Caches and Hash Trees for Efficient Memory Integrity Verification*

Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk† and Srinivas Devadas
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139, USA
{gassend,suh,declarke,marten,devadas}@mit.edu

Abstract

We study the hardware cost of implementing hash-tree based verification of untrusted external memory by a high performance processor. This verification could enable applications such as certified program execution.

A number of schemes are presented with different levels of integration between the on-processor L2 cache and the hash-tree machinery. Simulations show that for the best of our methods, the performance overhead is less than 25%, a significant decrease from the $10\times$ overhead of a naive implementation.

1. Introduction

Secure processors (e.g., [16] [15], [10]) try to provide applications running on them with a private and tamper-proof execution environment. In desktop machines they are typically present as coprocessors, and are used for a small number of security critical operations. The ability to provide the same protection for the primary processor would multiply the amount of secure computing power, making possible applications such as copy-proof software and certification that a computation was carried out correctly.

In this paper we focus on providing a tamper-proof environment for programs to run in (we do not deal with privacy of data), in particular in the case of physical attacks on the components located around the processor. For that, the main primitive that has to be developed is memory verification, to prevent a physical attacker form tampering with the system bus to change a running program's state. The processor must detect any form of memory corruption. Typically, upon detecting memory corruption the processor should abort the tasks that were tampered with to avoid producing incorrect results. For it to be worthwhile, the verification scheme must not impose too great a performance penalty on the computation, or the benefits of using the primary processor are lost.

In this paper, we describe hardware schemes to efficiently verify all or a part of untrusted external memory using a limited amount of trusted on-chip storage. Our schemes use hash trees and caches to efficiently verify memory. Naive schemes where the hash tree machinery is placed between caches, e.g., between L2 and external memory, 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 schemes, we integrate the hash 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 verification using our integrated hash tree/caching scheme is less than 25%, whereas the overhead of verification for a naive scheme can be as large as $10\times$. We show tradeoffs between external memory overhead and secure processor performance.

We describe related work in Section 2. The assumed model is presented in Section 3, and motivating applications are the subject of Section 4. An on-line caching scheme for memory verification is described in Section 5. Finally, in section 6 we evaluate the schemes on a superscalar processor simulator.

2. Related Work

In [12], hash trees were proposed as a means to update and validate data hashes efficiently by maintaining a tree of hash values over the objects.

Blum et al. addressed the problem of securing various data structures in untrusted memory. One scheme is to use

*This work was funded by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership.

†Visiting researcher from Philips Research, Prof Holstlaan 4, Eindhoven, The Netherlands.

a hash tree rooted in trusted memory [3]. This scheme has a $O(\log(N))$ cost for each memory access.

Maheshwari, Vingralek and Shapiro use hash trees to build trusted databases on top of trusted storage [11]. 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 hash 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 5 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 [14] address the problem of managing persistent state in DRM systems. Because of the large overhead of computing a MAC for each memory reference, they discount the possibility of securing volatile storage. They assume that volatile memory is inside the security perimeter.

In [7] allusions are made to a smartcard system that would use a hash 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 hash tree.

3. Model

In this paper we are considering a computer similar to a typical desktop machine. It is built around a processor with a large on-chip cache (large enough to privately perform on-chip some simple cryptographic operations). We will assume that the processor is invulnerable to physical attack, meaning that its internal state cannot be tampered with or observed. All the rest of the computer system, in particular the memory is untrusted, which means that it can be observed or modified by an adversary. The processor is also equipped with a unique secret that it will use to produce digital signatures. [1]

The objective of the system is to allow a user to perform a computation involving both the processor and external memory. During the computation, the processor can be asked to perform cryptographic primitives that involve its secret (the reason for the cryptographic primitives will be illustrated in section 4.1). The system must be able to detect with high probability if an adversary has tampered with off-chip components in a way that could compromise the result of the computation. However, this tamper detection must not impose too large a performance penalty on the computation being carried out.

The objective of the adversary is simply to tamper with off-chip devices (i.e., 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 integrity verification mechanism for the off-chip memory. In the next section we show how this integrity can be used in applications.

4. Applications

4.1. Certifying the Execution of a Program

In our model, memory verification is useful only if the processor is equipped with a secret and the ability to do some cryptography. This section illustrates why this is the case through the example of distributed computing.

