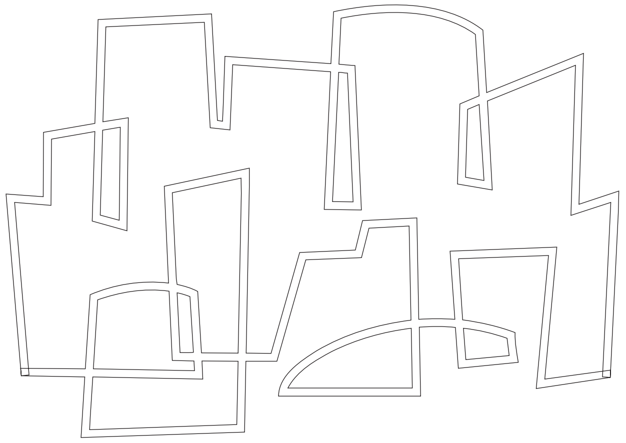
Checking the Integrity of Memory in a Snooping-Based Symmetric Multiprocessor (SMP) System

Dwaine Clarke,G. Edward Suh, Blaise Gassend,
Marten van Dijk, Srinivas Devadas

2004, July

Computation Structures Group
Memo 470



Checking the Integrity of Memory in a Snooping-Based Symmetric Multiprocessor (SMP) System

Dwaine Clarke, G. Edward Suh, Blaise Gassend,
Marten van Dijk[*] and Srinivas Devadas

MIT Computer Science and Artificial Intelligence Laboratory
{declarke, suh, gassend, marten, devadas}@mit.edu

July 26th, 2004

Abstract

We study schemes to implement memory integrity checking in a symmetric multiprocessor (SMP) system. We focus on a snooping-based SMP system, where the processors are connected by a single bus. The problem has two parts, checking the integrity of the cache coherence protocol, and checking the integrity of off-chip memory. To check the integrity of the cache coherence protocol, we introduce schemes to authenticate the bus. To check the integrity of off-chip memory, we extended existing uniprocessor memory integrity checking schemes to an SMP system.

Previous work on memory integrity checking [GSC$^+$03, CDvD$^+$03, SCG$^+$03b] described checking the integrity of memory on a uniprocessor. This scenario is simpler for two reasons. In a uniprocessor system, because there is just one processor, there is no cache coherence protocol that needs to be checked. Also, with respect to the off-chip RAM, a uniprocessor system just has to determine whether the value that it has read from an address is the same value that it had last written to that address. An SMP system requires that each processor, when it reads a value, be able to distinguish whether that value was written by one of the other processors as opposed to being written by an adversary, as well as determine the freshness of the value.

1 Introduction

Secure processors [SCG$^+$03a, LTM$^+$00, CJPL02, SW99, Yee94] can enable many useful applications such as copy-proof software, Digital Rights Management, and certifying the execution of a program. In order to build secure computing systems, one of the problems that must be solved is checking the integrity of untrusted external memory. The memory is typically accessed via an external memory bus. While the secure processors are running, an adversary may corrupt the memory values in the external memory. The secure processors should be able to detect any form of physical or software corruption of the memory. (Typically, the secure processors will stop the execution of running process(es) when they detect memory corruption. In this paper, we are not concerned with the recovery of corrupted data. An adversary may be able to destroy the entire contents of memory causing an unrecoverable error. Thus, the secure processors we consider do not protect against denial-of-service attacks.)

[*]Visiting researcher from Philips Research, Prof Holstlaan 4, Eindhoven, The Netherlands.

In this paper, we describe hardware schemes to cryptographically check the integrity of a program's operations on untrusted external memory in a snooping-based symmetric multiprocessor (SMP) system. In an SMP system, multiple processors operate in parallel and share a single address space. Any processor is able to access any memory address using store and load operations. Each processor takes the same time to access the external memory. The problem of checking the integrity of memory in an SMP system has two components: (1) checking the integrity of the cache coherence protocol, (2) checking the integrity of the operations on the external memory. This paper focuses on a snooping-based SMP system, where the processors are connected by a single bus.

To check the integrity of the cache coherence protocol, the bus is authenticated. We present a bus authentication protocol in which a sequence of bus transactions is checked. The bus authentication protocol provides an authentic communication layer over which any cache coherence protocol can be implemented.

For checking the integrity of operations on external memory, we present a hash tree-based approach and a log hash-based approach. Each approach enables the processors to check the integrity of an arbitrarily-large external memory using small fixed-sized trusted on-chip storages. The hash tree-based approach checks, after each load operation, whether the external memory was behaving correctly, whereas the log-based approach checks, after a sequence of memory operations, whether the external memory was behaving correctly.

The outline of the paper is as follows. Section 2 describes related work, and the main contributions of this work. In Section 3, we describe the model. Section 4 gives an example of a popular cache coherence protocol; this example will be used as a reference throughout the paper. Though the work references this cache coherence protocol, other protocols can be adapted and used with our schemes. In Section 5, we describe the bus authentication protocol. In Section 6, we describe the approaches to checking external untrusted memory. Section 7 examines performance. Section 8 gives a description of how to extend the work to a hybrid hash tree-log hash SMP memory integrity checker, which can have performance benefits over both the hash tree and log hash checkers. Section 9 concludes the paper.

2 Related Work and Main Contributions

Previous work on memory integrity checking was done for the uniprocessor case [GSC+03, CDvD+03, SCG+03b]. This scenario is simpler for two reasons. In a uniprocessor system, because there is just one processor, there is no cache coherence protocol that needs to be checked. Also, with respect to the off-chip RAM, a uniprocessor system just has to determine whether the value that it has read from an address is the same value that it had last written to that address. An SMP system requires that each processor, when it reads a value, be able to distinguish whether that value was written by one of the other processors as opposed to being written by an adversary, as well as determine the freshness of the value. The main contributions of this work are the introduction of a bus authentication scheme, and the extension of the uniprocessor memory integrity checking algorithms to an SMP system.

