Caches and Merkle Trees for Efficient Memory Authentication

Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, Srinivas Devadas
MIT Laboratory for Computer Science

September 23, 2002

Abstract

We describe a hardware scheme to authenticate all or a part of untrusted external memory using trusted on-chip storage. Our scheme uses Merkle trees and caches to efficiently authenticate memory. Proper placement of Merkle tree checking and generation is critical to ensure good performance. Naïve schemes where the Merkle tree machinery is placed between caches can result in a large increase in memory bandwidth usage. We integrate the Merkle tree machinery with one of the cache levels to significantly reduce memory bandwidth requirements.

We present an evaluation of the area and performance costs of various schemes using simulation. For most benchmarks, the performance overhead of authentication using our integrated Merkle tree/caching scheme is less than 25%, whereas the overhead of authentication for a naïve scheme can be as large as 10×. We explore tradeoffs between external memory overhead and processor performance.

Introduction

Secure processors (e.g., [Yee94] [SW99], [LTM+00]) can enable many applications such as copy-proof software and certification that a computation was carried out correctly. Additionally they hold promise in helping to implement secure computing systems that are resistant to viruses and other forms of attacks [CPL].

Even if secure processors are built, in order to build secure computing systems, one of the problems that needs to be solved is the authentication of untrusted external memory. This memory is typically accessed via an external memory bus, or may be networked in some fashion to the secure processor. While the secure processor is running, an adversary may corrupt the memory values in the external memory. The secure processor should be able to detect any form of memory corruption. Typically, the secure processor will cease the execution of the running task(s) when it detects memory corruption.[1] To achieve security and high performance, it is critical to have an efficient authentication scheme that does not slow down the processor. In this paper, we describe hardware schemes to efficiently authenticate all or a part of untrusted external memory using a limited amount of trusted on-chip storage.

We propose a on-line scheme for memory authentication. Our scheme uses hash trees (also called Merkle trees) and caches to efficiently authenticate memory. Naïve schemes where the Merkle tree machinery is placed between caches, e.g., between L1 and L2 caches, can result in a factor of $\log N$ increase in memory bandwidth usage (where N is the memory size), thereby degrading performance significantly. In our proposed scheme, we integrate the Merkle tree machinery with one of the cache levels to significantly reduce memory bandwidth requirements.

We present an evaluation of the area and performance costs of various on-line schemes using simulation. For most benchmarks, on a superscalar processor, the performance overhead of authentication using our integrated Merkle tree/caching scheme is less than 25%, whereas the overhead of authentication for a naïve scheme can be as large as 10×. We show tradeoffs between external memory overhead and secure processor performance.

We describe related work in Section 1. The assumed model is presented in Section 2, and motivating applications are the subject of Section 3. An on-line caching scheme for memory authentication is described in Section 4. We evaluate the scheme implemented on a superscalar processor in Section 5.

1 Related Work

Merkle trees [Mer80] were proposed as a means to update and validate data hashes efficiently by maintaining a tree of hash values over the objects.

[1] In this paper, we are not concerned with the correction of corrupted data in memory. 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.

Blum et al. addressed the problem of securing various data structures in untrusted memory using a hash tree rooted in trusted memory [BEG+91]. They consider both off-line detectors that use a hashed trace of all accesses (including values read or written, locations accessed and times of access) and an on-line detector, which uses Merkle trees. The on-line scheme described by Blum et al. has a $O(\log(N))$ cost for each memory access. The off-line scheme targets the detection of memory errors, and does not secure memory against attacks from an adversary.

Maheshwari, Vingralek and Shapiro use Merkle trees to build trusted databases on top of trusted storage [MVS00]. This work is similar to ours in that trusted memory can be viewed as a cache for untrusted disk storage – their scheme exploits memory locality to reduce disk bandwidth. Our work addresses the issues in implementing Merkle tree machinery in hardware and integrating this machinery with an on-chip cache to reduce the $\log N$ memory bandwidth overhead. The caching algorithm of Section 4 is more general in that a single hash can be used for multiple cache blocks. This scheme can potentially reduce untrusted memory size overhead and cache pollution without increasing cache block size.

Shapiro and Vingralek [SV01] address the problem of managing persistent state in DRM systems. They assume that each memory reference results in a MAC computation and therefore discount the possibility of securing volatile storage because it requires large overheads. They assume that volatile memory is inside the security perimeter.

In [DS02] allusions are made to a smartcard system that would use a Merkle tree with large pages of RAM at its leaves, combined with caching of pages in internal memory. Their discussion, however is strongly directed towards smartcard applications, and they do not appear to consider caching nodes of the Merkle tree.

2 Model

In this paper we are considering a computer system with the following properties:

- There is a high performance processor that contains a secret that allows it to produce keys to perform cryptographic operations such as signing or encrypting that no other processor could do for it. This secret can be a private key from a public key pair as in XOM [LTM+00], or it can be a Physical Unknown Function [GCvDD02]. Symmetric key schemes are inapropriate as we want many mutually mistrusting principals to be able to use the system.
- Operations that take place inside the processor are assumed to be private and tamper-evident.
- The processor has sufficient on-chip storage to perform its cryptographic operations on-chip.
- The processor has a trusted on-chip cache.
- Everything outside the processor is untrusted, in particular the memory. By untrusted, we mean that there is an adversary who knows everything about the system except for the processor's secret.

The objective of the system is the following:

- A user wants to use the system to perform a computation that produces one or many results, to which cryptographic primitives are then applied. The reason for the cryptographic primitives will be illustrated in section 3.1.
- The computation will involve the processor and external memory.
- We want the computation to be carried out at speeds that are as close as possible to the speed of a conventional insecure processor.
- We want a high probability of detecting results that contain errors induced by tampering from the adversary. This probability must be high even if the adversary chooses the program to run.

The objective of the adversary is the following:

- The adversary wants to tamper with the memory in such a way that the system produces an incorrect result that looks correct to the user.

In this paper we will solve this problem by providing an authentication mechanism for the off-chip memory. In the next section we show how this integrity can be used in applications.

3 Applications

3.1 Certifying the Execution of a Program

The main focus of this paper is to explore memory authentication. However, memory authentication is only useful if it is performed on a processor that contains a secret, and the processor is able to perform some simple cryptography on it. We will not go into the details of the cryptography that is necessary, but for ease of understanding, we provide an example of use.

Alice has a problem to solve that requires a lot of computing power. Bob has a computer that is idle, and that he is willing to rent to Alice. If Alice gives

Bob her problem to execute, and Bob gives her a result, how can she be sure that Bob actually carried out the computation? How can she tell that Bob didn't just invent the result?

