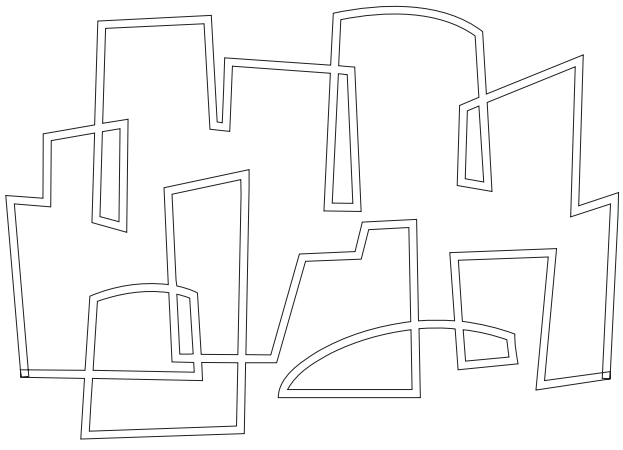
Hardware Mechanisms for Memory Authentication

Edward Suh, Dwaine Clarke, Blaise Gassend,
Marten van Dijk, Srini Devadas

2002, November

Computation Structures Group
Memo 460



Hardware Mechanisms for Memory Integrity Checking

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

Abstract

Memory integrity verification is a useful primitive when implementing secure processors that are resistant to attacks on hardware components. This paper proposes new hardware schemes to verify the integrity of untrusted external memory using a very small amount of trusted on-chip storage. Our schemes maintain incremental multiset hashes of all memory reads and writes at run-time, and can verify a *sequence* of memory operations at a later time. We study the advantages and disadvantages of the two new schemes and two existing integrity checking schemes, MACs and hash trees, when implemented in hardware in a microprocessor. Simulations show that the new schemes outperform existing schemes of equivalent functionality when integrity verification is infrequent.

1 Introduction

Secure processors (e.g., [19] [18], [9]) 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 attacks on the components located around the processor. For that, the main primitive that has to be developed is memory verification, to prevent a attacker from tampering with the off-chip memory 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 new hardware schemes to efficiently verify all or a part of untrusted external memory using a limited amount of trusted on-chip storage. Our schemes maintain incremental multiset hashes of all memory reads and writes at run-time, and verify a *sequence* of memory operations at a chosen later point of time. We study the advantages and disadvantages of our two new schemes and two existing integrity checking schemes, MACs and hash trees, when viewed as hardware mechanisms in a microprocessor. We also describe how memory integrity checking schemes can be integrated into a multitasking environment with mutually mistrusting processes.

Simulations show that our new schemes outperform two existing schemes when integrity verification is infrequent. In these cases, the performance overhead of our schemes is less than 5% in most cases and 15% in the worst case. On the other hand, the hash tree scheme is the best choice for data integrity checking when each memory operation needs to be verified before continuing the execution. The hash tree scheme has less than 25% overhead for many cases, but may cause more than 50% degradation when on-chip caches are small. MAC'ing data blocks has very low overhead even for frequent checks, comparable to our new schemes with infreqeuent checks. However, it is only secure for static instruction verification.

The assumed model is presented in Section 2, and motivating applications are the subject of Section 3. Existing mechanisms for memory verification and our new schemes are described in Section 4. Memory checking in multi-process environments is discussed in Section 5. We compare the various schemes in in Section 6. In section 7 we evaluate the schemes on a superscalar processor simulator. We discuss related work and conclude the paper in Section 8.

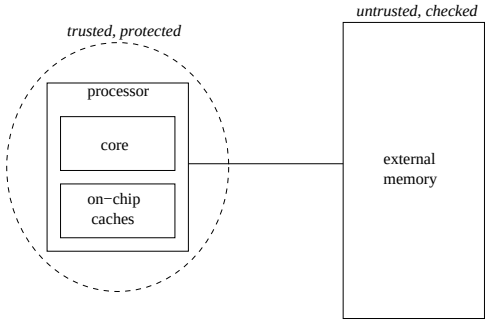


Figure 1: Model.

2 Model

In this paper, we consider a system that is built around a microprocessor with external memory. Figure 1 illustrates the general model. The processor's core is trusted, and we assume that the processor is invulnerable to physical attack, meaning that its internal state cannot be tampered with or observed. External memory is untrusted. The objective of an adversary is to alter the contents of external memory in such a way that the system produces an incorrect result that looks correct to the system user.

There are two possibilities. In the first case, the processor, including its on-chip caches, and some core functionality of the operating system is trusted. In this case, trusted core software manages virtual memory and therefore we need to check the operations on the physical address space. In the second case, just the processing core is trusted. Everything else, including the operating system and the external memory, is untrusted. In this case, we protect virtual memory with the integrity checking mechanisms. Section 5 studies the two cases.

In both cases, the adversary can attack off-chip memory, and the processor needs to check that it behaves like valid memory. *Memory behaves like valid memory if the value the processor loads from a particular address is the most recent value that it stored to that address.* If the contents of the off-chip memory have been altered by an adversary, the memory may not behave correctly (like valid memory). Each scheme in this paper allows the processor to detect, with high probability, if such tampering has occurred. The probability is high, irrespective of whichever programs the processor executes. If tampering is detected, the processor raises an integrity exception.

The processor can contain a secret that allows it to produce keys to perform cryptographic operations, such as signing, that no other processor could do for it. This secret can be a private key from a public key pair, as in XOM [9]. If the processor has ascertained that the program it has run was executed in an authentic manner, it can use the key to generate a certificate. The certificate is used to prove to some entity that the program's execution was not tampered with while it ran on the processor, and that the program produced a particular set of results when run on the processor.

3 Applications

3.1 Certifying a Program Execution

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 private key. The corresponding public key is published by the processor's manufacturer. Alice sends the processor her program. 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. It then uses its key to create a certificate with a hash of the program and the program's result, and sends the certificate to Alice. When Alice verifies the certificate, Alice has certain assurances that the program was executed by the processor in an authentic manner.

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 performed correctly.

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 the framework for certified execution in multitasking systems. Both could benefit from memory verification to resist physical attacks.

3.2 Palladium

Microsoft's proposed security model, Palladium [3], 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. (While we do not address the problem of ensuring privacy of data from a physical attacker in this paper, encrypting the data that goes off-chip can keep the data in memory secret.)

3.3 XOM architecture

The eXecute Only Memory (XOM) architecture [9] 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 keep the data secret. 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.

However, XOM's integrity mechanism is vulnerable to replay attacks, which was also pointed out in [17]. Indeed, in XOM there is no way to detect whether data in external memory is fresh or not.[1] 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.

XOM can be fixed in a simple way by combining it with memory verification. XOM provides protection from an untrusted OS, and memory verification will provide protection from untrusted off-chip memory.

[1] 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.

4 Integrity Checking Methods

Section 4.1 summarizes two conventional techniques that can be used to check the memory integrity of microprocessors. Section 4.2 introduces new schemes which have a performance advantage over conventional methods as we shall see in Section 7.

In our description of algorithms, we use a term *chunk* as the minimum memory block that is read from and written to memory for integrity checking. If a word within a chunk is accessed by a processor, the entire chunk is brought into the processor. In the simplest instantiation, a chunk can be a L2 cache block.