3 Model

Figure 1 illustrates the model we use. Each checker keeps and maintains some small, fixed-sized, trusted state. The untrusted RAM (external memory/main memory) is shared, untrusted and arbitrarily large. The finite state machines (FSMs) and untrusted RAM are connected by a single bus. The FSMs share a single address space and each FSM is able to access any memory address

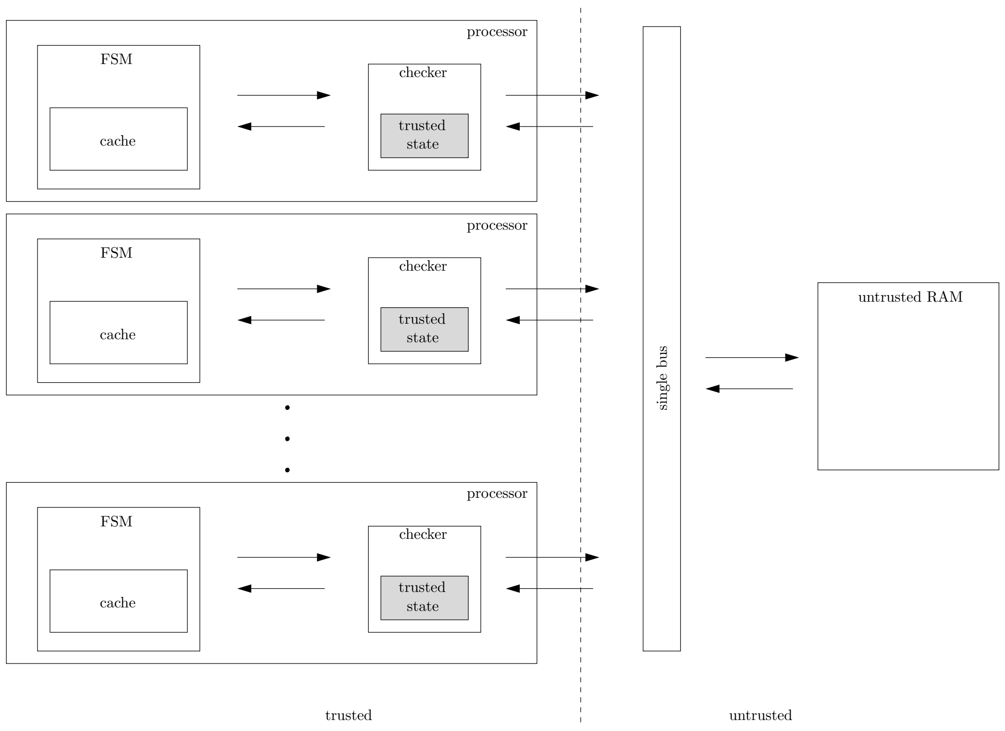


Figure 1: Model

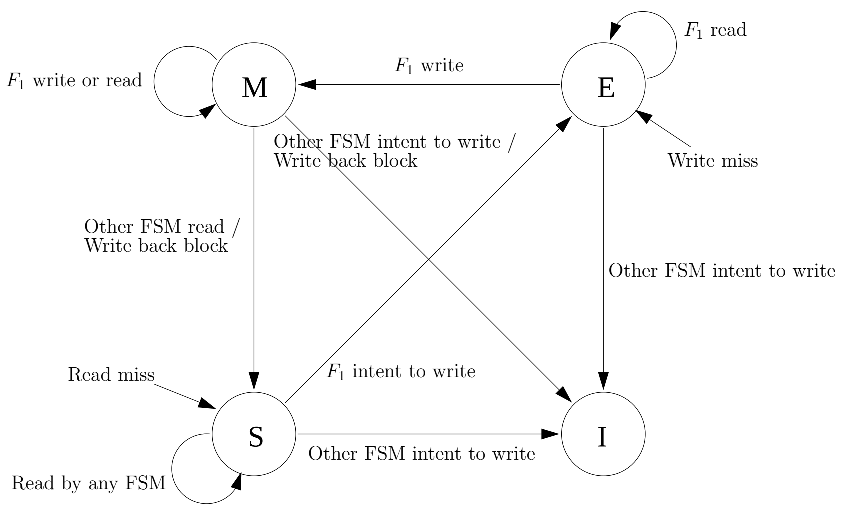
using store and load operations. Each FSM also maintains a fixed-sized *cache*. The cache is initially empty, and can store data frequently accessed by the FSM.

Each finite state machine (FSM) generates messages, including store and load operations, and messages particular to the cache coherence protocol. The checker associated with the FSM updates its trusted state on each of these messages. The checkers use their trusted state to (1) check the authenticity of the bus, and (2) check the integrity of the untrusted RAM. Data that is loaded into the cache is checked by the checker and the FSM can trust the data once it is loaded into the cache.

The trusted computing base (TCB) consists of the FSMs and their caches, and the checkers and their trusted states. In the model, each FSM is an unmodified processor running a user program. Each checker is special hardware that is added to the processor to check the integrity of memory. Thus, the TCB consists of all of the processors. The untrusted RAM is off-chip memory. The bus is also off-chip.

To check the integrity of memory in an SMP system, both the integrity of the cache coherence protocol and the integrity of the untrusted RAM must be checked. Both of these parts must be checked before the FSMs export results to other programs or users.

To check the integrity of the cache coherence protocol, the bus is authenticated. *An authentic bus means that each checker sees the same messages in the same sequence.* Each checker records all of the messages it sees on the bus, including the messages it sends, and the sequence in which it saw the messages. To check the authenticity of the bus, the checkers securely exchange their records and check that all of the records are the same. Because each checker also records each message it sends, and because each checker also includes its FSM id in each message it sends, checking the authenticity of the bus also includes checking the integrity and sender of each checker message. All other messages are treated as if they are coming from the untrusted RAM, and, in this case, bus



F_1 read
F_1 write or read
F_1 write
M
E
Other FSM intent to write / Write back block
Write miss
Other FSM read / Write back block
Other FSM intent to write
F_1 intent to write
Read miss
S
I
Other FSM intent to write
Read by any FSM

Figure 2: MESI Cache Coherence Protocol

authentication just means checking that each checker saw the same messages in the same sequence (the integrity of these messages will be checked by the scheme which checks the integrity of the untrusted RAM). The idea of bus authentication is to provide an authentic communication *layer* over which any cache coherence protocol can be implemented. In our model, the bus is assumed to be actively controlled by an adversary. The adversary may alter the bus transmissions or insert its own transmissions.

To check the integrity of the untrusted RAM, the checkers check that the untrusted RAM behaves like valid RAM. *RAM behaves like valid RAM if the data value that a checker reads from a particular address is the same data value that has most recently been written to that address by a checker.* In our model, the untrusted RAM is assumed to be actively controlled by an adversary. The untrusted RAM may not behave like valid RAM if the adversary somehow alters it. In checking the integrity of the untrusted RAM, a simple solution such as calculating a message authentication code (MAC) of the data value and address, writing the (data value, MAC) pair to the address, and using the MAC to check the data value on each read, does not work. The approach does not prevent replay attacks: an adversary can replace the (data value, MAC) pair currently at an address with a different pair that was previously written to the address. In the approaches described in this paper, the checkers update and use their fixed-sized trusted states to help check the freshness of the data values they read.