Alice has a problem to solve, expressed as a program 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 program 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?

Our way of solving the problem is to have a processor that has a secret key. The corresponding public key having been published by its manufacturer.

Alice sends this processor her program. The processor combines its secret key with Alice's program through a collision resistant scheme to produce a key that is unique to the processor-program pair. If Alice ever sees data signed by this key, she will be sure that it originated from her program on Bob's processor. 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 there is no major difficulty. However, for most algorithms, Alice will need to use external memory. *How can she be sure that Bob isn't tampering with the memory bus to make her program terminate early while still producing a valid certificate for an incorrect result?* Our answer, of course, is to use memory integrity verification.

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

[1] This secret can be a private key from a public key pair as in XOM [10], or it can be a Physical Random Function [8]. Symmetric key schemes are inappropriate as we want many mutually mistrusting principals to be able to use the system.

performed correctly.[2] If her program was correct then Alice has the correct result.

It is the combination of memory verification and cryptographic signature using a secret key that make this example possible. Without the key, it would be impossible to distinguish if results were produced on a real processor or in a simulator (on which any kind of internal tampering is easy). In our model, without the ability to perform some kind of cryptography, memory verification would be useless except to detect faults in the memory, which could be detected much more cheaply with simple error detecting codes.

Of course, in real systems Bob will want to continue using his computer while Alice is calculating. In the next sections, we describe Palladium and XOM which could provide provide the framework for certified execution in multitasking systems. Both could benefit from memory verification to resist physical attacks.

4.2. Palladium

Microsoft's proposed security model, Palladium [5], may be enhanced by memory verification. Indeed, Palladium works by providing a way for applications to be executed in a secure context. However, currently, Palladium only concerns itself with enforcing protection from other software. So hardware attacks remain possible. With memory verification, applications could get guarantees that their data has not been modified, even by a physical attacker (we do not address the problem of ensuring privacy of data from a physical attacker, though).

4.3. XOM architecture

The eXecute Only Memory (XOM) architecture [10] is designed to run security requiring applications in secure compartments from which data can escape only on explicit request from the application. Even the operating system cannot violate the security model.

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.

4.3.1 Exploiting Replay Attacks

However, XOM's integrity mechanism is vulnerable to replay attacks, which was also pointed out in [14]. Indeed, in XOM there is no way to detect whether data in external memory is fresh or not.[3] An adversary can do replay attacks by having the memory return stale data that was previously stored at the same address during the same execution. In particular, XOM will not notice if only the first write to an address is ever actually performed.

This flaw in XOM's integrity checking could be exploited to violate the privacy of some programs. Consider the following example where `outputdata` copies data out of the secure compartment:

```
for (i = 0; i < size; i++)
{
  outputdata(*data++);
}
```

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. If 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 prerecorded 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 was initially intended. If `data` is stored on the stack at an address that previously contained a return address, a replay of the return address would even reveal a large portion of the program's code.

To pull this attack off, the adversary would presumably turn off caching and single step through the program (even if single stepping is forbidden, frequent interrupts are almost as good). In this way, he can observe the program's memory access patterns and 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. In fact, it might be possible to find a suitable loop simply by observing patterns of system calls. All the adversary has left to do is guess the location of `i` in the stack (the approximate position on the stack will be apparent from the memory access pattern).

Though this attack may seem involved, and the code sample is somewhat unlikely, it is quite plausible that a

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

[3] Limited freshness guarantees are provided by using a different key for each execution, but the method cannot be extended to checking the freshness of the memory.

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 [1].

4.3.2 Correcting XOM

XOM can be fixed in a simple, though not optimal, way by combining it with our memory verification method. Essentially, XOM was attempting to provide two forms of protection: protection from an untrusted OS, and protection from untrusted off-chip memory. XOM does a good job of the former, but fails on the latter. Our memory verification techniques would provide XOM with a secure foundation to work on.

5. Integrity Verification Algorithm

5.1. Hash Trees

We verify the integrity of memory with a hash tree (also called a Merkle tree, see [12]). 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.