One way of solving this is by having a processor that has been certified by its manufacturer, that contains a secret, and that is equipped to deal with problems such as the one Alice has.

Alice sends this processor her problem expressed as a program. The processor uses Alice's program combined with its secret key through a collision resistant scheme to produce a key that is unique to the processor-program pair. The processor then executes Alice's program without allowing any interference from external sources. The processor executes the program in a deterministic way to produce the result Alice desires. It then uses the key it generated to sign the result before sending it to Alice.

As long as Alice's computation can all be done on the processor, things go well. However, for most algorithms, it is likely that Alice will need to use external memory. *How can she be sure that Bob isn't tampering with the memory bus to make Alice's program terminate early while still producing a valid certificate for an incorrect result?* This is precisely the question that we try to answer in this paper by providing an efficient means to authenticate memory operations.

When Alice receives the signed result, she is able to check it. At that point she knows that her program was executed on a trusted processor, and that the external memory performed correctly.[2] If the program did not contain a bug then Alice has the correct result.

Without the ability to perform some kind of cryptography, the memory authentication would be useless except to detect faults in the memory (which could be detected much more cheaply with simple error detecting codes). Indeed, executions carried out on the real processor would be identical to results carried out on a processor simulator, on which any kind of tampering can be done with the data.

Of course, in real systems Bob will want to continue using his computer while Alice is calculating. Systems like Palladium or XOM provide this functionality as we shall see in the following sections. But in each case, they still need authenticated memory.

3.2 Palladium

Microsoft's proposed security model, Palladium [CPL], may be enhanced by authenticated memory. Indeed, Palladium works by providing a mechanism whereby the Operating System (OS) can prove to an application that it is trusted, and that it was loaded properly (so no mischievous code was able to load before the OS). The application program, knowing that it is being protected by a security kernel that was properly loaded, can be confident that it will receive whatever protection it requires for proper execution.

But in order to make this model work, it has to be impossible for a hacker to break the system's security by modifying the secure kernel as it goes over the bus between external memory and processor. This can be done by always keeping the security kernel on the processor. But this wastes on-chip storage when the security kernel is not in use (most of the time). Another option is to use authenticated memory such as we describe it in this paper.

3.3 XOM architecture

The eXecute Only Memory (XOM) architecture [LTM+00] is designed to run security requiring applications in secure compartments that can only communicate with the rest of the world on an explicit request from the application.

This protection is achieved on-chip by tagging data with the compartment to which it belongs. In this way, if a program executing in a different compartment attempts to read the data, the processor detects it and raises an exception.

For data that goes off-chip, XOM uses encryption to preserve privacy. Each compartment has a different encryption key. Before encryption, the data is appended with a hash of itself. In this way, when data is recovered from memory, XOM can verify that the data was indeed stored by a program in the same compartment. XOM prevents an adversary from copying encrypted blocks from one address to another by combining the address into the hash of the data that it calculates.

3.3.1 Exploiting Replay Attacks

However, XOM's integrity mechanism is vulnerable to replay attacks, which was also pointed out in [SV01]. Indeed, in XOM there is no way to detect whether data in external memory is fresh or not. Freshness appears to be provided for from one execution of a secure application to another by having a mutating key (essentially, a different key for each execution). Within a single execution of the secure application, however, the key cannot be changed without making data that was stored at the beginning of execution unreadable. Therefore, an adversary can do replay attacks by having the memory return stale data that had previously been stored at the same address during the same execution. In particular,

[2] If Alice's program stored data on disk, we assume that it took measures to authenticate the data.

XOM will not notice if writes to memory are never performed except when memory is first initialized.

This flaw in XOM's authentication could be exploited to violate the privacy of some programs. Consider the following example:

```
for (i = 0; i < size; i++)
{
  outputdata(*data++);
  /* outputdata copies data
    out of the secure
    compartment */
}
```

If `outputdata` causes `i` to be swapped to memory, and if `i` and `data` are not in the same cache line, then an attacker can cause the loop to be executed many more times than it should. Assuming the attacker knows where `i` is stored, he can record the value of `i` during an iteration of the loop, and then replace the incremented value by the pre-recorded value each time through the loop. In this way, `outputdata` gets called with each data value up to the end of the data segment, thus revealing a lot more to the outside world than initially intended. If `data` is stored on the stack, it might be possible to replace it with an address in the code segment to reveal a large portion of the program's code.

To pull this attack off, the adversary would presumably single step the program,[3] flushing the cache between steps. In this way, he can obtain a cache-line level observation of the program's memory access patterns. By observing loop like patterns in the program counter, the adversary can search for loops. Loops that cause data to be copied out of the secure compartment can be identified by the unencrypted data that they are writing to memory, or, even better, by the unprotected system calls that are being called with the data.[4] All the adversary has left to do is guess the location of `i` in the stack (the general position of the stack will be apparent from the memory access pattern).

Though this attack may seem involved, and despite the fact that the code sample is somewhat unlikely, it is quite plausible that a complex program will contain similar vulnerabilities, which a motivated adversary could find and exploit. There is a wealth of examples from the smartcard world where attacks of similar type have been carried out to extract secret information, as can be seen in [AK97].

[3] If single stepping is forbidden, a similar effect can probably be obtained by generating an interrupt after a small number of memory accesses. With luck, it would even be possible to turn off the processor's caches.

[4] In fact, it might be possible to find a suitable loop simply by observing patterns of system calls.

3.3.2 Correcting XOM

XOM can be fixed in a simple, though not optimal, way by combining it with our memory authentication method. Essentially, XOM was attempting to provide two forms of protection: protection from an untrusted OS, and protection from untrusted off-chip memory. Its method for dealing with the untrusted OS seems much more open an approach than the one that is used by Palladium. It would make all the Palladium applications possible without forcing the use of a certified (and presumably commercial) OS. As far as the protection of off-chip memory goes, XOM fails because memory is not properly authenticated. Protecting memory integrity with hash trees would solve XOM's problem.

4 Authentication Algorithm

4.1 Hash Trees

We verify the integrity of memory with a hash tree, also called a Merkle tree (see [Mer80]). In a hash tree, data is located at the leaves of a tree. Each node contains a collision resistant hash of the data that is in each one of the nodes or leaves that are below it. A hash of the root of the tree is stored in secure memory where it cannot be tampered with. Figure 1 shows the layout.