4 MESI Cache Coherence Protocol

This section gives an overview of a popular cache coherence protocol, the MESI protocol [HP97, Asa04]. We will refer to this protocol throughout the paper.

In an SMP system, the same address can be found in the caches of different FSMs because different FSMs can read or write to the same address. Because of this, a cache coherence protocol is needed to help ensure that the data value that an FSM reads from an address in its cache is the same data value that had most recently been written to that address by an FSM. Without a cache coherence protocol, an FSM might read a stale value when it reads from an address in its cache because the latest value may be in the cache of another FSM.

In the MESI cache coherence protocol, each cache block can be in one of four states:

M Modified Exclusive. This cache is the only cache with a copy of the block, and the block is dirty.

E Exclusive, unmodified. This cache is the only cache with a copy of the block, and the block is clean.

S Shared. A copy of this block may be in at least one other cache.

I Invalid. This cache block is invalid.

The state transition diagram for a single cache block of an FSM, F_1, is shown in Figure 2.

Each cache is a write-back cache. This means that when a write occurs, the new value is written only to the block in the cache. The modified block is written to the untrusted RAM when it is replaced in the cache.

State transitions in the cache coherence protocol are solely dependent on messages from the FSMs. (As described in Section 3, the checkers authenticate, via bus authentication, the senders, integrity, and sequences of all FSM messages.) Transitions in the state of a cache block occur when an FSM misses in its cache when it is reading an address (read miss), misses in its cache when it is writing to an address (write miss), or hits its cache when it is writing to an address (write hit). When the FSM hits in its cache when it reads an address (read hit), the cache block's state does not change.

On a read miss, the FSM writes back the old block to the untrusted RAM if it is in state M and makes a request for the new block, using the new block's address. All of the other FSMs see this request. If one of them has the block in its cache in state M, it aborts the request, changes its block's state in its cache to state S, and writes its block back to the untrusted RAM. The FSM which requested the block will see the write back. It will read the block as it is being written back, and put it into its cache in state S. If the block is not in state M in any of the caches, the FSM which requested the block will end up reading the block from the untrusted RAM and will put it into its cache in state S.

The protocol treats a write hit to a shared block the same way as it treats a write miss. Thus, on a write miss or a write hit to a shared block, the FSM writes back the old block to the untrusted RAM if it is in state M and places a write miss on the bus, again using the new block's address. All of the FSMs with the block in their caches see the write miss and invalidate the block by changing the block's state in their caches to state I. The block is changed to state E in the cache of the FSM performing the write, and when the write is performed, the FSM then changes the block's state to state M. Because an FSM invalidates copies of a block in other FSM caches when it wants to perform a write, the MESI protocol is a *write-invalidate* cache coherence protocol [HP97].

Any valid cache block is either in state S in one or more caches, or in one of the exclusive states, state M or E, in exactly one cache. The essence of the protocol is that a transition to one of the exclusive states is required before an FSM can write to a block. If another FSM subsequently wants to read the block, it incurs a read miss; the cache with the modified exclusive copy of the block generates a write back to supply the block, and changes its cache block's state to state S, which will force a later write to require exclusive ownership again.

5 Bus Authentication Protocol

For the bus authentication protocol, each checker's fixed-sized state consists of one running hash, RHash; a counter, MAC-based-Counter; a secret key; and optionally, a second counter, Recording-Counter. All of the checkers' secret keys are the same. The protocol has two stages. During runtime, each checker uses the recording operations in Section 5.1 to record all of the messages it sees on the bus, including the messages it sends, and the sequence in which it saw the messages. To check the authenticity of the bus, the checkers use the MAC-based operations in Section 5.2 to securely exchange their records and check that all of the records are the same. If all of the records are the same, the bus is authentic (c.f. Section 3)[1].

5.1 Recording Stage

Figure 3 shows the `recording-send-recording-receive` interface each FSM calls to use the checker to send and receive messages in the recording stage. The running hash, RHash, can implemented using a standard hash function, like SHA-1 [NIS95], or an incremental hash function like XOR-MAC [BGR95]. In either case, the update operation (+=) is very efficient.

Each checker updates its running hash with either the message it sent or a message it received. Depending on whether it is implemented with a standard hash function like SHA-1, or with an incremental hash function like XOR-MAC, the running hash will implicitly or explicitly record the sequence of the strings with which it is updated. If the implementation of the running hash already records the sequence of the strings with which it is updated, such as when it is implemented with SHA-1, the Recording-Counter can be removed from the protocol. Figure 4 shows the `recording-send-recording-receive` interface each FSM calls to use the checker to send and receive messages in the recording stage in this case.

If the sequence of the messages is changed, the running hashes will not be the same when a check is performed. Because each checker records each message it sends, and because each checker also includes its FSM id in each message it sends, if an adversary tampers with the integrity or sender of one of the checker's messages, the running hashes will also not be the same. (All other messages are treated as if they are coming from the untrusted RAM. An adversary may try to tamper with the integrity of these messages, or insert its own messages. The schemes which check the integrity of the untrusted RAM check these messages, and will detect these attacks.)

5.2 MAC-based Stage

Figure 5 shows the `MAC-based-send-MAC-based-receive` interface each FSM calls to use the checker to send and receive messages in the MAC-based stage. To check the sequence of messages, the checkers communicate their running hashes to each other, using the operations in this stage. If all of the checkers' running hashes are the same, the bus is authentic (c.f. Section 3). In this stage, each checker uses its MAC-based-Counter and its secret key. All of the checkers secret keys are the same, and are used to create message authentication codes.

When either the MAC-based-Counter, or, if it is used, the Recording-Counter, in any of the checkers reaches its maximum value or a value near to its maximum value, the secret key must be changed on all of the checkers. Standard key agreement protocols for secret keys can be used to accomplish this. For example, one leader checker can pick a new secret key, then broadcast it encrypted and MAC'd on the bus; that is, the leader checker performs `MAC-based-send`(encrypted key). The other checkers would perform an `MAC-based-receive` operation to receive the message, and decrypt the encrypted key to get the new key. The counters are then reset, and the new key used to create further MACs[2].

[1]The protocol as we describe it authenticates a sequence of bus messages. We do note that the operations in Section 5.2 can form a separate bus authentication protocol by themselves, which can authenticate each individual bus message; while this is a valid, secure bus authentication protocol, it is likely to consume much more bandwidth, and thus be less efficient, in practice.

