the current counter stored in the receive table. An alarm can be raised when this situation is detected. However, we note that out-of-order message delivery can also cause out-of-order counter values. Thus, false positives due to out-of-order delivery of messages could occur.

We may like to distinguish between a replay attack and out-of-order message delivery more definitively. To achieve that, first we observe that data messages are either responses to data request/intervention/invalidation messages, or a natural write back of modified lines. In the former case, we can send an authenticated counter value along with the request/intervention/invalidation message. We will refer to this counter value as the *originator counter.* The receiving processor then sends the data message reply augmented with the authenticated originator counter value. The originator of the request/intervention/invalidation message (the receiver of the data message) keeps a list of outstanding transactions and their corresponding originator counter values. If the received data message has an originator counter not matching any in its outstanding transactions, it has detected a real replay attack. For the latter case of natural write backs, the sender of the written-back cache line can keep track of outstanding write back transactions and their originator counter values. Upon receiving the written back cache line, the home node is now required to send a fully authenticated write-back acknowledgment that contains the originator counter from the write back message. Upon receiving the acknowledgment, the sender detects a replay attack if the originator counter in the acknowledgment message does not match any one from its outstanding write back transactions. This protection works because, even though an attacker knows the value of the originator counter, he could not have produced a reply containing the encrypted and authenticated originator counter unless the attacker has the encryption/authentication key. So the message reply received by the originator must have been produced by a legitimate sender. In addition, the reply message is accepted only once by the originator because it keeps a list of outstanding transactions, and a transaction is removed from the list if its legitimate reply has been received.

With the ability to detect replay attacks, combined with full protection of data messages, we have satisfied all the encryption and authentication security requirements discussed in Section 4.

5.5 Implementation Issues

The size of the counters for encrypting processor-processor messages can be chosen to be large enough so that they will not overflow for the expected life of the system, but not too large to cause too much increase in bandwidth. In this work, we use 64-bit counters to avoid counter overflows for many years, and all of our evaluation results take into account the extra bandwidth necessary to transmit these counters with the encrypted data and MAC. 64-bit counters do not present a storage problem in our schemes because we only deal with processor-processor communication, so our counters are only needed in send and receive tables.

Finally, we apply uniprocessor counter-mode encryption similar to [24], and Merkle tree authentication to the each node's main memory including data and directory information, using state-of-the art GCM-based Merkle tree uniprocessor authentication similar to [23]. To avoid coherence problems for uniprocessor counters and Merkle trees, each node only protects its own local memory. A reply to a remote request is protected with uniprocessor processor-memory protection when it is brought from the main memory to the local node's processor, and with our processor-processor protection when it is sent from the local node to the remote requester.

6. EXPERIMENTAL SETUP

In order to evaluate our approaches for data protection in DSM systems, we added DSM support and implemented our proposed mechanisms in SESC [10], a detailed execution-driven simulator. Table 1 shows the relevant architectural features of our simulated system. The important features of our simulated DSM system to note are the use of the MESI coherence protocol with a hypercube network and fixed-path routing protocol. Also note that the on-chip counter cache is used to store counters for use in processor-memory data encryption, not for storing counters for our schemes. In addition, we model an 80 cycle latency for the AES engine, which similar to the 37ns implementation described in [11] on a 2 GHz processor.

We assume round-robin page allocation. More optimized systems may employ first touch policy which minimizes accesses to remote memory. Note that round-robin allocation stresses our schemes more compared to an unprotected system because of the high number of remote memory accesses, and each remote memory access results in processor-processor communication. Our simulations also take into account the extra bandwidth usage due to sending counters, MACs, and other information along with data messages in our processor-processor data protection schemes.

Table 1: Architectural Parameters.