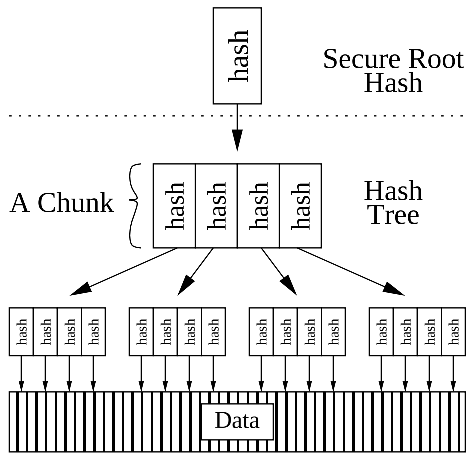


Figure 1: A hash tree. Chunks contain hashes that authenticate data in chunks lower in the tree.

To check that a node or leaf in a hash tree has not been tampered with, we check that its hash matches the hash that is stored in its parent node, and that the parent node has not been tampered with. We repeat this process on the parent node, and on its parent node, all the way to the root of the tree that has its hash stored in secure memory. When writing to the hash tree, it is similarly necessary to update all the nodes above the modification up to the secure root hash.

An m-ary hash tree allows integrity verification with a constant factor overhead in memory consumption of $m/(m-1)$. With a balanced tree, the number of hash checks to perform is $\log_m(N)$, where N is the amount of memory to be protected, expressed in multiples of the size of a hash. The cost of each hash computation is proportional to m (i.e., the amount of data to hash).

These costs of using a hash tree are quite modest considering that they allow a very small amount of secure memory (typically 128 to 160 bits) to verify the integrity of arbitrarily large amounts of memory. For a 4-ary tree, one quarter of memory ends up being used by hashes, which is large, but not unacceptably so. However memory bandwidth is a major constraint in high performance systems, and the number of hashes that must be read from memory increases very quickly with memory size and can easily exceed 10. Unless this access pattern is optimized, system performance is sure to be dismal.

4.2 Hash Trees in the Memory Hierarchy

Proper placement of the hash tree checking and generation machinery is critical in ensuring good performance. On first thought, the machinery could be placed between two layers of the memory hierarchy. The higher layers would not know about the hash tree. On a miss, they would use the hash tree machinery to read and authenticate data from the lower part of the hierarchy. There are two logical places for these operations to be carried out:

1. One possibility is to place the hash tree machinery between L2[5] and external memory. This way cached data can be retrieved very fast without having to go through any integrity verification. However, this option implies that L2 cache misses will result in a whole path from leaf to root of the hash tree being fetched from external memory on a cache miss, which is unacceptably slow.

2. Another possibility is to place the hash tree machinery between L1 and L2. Thus, frequently used paths in the hash tree will end up being cached in L2 and will not put such a high load on memory bandwidth. The drawback is that accessing L2 becomes much more costly since each access to L2 involves checking a whole path in L2.

[5]We are assuming a processor with on-chip L1 and L2 cache.

Both schemes share a common drawback. Each access that goes through the hash tree layer requires $\log_m(N)$ hash checks, and consequently accessing all the data that those hashes are to be computed on. In Section 5 we use scheme 1 as a representative naïve scheme, and refer to it as `naive`. The following section shows an optimized hash tree implementation that gets around this problem and which makes hash trees a reasonable choice in a high performance processor.

4.3 Making Hash Trees Fast: `chash`

To make hash trees fast, we have to merge the hash tree machinery with one of the cache levels. Values that are stored in this combined cache are trusted, which allows accesses to be performed directly on cached values without any hashing, giving us the advantage of scheme 1 above. At the same time, data that is needed for hashing can now be read from the cache. This reduces the latency to the data as in scheme 2, and has the additional advantage that if a hash comes from the cache, it is trusted, and therefore it is not necessary to continue checking hashes up to the root of the tree.

The following algorithms show how the combined cache that has all these nice properties can be implemented. In these algorithms the word *cache* refers to the combined cache (which is assumed to be trusted), and the word *memory* refers to the next level in the memory hierarchy.[6] The memory is divided into *chunks* that are the basic unit that hashes are computed on. For now we will consider that chunks conicide with cache blocks.

ReadAndCheckChunk: Reads data from external memory and checks it.

1. Read the chunk from memory.

2. Return the chunk to the caller so that it can start speculative execution.

3. Start hashing the chunk that we just read. In parallel, recursively call ReadAndCheck to fetch the chunk's hash from its parent chunk. If the chunk is in fact the root chunk, its hash is fetched directly from secure memory instead of calling ReadAndCheck.

4. Compare the hash we just computed with the one in the parent chunk. If they do not match, raise an exception.

ReadAndCheck: Called when the processor executes a read instruction.

[6]We will work with L2 as the cache, and off-chip RAM as the memory.

1. If the data is cached, return the cached data. We are done.
2. Call ReadAndCheckChunk on the data's chunk.
3. Put the read chunk into the cache.
4. Return the requested data.

Write: Called when the processor executes a write instruction.

1. If the data to be modified is in the cache, modify it directly. We are done.
2. Otherwise, use ReadAndCheckChunk to get the chunk data, and put it into the cache (we are implementing a write-allocate cache here).
3. Modify the data in the cache.

Write-Back: Called when a dirty cache block is evicted.

1. Compute the hash on the modified chunk.
2. In a way that makes both changes visible simultaneously, write the chunk to memory and change its hash in the parent chunk using the Write operation described above (unless it is the root chunk, in which case the hash is stored in secure memory).

Intuitively, with this algorithm, when a node of the hash tree is loaded into the cache, it is used as the root of a new hash tree. This is valid because the node is now stored in secure on-chip storage, and thus no longer needs to be protected by its parent node in the main hash tree. The performance advantage results because the new tree is smaller than the original tree, which reduces the path from leaf to node. As far as correctness goes, the algorithm's essential invariant is that at any time, a hash contained in the tree is a hash of the child chunk in memory.[7] On writes, the hash only gets recomputed when the changes are written back.

Note that this algorithm implements a write-allocate cache. This is sensible since performing a word write requires the word's whole chunk to be read in for hashing anyways. Nevertheless, a useful optimization can be made, inspired by normal cache technology: if write allocation simply marks unwritten words as invalid rather than loading them form memory, then chunks that get entirely overwritten don't have to be read from memory and checked. This optimization eliminates one chunk read from memory and one hash computation.

[7]This invariant is in fact a bit too strong for this algorithm, but will be necessary for the versions that are described in the next sections. We could reduce the invariant to: hashes of uncached chunks must be valid, hashes of cached chunks can have an arbitrary value. The last step of the write-back algorithm can then take place in two steps: update the hash, then write the hash back to memory.