[2]The initial establishment of the same secret key in each checker is a bootstrapping issue. One approach is to

In this stage of the bus authentication protocol, the fixed-size state each checker uses is:

- 1 running hash: RHash. Initially RHash is 0.
- 1 counter: Recording-Counter. Initially Recording-Counter is 0.

```
recording-send($msg$)
    checker $C_i$ sends $msg$ on the bus:
        1. $C_i$ sends $(i, msg)$ on the bus.
        2. Let $c$ be the value of of $C_i$'s Recording-Counter.
        3. $C_i$ updates its RHash with a hash of $(i, msg, c)$, i.e., RHash += $(i, msg, c)$.
        4. $C_i$ increments $C_i$'s Recording-Counter.

recording-receive($j, msg$)
    $C_i$ receives $C_j$'s message, $i \neq j$, from the bus:
        1. $C_i$ receives $(j, msg)$ from the bus. If $(i \neq j)$, continue; else ignore the message.
        2. Let $c$ be the value of $C_i$'s Recording-Counter.
        3. $C_i$ updates its RHash i.e. RHash += $(j, msg, c)$.
        4. $C_i$ increments $C_i$'s Recording-Counter.
```

Figure 3: `send` and `receive` operations in the recording stage of the bus authentication protocol

In this stage of the bus authentication protocol, the fixed-size state each checker uses is:

- 1 running hash: RHash. Initially RHash is 0.

```
recording-send($msg$)
    checker $C_i$ sends $msg$ on the bus:
        1. $C_i$ sends $(i, msg)$ on the bus.
        2. $C_i$ updates its RHash with a hash of $(i, msg)$, i.e., RHash += $(i, msg)$.

recording-receive($j, msg$)
    $C_i$ receives $C_j$'s message, $i \neq j$, from the bus:
        1. $C_i$ receives $(j, msg)$ from the bus. If $(i \neq j)$, continue; else ignore the message.
        2. $C_i$ updates its RHash i.e. RHash += $(j, msg)$.
```

Figure 4: `send` and `receive` operations in the recording stage of the bus authentication protocol when the implementation of RHash records the sequence of strings with which it is updated.

In this stage of the bus authentication protocol, the fixed-size state each checker uses is:

- 1 secret key.
- 1 counter: MAC-BASED-COUNTER. Initially MAC-BASED-COUNTER is 0.

```
MAC-based-send($msg$)
    checker $C_i$ sends $msg$ on the bus:
        1. Let $c$ be the value of $C_i$'s MAC-BASED-COUNTER.
        2. $C_i$ sends $((i, msg, c), MAC(i, msg, c))$ on the bus.
        3. $C_i$ waits for acknowledgements from each of the other checkers.
        4. After receiving all of the acknowledgements, $C_i$ increments $C_i$'s MAC-BASED-COUNTER.

MAC-based-receive($(j, msg, c), MAC(j, msg, c)$)
    $C_i$ receives $C_j$'s message, $i \neq j$, from the bus:
        1. $C_i$ receives $((j, msg, c), MAC(j, msg, c))$ from the bus. If $(i \neq j)$, continue; else ignore the message.
        2. $C_i$ checks MAC.
        3. $C_i$ checks that $c$ is the same as the value of $C_i$'s MAC-BASED-COUNTER.
        4. If all of the checks pass, accept the message.
        5. $C_i$ increments $C_i$'s MAC-BASED-COUNTER.
        6. $C_i$ sends an acknowledgment, $((i, c, \text{ack}), MAC(i, c, \text{ack}))$, on the bus.
```

Figure 5: `send` and `receive` operations in the MAC-based bus-authentication protocol

In the protocol, the MAC is used to certify that the message originated from a checker and that the message was not tampered with. The nonces are used to certify that the messages are accepted in the same sequence that they were sent on the bus. The acknowledgments ensure that the bus has not been partitioned so that only some of the checkers receive a message [3] [4].

6 External Shared Memory

The protocol in Section 5 is used to check the authenticity of the bus (c.f. Section 3). In this section, we assume that the bus is authentic, and describe schemes to check the integrity of operations on the untrusted RAM.

To check the integrity of operations on the untrusted RAM, we adapt the hash-tree based and log-based uniprocessor algorithms in [GSC$^+$03, CDvD$^+$03, SCG$^+$03b] to a snooping-based SMP system. These algorithms check that untrusted RAM has been behaving like valid RAM

have the manufacturer of the SMP system put the same secret key in each checker. Another approach is for the manufacturer to put a public-private key pair in each checker; the manufacturer also puts a certificate for the checker's public key, and the manufacturer's public key in each checker. The checkers can then use a public-key protocol to dynamically establish a shared secret when the FSMs are first started [KPS95].

[3] In the acknowledgment, a checker only needs to acknowledge the nonce because, with a bus, there can only be one message per nonce.

[4] An implementation should be able to distinguish between a *msg* and an acknowledgement. For instance, if necessary, a *msg* could be prepended with a 0, and an acknowledgement prepended with a 1.

(c.f. Section 3).

6.1 Hash Tree Approach

Section 6.1.1 gives a brief description of our previous work in implementing the hash tree approach in a uniprocessor. For clarity, we then examine, in Section 6.1.2, how to extend the approach to SMP in the case where the FSMs do not have caches. Section 6.1.3 then describes how to extend the approach to SMP in the case where the FSMs do have caches.

6.1.1 Uniprocessor

In the hash-tree based approach on a uniprocessor, a tree of hashes is maintained over the data, and the root of the tree is kept in an FSM. On each FSM store, the checker updates the path from the data leaf to the root. On each FSM load, the checker checks the path from the data leaf to the root before the FSM treats the data as valid. The hash tree is used to check, after *each* load operation, whether the untrusted RAM performed correctly.

In [GSC+03], we demonstrated how to implement the hash tree scheme in hardware in a uniprocessor and properly place the hash tree checking and generation in the memory hierarchy to ensure good performance. If, naïvely, the cache just contains data, the logarithmic overhead of using the hash tree can be significant, as large as 10x. The performance overhead can be dramatically reduced by caching the internal hash nodes on-chip with regular data. The FSM trusts data leaves and hashes stored in the cache, and can perform memory accesses directly on them without any hashing. Therefore, instead of checking the entire path from the data leaf to the root of the tree, the checker checks the path from the leaf to the first hash it finds in the cache. This hash is trusted and the checker can stop checking. When a data leaf or hash is evicted from the cache, the checker brings its parent into the cache (if it is not already there), and updates the parent in the cache. With this optimization, the overhead of using the tree can be reduced to, generally, less than 25%.