Processor	2 GHz, 3-way out-of-order issue
Memory	L1-Inst: 16KB, 2-way, 64B line, WT, 2 cycles L1-Data: 16KB, 2-way, 64B line, WT, 2 cycles L2-Unif:256KB, 8way, 64B line, WB, 10 cycles Round robin page allocation, 4KB pages Memory bus: 1 GHz, 4-Byte wide, split-trans. RT memory latency: 200 cycles (uncont.)
Proc-Mem Encryption	Counter-mode, 64-bit counters SN Cache: 32KB, 4-way, 64B line, WB GCM-based Merkle Tree authentication
Proc-Proc Encryption	Counter-mode encryption, 64-bit counters Send and receive table size depend on scheme. Authenticated-Encryption [14]
Encryption HW	1 AES unit for all encr./decr. and auth. 16 stage AES pipe: 80 cycle lat., 5 cycle occp.
Network	Hypercube, fixed-path routing Link BW 6GB/s, 50ns hop delay (as in [20])
Coherence Prot.	MESI, full bit vector, home-based directory, reply forwarding

We use the SPLASH-2 benchmark suite [22] to evaluate our techniques. The relevant application parameters are shown in Table 2, along with the *global* L2 cache miss rate of each application. Applications are simulated from start to completion without fast forwarding or sampling.

Table 2: Applications used in our evaluation. The L2 global miss rate is the number of L2 misses divided by the number of L1 cache accesses.

Application	Input	Global L2 Miss Rate
Barnes	16K particles	0.18%
Cholesky	tk25.O	0.41%
FFT	1M points	2.69%
FMM	8K particles	0.22%
LU	512x512 matrix,16x16 blocks	0.20%
Ocean	258 x 258 ocean	2.12%
Radiosity	-test	6.23%
Radix	4M keys, radix 1024	1.95%
Raytrace	car	1.03%
Volrend	head	0.78%
Water-n2	512 molecules	0.12%
Water-sp	512 molecules	0.08%

7. EVALUATION

In this section we use several different DSM configurations to evaluate the data protection and counter management schemes proposed in this paper. The *P2M Only* configuration refers to protection for data in main memory and for processor-memory communication within each DSM node, using previously proposed uniprocessor counter-mode memory encryption and Merkle Tree authentication. Note that this configuration has no protection for data communicated between nodes. The *Direct* configuration uses the AES block cipher and SHA-1 MAC generation algorithm to encrypt and authenticate processor-processor data communication. In order to not penalize the *Direct* configuration too greatly, an 80 cycle latency is assumed for both AES and SHA-1, which is optimistic compared to over 300ns in a recent hardware implementation [11]. Lastly, *Private*, *Shared*, and *CachedX*, where *X* denotes the number of send and receive table entries, represent our protection schemes described in Section 5 for processor-processor data protection. All schemes also include the mechanisms of *P2M Only* for protecting processor-memory data communications. All figures showing execution time overheads are relative to a baseline DSM system with no support for data protection.

7.1 DSM Data Protection Schemes

In order to evaluate the performance of our processor-processor communication data secrecy and integrity protection, Figure 5 compares the execution time overheads of the *P2M Only* and *Direct* configurations with our approaches, *Private*, *Shared*, and *Cached4* on a 16-processor DSM system. With these configurations, the total send and receive table size of *Shared* and *Cached4* are a half and a quarter of that of *Private*, respectively. This figure shows that adding just processor-memory data protection (*P2M Only*) results in a very modest overhead of under 5% for most benchmarks and on average. However, the straightforward approach to processor-processor data protection (*Direct*) results in an average slowdown of 15% across all applications, and more than 20% in *cholesky*, *ocean*, *radiosity*, and *volrend*, even with a very optimistic assumption on the MAC generation latency. This execution time overhead is not likely to be tolerable for real-world DSM systems, motivating the need for more efficient techniques.

As shown in the figure, each of our proposed techniques is able to reduce the overhead of processor-processor data protection significantly over *Direct*, with *Private* being the best in terms of execution time overhead at only 6%, and with *Shared* and *Cached4* performing just slightly worse despite much smaller storage overheads. The figure also shows that *Cached4* gives better performance than *Shared* in almost all applications, although it has only half the table storage overhead of *Shared*. However, on average it is only slightly better because of the pathological case in *barnes*, where *Cached4* has twice the performance overhead of *Shared*. Finally, we note that the majority of the overhead in most applications comes from processor-memory protection rather than from our processor-processor protection.

We also observe that the overheads for *ocean* and *radiosity* are significantly larger than the average, so we examine their behavior more closely. For *ocean*, most of the overhead comes from processor-memory protection, as opposed to our mechanisms. Prior studies on uniprocessors [5, 17, 21, 23, 24] showed that two main components of processor-memory overhead are counter cache misses and reduced L2 capacity due to storage of Merkle Tree nodes. Ocean's counter cache miss rate is very high (30%) and its L2 misses increase by 50% due to Merkle Tree caching. For *radiosity*, processor-memory and processor-processor protection each account for about half of the total overhead. We observe that the main cause of overhead is over-utilization of the AES unit due to frequent memory traffic (high L2 miss rate as shown in Table 2). The average time that a pad generation request is buffered until it starts to be serviced by the AES unit for *radiosity* is 126 cycles, which is 3 to 10 times higher than for other applications. One possible solution to reduce *radiosity's* overhead is to use a second or deeper-pipelined AES unit.