4.4 Multiple Cache Blocks per Chunk: `mhash`

In the algorithm described above, we have assumed that there is exactly one cache block per chunk. Since the cache block is usually chosen to optimize the performance of the processor when security is turned off, it turns out that the chunk size is completely constrained before the memory integrity option is even considered. To allow more flexible selection of the chunk size, let us consider an improved algorithm that does not require that chunks coincide with cache blocks.

The modified algorithm is described below. Parts that are unchanged appear in small type. Note that ReadAndCheckChunk returns the data that is in memory. This data will be stale when the cache contains a dirty copy of some cache blocks.

ReadAndCheckChunk

1. Read cache blocks that are clean in the cache directly from the cache. Read the rest of the chunk from memory.
2. Return the chunk to the caller so that it can start speculative execution.
3. Start hashing the chunk that we just read. In parallel, recursively call ReadAndCheck to fetch the chunk's hash from its parent chunk. If the chunk is in fact the root chunk, its hash is fetched directly from secure memory instead of calling ReadAndCheck.
4. Compare the hash we just computed with the one in the parent chunk. If they do not match, raise an exception.

ReadAndCheck

1. If the data is cached, return the cached data. We are done.
2. Call ReadAndCheckChunk on the data's chunk.
3. Put the read chunk into the cache, except for cache blocks that are already cached in the dirty state.
4. Return the requested data.

Write

1. If the data to be modified is in the cache, modify it directly. We are done.
2. Otherwise, use ReadAndCheckChunk to get blocks that are missing from the cache. Write them to the cache (we are implementing a write-allocate cache here).
3. Modify the data in the cache.

Write-Back

1. If the chunk is not entirely contained in the cache, use ReadAndCheckChunk to get the missing data.
2. Mark all the chunk's cached blocks as clean.
3. Compute the hash on the modified chunk.
4. In a way that makes both changes visible simultaneously, write the blocks that were dirty to memory and change its hash in the parent chunk using the Write operation described above (unless it is the root chunk, in which case the hash is stored in secure memory).

This algorithm can be further optimized by replacing the hash function by an incremental MAC (Message Authentication Code). This MAC has the property that single cache block changes can be applied without knowing the value in the other cache blocks. An example of such a MAC is presented in [BGR95], it is based on a conventional MAC function h_k and an encryption function $E_{k'}$:

$$M_{k,k'}(m_1, \cdots, m_n) = E_{k'}(h_k(1, m_1) \oplus \cdots \oplus h_k(n, m_n))$$

Given a value of the MAC, it can be updated when m_i changes by decrypting the value, subtracting the old value of $h_k(i, m_i)$, adding the new value of $h_k(i, m_i)$, and finally encrypting the new result.

With this hash function, the Write-Back operation can be optimized so that it is not necessary to load the whole chunk from memory if part of it isn't in the cache.

Write-Back

1. Read the parent MAC using the ReadAndCheck operation.
2. Read the old value of the cache block from memory directly (without checking it so that we don't have to read the whole chunk).
3. Calculate the new value of the MAC by doing an update.
4. In a way that makes both changes visible simultaneously, write the chunk to memory and change its hash in the parent chunk using the Write operation described above (unless it is the root chunk, in which case the hash is stored in secure memory).

We will refer to the optimized algorithm as `mhash` in Section 5.

4.5 Simplified Memory Organization

We have chosen to adopt a very simple memory organization in which all of physical memory is authenticated. Physical memory is assumed to be present as a contiguous segment beginning at address 0 that we want to authenticate completely. While this assumption is quite restrictive as far as real systems go, it is quite adequate for our purposes of studying the performance cost of protecting RAM with hash trees.

The layout of the hash tree in RAM is equally simple. The memory is stored in equal sized chunks. Each chunk can store data or can store m hashes. Chunk are numbered consecutively starting from zero so that a chunk's number multiplied by the size of the chunk produces the chunk's starting address.

To find the parent of a chunk, we subtract one from the chunk's number divided by m and round down. If the result is negative then the chunk's hash is stored in secure memory. Otherwise, the result is the parent chunk's address. The remainder of the division indicates the index of the chunk's hash in its parent chunk.

The resulting tree is almost a balanced m-ary tree. In general, the m balanced subtrees aren't quite balanced as there aren't enough elements to fill the last level completely.

The interesting features of this layout are that it is very easy to find a chunk's parent when m is a power of two, and all the leaves are contiguous.

4.6 Real Life Issues

4.6.1 Direct Memory Access

For now we have considered that all of memory is authenticated. This assumption breaks down when data is inserted directly into memory by a device through Direct Memory Access (DMA).

One way of dealing with this is to set aside an unprotected area of memory for use in DMA transfers. Once the transfer is done, programs can copy the data into a secure buffer before using it. This method has the drawback of requiring an extra copy of the data. But in many systems this is not a major penalty as the operating system typically performs a copy of incoming data from a kernel buffer into user space anyways.

Data coming from the DMA is untrusted (since it comes from off chip), so once it has been brought into authenticated memory, it must be authenticated by the application program using some scheme of its choosing.

For safety, the processor should only allow reads to unprotected memory when a special ReadWithoutChecking instruction is used. That way a program cannot be tricked into reading unauthenticated data when it expects authenticated data.

4.6.2 Initialization

So far we have considered how the processor executes when memory is authenticated. It is important to consider how to initialize secure mode since a flaw in this step would make all our efforts futile. Here is the proposed procedure:

1. Turn on hashing, but for now do not bother checking hashes during the ReadAndCheckChunk. In this mode hash trees will be computed, but no checking is done.
2. Touch (write to) each chunk that is to be covered by the hash tree. In this way each chunk ends up in the cache in a dirty state. As chunks are written back, higher levels of the hash tree will get updated.

3. Flush the cache. This forces all the dirty chunks to be written back to memory. These write-backs will cause their parent nodes to appear in the cache in a dirty state. The parents will in turn be written back to the cache, and so on until the whole tree has been computed.[8] [9]
4. Turn on the memory authentication failure exceptions.
5. Generate the key that will be used by this program for cryptographic purposes (see Section 3.1).

At this point, the program is running in secure mode, and its key has been generated. It can now run and eventually sign its results, unless tampering takes place resulting in the destruction of the program's key.

4.7 Precise Exceptions

In the algorithms presented above, it has been stated that hash checks related with ReadChunk can be completed in the background while the returned value is used speculatively. The question is: how far can execution continue speculatively?

Since there is no general way of recovering from tampering other than restarting the program execution from scratch, it appears that there is no need to make memory authentication failure exceptions precise. Therefore, execution can proceed without waiting for authentication checks to complete, and speculative instructions can commit.