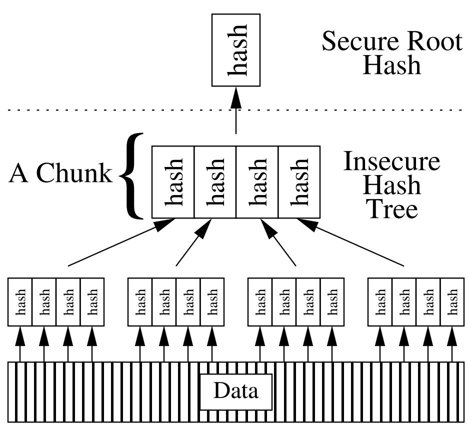


Figure 1. A hash tree. Chunks contain hashes that attest to the integrity of data in chunks lower in the tree.

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. Repeating this process recursively, we check nodes up to the root of tree. The root does not need to be checked as it is stored in secure memory. Similarly, a change to a leaf requires that all the nodes between it and the root be updated.

An m-ary hash tree allows integrity verification with a constant factor overhead in memory consumption of $1/(m-1)$. With a balanced tree, the number of hash checks per read 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 are quite modest since they allow a very small amount of secure storage (typically 128 to 160 bits) to verify the integrity of an arbitrarily large memory. For a 4-ary tree, one quarter of memory is used by hashes, which is large, but not unacceptably so. Memory bandwidth is a greater concern as it is the limiting factor in many high performance systems. For typical memory sizes there can be tens of hash reads for each data access, a sure performance killer. Therefore, we must focus on limiting the number of hash reads.

5.2. 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 verify data from the lower part of the hierarchy. Assuming a processor with on-chip L1 and L2 caches, there are two a priori places where the hash tree machinery can be situated. Between L1 and L2, or between L2 and external memory.

In either case the memory level directly below the checker sees its activity increase by an order of magnitude. Indeed, each access from above generates $\log_m(N)$ hash accesses for the level below the checker. In Section 6 we use the scheme where the machinery is between L2 and external memory as a representative naive scheme, and refer to it as `naive`. The following section shows an optimized hash tree implementation that integrates hashing with caching to reduce the gap between memory checking and performance.

5.3. Making Hash Trees Fast: `chash`

To make hash trees fast, we merge the hash tree machinery with one of the cache levels. Values that are stored in this cache are trusted, which allows accesses to be performed directly on cached values without any hash operations. At the same time hash accesses can now be directed to the same cache. This reduces the latency to the hash data. But the major advantage is that hash accesses to the cache do not immediately generate other hash operations, so a cache miss on a leaf no longer systematically leads to $log_m(N)$ hash operations.

The following algorithms show how the integrated cache can be implemented. In these algorithms the word *cache* refers to the integrated cache (which is assumed to be trusted), and the word *memory* refers to the next level in the memory hierarchy.[4] The memory is divided into *chunks* that are the basic unit that hashes are computed on. For now we will consider that chunks coincide with cache blocks.

ReadAndCheckChunk: Reads data out of 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 the topmost 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.

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 topmost chunk, in which case the hash is stored in secure memory).

[4]We will work with L2 as the cache, and off-chip RAM as the 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 new tree is smaller which reduces the cost of subsequent accesses to it. As far as correctness goes, the algorithm's essential invariant is that at any time, nodes contain hashes of their children as they are in memory.[5] 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 from 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.

5.4. Multiple Cache Blocks per Chunk: `mhash`

In the `chash` algorithm, we assume that there is exactly one cache block per chunk. Since performance when memory verification is off is the utmost priority, the cache block is usually chosen to optimize the performance of the processor in that case. Consequently, the cache block size is completely constrained before memory integrity verification performance 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 the topmost 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.

[5]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.

ReadAndCheck

1. If the data is cached, return the cached data. We are done.
2. Call ReadAndCheckChunk on the data's chunk.
3. Put uncached blocks of the the read chunk into the cache.
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 topmost chunk, in which case the hash is stored in secure memory).

5.5. Incremental Hashing : `ihash`

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 [2], it is based on a conventional MAC function h_k, an encryption function $E_{k'}$ and the XOR operator $\oplus$:

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

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 (we don't need to check this value so we can avoid reading 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 topmost chunk, in which case the hash is stored in secure memory).

As it is, the scheme that is presented above is incorrect because in step 2, we do not check the old value of the block that we read from memory. There are two possible attacks.