4.1 Conventional Techniques

4.1.1 Message Authentication Code: `MAC`

A hash of a message is a fixed length cryptographic[2] fingerprint of the message. A message authentication code (MAC) is a hash computed over the message using a secret key and attached to the message, which often used to authenticate a message. Later, a receiver recomputes the MAC of the received message and compares it with the attached MAC. If it is equal to the attached MAC the receiver knows that the message it received is authentic, that is, is the original message sent by the sender.

The idea can be simply extended to memory integrity checking for static data, like most instructions, in processors. We divide the memory space into multiple *chunks*. A processor contains a secret key on-chip, and associates a MAC for each chunk. When the processor reads a block from the memory, it reads the entire chunk that the block belongs to and recomputes the MAC of the loaded chunk and compares this with the MAC stored in the memory. To prevent an adversary from copying content at one chunk to another chunk, the MAC is computed over the chunk in combination with its address. For this reason we call this scheme the addressed MAC (`MAC`) scheme.

4.1.2 Cached Hash Tree: `CHTree`

Unfortunately, `MAC` cannot be used to check the integrity of dynamically changing data because it is vulnerable to replay attacks. The valid MACs guarantee that a chunk is stored by the processor, but do not guarantee that it is the most recent copy. For this reason, `MAC` can only be applied to instructions,

[2] It is hard to find two distinct messages with the same hash. This property is called collision-resistance.

which are static for a program, not to data, which dynamically changes.[3]

Hash trees (or Merkle trees) are often used to verify the integrity of dynamic data in untrusted storage [12]. Figure 2 illustrates a hash tree. Similar to `MAC`, the memory space is divided into multiple chunks, denoted by V_1, V_2, etc. The chunks are the leaves of the hash tree. A parent is the hash of the concatenation of its children. The root of the tree is stored on-chip where it cannot be tampered with.

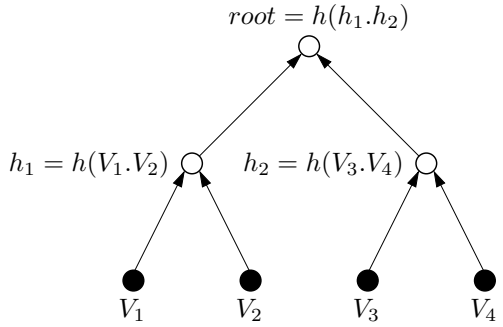


Figure 2: A binary hash ($m = 2$) hash tree. Each internal node is a hash of the concatenation of the data in the node's children.

To check the integrity of a node in the tree, the processor (i) reads the node and its siblings from the memory, (ii) concatenates their data together, (iii) computes the hash of the concatenated data, and (iv) checks that the resultant hash matches the hash in the parent. The steps are repeated all the way to the root of the tree.

To update a node, the processor checks its integrity as described in the previous paragraph while it (i) modifies the node, and (ii) recomputes and updates the parent to be the hash of the concatenation of the node and its siblings. These steps are repeated to update the whole path from the node to the root, including the root.

Hash trees allow dynamically changing data in an arbitrarily large storage to be verified and updated with one small root hash on-chip (128 bits for MD5 [15], 160 bits for SHA-1 [14]). With a balanced m-ary tree, the number of nodes to check on each memory access is $log_m(N)$, where N is the number of chunks to be verified. The logarithmic overhead of using the hash tree can be significant. For example, [7] showed that applying the hash tree to a processor can slow down the system by as much as factor of ten. The experiments used 4-GB memory and 128 bit hashes.

[3]Even if the instructions do not change for a program, we need to make sure that the MACs are computed in a different way for each program to prevent copying instructions from one program to another. To achieve this, we make the secret key a function of the processor's key and a hash of the program.

The performance overhead of using a hash tree can be dramatically reduced by caching the internal hash nodes on-chip with regular data. The processor trusts data stored in the cache, and can perform memory accesses directly on them without any hashing. Therefore, instead of checking the entire path from the chunk to the root of the tree, the processor checks the path from the chunk to the first hash it finds in the cache. This hash is trusted and the processor can stop checking. When a chunk or hash is ejected from the cache, the processor brings its parent into the cache (if it is not already there), and updates the parent in the cache.

Previous work [7] showed that `CHTree` clearly outperforms the hash tree in all cases. The performance overhead of `CHTree` is often less then 25%. Therefore, in this paper, we only use `CHTree` for comparison. Also, for simplicity, we make the chunks the same as the L2 cache blocks. (For more details and variants of `CHTree`, see [7].)

4.2 Log Hash Integrity Checking

`CHTree` checks the integrity of memory after every processor memory access. However, checking the integrity of every access implies unnecessary overhead when we are only interested in the integrity of a *sequence* of memory operations. For example, in the certified execution application, knowing exactly which operation has failed is not useful.

We introduce a new approach of verifying memory integrity with low run-time overhead. The approach is based on the work presented by Blum et. al [1] on memory correctness checking; we have extended it to be implemented with collision-resistant multiset[4] hash functions (which we will describe shortly), and to incorporate caches.

Intuitively, the processor maintains a read log and a write log of all of its operations to off-chip memory. At runtime, the processor updates logs with minimal overhead so that it can verify the integrity of a *sequence* of operations at a later time. To maintain the logs in a small fixed amount of trusted on-chip storage, the processor uses multiset hash functions. When the processor needs to check its operations, it performs a separate *integrity-check* operation using the trusted state.

A multiset hash maps multisets into a small fixed-sized bit string. It is incremental in that it is efficient to update it when a new element is added to the multiset. We use `MSet-XOR MAC` [4] based on the hash function MD5. `MSet-XOR MAC` requires one MD5 operation using a secret key in the processor, and one

[4]A multiset is an unordered group of elements where an element can occur as a member more than once [16].

XOR operation to update the multiset hash incrementally. `MSet-XOR MAC` is set-collision resistant in that it is hard to find a set and a multiset which produces the same hash. For our purpose, since the multiset hashes are used to maintain logs, we refer to them as log-hashes, and refer to our schemes as log-hash schemes.

4.2.1 Baseline Algorithm: `LHash`

Figure 3 shows the steps of the baseline Log Hash (`LHash`) integrity checking scheme. We describe the operations assuming that the chunk is the same as a L2 cache block and the cache is write-allocate. To verify a sequence of memory operations, the processor keeps two log hashes (ReadHash and WriteHash) and a counter (Timer) in trusted on-chip storage. ReadHash maintains information of data read from memory, and WriteHash maintains information of data written to memory. Because our log hashes maintains set information, Timer is used to mark the order of memory operations. We denote the (ReadHash, WriteHash, Timer) tuple as the object $\mathcal{T}$.

Whenever there is a new set of chunks of memory that need to have their integrity verified, the processor performs an add-chunks operation to add it to WriteHash. This operation effectively remembers the initial value of the chunks in WriteHash. For example, if the scheme is verifying a virtual memory space, whenever a new page is allocated, all of the chunks in the page are initialized using add-chunks. If the scheme verifies the entire physical memory, the chunks in the entire memory are initialized at boot time.

At runtime, the processor calls read-chunk and write-chunk to properly update the logs as it reads and writes chunks. When a chunk gets evicted from the cache, the processor logs the evicted chunk's value by calling write-chunk. The chunk is associated with a new time stamp by incrementing Timer and using the new value. WriteHash is updated with the hash of the corresponding address-chunk-time stamp triple. If the chunk is dirty, the chunk and the time stamp are written back to memory; if the chunk is clean, only the time stamp is written back to memory.