There is however an exception to this rule as far as cryptographic operations go. We saw in section 2 that the program can perform cryptographic operations using a key that is a function of the running program. These operations must not allow their results to be seen outside the processor before all preceding hash checks have passed. Otherwise an adversary would be able to make a change to some data just before a program performs a cryptographic primitive on it. With luck, the result of the operation could be sent off-chip before the hash checks completed, and the adversary would have tricked the system into applying its cryptography on data to which it shouldn't have been applied. This attack would allow the adversary to sign, encrypt or decrypt a message of his choice even though he does not have the key.

Therefore, cryptographic instructions must act as barriers for speculative execution of instructions that rely on unauthenticated data. They will stall at the commit stage until all hash checks have completed. For debugging purposes, it might be desirable to provide a mode in which all instructions behave as barriers.

[8] In fact, with this procedure, each hash might be computed a number of times that is equal to the arity of the tree. The procedure that is described here could be optimized to produce only one computation of each hash, but this would require added assumptions about the instruction set architecture to describe precisely, and would not impact the security of the scheme.

[9] In the case where incremental MACs are used, all MAC computations are incremental. So this cache flushing trick would not work. Therefore, the initialization must be modified so that it actually computes a MAC from scratch.

5 Evaluation

This section evaluates our memory authentication scheme using a processor simulator. First, we describe the hardware implementation of our algorithm and the additional logic required by our scheme. Next, the simulation framework used for the experiments is described. Performance and memory space overheads of the scheme are then discussed. We also study the effects of various architectural parameters on memory authentication performance. Finally, the incremental MAC based method to reduce the memory overhead is evaluated.

5.1 Hardware Implementation

We describe the implementation of the `chash` scheme. The `mhash` scheme uses the same datapaths but requires additional control.

A hash checking/generating unit is added next to the L2 cache. Whenever there is a L2 cache miss, a new cache block is read from the main memory, and added to the hash read buffer unit which checks authenticity (Figure 2 (a)). The hashing unit computes a hash of the new cache block, and compares with a previously stored hash, which is read from the L2 cache (or a root hash register if the hash happens to be the root of the tree). If two hashes do not match each other, a security exception is raised.

Similarly, when a cache block gets evicted from the L2 cache, it is stored in the hash write buffer unit while writing back the block to the main memory (Figure 2 (b)). The hash unit computes a new hash of the evicted block and stores the hash back into the L2 cache.

5.2 Logic Overhead

To evaluate the cost of computing hashes, we considered the MD5 [Riv92] (and SHA-1 [DEJ01]) hashing algorithms. The core of each algorithm is an operation that takes a 512-bit block, and produces a 128-

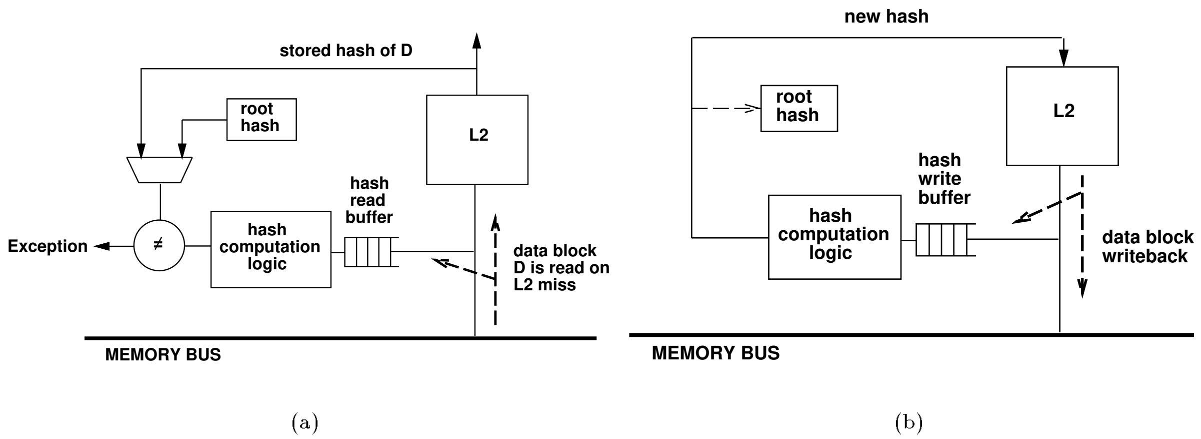


Figure 2: Hardware implementation of the `chash` scheme. (a) L2 cache miss: read from the memory. (b) L2 write back: write to the memory.

bit (or 160-bit, respectively) digest.[10]

In each case, simple 32-bit operations are performed over 80 rounds. In each round there are 2 to 4 logic levels, as well as 2 adders. We noted that with suitable skewing of the adders, rounds can be performed in one cycle per round on average.

The total amount of 32-bit logic blocks that is required for the 80 rounds is 260 adders, 32 multiplexers, 16 inverters, 16 or gates and 48 xor gates (for SHA-1, 325 adders, 60 and gates, 40 or gates, 20 multiplexers and 272 xor gates). If these were all laid out, we would therefore need on the order of 50,000 1-bit gates altogether. In fact, the rounds are very similar to each other so it should be possible to have a lot of sharing between them. To exploit this we chose a hash throughput of one per 20 cycles. This should allow the circuit size to be divided by a factor of 10 to 20.

5.3 Simulation Framework

Our simulation framework is based on the SimpleScalar tool set [BA97], which models speculative out-of-order execution. To model the memory bandwidth usage more accurately, separate address and data buses were implemented. All structures that access the main memory including a L2 cache and the hash unit share the same bus.

The architectural parameters used in the simulations are shown in Table 1. SimpleScalar is configured to execute Alpha binaries, and all benchmarks are compiled on EV6 (21264) for peak performance.

Architectural parameters	Specifications
Clock frequency	1 GHz
L1 I-caches	64KB, 2-way, 32B line
L1 D-caches	64KB, 2-way, 32B line
L2 caches	Unified, 1MB, 4-way, 64B line
L1 latency	2 cycles
L2 latency	10 cycles
Memory latency (first chunk)	80 cycles
I/D TLBs	4-way, 128-entries
Memory bus	200 MHz, 8-B wide (1.6 GB/s)
Fetch/decode width	4 / 4 per cycle
issue/commit width	4 / 4 per cycle
Load/store queue size	64
Register update unit size	128
Hash latency	80 cycles
Hash throughput	3.2 GB/s
Hash read/write buffer	16
Hash length	128 bits

Table 1: Architectural parameters used in simulations.