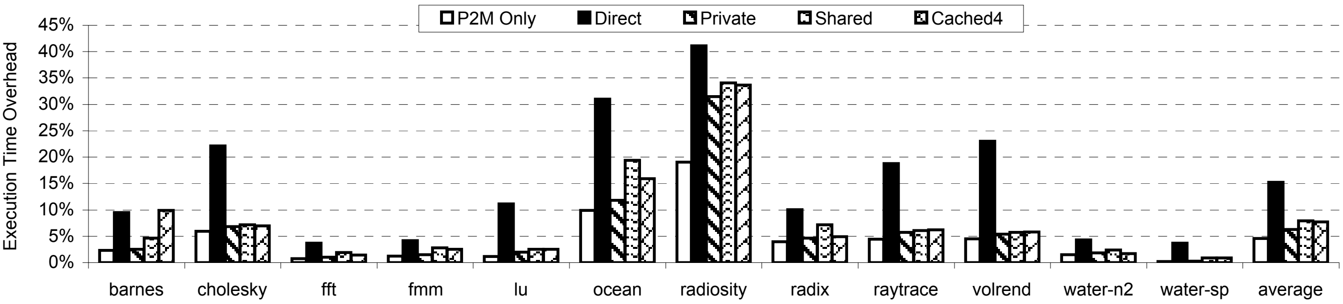


Figure 5: Execution time overheads for processor-memory data protection only, versus schemes with additional processor-processor data protection on 16 processors.

In order to gain more insight into the performance of our techniques from the previous figure, we introduce the terms *pad hit*, *pad half-miss*, and *pad miss*. A pad hit refers to the case in which the correct encryption and authentication pads have been fully pre-generated and only an XOR is needed to encrypt, decrypt, or authenticate the data. A pad half-miss refers to the case in which one or both of the correct encryption and authentication pads are in the process of being generated, so only part of the pad generation latency is hidden. Lastly, a pad miss refers to the case in which the correct pads are not currently in the table or in the AES pipeline, so they must be generated directly and none of the pad generation latency is hidden.

Figure 6 shows the average (across all benchmarks) pad hit, pad half-miss, and pad miss rates. In *Private*, the pad miss rate is nearly zero, so this scheme has the lowest execution time overheads. This is expected, because a dedicated counter and its pre-generated set of pads are kept for each processor pair in *Private*. In *Shared*, we reduce the storage overhead relative to *Private* by using one counter and pad in place of the send table. However, this results in an increased pad half-miss rate for sending data, and a relatively large miss rate in the processors' receive tables for receiving and decrypting data. This is due to the fact that a processor's decryption pads are only kept synchronized with a particular sender's encryption pad if consecutive messages from a particular sending processor go to the same receiving processor. Finally, we see that the table performance of *Cached4* is between *Private* and *Shared*. The pad half-miss rate for sending data is almost as low as in the *Private* scheme, and the pad miss rate for receiving data is significantly lower than that observed for *Shared*.

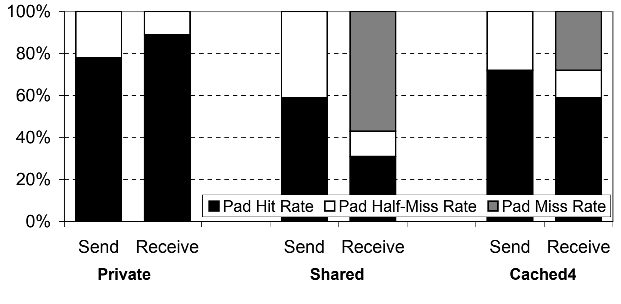


Figure 6: Send and receive table performance for our schemes on 16 processors.