The processor calls read-chunk to bring a chunk from the cache. The time stamp associated with the chunk is checked to be less than or equal to the current value of Timer. Because the processor only maintains hashes, the time stamps are used to ensure that the chunk the processor reads from memory is the most recent chunk it stored to memory and that its memory accesses are not being reordered by an adversary (we refer to [4] for a detailed argument).

Initialization Operation
add-chunks($\mathcal{T}$, set of `Address-Chunk` pairs):

1. Increment $\mathcal{T}$.Timer. `TimeStamp` = $\mathcal{T}$.Timer.
2. For each pair:
   (a) Store (`Chunk`, `TimeStamp`) at address, `Address`, in memory.
   (b) Update $\mathcal{T}$.WriteHash with the hash of (`Address`·`Chunk`·`TimeStamp`).

Run-Time Operations

- For a cache eviction
  write-chunk($\mathcal{T}$, `Address`, `Chunk`):
  1. Increment $\mathcal{T}$.Timer. `TimeStamp` = $\mathcal{T}$.Timer.
  2. Update $\mathcal{T}$.WriteHash with the hash of (`Address` · `Chunk` · $\mathcal{T}$.TimeStamp).
  3. If a block is dirty, write `(Chunk, Timer)` back to memory. If the block is clean, only write `Timer` back to memory (we do not need to write `Chunk` back to memory).
- For a cache miss, do read-chunk($\mathcal{T}$, `Address`):
  1. Read the `(Chunk, TimeStamp)` pair from `Address` in memory and bring `Chunk` into the cache.
  2. If `TimeStamp` > $\mathcal{T}$.Timer, raise an integrity exception.
  3. Update $\mathcal{T}$.ReadHash with the hash of (`Address` · `Chunk` · `TimeStamp`).

  and store `Chunk` in cache.

Integrity Check Operation
integrity-check($\mathcal{T}$):

1. `New`$\mathcal{T}$ = (0, 0, 0).
2. For each chunk address covered by $\mathcal{T}$, check if the chunk is in the cache. If it is not in the cache,
   (a) read-chunk($\mathcal{T}$, address).
   (b) add-chunks(`New`$\mathcal{T}$, address, chunk), where chunk is the chunk read from memory in Step 2a.
3. Compare ReadHash and WriteHash. If different, raise an integrity exception.
4. If the check passes, $\mathcal{T}$ = `New`$\mathcal{T}$.

Figure 3: `LHash` Integrity Checking Algorithm.

ReadHash is updated with the hash of the address-chunk-timer triple.

The WriteHash maintains information on the chunks that, according to the processor, should be in memory at any given point in time. The ReadHash maintains information on the chunks the processor reads from memory. During runtime, compared to ReadHash, WriteHash is updated once more per address. Therefore, to check the integrity of operations, all addresses covered by $\mathcal{T}$ are read and ReadHash gets updated accordingly. If ReadHash is equal to WriteHash, and assuming that all of the time checks passed, then the memory was behaving correctly during the processor's sequence of operations. This checking is done in the integrity-check operation.

The processor performs an integrity-check operation when a program needs to check a sequence of operations, or when Timer is near its maximum value. Unless the check is at the end of a program execution, the processor will need to continue memory verification after an integrity-check operation. To do this, the processor initializes a new WriteHash while it reads memory during an integrity-check. If the integrity check passes, WriteHash is set to the new WriteHash, and ReadHash and Timer are reset. The program can then continue execution as before.

To avoid reading the entire virtual or physical memory space on a integrity-check operation, the processor can incrementally add chunks on demand and use a table to maintain the list of chunks ever touched. For example, the processor can use the program's page table to keep track of which pages it used during the program's execution. When there is a new page allocated, the processor calls add-chunks for all chunks in the page. When the processor performs an integrity-check operation, it walks through the page table in an incremental way and reads all chunks in a valid page.

In this scheme, the page table does not need to be trusted. If an adversary changes the page table so that the processor initializes the same chunk multiple times or skip some chunks during the check operation, the integrity check will fail in that ReadHash would not be equal to WriteHash.

In our description and implementation, we have used two log hashes, WriteHash and ReadHash. It is possible to implement the scheme using one log hash, RWHash. When a chunk is evicted from the cache, the hash of its corresponding triple is 'added' to RWHash; when a chunk is brought into the cache, the hash of its corresponding triple is 'subtracted' from RWHash. In essence, RWHash is the difference of WriteHash and ReadHash. If at the end of the integrity-check operation, LHash is equal to 0, the integrity check is successful.

4.2.2 Making Checks Fast: `H-LHash`

In `LHash`, the $\mathcal{T}$ stored in the processor covers all of the memory space to be verified. As a result, every integrity-check operation involves reading all of the chunks in this memory space. Although the run-time overhead of the scheme is minimal, the overhead of this integrity check can be significant. We propose a hierarchical scheme (`H-LHash`) where the processor only needs to read the subset of the memory space touched since the last integrity-check operation when performing an integrity check.

In the hierarchical scheme, the memory space is divided into multiple *subspaces*. A subspace is a block of memory that is verified by one $\mathcal{T}$, (ReadHash, WriteHash, Timer) tuple. For example, the subspace for the `LHash` scheme is the entire memory space to be verified. The `H-LHash` scheme can be considered as a hash tree where each node is a $\mathcal{T}$ that verifies its children using the `LHash` scheme. Similar to the hash tree scheme (Figure 2), chunks at the leaves form a data subspace. The children of each internal node form a subspace of chunks containing $\mathcal{T}$s. For example, in Figure 2, V_1 and V_2 form a data subspace that is verified by h_1. h_1 and h_2 form a $\mathcal{T}$ subspace that is verified by *root*. In this way, each parent is a log hash of its children.

Our hierarchical scheme reduces the overhead of traversing the entire path from the leaf to the root using the cache in a manner similar to that in `CHTree`. When a chunk is read from or written to the memory, the processor needs to properly update ReadHash or WriteHash. If its parent $\mathcal{T}$ is in the cache, it is trusted and can be updated using read-chunk or write-chunk. On the other hand, if the parent is not in the cache, it is brought into the cache and the grandparent $\mathcal{T}$ is updated. Unlike the `CHTree` scheme, this scheme does not require reading all the siblings to update the parent. Therefore, it is possible to have a wide tree structure without incurring significant overhead [5]. Refer to Figure 4 for detailed algorithms.

During the integrity check operation, the tree is traversed in a depth first manner. However, the search *only traverses children for which the corresponding* ReadHash *is not equal to 0*; in other words it only traverses children which have been updated since the last integrity check. This means that only data chunks in the subspaces which have been used since the last integrity check will be read during the current check. Compared to `LHash`, using the tree increases the cost at run time only slightly provided the

[5] In the `H-LHash` scheme, $\mathcal{T}$ verifies a subspace that can contain many chunks, whereas each hash verifies one chunk in the `CHTree` scheme.

Get Parent operation
get-parent(Address):

1. addr = Parent(Address).
2. If $\mathcal{T}_{parent}$ stored at addr is not in cache, and is not the root, then
   (a) $\mathcal{T}_{grandparent}$ = get-parent(addr).
   (b) Get $\mathcal{T}_{parent}$ from the chunk read in read-chunk($\mathcal{T}_{grandparent}$, addr).
3. If $\mathcal{T}_{parent}$ is not the root, store $\mathcal{T}_{parent}$ in the cache.
4. Return $\mathcal{T}_{parent}$.