For all the experiments in this section, nine SPEC2000 CPU benchmarks [Hen00] are used as representative applications: `gcc`, `gzip`, `mcf`, `twolf`, `vortex`, `vpr`, `applu`, `art`, and `swim`. These benchmarks show varied characteristics such as the level of ILP (instruction level parallelism), cache miss-rates, etc. By simulating these benchmarks, we can study the impact of memory authentication on various types of applications.

To capture the characteristics of benchmarks in the middle of computation, each benchmark is simulated for 100 million instructions after skipping the first 1.5 billion instructions. In the simulations, we ignore the initialization overhead of the hash tree. Given the fact that benchmarks run for a long time, the overhead should be negligible compared to the steady-

[10]In fact, for variable length messages, the output from the previous 512-bit block is used as an input to the function that digests the next 512-bit block. Since we are dealing with fixed-length messages of less than 512 bits, we do not need this.

state performance.

5.4 Performance Impact of Memory Authentication

On-line memory authentication requires computing and checking a hash for every read from off-chip memory. At the same time, a new hash should be computed and stored on a write-back to memory. Memory authentication implies even more work for memory operations, which already are rather expensive. Therefore, the obvious first concern of memory authentication is its impact on application performance.

Fortunately, computing and checking hashes do not always increase memory latency. We can optimistically continue computation as soon as data arrives from the memory while checking their authenticity in the background. Checking the authenticity of data hurts memory latency only when read/write buffers are full.

Authenticating memory traffic, however, can degrade the memory performance in two ways: L2 cache pollution and memory bandwidth pollution. First, if we cache hashes in the L2 cache, hashes contend with regular application data and can degrade the L2 miss-rate for application data. On the other hand, loading and storing hashes from/to the main memory increases the memory bandwidth usage, and may steal bandwidth from applications.

Figure 3 illustrates the impact of memory authentication on application performance. For six different L2 cache configurations, the IPCs (instructions per cycle) of three schemes are shown: a standard processor (`base`), memory authentication with caching the hashes with a single cache block per chunk (`chash`), and memory authentication without caching (`naive`).

The figure first demonstrates that the performance overhead of memory authentication can be surprisingly low if we cache hashes. Even though the on-line memory authentication algorithm based on a hash tree can cause tens of additional memory accesses per L2 cache miss, the performance degradation of `chash` compared to `base` is less than 50% in the worst case (`mcf` in the 64B, 256KB case). Moreover, the performance degradation decreases rapidly as either the L2 cache size or the block size increases. For a 4-MB L2 cache, all nine benchmarks run with less than 20% performance hit.

The importance of caching the hashes is also clearly shown in the figure. Without caching (`naive`), some programs can be slowed down by factor of ten in the worst case (`swim` and `applu`). In the case of the naïve scheme, even increasing the cache size or the cache block size does not reduce the overhead. For example, `applu` is still ten times slower than the base case with a 64-B, 4-MB L2 cache.

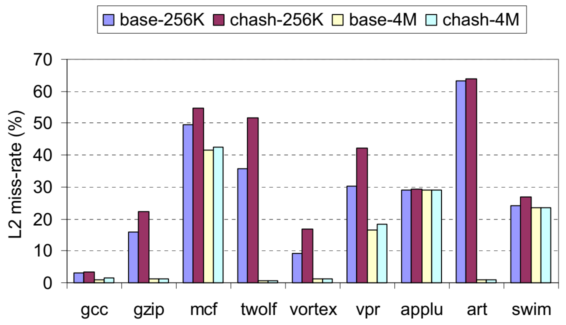


Figure 4: L2 cache miss-rates of program data for a standard processor (`base`) and memory authentication with caching (`chash`). The results are shown for 256-KB and 4-MB caches with 64-B cache blocks.

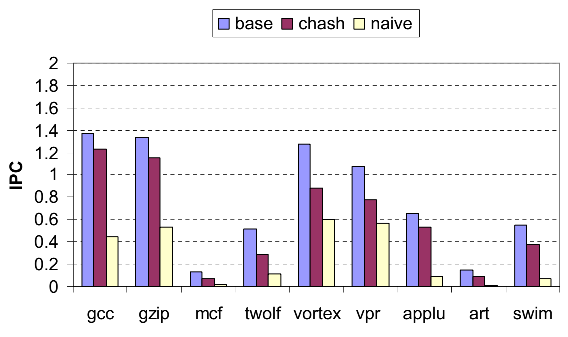
Finally, Figure 3 shows the effect of changing the L2 cache size and the L2 block size on the performance. Having a larger L2 cache reduces authentication performance since it reduces the number of off-chip accesses. A large L2 cache is likely to result in better hash hit-rate without hurting application hit-rate. Having a larger L2 block also reduces the overhead of memory authentication by having less levels in the hash tree. However, a non-optimal L2 block size can degrade the baseline performance as shown in Figure 3.

In the following subsections, we discusses the performance considerations of memory authentication in more detail.

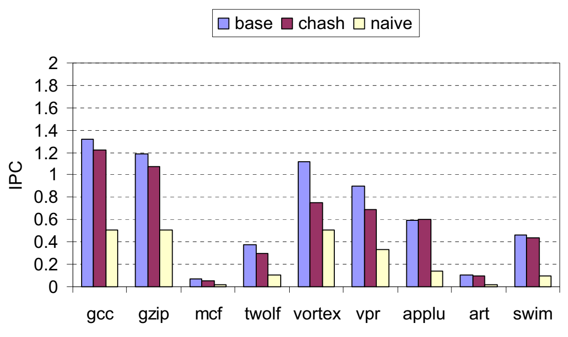
5.4.1 Cache Contention

Since we cache hashes sharing the same L2 cache with a program executing on a processor, both hashes and application data contend for L2 cache space. This can increase the L2 miss-rate for a program and degrade the performance.

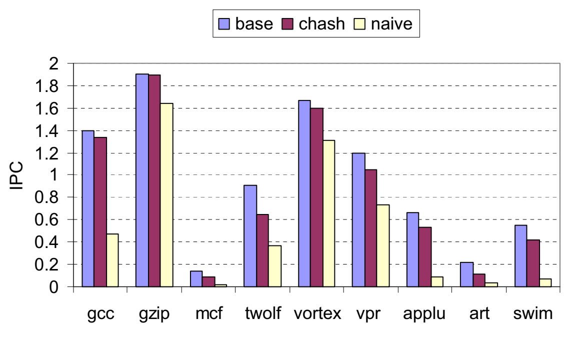
The effect of cache contention is studied in Figure 4. The figure depicts the L2 miss-rates of the baseline case and memory authentication with caching. As shown, for a small L2 cache, the miss-rate can be noticeably increased by caching the hashes. In fact, cache contention is the major source of performance degradation for `twolf`, `vortex`, and `vpr`. However, as the L2 cache size increases, cache contention is alleviated. For example, with a 4-MB L2 cache, none of the benchmarks show noticeable L2 miss-rate degradation. We note that increasing the L2 block size (block = chunk) alleviates cache contention by reducing the number of hashes to cover a given memory space (not shown).