6.1.2 SMP without caches

On an SMP system, one tree may be used to protect the untrusted RAM with its root managed by one checker in one FSM. This checker monitors all of the writes and loads to the untrusted RAM by all of the checkers and checks the operations using its root. The workload of this approach can be distributed if the untrusted RAM is segmented, with each segment maintained by a separate tree managed by a different checker in a different FSM. The appropriate checker monitors for write and load operations on addresses under the protection of the tree it maintains, and updates/checks its tree accordingly. In this approach, to maintain the property that a hash tree can check, after each load operation, whether the untrusted RAM was behaving like valid RAM, the checker that checked a particular load operation would have to send a message on the bus and the authenticity of the bus would have to be checked every time it was not the checker's own FSM that was performing the load operation. The message would be to inform the FSM that was using the value on whether or not the value is correct. These messages and the frequent bus authentication checks would consume much bus bandwidth. In this approach, if the guarantee that the hash tree checks each load operation is relaxed such that the hash tree checks a sequence of operations, an approach that uses less bandwidth can be used: when a checker wants to check a sequence of its FSM's operations, it can send a message on the bus asking if there have been any load operations whose check has failed; the checker(s) responsible for checking the untrusted RAM would then reply on the bus informing it as to whether any checks have failed. The authenticity of the bus would then be checked. This is more practical because notification messages are only sent, and the authenticity of the bus only checked, when a checker needs to check a sequence of operations, reducing the bandwidth consumption.

Another approach for using a hash tree to check memory integrity is for each checker to maintain its own tree over the untrusted RAM. Each checker has its own root hash which it uses to check the integrity of the untrusted RAM. Whenever any FSM performs a store, all of the checkers update their root hashes. However, the checker only needs to check load operations when its own FSM performs loads. In this approach, a checker can determine, on each load, whether the untrusted RAM has been behaving like valid RAM without using extra bus messages.

6.1.3 SMP with caches

The same basic uniprocessor caching strategy can be used with the cache coherence protocols to ensure good performance of the hash tree scheme in an SMP system. In the case there is one checker which checks the untrusted RAM, or a part of the untrusted RAM, whenever any FSM brings data into one of the caches from the untrusted RAM, the checker responsible for checking the data checks it using its tree and cache; the data is stored in the cache of the FSM that requested it and the hashes that were used to check the data are stored in the cache of the checker that checked the data. When a data or hash block is evicted from a cache, the appropriate checker brings the parent into its cache (if it not already there) and updates the parent in the cache.

In the case that each checker maintains a tree over the untrusted RAM, each checker will be storing hashes in its cache. The cache coherence protocol will apply to these hashes just like it applies to data values. For instance, if a particular hash is stored in multiple caches and a checker wants to update it as its FSM is performing a store operation, it requests exclusive ownership of the hash and invalidates the copies in the other caches.

6.2 Log-based Approach

Again, section 6.2.1 gives a brief description of our previous work in implementing the log-based approach in a uniprocessor. For clarity, we then examine, in Section 6.2.2, how to extend the approach to SMP in the case where the FSMs do not have caches. Section 6.2.3 then describes how to extend the approach to SMP in the case where the FSMs do have caches.

6.2.1 Uniprocessor

Figure 6 shows the basic `put` and `take` operations that are used internally in the checker. Figure 7 shows the interface the FSM calls to use the checker to check the integrity of the untrusted RAM.

In Figure 6, the checker maintains two multiset hashes and a counter. In the untrusted RAM, each data value is accompanied by a time stamp. Each time the checker performs a `put` operation, it appends the current value of the counter (a time stamp) to the data value, and writes the (data value, time stamp) pair to the untrusted RAM. When the checker performs a `take` operation, it reads the pair stored at an address, and, if necessary, updates the counter so that it is strictly greater than the time stamp that was read. The multiset hashes are updated ($+_{\mathcal{H}}$) with (a, v, t) triples corresponding to the pairs written or read from the untrusted RAM.

Figure 7 shows how the checker implements the `store-load` interface. To initialize the untrusted RAM, the checker `puts` an initial value to each address. When the FSM performs a `store` operation, the checker `gets` the original value at the address, then `puts` the new value to the address. When the FSM performs a `load` operation, the checker `gets` the original value at the address and returns this value to the FSM; it then `puts` the same value back to the address. To check the integrity of the untrusted RAM at the end of a sequence of FSM stores and loads, the checker `gets` the value at each address, then compares WriteHash and ReadHash. If WriteHash is equal to ReadHash, the checker concludes that the RAM has been behaving correctly.

Intermediate `checks` can also be performed with a slightly modified `check` operation [CGS+02]. First, the checkers create a new Timer and WriteHash. Then, as the untrusted RAM is read to perform the `check`, `put` operations are performed to reset the time stamps in untrusted RAM and the new WriteHash.

Because the checker checks that WriteHash is equal to ReadHash, substitution (the RAM returns a value that is never written to it) and replay (the RAM returns a stale value instead of the one that is most recently written) attacks on the RAM are prevented. The purpose of the time stamps is to prevent reordering attacks in which RAM returns a value that has not yet been written so that it can subsequently return stale data. Suppose we consider the `put` and `take` operations that occur on a particular address as occurring on a timeline. Line 3 in the `take` operation ensures that, for each `store` and `load` operation, each write has a time stamp that is strictly greater than all of the time stamps previously read from the untrusted RAM. Therefore, the first time an adversary tampers with a particular (data value, time stamp) pair that is read from the untrusted RAM, there will not be an entry in the WriteHash matching the adversary's entry in the ReadHash, and that entry will not be added to the WriteHash at a later time. The proof that this scheme is secure is in [CDvD+03].

In [SCG+03b], we demonstrated how to implement the log-based scheme in hardware in a uniprocessor to ensure good performance. The cache contains just blocks of data values alone, while the untrusted RAM contains (block, time stamp) pairs. When the cache evicts a block, the checker `puts` the (address, block) pair into the untrusted RAM (if the block is clean, only the time stamp needs to be written back to the untrusted RAM). When the cache brings in a block from the untrusted RAM, the checker `takes` (address) from the untrusted RAM. The `check` operation iterates through the addresses. For each address, if it is in the cache, no action is taken for the address. If the address's block is not in the cache, the (address) is `taken` from the untrusted RAM (and the corresponding (address, block) pair `put` back into the untrusted RAM if the `check` is an intermediate `check`). If PutHash is equal to TakeHash at the end, the `check` is successful.