Initialization Operation
hier-add-chunks(set of Address-Chunk pairs):

1. $\mathcal{T}_{parent}$ = get-parent(Address).
2. add-chunks($\mathcal{T}_{parent}$, set of Address-Chunk pairs).

Run-Time Operations

- For a cache eviction
  hier-write-chunk(Address, Chunk):
  1. $\mathcal{T}_{parent}$ = get-parent(Address).
  2. write-chunk($\mathcal{T}_{parent}$, Address, Chunk)
- For a cache miss, do
  hier-read-chunk(Address):
  1. $\mathcal{T}_{parent}$ = get-parent(Address).
  2. read-chunk($\mathcal{T}_{parent}$, Address).

  and store Chunk read in step 2 in the cache.

Integrity Check Operation
hier-integrity-check($\mathcal{T}$):
Just like integrity-check($\mathcal{T}$), with the extra step:

2(c) For each $\mathcal{T}_{child}$ in chunk read in step 2(a), if $\mathcal{T}_{child}$.READHASH$\neq$0, then hier-integrity-check($\mathcal{T}_{child}$).

Figure 4: H-LHash Integrity Checking Algorithm

tree is not deep, but the cost of regular intermediate integrity checks can decrease significantly.

4.3 Implementation

4.3.1 Memory Layout

To implement the memory checking schemes, the layout of data, hashes, and time stamps should be determined. The layout should be simple enough for hardware to easily compute the address of the corresponding hash or time stamp from the address of a data chunk. We give an example layout for the H-LHash scheme where we use the top of the memory space for tree nodes and the bottom of the space for time stamps.

Nodes ($\mathcal{T}$ for H-LHash) in a tree are linearly laid out at the top of the memory space in a breadth-first manner. The root is at address 0. The second level nodes follow, and so on. The address of a parent node can be easily found from the child address by

$$Parent(Addr) = \frac{Addr - B_{Node}}{B_{SubSpace}} \times B_{Node}.$$

B_{Node} is the size of a node ($\mathcal{T}$) in bytes, and $B_{SubSpace}$ is the size of the memory space covered by each node. For simplicity, we choose B_{Node} and $B_{SubSpace}$ to be a power of two.

Time stamps are laid out linearly at the bottom of the memory space starting at TS_{Base}. Therefore, the address of a time stamp can be computed by

$$TimeStampAddr = TS_{Base} + \frac{Addr - B_{Node}}{B_{Chunk}} \times B_{TS}.$$

B_{Chunk} is the chunk size, which is a multiple of B_{Node}, and B_{TS} is the size of a time stamp.

The MAC scheme can use the same layout with time stamps for MACs, and the CHTree scheme can use the same layout with tree nodes for hash trees.

4.3.2 Hardware Modules

We describe the implementation of the integrity checking mechanisms based on the H-LHash scheme. The LHash scheme is a simplified version of the hierarchical scheme. MACs and hash trees also require very similar datapaths [7].

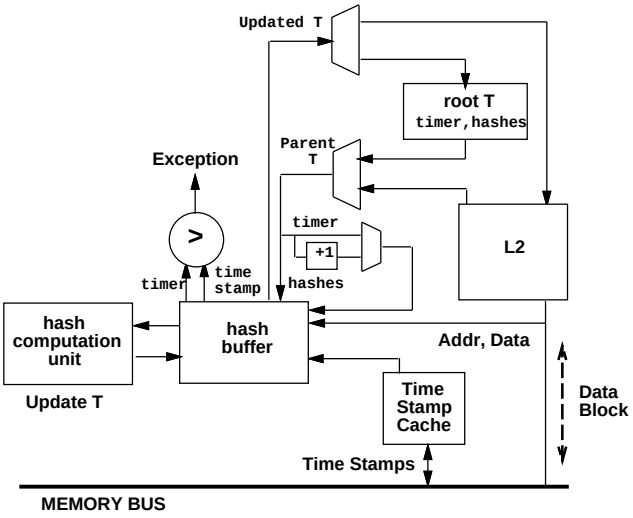


Figure 5: Hardware implementation of the H-LHash scheme.

The integrity checking module is added next to the on-chip Level 2 (L2) cache as shown in Figure 5.

Whenever there is a cache miss, the time stamp is checked and the $(Address, Data, Time)$ triple from the memory is added to a hash buffer. The unit computes the hash of the triple, and updates the parent's ReadHash by accessing the L2 cache (or the root register). If a cache block gets evicted, the new time stamp is read from a counter, the $(Address, Data, Time_{new})$ from the cache is added to a hash buffer, and the time stamp is written back to memory. The unit compute the hash of the triple, and updates the parent's ReadHash by accessing the L2 cache (or the root register).

As an optimization, a small time-stamp cache can be added. Modern processors often have a 8-B or wider memory bus, while we can use 4-B time stamps. In this case, directly accessing time stamps one by one will waste one half of bandwidth used for time stamps. To avoid this inefficiency, we add a small non-write-allocate cache with the cache block size the same as the memory bus width. All accesses to time stamps go through this cache.

5 Multiprocess Environments

Up to now, we have been working as if the program for which we want to do memory checking has full control of the processor. This is far from reality on most computing platforms, so to really make them useful, we have to show how our primitives could be integrated into a multitasking environment with mutually mistrusting processes. In this section we summarize two methods to achieve this goal.

5.1 Checking Physical Memory

So far, we have implicitly been checking physical memory. It is possible to continue doing so in multiprocessor systems on which some core functionality of the operating system is trusted. We shall call the trusted part of the operating system the *nexus* as in Palladium [3]. The goal is for many mutually mistrusting applications to be able to run concurrently without being vulnerable to tampering from any software other than the nexus, or to a physical attacker.[6]

In this model, the nexus configures the processor to perform memory checking on some contiguous block of memory. The data and the checker's meta-data are laid out in that block exactly as in the single process case. The memory checking directly makes the block of memory immune to physical tampering.

Next, mutually mistrusting pieces of software must be protected from each other. The processor's traditional protection mechanisms such as page tables, and privilege levels are adequate to protect applications from each other. To protect applications from potentially malicious parts of the operating system (device drivers, for example), we arrange, as in Palladium, to have the nexus run at a higher privilege level than the rest of the operating system.

The final problem is to give applications a way of checking that they are running under the control of a valid nexus. A collision resistant hash of the nexus is stored in a special register in the processor that applications can read to determine the nexus that they are running under.

Overall, this strategy is very close to what is done in Palladium, and we believe that memory checking would be easy to add to Palladium.[7]

5.2 Checking Virtual Memory

The strategy that was described in the previous section has a few drawbacks. First, the operating system needs to be aware of the memory checking mechanisms. Updating the operating system could involve a considerable development effort that would have to be done before any protected application could run. Moreover, it would be satisfying to provide, as in XOM [9], a security mechanism that is free of any trust in the operating system.

To get rid of the nexus, it is necessary to add more functionality to the hardware (in practice this functionality could be provided in part by firmware on the processor). The result will be quite similar to XOM, with encryption replaced by memory checking.

First, the processor must have a notion of protected process. Indeed, if many protected processes are coexisting, and the operating system is untrusted, then the processor must be able to distinguish processes from each other without help from the operating system. Therefore, when a secure process starts, the processor has to make note of the fact, and allocate some state for the process.

The state the processor has to keep for each process is the root of its secure storage and the value of the process's registers. While the protected process is running, memory checking is activated. Whenever the process loses control the processor records its registers. When the process is resumed, the processor checks that the registers match the recorded values.