7.2 Scalability of the *Cached* Scheme

Since the motivation for our *Cached* scheme was to have an approach with a small storage overhead that could scale to work well on an arbitrary number of processors, we now evaluate its effectiveness in achieving these goals. Figure 7 shows the execution time overhead of *Cached8* as the system size varies from 16 to 32 to 64 processors. As before, note that the overhead shown for each system size is relative to a DSM system of the same size, but without any data protection mechanisms. This figure shows that for most benchmarks, and on the average, the execution time overhead actually decreases as the number of processors increases, but in some cases the reverse is true. This can be explained by two opposing performance factors we observe as the system size increases. First, the size of the interconnection network size increases as the number of processors increases, and therefore data sent from one processor to another travels more link hops on average to reach its destination. This amortizes the encryption/decryption and authentication latency for sending and receiving data over a greater total network delay, and reduces the execution time overhead relative to the baseline system. The second factor is the pad miss rate at the processors' receive tables. As the system size increases, the 8 table entries cover a smaller percentage of the total number of processors. Therefore we observe increasing pad miss rates for our applications as the DSM size grows. Overall, no matter which factor dominates the total execution time overhead, this figure shows that our *Cached8* scheme indeed scales quite well to relatively large DSM systems, even with a small, fixed, table storage overhead of $\frac{2\times8\times705}{8}$ bytes, or slightly less than 1.5 KB.

To further evaluate our *Cached* scheme, we also determine how well this scheme performs at correctly caching counters and pre-generated pads of frequently communicating processor pairs, even with small table sizes and a large number of processors. Figure 8 shows the pad miss rate at the receive tables for *Cached8* with 16, 32, and 64 processors. This figure shows that our *Cached8* scheme is effective at storing the correct pre-generated pads most of the time. Even with small, 8-entry tables, the pad miss rate only increases from 15% to 26% to 31% on average for 16, 32, and 64 processor DSM systems respectively. This is promising because it indicates that even as the DSM size doubles, the pad miss rate for our tables only suffers slightly.

7.3 Sensitivity Analysis

In this section, we present several sensitivity studies to confirm that our schemes perform well for a variety of system parameters. We first evaluate our schemes under various AES latencies and occupancies. Then we evaluate the performance of our schemes with various DSM system sizes. Lastly we examine the performance of our schemes as the size of the L2 cache is varied.

Figure 9 shows the performance of *Direct*, *Private*, *Shared*, and *Cached4* averaged across all benchmarks as the AES latency and occupancy is increased from 1× (the default value) to 2× and 4×. The 4× latency corresponds to a 320-cycle AES latency and 20-cycle occupancy, which is very

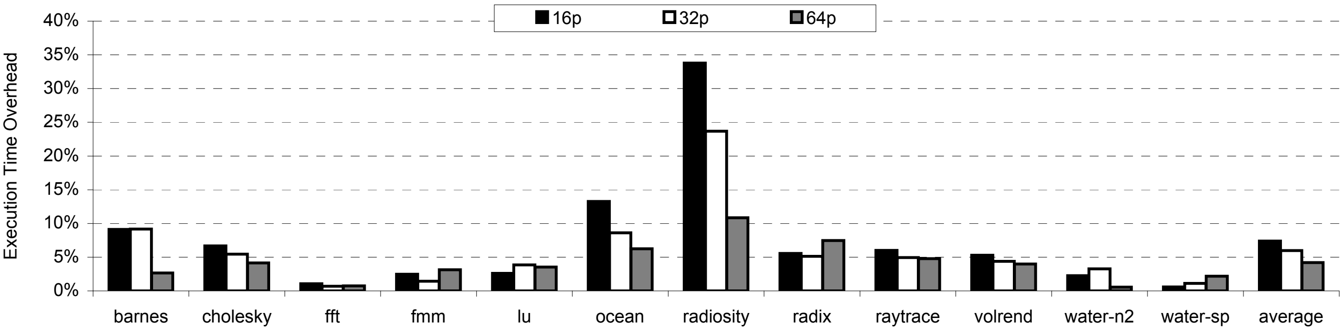


Figure 7: Execution time overheads for our *Cached8* scheme across 16, 32, and 64 processors.