In essence, in the log-based approach, each checker uses multiset hash functions [CDvD+03] to efficiently log the minimum information necessary to check operations on untrusted RAM. Because only the minimum information is collected at program runtime, the performance overhead of the scheme can be very small. If sequences of operations are checked the log-based scheme will perform better than the hash tree scheme; if checks are frequent, hash trees will perform better [SCG+03b].

6.2.2 SMP without caches

For an SMP system, we will not examine the case where just one checker maintains a (WriteHash, ReadHash, Timer) triple. This approach will not be very efficient when we add caching to the system in Section 6.2.3 because it will require that, instead of just throwing away clean blocks when they are evicted from the cache, clean blocks would have to be transmitted on the bus so that the checker which checks the untrusted RAM can properly update its trusted state. This would consume too much extra bus bandwidth, and will not be as efficient as the approach where each checker maintains its own triple.

Therefore, for the log-hash scheme in an SMP system, each checker maintains its own (WriteHash, ReadHash, Timer) triple. We will first describe how the operations in Figures 6 and 7 are modified to an SMP system, then describe why the modified scheme is secure.

Each checker updates its own WriteHash and ReadHash on `put` and `take` operations. The `put` operation is the same as in Figure 6, but the `take` operation is modified. Figure 8 shows the

The checker's fixed-sized state is:

- 2 multiset hashes: WriteHash and ReadHash. Initially both hashes are 0.
- 1 counter: Timer. Initially Timer is 0.

```
put(a, v)
    writes a value v to address a in the untrusted RAM:
        1. Let t be the current value of Timer. Write (v, t) to a in the untrusted RAM.
        2. Update WriteHash: WriteHash $+_{\mathcal{H}}=$ hash(a, v, t).

take(a)
    reads the value at address a in the untrusted RAM:
        1. Read (v, t) from a in the untrusted RAM.
        2. Update ReadHash: ReadHash $+_{\mathcal{H}}=$ hash(a, v, t).
        3. Timer = max(Timer, t + 1).
```

Figure 6: `put` and `take` operations for a uniprocessor

```
initialize() initializes RAM.
        1. put(a, 0) for each address a.

store(a, v) stores v at address a.
        1. take(a).
        2. put(a, v).

load(a) loads the data value at address a.
        1. v = take(a). Return v to the caller.
        2. put(a, v).

check() checks if the RAM has behaved correctly (at the end of operation).
        1. take(a) for each address a.
        2. If WriteHash is equal to ReadHash, return true.
```

Figure 7: Log-based integrity checking of untrusted RAM for a uniprocessor

new `take` operation. The principal difference is in step 3 of the `take` operation: all of the checkers snoop the time stamp on the bus when one of the checkers performs a `take` operation, and they all update their TIMERS accordingly.

With respect to the `store-load` interface, the `initialize`, `store`, and `load` operations are the same as in Figure 7. The `check` operation is modified. Figure 9 shows the new `check` operation. Each of the checkers transmits its WRITEHASHES and READHASHES on the bus (as with all other checker bus transmissions, these transmissions are authenticated, c.f. Section 3). Each checker will see all of the transmissions of the hashes. Each checker can thus sum all of the WRITEHASHES and all of the READHASHES. The summation of the hashes is a hash of the union of the multisets represented by each hash. If the summed WRITEHASH is equal to the summed READHASH, then the checker knows that the RAM was behaving like valid RAM.

The operations of reading the untrusted RAM on a `check` operation can be distributed among different checkers. For instance, if the untrusted RAM has n banks, n checkers can read the untrusted RAM in parallel, each reading $1/n$-th of the untrusted RAM. Each of these checkers would update their own READHASHES when they are reading the untrusted RAM. A similar parallelization optimization also works when writing to the untrusted RAM to initializing it or re-initialize after an intermediate `check`. The benefit of the parallelization is that checking periods can be n times as frequent for the same performance overhead.

The reason that all of checkers snoop the time stamp on the bus and update their TIMERS accordingly whenever one of the checkers performs a `take` operation is to keep the TIMERS synchronized on all of the checkers. The bus authentication protocol ensures that each checker sees the same messages from the untrusted RAM in the same sequence. All of the checkers see the same time stamp value; otherwise, the bus authentication check will fail. The security of this SMP log-hash scheme thus reduces to the security of the original uniprocessor scheme because there is, in effect, only one global TIMER among all of the checkers[5].

Because the WRITEHASHES and READHASHES are maintained only by checkers, the `check` is able to distinguish whether a value was written by one of the checkers or was tampered with by an adversary. Each of the messages from the untrusted RAM is tagged with the id of the FSM for which the message is intended. If an adversary inserts its own message, the checker of the FSM whose id is in the message will record the message in its READHASH; however, when the `check` operation is performed, there will not be a corresponding entry in any of the WRITEHASHES, and the `check` operation will not pass. Thus, if the `check` operation passes, then the data value that a checker reads from a particular address is the same data value that had most recently been written to that address by a checker (i.e. if the `check` operation passes, then the untrusted RAM has been behaving like valid RAM, c.f. Section 3).

6.2.3 SMP with caching

Care must be taken when applying caching to the log hash scheme in an SMP system. The issue is determining when a `take` from the untrusted RAM or a `put` to the untrusted RAM should occur. When bringing a block into a cache, the cache's checker should `take` the address from the untrusted RAM only if none of the other caches has the cache block; otherwise the cache should just read the block from a cache that has the block. When a dirty block is evicted from a cache, because the cache is the only cache with the block, the cache's checker should `put` the (address, block) pair in the untrusted RAM. When a clean block is evicted from a cache, if the block is in none of the

[5]As a note, instead of all of the checkers snooping the time stamp on the bus whenever one of the checkers performs a `take` operation, the checker that is performing the `take` operation could broadcast the new value of its TIMER on the bus to keep all of the TIMERS synchronized; this would require extra bus messages, however.

```
take(a)
    reads the value at address $a$ in the untrusted RAM:

        1. Read $(v, t)$ from $a$ in the untrusted RAM.
        2. Update ReadHash: ReadHash $+_{\mathcal{H}}=$ hash$(a, v, t)$.
        3. All of the checkers perform Timer $=$ max(Timer, $t + 1$).
```

Figure 8: `take` operation in an SMP system

```
check()  checks if the RAM has behaved correctly (at the end of operation).

        1. take(a) for each address $a$.
        2. Each of the checkers transmit its WriteHashes and ReadHashes on the bus. Each checker
           will see all of the transmissions of the hashes. Each checker can thus sum all of the Write-
           Hashes and ReadHashes.
        3. If the summed WriteHash is equal to the summed ReadHash, return true.
```

Figure 9: `check` operation in an SMP system

other caches, the (address, block) pair should be `put` in the untrusted RAM (only the time stamp needs to be written to the untrusted RAM); otherwise, the cache should not write to the untrusted RAM as one of the other caches has the block. These modifications are necessary to ensure correct operation of the log hash scheme, i.e. if these modifications are not made, the `check` operation may fail, even if the RAM has been behaving like valid RAM. As in the previous sections, if the `check` operations passes, then the RAM has been behaving like valid RAM.