Suppose value d_o is replaced by d_n at some address, the value read from memory in step 2 is d'_o, and the following read operation to that address returns d'_n (if the memory is correct primed values should equal unprimed ones). The check that is performed on the next memory read compares $h_k(i, d_o) \oplus h_k(i, d'_o) \oplus h_k(i, d_n)$ with $h_k(i, d'_n)$ (we have omitted all the terms in the sum for indices other than i because the difference in index prevents the terms from interacting). If the memory performs correctly, then the d_o terms cancel, and the d_n terms match. All is well. Unfortunately, there are other terms that might cancel out. The check passes if $d_n = d'_o$ and $d'_n = d_o$, which means that the adversary can leave the old value in the memory by correctly predicting the new value. The check also passes if $d_n = d_o$ and $d'_n = d'_o$, which means that if the value at the address was in fact unchanged, the adversary can store a value of his choosing in the memory.

Both of these issues can be fixed by storing a one-bit timestamp for each cache block along with the MAC in the parent chunk. This timestamp flips each time the cache block is written back, and is incorporated in the MAC (replace $h_k(i, m_i)$ by $h_k(i, m_i, b_i)$ where b_i is the timestamp). The two attacks are defeated because the timestamp prevents d_n terms from being identical to d_o terms. A one bit time stamp is sufficient because once a cache block has been written back our algorithm always makes it undergo a read with a check before it is dirtied again, so the accumulation of unchecked blocks can never be worse than in the scenario we studied above.

5.6. Simplified Memory Organization

We have chosen to adopt a very simple memory organization in which we want to verify a contiguous segment of memory starting at address 0. While this assumption is quite restrictive as far as real systems go, it allows us to study the performance of our schemes without going into the details of a particular memory organization.

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.

In this paper, we assume that the cache is physically tagged, and that we are doing verification of physical memory. This model is well suited to Palladium. For XOM where an untrusted operating system is responsible for virtual memory management, it could be desirable to do virtual memory verification instead. In that case ensuring correctness when multiple applications have data in the cache is a difficult problem that has yet to be studied in detail.

5.7. Real Life Issues

5.7.1 Direct Memory Access

New problems appear if we want to allow devices to write to protected regions of memory through Direct Memory Access (DMA). Indeed, since the processor is not involved in the transfer, the hash tree does not get updated to reflect the new data. This is in fact the desired behavior as the data has an untrusted origin.

There are two ways of dealing with the difficulty. Marking a subtree of the hash tree as unprotected, doing the DMA transfer, and then rebuilding the relevant part of the tree; or doing the DMA transfer into unprotected memory, and then copying it into protected memory. Inevitably, all the data has to be processed by the processor before it is protected by the hash tree. Finally, once the data is protected, its integrity must be checked by some scheme of the application program's choosing.

Note that 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 unprotected data when it expects protected data.

5.7.2 Initialization

So far we have considered how the processor executes when memory is being verified. 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 the hashing algorithm for writes but not for reads. In this mode hash trees will be computed, but no exceptions are raised.
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.[6] [7]
4. Turn on the memory verification failure exceptions.
5. Generate the key that will be used by this program for cryptographic purposes (see Section 4.1).

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.

5.8. Precise Exceptions

If tampering with memory is ever detected, an exception is raised. Since this exception should only occur when someone is physically attacking the machine, graceful recovery is not needed. Therefore this exception need not be precise. Consequently, it is possible to commit instructions even if they speculatively used data that is still being checked in the background.

The only exception is for instructions that involve the processor's secret. 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 signs it, and get the signature off-chip before the tampering has been detected.

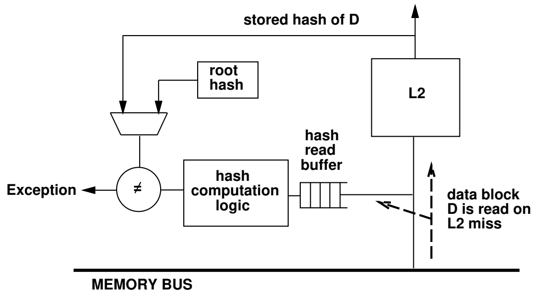
Therefore, cryptographic instructions must act as barriers, and only commit once the checks for all the instructions that precede them have completed.