The state that the processor has to maintain can be fixed, as in XOM, or it can be stored in main memory and protected by memory checking. In the latter case, some mechanism has to be provided for the processor to allocate some memory for its state

[6] Palladium only protects software from software attacks. We additionally protect integrity from a physical attacker.

[7] One interesting difference with Palladium is that we no longer need support from the bus controller to guarantee execution integrity. Indeed, protection from DMA attacks is provided by the memory checking functionality.

(a user level daemon could do the job as any failure to do its job correctly would be promptly detected by the memory checking algorithm, and there is no need to involve the operating system).

In this scheme, memory authentication is carried out on each process's virtual memory. The process invokes an instruction that activate memory checking on some contiguous block of virtual memory. The data and checker meta-data is laid out in this block in the usual way. Since the checker algorithms need to operate at the level of the L2 cache, this layout strategy implies that the checker algorithm can determine the virtual address of a block to be written back to compute the virtual address of the corresponding meta-data, and then convert the virtual address into a physical address. To make these conversions possible the L2 cache should contain virtual addresses, and the checker needs a table similar to the TLBs for virtual to physical translation.

In many programs virtual memory is sparsely filled, with the stack at high addresses, and code and heap at low addresses. This model can accommodated by checking a large sparse contiguous block of memory. Pages of checker data that are not needed because the corresponding address is unused can be left unallocated.

5.2.1 Context Switching and the Cache

A major problem arises during context switch from a protected task. Indeed, at the time of the context switch, the on-chip cache can contain dirty blocks to be written back to checked memory. However, during the context switch, the virtual memory mapping changes. Consequently, at write-back time, the processor no longer has the old process's page table it would need to find the checker meta-data for the block.

The simple solution to the problem is to write back all of a protected task's data before a context switch. The drawbacks of this method are of course a performance penalty, because the processor will be idle during the write back, and also a significant increase in interrupt latency. Both problems can be mitigated by adding tag bits to the cache to allow the processor to keep track of which process's data space cached data corresponds to. Just a single tag bit to distinguish between the previous task and the current task could drastically reduce the performance penalty. Along with the tags, extra state has to be added to the processor so that it can remember the page table locations for each tag value.

A final precaution must be taken for data in the cache when a context switch occurs *to* a protected process. Indeed, data that is in the cache at the time of the context switch has not necessarily been checked by the new process, but being in the cache, it is assumed to be correct. The simple solution is to flush the cache on a context switch. Once again, the simple solution can be improved by adding tag bits that identify which protected process trusts the data, so that a protected process can keep data in the cache when it is not running.

It should be noted that for global address space architectures, the caching problems are greatly simplified as there are, by definition, enough tag bits in the cache to identify each process's data.

5.3 Summary

In this section we have considered two ways of implementing memory checking in multiprocessing systems. These are just two extreme cases, one where everything but the actual memory checking is done in a trusted nexus[8] that is part of the operating system, and one where it is done without any operating system intervention. Real implementations will probably choose intermediate solutions depending on the other constraints that drive their architectures. In particular, the need to allow sharing of data between protected processes in a flexible way favors the software approach.

6 Comparisons

This section compares four integrity checking mechanisms described in this paper: `MAC`, `CHTree`, `LHash` and `H-LHash`. All four schemes can be implemented in either software or hardware, but we focus on the case when we use them as a hardware mechanism in microprocessors.

6.1 Restrictions

The `MAC` scheme can only be applied to static data such as program instructions because it cannot prevent replay attacks. The other three schemes can be used for data integrity protection.

6.2 Performance Overhead

The major component to determine the run-time performance overhead of the integrity checking scheme is the additional bandwidth usage. The `MAC` scheme uses $\frac{B_{MAC}}{B_{Chunk}}$ times more bandwidth compared to a processor without integrity checking, where B_{MAC}

[8] It would be possible to be a little more extreme and have software actually perform the memory checking. In that case, though, the processor would require some dedicated on-chip storage for the memory checking code to avoid the circular problem of having to check the memory checking code. We believe, however, that this solution would be too slow to be worthwhile.

is the size of a MAC in bytes and B_{Chunk} is the size of a chunk. The bandwidth usage of `CHTree` depends on the average number of hash accesses that each memory access incurs (z_{hash}). The overhead is $z_{hash} \cdot \frac{B_{Hash}}{B_{Chunk}}$, where B_{Hash} is the size of a hash. `LHash` reads and writes time stamps and thus has the bandwidth overhead of $2 \cdot \frac{B_{Time}}{B_{Chunk}}$ [9] , where B_{Time} is the size of a time stamp. For `H-LHash`, let z_{lhash} be the average number of log hash accesses for each memory access. Each log hash is $B_{Time} + 2B_{Hash}$ bits large, and on a cache miss or eviction, its parent must be updated, with the overhead being the time stamp that is read or written to memory. The leaf's parent must also be updated, with a cost that is the same as `LHash`. Thus, the bandwidth overhead of `H-LHash` is $\frac{z_{lhash}(2B_{Time}+2B_{Hash}) + 2B_{Time}}{B_{Chunk}}$.

For a typical configuration ($B_{MAC} = B_{Hash} = 16$, $B_{Chunk} = 64$, $B_{Time} = 4$, $z_{hash} = 1.5$, $z_{lhash} = 0.04$), the overhead is `MAC`=25%, `CHTree`=37.5%, `LHash`=12.5%, `H-LHash`=15%. Therefore, the `LHash` scheme is likely to perform the best when checking is infrequent.

However, the `LHash` scheme incurs significant overhead of reading memory space for each `integrity-check` operation. Therefore, the performance of the scheme will degrade as we perform more frequent check operations.

On the other hand, the `CHTree` scheme pollutes the L2 cache with hashes and degrades the cache performance of an executing program. This effect also degrades the performance especially for systems with small caches.

`H-LHash` moves some of the cost of the `integrity-check` operation over to run time. Its run time cost can, thus, be expected to be slightly more than that of `LHash`, but its integrity checking cost should be significantly less than that of `LHash`.

The performance of the four schemes are studied in detail using a processor simulator in the next section.

6.3 Memory Space Overhead

All the integrity checking schemes need memory space in addition to the data they verify. `MAC` stores a MAC of the data chunk, `CHTree` stores hashes, and `LHash` stores time stamps. `H-LHash` stores `time stamp`-ReadHash-WriteHash triples. We evaluate the overhead of addtional memory space compared to the data chunks. The overhead is approximately $\frac{B_{MAC}}{B_{Chunk}}$ for `MAC`, $\frac{1}{m_{CHTree}-1}$ for `CHTree` with a m_{CHTree}-ary hash tree, $\frac{B_{Time}}{B_{Chunk}}$ for `LHash`, and $((1 + \frac{1}{m_{HLHash}-1}).(1 + \frac{B_{Time}}{B_{Chunk}}) - 1)$ for `H-LHash`.

[9] This is an upper bound, since a write-allocate cache will consume $2 \cdot B_{Chunk}$ bandwidth not just B_{Chunk}.

For typical values ($m_{CHTree} = 4$, $m_{HLHash} = 64$), the overheads are 25%, 33%, 6.25%, and 7.9% for `MAC`, `CHTree`, `LHash`, and `H-LHash` respectively. Therefore, `LHash` has significantly less space overhead compared to the other schemes.

6.4 Logic Overhead

The major logic component to implement the schemes is a hash (MAC) computation unit. The mechanisms only need a few buffers and a small amount of on-chip storage other than the hash unit.