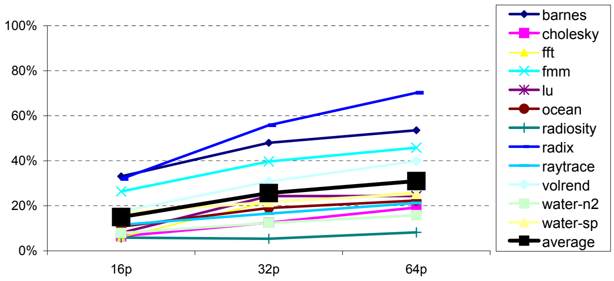


Figure 8: Pad miss rate at the receive tables for our *Cached8* scheme across 16, 32, and 64 processors.

pessimistic. This figure shows that as the AES latency and occupancy increase, the overhead of all schemes increases, especially for the 4× configuration. We investigate this further and found that most of the increase in overheads is due to the increase in occupancy of the AES unit which cannot keep up with the arrival rate of encryption and authentication requests for both processor-memory and processor-processor communication. Therefore, these additional overheads can be reduced by providing if more than one AES unit. Also note that with one AES unit the performance gap between the performance of *Direct* and our schemes becomes larger (a percentage-point difference of 7% between *Direct* and *Cached4* for 1× vs. 15% for 4×). This indicates that our latency-hiding techniques work even better with longer AES unit latencies and AES queueing latencies.

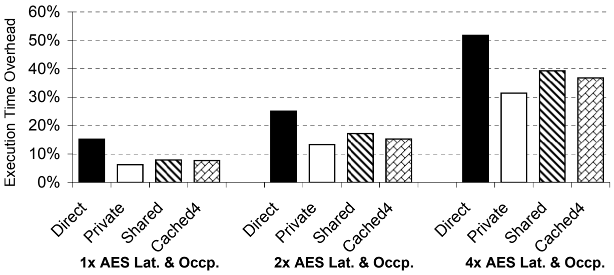


Figure 9: Execution time overheads for 1x, 2x, and 4x the base AES latency and occupancy.

Figure 10 examines the execution time overhead of *Direct*, *Private*, *Shared*, and *Cached(p/4)* (where p = number of processors) as size of the DSM system varies from 16 (the default value) to 32 to 64 processors. Again, because of the amortization of the encryption/decryption and authentication delays over more network hops in a large system, we observe that our schemes perform well on large systems. We note that even on a 64 processor system, however, the overhead of *Direct* is still almost 10%, and each of our schemes reduces this overhead by almost 50% or better.

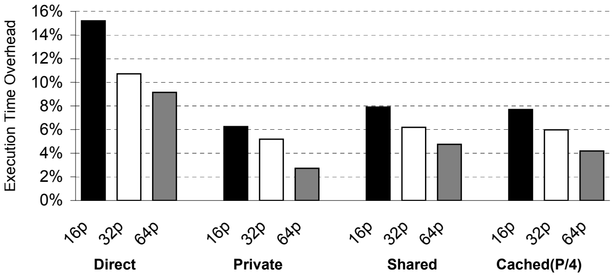


Figure 10: Execution time overheads across 16, 32, and 64 processor DSM systems.

Figure 11 validates our schemes against a number of L2 cache sizes, ranging from 256KB (the default value) to 512KB to 1MB. For the increased cache sizes of 512KB and 1MB, we adjust the L2 access latencies using the Cacti 3.0 toolkit [19]. This figure shows that in general the execution time overhead decreases as the L2 cache size increases because the L2 cache miss rate decreases and less data must be communicated between processors. This figure also validates that our schemes perform well across a variety of L2 cache sizes in a DSM system.

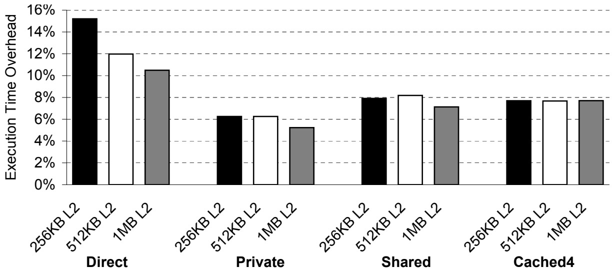


Figure 11: Execution time overheads for 256KB, 512KB, and 1MB L2 cache sizes.