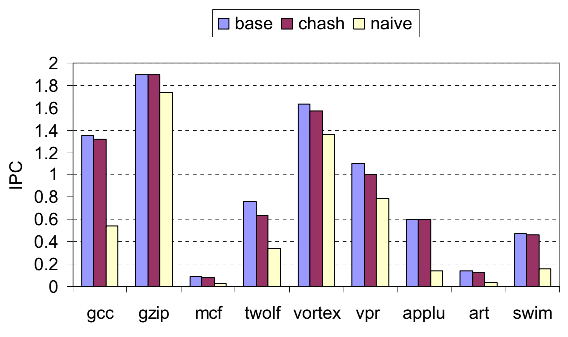
(a) 256KB, 64B



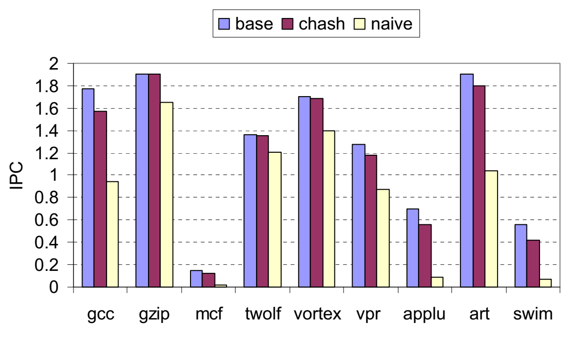
(b) 256KB, 128B



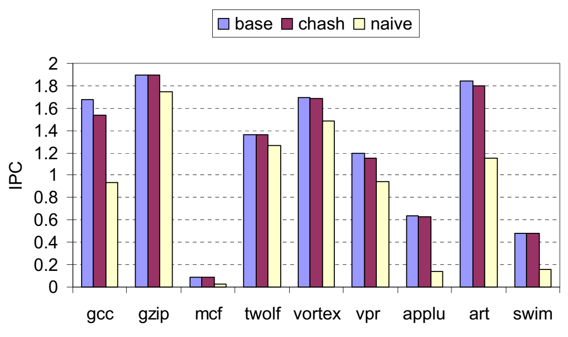
(c) 1MB, 64B



(d) 1MB, 128B



(e) 4MB, 64B



(f) 4MB, 128B

Figure 3: IPC comparison of three different schemes for various L2 cache configurations: standard processors without memory authentication (`base`), memory authentication with caching the hashes (`chash`), and memory authentication without caching hashes (`naive`). Results are shown for different cache sizes (256KB, 1MB, 4MB) and different cache block sizes (64B, 128B).

5.4.2 Bandwidth Pollution

Another major concern of the memory authentication scheme is the increase in the memory bandwidth usage. In the worst case, one L2 cache miss causes the entire hash hierarchy corresponding to the L2 block to be loaded from memory.

Fortunately, the simulation results in Figure 5 indicate that caching works very well for the hash tree. Figure 5 (a) shows the average number of hash blocks loaded from the main memory on a L2 cache miss. Without caching the hashes, every L2 miss causes thirteen additional memory reads for this configuration as shown by the naïve scheme. However, with caching, the number of additional memory reads is less than one for all benchmarks. As a result, the overhead of the memory bandwidth usage with caching is very small compared to the case without caching (Figure 5 (a)).

For programs that have low bandwidth usage, the increase of the bandwidth usage due to memory authentication is not a problem since loading the hashes just uses extra bandwidth. In our simulations, the bandwidth pollution is a major problem only for `mcf`, `applu`, `art`, and `swim` even though accessing hashes increases the bandwidth usage for all benchmarks.

5.5 Effects of Hash Parameters

There are two architectural parameters in our memory authentication scheme: the throughput of hash computation and the size of hash read/write buffers. This subsection studies the trade-offs in varying these parameters.

The throughput of computing hashes varies depending on how the logic is pipelined. Obviously, higher throughput is better for the performance, but requires larger space to implement. Figure 6 shows the IPC of various applications using memory authentication with caching for varying hash throughput.

As shown in the figure, having higher throughput than 3.2GB/s does not help at all. When the throughput lowers to 1.6GB/s, which is the same as memory bandwidth, we see minor performance degradation. If the hash throughput is lower than the memory bandwidth, it directly impacts and degrades the performance. In our experiments, the IPC degraded as much as 50% for `mcf`, `applu`, `art`, and `swim`. This is because the effective memory bandwidth is limited by the hash computing throughput. Therefore, the hash throughput should be slightly higher than the memory bandwidth.

Figure 7 studies the effect of the hash buffer size on the application performance (IPC). The hash read buffer holds a new L2 cache block while its hash gets

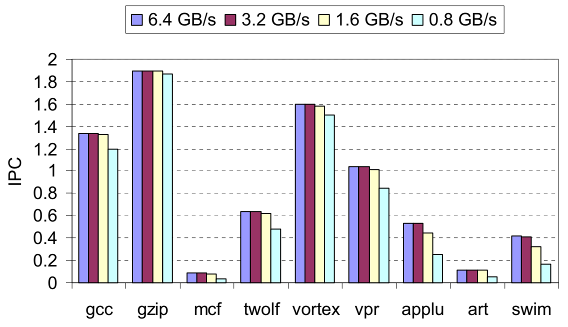


Figure 6: The effect of hash computation throughput on performance. The results are shown for a 1-MB cache with 64-B cache blocks. 6.4GB/s = one hash per 10 cycles.