To evaluate the cost of computing hashes, we considered the MD5 [15] (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. [10]

In each case, simple 32-bit operations are performed over 80 rounds. 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. Therefore, the circuit size is inversely proportional to the throughput of hash computation.

For `MAC` and `CHTree`, the hash of B_{Chunk} (typically 64 Bytes) needs to be computed for each memory read/write. For `LHash` and `H-LHash`, two hashes of $B_{Address} + B_{Chunk} + B_{Time}$ (typically 72 Bytes) for each memory read. Therefore, these schemes would have about 2-3 times more logic overhead compared to others. For typical memory throughput of 1.6GB/s, the circuit size will be around 10,000 to 20,000 1-bit gates.

Table 1 summarizes the discussion in this section. The typical values are shown in the parentheses.

7 Experiments

This section evaluates the `LHash` and `H-LHash` schemes compared to the two other schemes through detailed simulations. For all integrity checking scheme, we used the chunks that are the same as the cache blocks.

[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.

	MAC	CHTree	LHash	H-LHash
Characteristics				
Applications	Static	Static Dynamic	Static Dynamic	Static Dynamic
Check Period	Fixed	Fixed	Flexible	Flexible
Performance Overhead				
Run-Time B/W	Med (25%)	High (37.5%)	Low (12.5%)	Low (15%)
Cache Pollution	No	Yes	No	Yes
Check Overhead	No	No	Yes	Yes
Others				
Space Overhead	Med (25%)	High (33%)	Low (6.25%)	Low (7.9%)
Hash Logic	1x	1x	2-3x	2-3x

Table 1: Comparison of four integrity checking scheme. The hash logic size is relative to the size required for the MAC scheme.

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
TLB latency	160
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	160 cycles
Hash throughput	3.2 GB/s (MAC, CHTree
	6.4 GB/s (LHash, H-LHash
Hash buffer	32
Hash length	128 bits
Time stamps	32 bits
Time stamp cache	32

Table 2: Architectural parameters used in simulations.

7.1 Simulation Framework

Our simulation framework is based on the SimpleScalar tool set [2]. The simulator models speculative out-of-order processors. 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 integrity checking units share the same bus.

The architectural parameters used in the simulations are shown in Table 2. SimpleScalar is configured to execute Alpha binaries, and all benchmarks are compiled on EV6 (21264) for peak performance.

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 integrity checking schemes. Given the fact that benchmarks run for a long time, the overhead should be negligible compared to the steady-state performance.

7.2 Baseline Characteristics

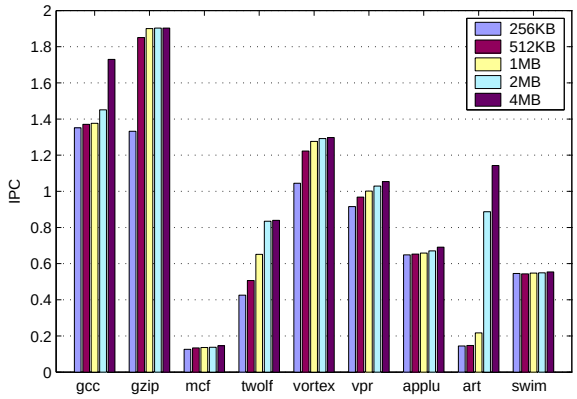
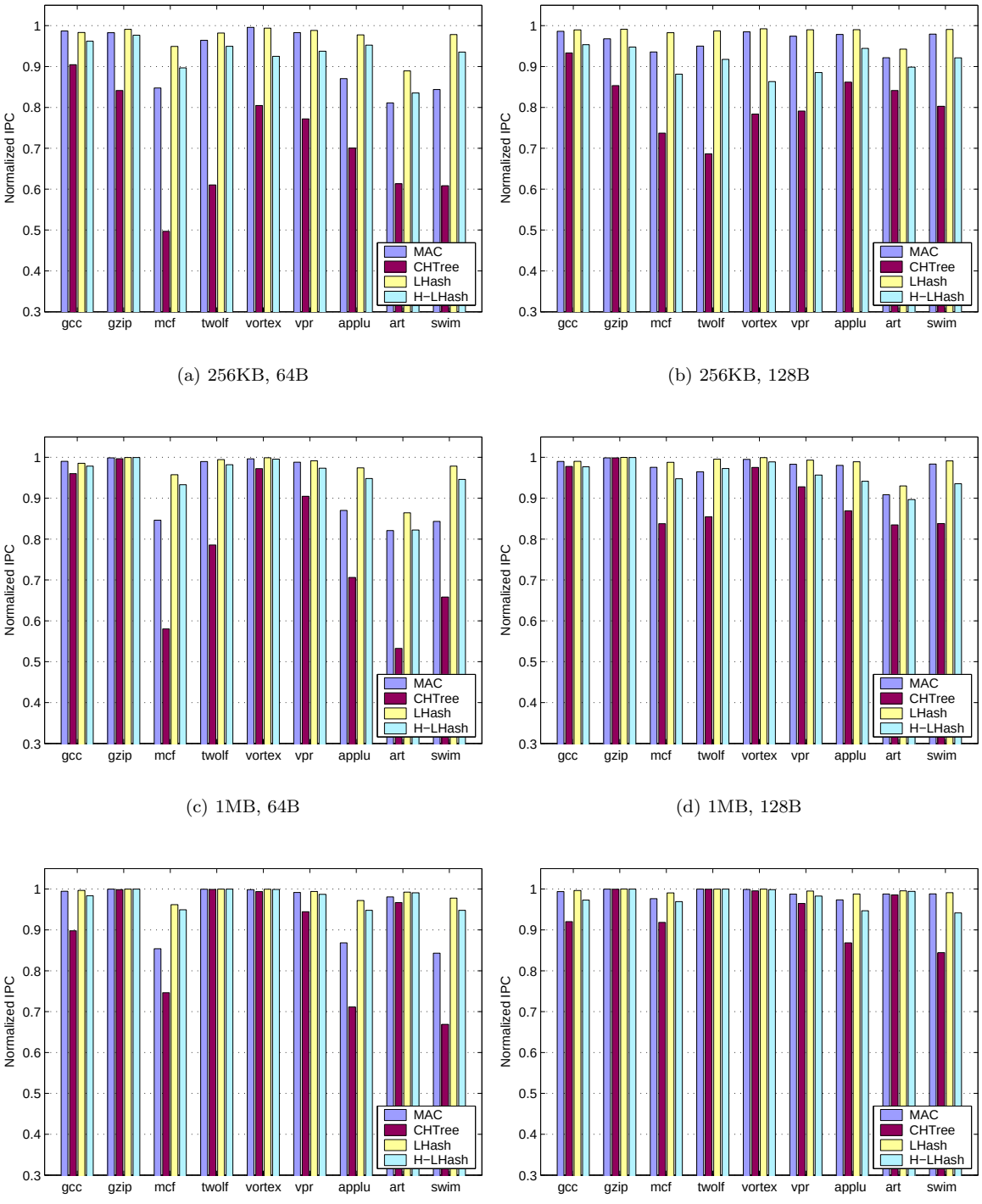


Figure 6: Baseline performance of simluated benchmarks. Results for five different L2 caches with 64-B blocks are shown.