8. CONCLUSIONS

While hardware memory encryption and authentication schemes have been studied in detail for uniprocessor and SMP systems recently, no such work has been done for DSM systems. This paper presented the first study of data protection with hardware memory encryption and authentication mechanisms for processor-processor communication in DSM systems. Through security analysis, we found that if coherence protocol anomalies are detected, only data messages need to be fully protected by encryption, authentication, and a mechanism against message replays. We find that all of our schemes perform well, and the performance overheads due to security protection decreases as the DSM size is increased. Through careful design, we found that our *Cached8* scheme with a fixed total storage overhead of roughly 1.5KB allows efficient protection even for large-scale DSMs. *Cached* also achieves relatively low performance overheads across different DSM sizes and L2 cache sizes. Across SPLASH-2 applications in a 16-processor DSM system, the average overhead is 6-8%, of which the majority comes from processor-memory protection rather than our processor-processor protection.

9. ACKNOWLEDGMENTS

This research was supported by NSF Early Faculty Career Awards CCF-0347425 and CCF-0447783, NSF award CCF-0429802, IBM Faculty Partnership Award, North Carolina State University and Georgia Institute of Technology. The authors would like to thank all of the anonymous reviewers for their helpful comments.

10. REFERENCES

[1] AMD. AMD Opteron Processor for Servers and Workstations. *http://www.amd.com/us-en/Processors/ ProductInformation/0,,30_8796_8804,00.html*, 2005.

[2] R. Anderson. Why cryptosystems fail. In *Proceedings of the 1st Conf. Computer and Communications Security (CCS '93)*, pages 215–227, 1993.

[3] R. K. B. Yang, S. Mishra. A high speed architecture for galois/counter mode of operation (gcm). In *Cryptology ePrint Archive: Report 2005/146*, 2005.

[4] D. Bartholomew. On Demand Computing – IT On Tap? *http://www.industryweek.com/ReadArticle.aspx? ArticleID=10303&SectionID=4*, June 2005.

[5] B. Gassend, G. Suh, D. Clarke, M. Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proc of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[6] T. Gilmont, J.-D. Legat, and J.-J. Quisquater. Enhancing the Security in the Memory Management Unit. In *Proc. of the 25th EuroMicro Conference*, 1999.

[7] A. Huang. *Hacking the Xbox: An Introduction to Reverse Engineering.* No Starch Press, San Francisco, CA, 2003.

[8] A. B. Huang. The Trusted PC: Skin-Deep Security. *IEEE Computer*, 35(10):103–105, 2002.

[9] IBM. IBM Power4 System Architecture White Paper. http://www-1.ibm.com/servers/eserver/pseries/ hardware/whitepapers/power4.html, 2002.

[10] J. Renau, et al. SESC. *http://sesc.sourceforge.net*, 2004.

[11] T. Kgil, L. Falk, and T. Mudge. ChipLock: Support for Secure Microarchitectures. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Oct. 2004.

[12] D. Lie, J. Mitchell, C. Thekkath, and M. Horowitz. Specifying and Verifying Hardware for Tamper- Resistant Software. In *IEEE Symposium on Security and Privacy*, 2003.

[13] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. MItchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[14] D. A. McGrew and J. Viega. The Galois/Counter Mode of Operation (GCM). http://csrc.nist.gov/ CryptoToolkit/modes/proposedmodes/gcm/, 2004.

[15] T. Olavsrud. HP Issues Battle Cry in High-End Unix Server Market. *ServerWatch, http://www.serverwatch.com/news/article.php/1399451*, 2000.

[16] W. Shi, H.-H. Lee, M. Ghosh, and C. Lu. Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 123–134, September 2004.

[17] W. Shi, H.-H. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[18] W. Shi, H.-H. Lee, C. Lu, and M. Ghosh. Towards the Issues in Architectural Support for Protection of Software Execution. In *Proceedings of the Workshop on Architectureal Support for Security and Anti-virus*, pages 1–10, October 2004.

[19] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. In *Technical Report WRL Technical Report 2001/2.* Compaq Western Research Laboratory, Aug 2001.

[20] Silicon Graphics, Inc. SGI Altix 3000 Data Sheet. *http://www.sgi.com/products/servers/altix*, 2004.

[21] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processor. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, 2003.

[22] S. Woo, M. Ohara, E. Torrie, J. 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.

[23] C. Yan, B. Rogers, D. Englender, Y. Solihin, and M. Prvulovic. Improving cost, performance, and security of memory encryption and authentication. In *Proc. of the International Symposium on Computer Architecture*, 2006.

[24] J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, 2003.

[25] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta. SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors. In *International Symposium on High-Performance Computer Architecture*, February 2005.