6. Evaluation

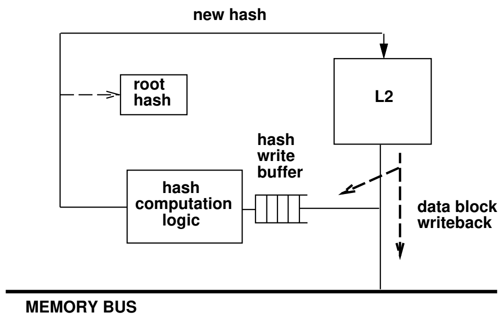
This section evaluates our memory verification schemes using a processor simulator.

[6] 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.

[7] In the case of `mhash`, 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.



(a)



(b)

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

6.1. Hardware Implementation

We describe the implementation of the `chash` scheme. The `mhash` and `ihash` schemes use the same datapaths but require 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 integrity (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 the hash unit computes a new hash and stores it back into the L2 cache (Figure 2 (b)).

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.

6.2. Logic Overhead

To evaluate the cost of computing hashes, we considered the MD5 [13] (and SHA-1 [6]) hashing algorithms. The core of each algorithm is an operation that takes a 512-bit block, and produces a 128-bit (or 160-bit, respectively) digest.[8]

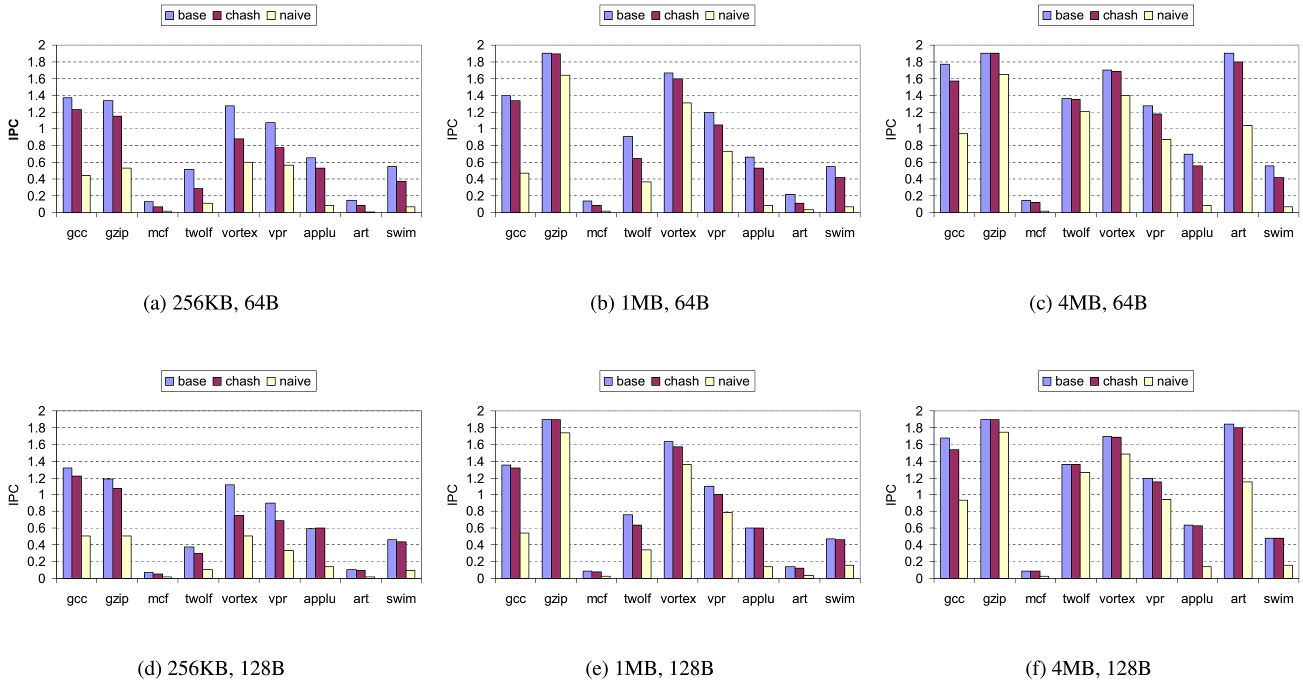
In each case, simple 32-bit operations are performed over 80 rounds. In each round there are 2 to 4 logic levels, as well as 2 adders. We assumed that with suitable skewing of the adders, rounds can be performed in one cycle per round on average. We now believe that this evaluation is optimistic. This is a minor point as longer latency implementations could be accommodated with no change in performance by adding a proportional number of entries in the hash buffers.