For all the experiments in this section, nine SPEC2000 CPU benchmarks [8] are used as representative applications: `gcc`, `gzip`, `mcf`, `twolf`, `vortex`, `vpr`, `applu`, `art`, and `swim`. Figure 6 illustrates the baseline characteristics of the benchmarks used in the experiments. For each benchmark, the IPCs are shown for five different L2 cache sizes with 64-B blocks. The benchmarks show varied characteristics such as the level of ILP (instruction level parallelism), cache miss-rates, etc. For example, `mcf`, `applu`, and `swim` show poor L2 cache performance for all sizes simulated, and heavily utilize the off-chip memory bandwidth (`bandwidth-sensitive`). On the other hand, other benchmarks are sensitive to cache sizes, and do not require high off-chip bandwidth (`cache-sensitive`). By simulating these benchmarks, we can study the impact of memory verification on various types of applications.

7.3 Run-Time Performance

We first investigate the impact of the integrity checking schemes on processor performance ignoring the overhead of integrity-check operations for `LHash` and `H-LHash` schemes. If applications export secrets or sign results at the end of execution, or relatively infrequently, the overhead is negligible and the results in this section represents the overall performance. For example, if a typical program executes for a minute, this corresponds to roughly 100 billion instructions, on a state-of-the-art 2- or 4-way superscalar 1 GHz processor. The memory-read and check operation typically takes less than a billion cycles, and if this



(a) 256KB, 64B

(b) 256KB, 128B

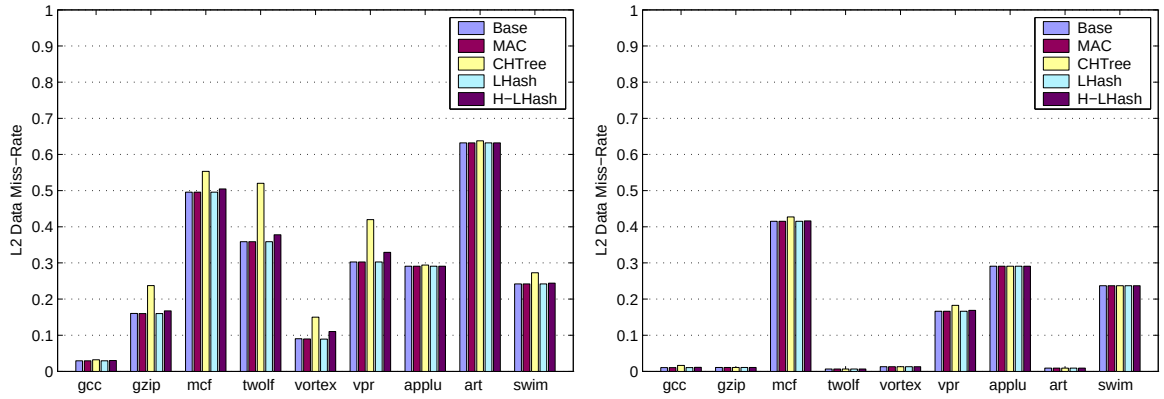
(c) 1MB, 64B

(d) 1MB, 128B

(e) 4MB, 64B

(f) 4MB, 128B

Figure 7: Run-time performance overhead of memory integrity checking: MACs (`MAC`), cached hash trees (`CHTree`), log-hashes (`LHash`), and hierarchical log-hashes (`H-LHash`). Results are shown for three different cache sizes (256KB, 1MB, 4MB) with cache block size of 64B and 128B. 32-bit time stamps and 128-bit MAC/hashes are used. In the `H-LHash` scheme, each $\mathcal{T}$ covers 4-KB memory space.



(a) 256KB, 64B

(b) 4MB, 64B

Figure 8: L2 cache miss-rates of program data for a standard processor (`Base`) and the ones with memory verification schemes (`MAC`, `CHTree`, `LHash`, and `H-LHash`). The results are shown for 256-KB and 4-MB caches with 64-B cache blocks.

is performed once, at the end of the execution, the overhead is very small.

Even for applications requiring rather frequent integrity checking, the overhead of an integrity-check operation is independent of cache configurations. Therefore, we study the impact of various cache configurations without considering `LHash` checking overhead. The effects of frequent integrity checking is studied in the following subsection.

Figure 7 illustrates the impact of integrity checking on the run-time program performance. For four different L2 cache configurations, the normalized IPCs (instructions per clock cycle) of four schemes are shown: MACs (`MAC`), cached hash trees (`CHTree`), log-hashes (`LHash`), and hierarchical log-hashes (`H-LHash`). The IPCs are normalized to the baseline performance with the same configuration.

The experimental results clearly demonstrate the advantage of the log-hash schemes (`LHash` and `H-LHash`) over the conventional integrity checking schemes when we can ignore the integrity-check overhead. For all cases we simulated, `LHash` outperforms both `MAC` and `CHTree`. The performance overhead of the `LHash` scheme is often less than 5% and less than 15% even for the worst case. The `MAC` scheme also performs very well for many cases (less than 5% overhead) and show only 20% performance degradation in the worst case. However, the `MAC` scheme is only secure for the static instruction verification. On the other hand, the cache hash tree `CHTree` has as much as 50% overhead in the worst case and 20-30% in general.

The run-time performance of the hierarchical log-hash scheme (`H-LHash`) is somewhat worse than the simple scheme (`LHash`) because it has to update a tree of hashes rather than just one. The hierarchical scheme has 5-10% overhead in general, and 20% in the worst case. However, the `H-LHash` scheme is still significantly better than the hash tree because it has considerably less levels (4) in the tree compared to the `CHTree` (13). It is possible to have many children in the hierarchical log-hash scheme (64-ary tree in the simulation) because we only need to read one node to update the parent. On the other hand, a 64-ary tree is not viable for the hash tree scheme because it needs to read all the children nodes to check or update the parent.

The figure also demonstrates the general effects of cache configuration on the memory integrity checking performance. The overhead of integrity checking decreases as we increase either cache size or cache block size. Larger caches results in less memory accesses to verify and less cache contention between data and hashes. Larger cache blocks reduce the space and bandwidth overhead of integrity checking by increasing the chunk size.

Memory integrity checking impacts the run-time performance in two ways: *cache pollution* and *bandwidth consumption*. The `CHTree` and `H-LHash` schemes store its hash nodes in the L2 cache with program data. As a result, the cache miss-rate of the program data can be increased by the schemes. On the other hand, all integrity checking schemes use additional off-chip bandwidth compared to the baseline case, to access MAC/hash or time stamps. Comsuming more bandwidth may delay the program memory accesses and increase the latency.

7.3.1 Cache Pollution

Figure 8 illustrates the effects of integrity checking on cache miss-rates. Since `MAC` and `LHash` do not store hashes in the cache, those schemes do not affect the L2 miss-rate. However, `H-LHash` slightly increases the miss-rate and `CHTree` can significantly increase miss-rates for small caches. In fact, the performance degradation of the `CHTree` scheme for *cache-sensitive* benchmarks such as `gcc`, `twolf`, `vortex`, and `vpr` in the 256-KB case (Figure 7) is mainly due to cache pollution. As you increase the cache size, cache pollution becomes negligible as you can cache both data and hashes without contention (Figure 8 (b)).