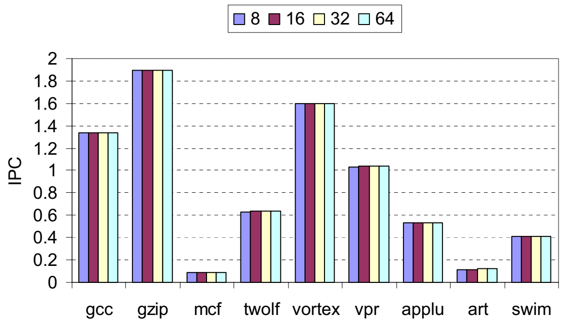


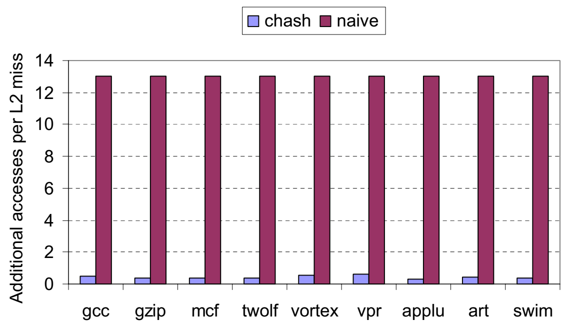
Figure 7: The effect of hash buffer size on performance. The results are shown for a 1-MB cache with 64-B cache blocks.

computed and checked with the previously stored hash. Similarly, the hash write buffer holds an evicted L2 cache block until a new hash of the block is computed and stored back in the L2 cache. A larger buffer allows more memory transactions to be outstanding. However, given the fact that the hash computation throughput is higher than the memory bandwidth, the hash buffer size does not affect the performance.

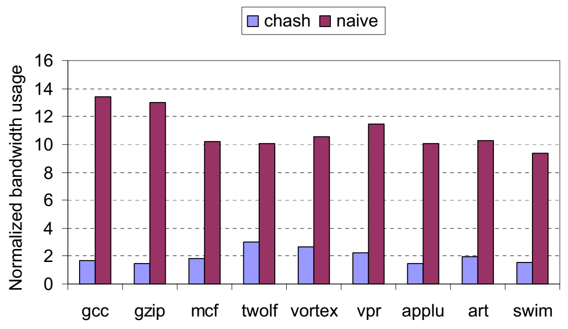
5.6 Reducing Memory Size Overhead

With one hash (128 bits) covering a 64-B cache line, 25% of main memory space is used to store hash values. This memory overhead also implies that these hash values will contend for the L2 cache space and comsume the memory bandwidth, which can result in performance degradation. Therefore, reducing memory overhead is essential to reduce the overall memory authentication overhead.

The most straightforward way to reduce mem-



(a)



(b)

Figure 5: Memory bandwidth usage for a standard processor, memory authentication with caching, without caching. The L2 cache is 1 MB with 64-B cache blocks. (a) The additional number of hash loads from memory per L2 cache miss. (b) Normalized memory bandwidth usage (normalized with `base`).

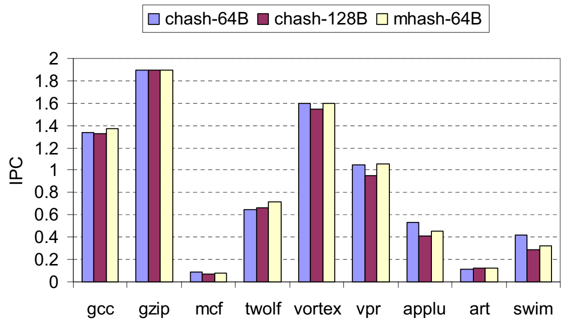


Figure 8: The performance of the **mhash** scheme with two cache blocks per chunk. The results are shown for a 1-MB cache.

ory overhead is to increase the L2 cache block size. As shown in Section 5.4, having 128-B L2 blocks rather than 64-B significantly reduces the performance degradation compared to the base case. On the other hand, large block sizes often result in poor baseline performance due to poor cache performance.

Another way to reduce the memory overhead is to make one hash cover multiple L2 cache blocks. However, in this case, all cache blocks covered by the same hash should be fetched to authenticate any one of them. Also, write back involves more memory operations. Therefore, the `mhash` scheme with 2 or more cache blocks per chunk tends to consume more bandwidth than the `chash` scheme.

Figure 8 compares the performance of using one hash per 64-B L2 block (`chash-64B`), one hash per 128-B L2 block (`chash-128B`), and one hash per two 64-B L2 blocks (`mhash-64B`). In general, `mhash-64B` performs comparable to `chash-64B`. For benchmarks with high bandwidth usage, `mhash-64B` performs worse than `chash-64B`. For benchmarks sensitive to L2 cache contention, it performs better. Therefore, the right algorithm should be chosen based on the type of main applications.

On the other hand, `mhash-64B` always outperforms `chash-128B`. Moreover, increasing the L2 cache block size can degrade performance even when the application do not use memory authentication scheme. Therefore, to reduce memory size overhead, it appears that is it always better to make one hash cover multiple cache blocks rather than increasing the cache block size.

Conclusion

We have presented a memory authentication scheme that can be used to build high performance secure computing platforms out of slightly modified general-purpose processors. By integrating the hash tree machinery with an on-chip (L2) cache, we arrived at a memory authentication scheme with reasonable overheads. The evaluations we have carried out show, for instance, that for large L2 sizes, performance overhead is $\approx 20\%$, area overhead is $\approx 10,000$ gates, and 12.5 to 25% of untrusted external memory is used up by hashes.

Ongoing work includes the investigation of off-line memory authentication schemes, and the generalization of authentication schemes to SMP systems.

Acknowledgments

We thank Chris Peikert and Ron Rivest for pointing us toward incremental hashing and cryptography. Thanks to Toliver Jue for valuable feedback.

References

[AK97] Ross Anderson and Markus Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *IWSP: International Workshop on Security Protocols, LNCS*, 1997.

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[BEG+91] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.

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

[CPL] Amy Carroll, Julia Polk, and Tony Leininger. Microsoft Palladium: A Business Overview. http://www.neowin.net/staff/users/Voodoo/Palladium_White_Paper_final.pdf.

[DEJ01] 3rd D. Eastlake and P. Jone. RFC 3174: US secure hashing algorithm 1 (SHA1), September 2001. Status: INFORMATIONAL.

[DS02] Premkumar T. Devanbu and Stuart G. Stubblebine. Stack and queue integrity on hostile platforms. *Software Engineering*, 28(1):100–108, 2002.

[GCvDD02] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Protocols and applications for controlled physical unknown functions. In *Laboratory for Computer Science Technical Report 845*, June 2002.

[Hen00] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

[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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 169–177, November 2000.

[Mer80] Ralph C. Merkle. Protocols for public key cryptography. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[MVS00] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of OSDI 2000*, 2000.

[Riv92] R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm, 1992. Status: INFORMATIONAL.

[SV01] William Shapiro and Radek Vingralek. How to Manage Persistent State in DRM Systems. In *Digital Rights Management Workshop*, pages 176–191, 2001.

[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.