The total number 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.

6.3. Simulation Framework

Our simulation framework is based on the SimpleScalar tool set [4], which models speculative out-of-order execu-

[8] 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.



(a) 256KB, 64B

(b) 1MB, 64B

(c) 4MB, 64B

(d) 256KB, 128B

(e) 1MB, 128B

(f) 4MB, 128B

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

tion. 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.

For all the experiments in this section, nine SPEC2000 CPU benchmarks [9] 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 verification 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-state performance. We also ignore the overhead of stalling due to cryptographic instructions (see section 5.8), as these operations are very infrequent. In distributed computation, for example, they will only be used every few seconds or even minutes.

6.4. Performance Impact of Memory Verification

Hash-tree based memory verification 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 verification implies even more work for memory operations, which already are rather expensive. Therefore, the obvious first concern with memory verification 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 integrity in the background. Checking the integrity of data hurts memory latency only when read/write buffers are full.

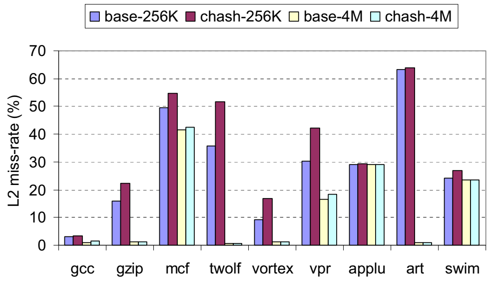


Figure 4. L2 cache miss-rates of program data for a standard processor (`base`) and memory verification with caching (`chash`). The results are shown for 256-KB and 4-MB caches with 64-B cache blocks.

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

The figure first demonstrates that the performance overhead of memory verification can be surprisingly low if we cache hashes. Even though the on-line memory verification 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 `naive` 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.

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 verification 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 verification 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 discuss the performance considerations of memory verification in more detail.

6.4.1 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.

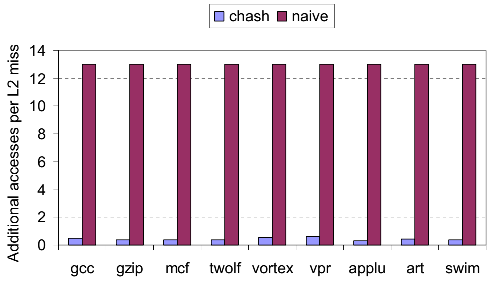
The effect of cache contention is studied in Figure 4. The figure depicts the L2 miss-rates of the baseline case and memory verification with caching. As shown, for a small L2 cache, the miss-rate can be noticeably increased by caching the hashes. In fact, cache contention is the major source of performance 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).

6.4.2 Bandwidth Pollution

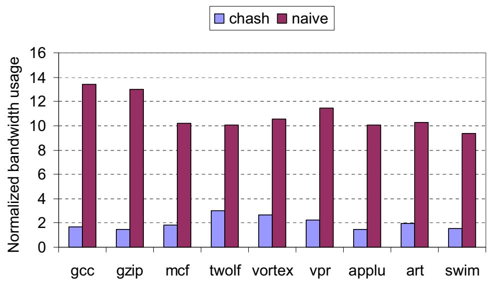
Another major concern of the memory verification scheme is the increase in the memory bandwidth usage. In the worst case, one L2 cache miss causes the entire hash hierarchy corresponding to the L2 block to be loaded from memory.

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 naive 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 with low bandwidth usage, the increase of the bandwidth usage due to memory checking is not a problem since loading the hashes just uses excess bandwidth. In our simulations, bandwidth pollution is a major problem only for `mcf`, `applu`, `art`, and `swim` even though accessing hashes consumes bandwidth for all benchmarks.



(a)



(b)

Figure 5. Memory bandwidth usage for a standard processor, memory verification 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`).

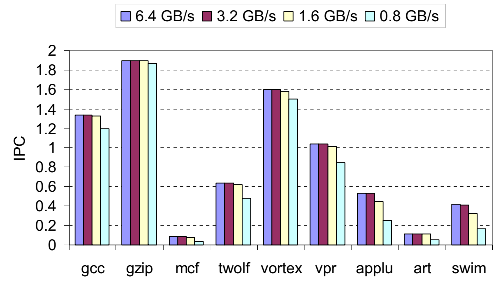


Figure 6. The effect of hash computation throughput on performance. The results are shown for a 1-MB cache with 64-B cache blocks. 6.4GB/s = one hash per 10 cycles.