The cache coherence protocol in Section 4 needs to be adapted. As before, transitions in the state of a cache block occur on a read miss, write miss, and write hit. We describe the changes to these transitions. The adapted protocol maintains cache coherence for similar reasons to the protocol in Section 4.

On a read miss, the checker `puts` the (address, block) pair in the untrusted RAM if the old block is in state M (the cache is evicting a dirty block) or if the old block is in state S and the checker's FSM is the only FSM with the block (the cache is evicting a clean block and it is the only cache with the block). The FSM then makes a request for the new block, using the new block's address. All of the other FSMs see this request. If one of them has the block in its cache in state M, E or S, it aborts the request, changes its block's state in its cache to state S if it was not already in state S, and sends its block on the bus, without writing the block back to the untrusted RAM. The FSM which requested the block will see the block. It will read the block and put it into its cache in state S. If the block is not in any of the caches, the (checker of the) FSM which requested the block will end up `taking` the (address) from the untrusted RAM and will put the block into its cache in state S.

A write hit to a shared block is treated the same way as a write miss. Thus, on a write miss or a write hit to a shared block, the checker `puts` the (address, block) pair in the untrusted RAM if the old block is in state M (the cache is evicting a dirty block) or if the old block is in state S and the checker's FSM is the only FSM with the block (the cache is evicting a clean block and it is the only cache with the block). The FSM then places a write miss on the bus, using the new block's address. All of the FSMs with the block in their caches see the write miss and invalidate the block by changing the block's state in their caches to state I. The block is changed to state E in the cache of the FSM performing the write, and when the write is performed, the FSM then changes the block's state to state M.

With reference to Figure 2, in each of the transitions from state M to state S and state M to state I, the block is not written back to the untrusted RAM. (The block is only written back to the untrusted RAM when the checker `puts` the (address, block) pair in the untrusted RAM; as described, this occurs when the block is evicted from the cache and the block was either in state M, or in state S and the cache was the only cache with the block.)

In an implementation, when a cache is bringing in a block, if one of the other caches has the block, it is supposed to supply the block instead of the block being `taken` from the untrusted RAM. This can easily be checked by the checker of the cache with the block setting an an error flag, maintained in its trusted state, if it sees the untrusted RAM returning a block that it already has it its cache; if the error flag is set, the `check` operation does not pass. When a cache is evicting a block, if the block is in state M, it knows it is the only cache with the block and its checker should `put` the (address, block) into the untrusted RAM. However, if the block is in state S, the block may really be shared by two or more caches, or may just be a clean block in the cache that is evicting it[6]. In the case that the cache is evicting a block in state S, it should put a message on the bus, using the block's address, asking if any of the other caches has the block. If one of the other caches has the block, it replies to the message. The cache performing the eviction waits for some predetermined number of cycles for at least one reply, and its checker only `puts` the (address, block) into the untrusted RAM if there is no reply within the predetermined number of cycles. Again, this approach can be easily checked. If there does happen to be another cache with the block, but the block is still `put` into the untrusted RAM, the checker of that cache can set an error flag, maintained in its trusted state; if the error flag is set, the `check` operation does not pass. When performing a `check`, a checker only `takes` blocks which are not in any of the caches from the untrusted RAM (and `puts` them back into the untrusted RAM if the `check` is an intermediate `check`). The checker can make a request for each of the blocks not in its cache; if one of the other caches has the block it can abort this request, and let the checker know that it already has the block. Again, the `check` operation can be distributed among different checkers as described in Section 6.2.2 to improve the performance of the log-hash scheme.

6.3 Direct Memory Access

In this section, we examine the issues that are posed to the checking of the untrusted RAM when there is Direct Memory Access (DMA). DMA allows a device controller to transfer data directly to or from the untrusted RAM without involving the FSMs. The mechanism is useful for high bandwidth devices because the FSMs can perform other functions while the data is being transferred.

Without DMA, all accesses to the untrusted RAM come from one of the checkers. With DMA, however, there is another path to the untrusted RAM, which does not involve any of the checkers. The way we deal with DMA in an SMP system is similar to the way we dealt with it for the uniprocessor case [GSC$^+$03]. An unprotected area of the untrusted RAM is set aside for DMA transfers. Once the DMA transfer is completed, the FSMs authenticate the data using some scheme of their choosing and copy the data into a secure buffer before using it.

Though the approach is secure, it has the drawback of requiring an extra copy of the data. However, in many systems, this is not a major penalty as the operating system typically performs a copy of the incoming data from a kernel buffer into user space anyways.

[6]We note that, in some variations of the MESI protocol, state S is only for if a block is shared by two or more caches, with state E for if the block is clean in exactly one cache.

7 Performance Implications

In general, the performance of hardware systems is heavily dependent on the amount of traffic on the bus.

We first study the performance of an SMP system that uses the hash tree-based memory integrity scheme. The approach where each checker maintains its own tree over the untrusted RAM is likely to be too expensive in practice because of the contention among the hashes in the different caches. Therefore, we study the performance of the approach where the untrusted RAM is segmented, with each segment maintained by a separate tree managed by a different checker in a different FSM. We note that one disadvantage of this approach is that there is not any load balancing. Thus, if one segment of memory is accessed more frequently than another, one checker will be performing more work than another. As described in Section 6.1.2, in the approach, the checkers should check sequences of memory operations, because again, it is likely to be too expensive to check each load operation.

During runtime, the extra overhead of memory integrity checking comes from the FSM id which prepends the FSM messages, the reduced cache hit rate of data values because the cache now also stores hashes, and the extra bandwidth consumption of fetching and writing hashes from and to the untrusted RAM. The runtime overhead is similar to the overhead of implementing hash tree-based integrity checking on a uniprocessor (c.f. Section 6.1.1).