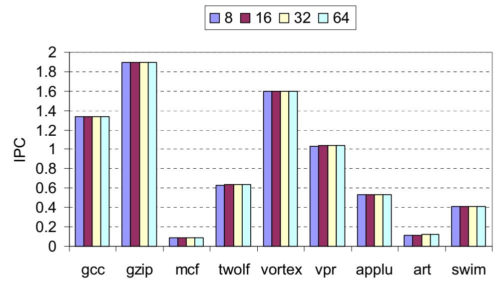


Figure 7. The effect of hash buffer size on performance. The results are shown for a 1-MB cache with 64-B cache blocks.

6.5. Effects of Hash Parameters

There are two architectural parameters in our memory verification 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 verification 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`, for a hash throughput of 0.8GB/s. 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 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.

6.6. 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.

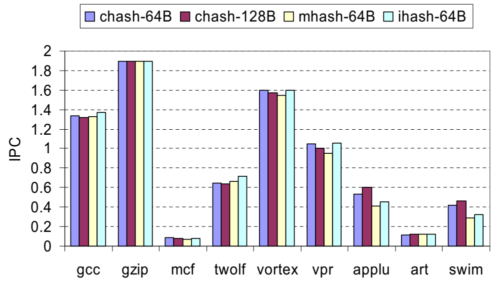


Figure 8. The performance of the `mhash` and `ihash` schemes with two cache blocks per chunk. The results are shown for a 1-MB cache.

In addition to wasting memory, these hash values can hurt performance by contending for L2 cache space and consuming memory bandwidth. Therefore we should be concerned with reducing memory overhead.

The simplest way to reduce memory overhead is to increase the L2 cache block size. As shown in Section 6.4, having 128-B L2 blocks rather than 64-B significantly reduces the performance degradation compared to the base case. However, large block sizes often result in poor baseline performance due to poor cache performance.

The other 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 verify any one of them. Also, write back involves more memory operations. Therefore, the `mhash` and `ihash` schemes 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`), one hash per two 64-B L2 blocks (`mhash-64B`) and one hash per two 64-B L2 blocks using incremental cryptography (`ihash-64B`).

Among the schemes with reduced memory overhead, `chash-128B` has the best performance, but would hurt baseline performance. Of the two remaining candidates, `ihash-64B` has the better performance, as expected. In fact it performs comparably to `chash-64B`, except for the benchmarks with the highest bandwidth usage.

7 Conclusion

In this paper we have shown how hash trees can be used to make general purpose computers capable of certified execution, by providing integrity verification of memory. The performance penalty of verifying memory contents turns out to be only $\approx 20\%$, so contrarily to what was previously assumed, it is not an unreasonable proposition. The good performance is achieved through careful integration of the hash tree machinery with the on chip (L2) cache.

8. Acknowledgments

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

References

[1] R. Anderson and M. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *IWSP: International Workshop on Security Protocols*, volume 1361 of *LNCS*, pages 124–142. Springer-Verlag, April 1997.

[2] 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.

[3] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.

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

[5] A. Carroll, J. Polk, and T. Leininger. Microsoft Palladium: A Business Overview. http://www.neowin.net/staff/users/Voodoo/Palladium_White_Paper_final.pdf.

[6] D. Eastlake, 3^{rd} and P. Jone. RFC 3174: US secure hashing algorithm 1, Sept. 2001. Status: INFORMATIONAL.

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

[8] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled Physical Random Functions. In *Proceedings of the 18th Annual Computer Security Conference*, December 2002.

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

[10] D. Lie *et al.* Architectural Support for Copy and Tamper Resistant Software. In *ASPLOS-IX*, pages 169–177, November 2000.

[11] U. Maheshwari, R. Vingralek, and W. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of OSDI 2000*, pages 135–150, 2000.

[12] R. C. Merkle. Protocols for public key cryptography. In *IEEE Symp. on Security and Privacy*, pages 122–134, 1980.

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

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

[15] 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.

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