When checking memory, the authenticity of the bus must also be checked. As described in Section 5, checking the authenticity of the bus entails the checkers using the MAC-based operations in Figure 5 to securely exchange their running hashes. If sequences of store and load operations are checked, the cost of this process is amortized over the memory operations. This amortized cost can be small, making the principal cost of hash tree memory integrity checking be the runtime cost. Thus, if sequences of memory operations are checked, the expected performance overhead of hash tree-based memory integrity checking on an SMP system will be similar to the overhead on a uniprocessor system.

We now study the performance implications of an SMP system which uses the log-based memory integrity scheme. As described in Section 6.2.2, the approach where just one checker maintains a (WriteHash, ReadHash, Timer) triple is not efficient because it will require that, instead of just throwing away clean blocks when they are evicted from the cache, clean blocks would have to be transmitted on the bus so that the checker which checks the untrusted RAM can properly update its trusted state. Therefore, we study the performance of the approach where each FSM maintains its own hashes and Timer. We note that, because each checker maintains its own triple to help check memory, this approach balances the load of memory integrity checking.

During runtime, the extra overhead of the memory integrity checking comes from the FSM id which prepends the FSM messages, the time stamps which accompany values written to and read from the untrusted RAM, and the bandwidth consumed by the request and reply messages transmitted when an FSM evicts a block in state S. Snooping-based SMP systems typically support up to 32 processors, thus, processor ids can be encoded in 5 bits. Time stamps can be encoded in 32 bits. The principal part of the request-reply messages is the address of the block in question; the address is also 32 bits large. Therefore, during runtime, the extra bus bandwidth consumption of log-based memory integrity is expected to be small, similar to the overhead on a uniprocessor system.

When a `check` is performed, the untrusted RAM that was used is read, and the WriteHashes and ReadHashes are transmitted on the bus to be summed by the checkers. The authenticity of the bus is also checked. This checking process can be expensive. However, if sequences of store and load operations are checked, the checking process's cost is amortized over the memory operations, and its *amortized cost* can be small. In this case, the principal overhead of log-based memory integrity checking comes from the runtime cost, which can also be small. Thus, if sequences of memory operations are checked, the expected performance overhead of log-based memory integrity checking on an SMP system is expected to be small. This result is very similar to the result we obtained with the log-based memory integrity checking scheme on a uniprocessor [SCG$^+$03b], which performed well when sequences of operations are checked, for similar reasons.

8 Hybrid hash tree-log hash scheme

The disadvantage of the log-hash scheme is that, if integrity checks are frequent, the log-hash scheme can perform worse than the hash-tree scheme, because of the overhead incurred reading the addresses used since the beginning of FSMs execution to perform the integrity check. It is desirable to construct a scheme which could take advantage of the benefits of both the hash-tree and log-hash schemes. The FSMs could use this scheme to maximize their performance during their execution.

The hybrid hash tree-log hash for a uniprocessor in [CGS$^+$02] can be adapted to a SMP system. In the hybrid approach, the checkers maintain both hash trees and the log hash multiset hashes and timer. Data is initially stored in the trees. The idea is that, when the FSMs want to perform a particular computation on a subset of the data in the storage, they can either work on the data using the tree, or take the data out of the tree, and work on it using the log-hash scheme. When the checkers perform an intermediate log-hash integrity check, the checkers only need to read addresses that were used since the last intermediate check, instead of having to read all of the addresses that the FSMs used since the beginning of their execution. If the intermediate log-hash integrity check is successful, data can then be returned to the tree if so desired.

The FSMs can dynamically employ several strategies during their execution that would maximize their performances. As an example of a strategy the FSMs might employ, if there is data the FSMs regularly use and data they use rarely, they can protect the data they use regularly with the log hash scheme, and protect the data they use rarely using the hash tree; this can reduce the number of addresses that are read to perform a log-hash integrity check. As a second example of a strategy, if, during some part of their execution, the FSMs will be regularly exporting results, the FSMs can protect the data using the hash trees, which has a smaller overhead when integrity checks are very frequent; if, during some other part of their execution, the FSMs perform a computation for which they will not export a result for some time, the data that is being used can be moved to the log hash scheme, which performs better when integrity checks are less frequent.

The checkers use the bus authentication protocol as described in Section 5, to check the authenticity of the bus whenever a sequence of operations needs to be checked.

9 Conclusion

The paper examined approaches to implementing memory integrity checking in an SMP system, focusing on a bus-based SMP system. We also examined extending the schemes to a hybrid hash tree-log hash memory integrity checking scheme for SMP.

References

[Asa04] Krste Asanovic. MIT Course 6.823: Computer System Architecture - Spring 2004. Lecture 16: Cache Coherence. `http://csg.lcs.mit.edu/6.823/handouts/lec16.pdf`, 2004.

[BGR95] M. Bellare, R. Guerin, and P. Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In *Advances in Cryptology - Crypto 95 Proceedings*, volume 963 of *LNCS*. Springer-Verlag, 1995.

[CDvD+03] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental Multiset Hash Functions and their Application to Memory Integrity Checking. In *Advances in Cryptology - Asiacrypt 2003 Proceedings*, volume 2894 of *LNCS*. Springer-Verlag, 2003.

[CGS+02] Dwaine Clarke, Blaise Gassend, G. Edward Suh, Marten van Dijk, and Srinivas Devadas. Offline integrity checking of untrusted storage. In *Technical Report MIT-LCS-TR-871*, November 2002.

[CJPL02] Amy Carroll, Mario Juarez, Julia Polk, and Tony Leininger. Microsoft "Palladium": A Business Overview. In *Microsoft Content Security Business Unit*, August 2002.

[GSC+03] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and merkle trees for efficient memory integrity verification. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, February 2003.

[HP97] John L. Hennessy and David A. Patterson. *Computer Organization and Design.* Morgan Kaufmann Publishers, Inc., 1997.

[KPS95] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security, Private Communication in a Public World.* Prentice Hall PTR, 1995.

[LTM+00] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[NIS95] NIST. FIPS PUB 180-1: Secure Hash Standard, April 1995.

[SCG+03a] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Int'l Conference on Supercomputing*, June 2003.

[SCG+03b] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th Int'l Symposium on Microarchitecture*, Dec 2003.

[SW99] S. W. Smith and S. H. Weingart. Building a High-Performance, Programmable Secure Coprocessor. In *Computer Networks (Special Issue on Computer Network Security)*, volume 31, pages 831–860, April 1999.

[Yee94] Bennet S. Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, 1994.