7.3.2 Bandwidth Consumption

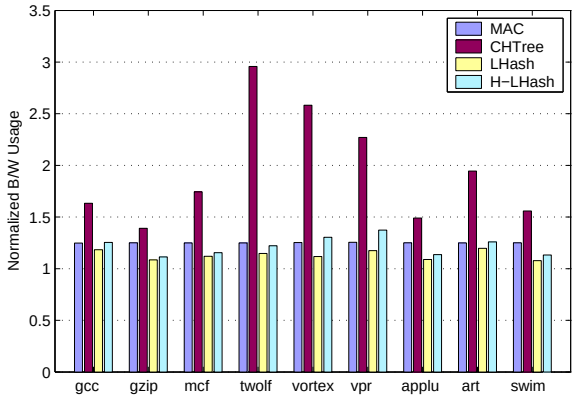


Figure 9: Off-chip bandwidth consumption of memory verification schemes (`MAC`, `CHTree`, `LHash`, and `H-LHash`). The L2 cache is 1 MB with 64-B cache blocks. The bandwidth consumption is normalized to the baseline case.

The bandwidth consumptions of the integrity checking schemes are shown in Figure 9. The `MAC` scheme has the constant overhead of 25% in this case since it always accesses one MAC for each cache block access. The `LHash` scheme theoretically consumes 6.25% to 12.5% of additional bandwidth compared to the baseline. In our processor implementation, however, it consumed more (8.5% to 20%) because our bus width is 8B while the time stamps are only 4B. The `CHTree` and `H-LHash` schemes consume additional bandwidth depending on the L2 cache performance on hashes. Because `CHTree` needs a deep tree, it consumes much more bandwidth than others. For *bandwidth sensitive* benchmarks, the bandwidth overhead directly translates into the performance overhead. This makes log-hash schemes much more attractive even for processors with large caches where cache pollution is not an issue.

7.4 Overall Performance

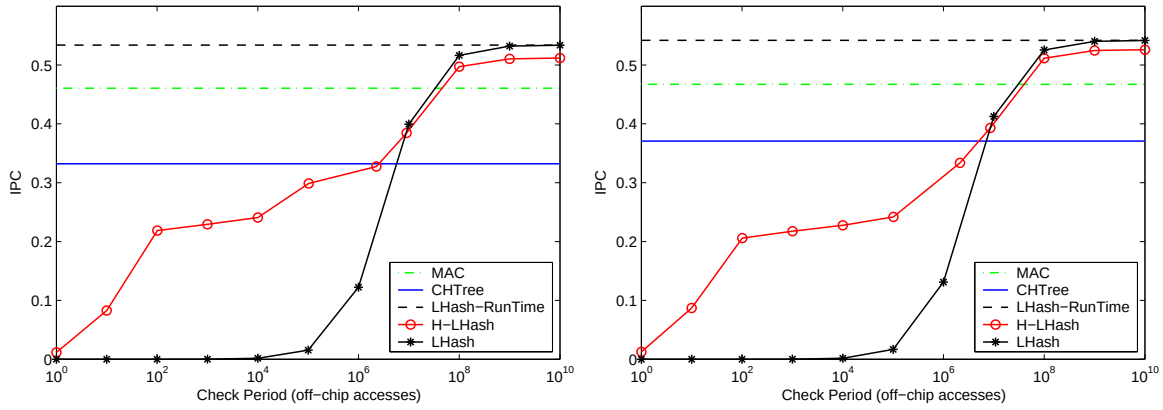
The last subsection clearly demonstrated that the `LHash` scheme outperforms the others when integrity-check operations are infrequent. However, applications may need to check memory integrity more often for various reasons such as exporting a secret to other programs, signing the results, etc. In these cases, we cannot ignore the overhead of the checking operation. In this subsection, we compare the integrity checking schemes including the overhead of periodic integrity-check operations.

We assume that the log-hash schemes check memory integrity every T memory accesses. A processor executes a program until it makes T main memory accesses, then checks the integrity of the T accesses by performing an integrity-check operation. Obviously, the overhead of the checking heavily depends on the characteristics of the program as well as the check period T. We use two representative benchmarks `swim` and `twolf` – the first consumes the largest amount of memory and the second consumes the smallest. `swim` uses 192MB of main memory and `twolf` uses only 2MB of memory. A processor only verifies the memory space used by a program.

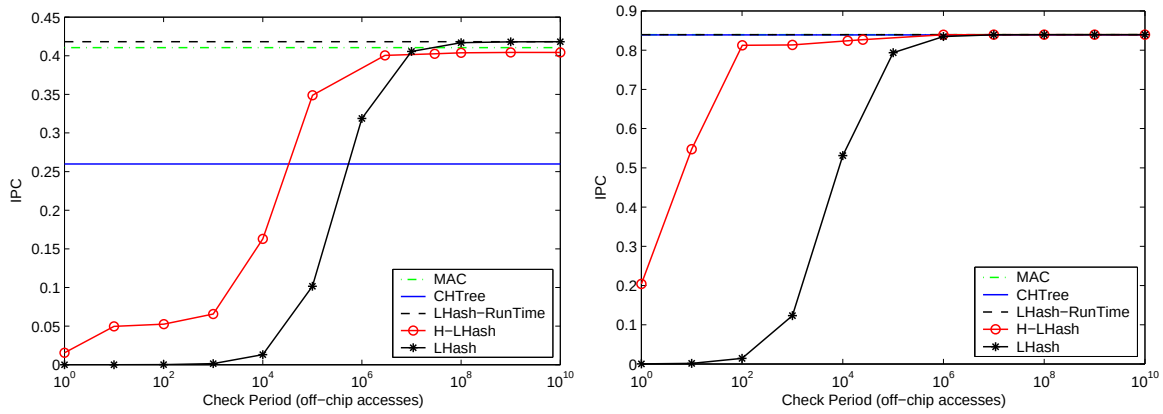
Figure 10 compares the performance of the four memory integrity checking schemes for varying check periods. The performance of the conventional schemes (`MAC` and `CHTree`) are indifferent to the checking period since they have no choice but to check the integrity after each access. Effectively, these schemes always have a checking period of one memory access.

On the other hand, the performance of the log-hash schemes (`LHash` and `H-LHash` heavily depends on the checking period. The `LHash` scheme is infeasible when the application needs to assure the memory integrity after a small number of memory accesses. In this case, either MACs or hash trees should be used since the IPC of the `LHash` scheme is effectively zero. As the checking period increases, the performance of `LHash` improves, and there is a break-even point between a conventional scheme and the `LHash` scheme. For a long period such as hundreds of millions to billions of accesses, both `LHash` and `H-LHash` converges to the run-time performance. In the experiments, the break-even point with the hash tree scheme is around 10^5 to 10^6 memory accesses. `twolf` has a smaller break-even point because it needs to read less amount of data per check.

The hierarchical log-hash scheme (`H-LHash`) dramatically improves the `LHash` scheme for short checking periods, with a slight performance degradation for long checking periods. With the hierarchical scheme, the performance is acceptable even for short peri-



(a) `swim, 256KB` (b) `swim, 4MB`



(c) `twolf, 256KB` (d) `twolf, 4MB`

Figure 10: Performance comparision between the offline scheme and the online scheme for various checking periods. Results are shown for 256-KB and 4-MB L2 caches with 64-B blocks. 32-bit time stamps and 128-bit hashes are used.

ods and it reduces the break-even point as well. In general, the hierarchical scheme offers a trade-off between the check overhead and the run-time performance. We can increase the *subspace* that each $\mathcal{T}$ covers to reduce the number of chunks to be read for each check. Therefore, a smaller subspace will reduce the check overhead. However, smaller subspaces result in a deeper tree structure, which degrades the run-time performance.

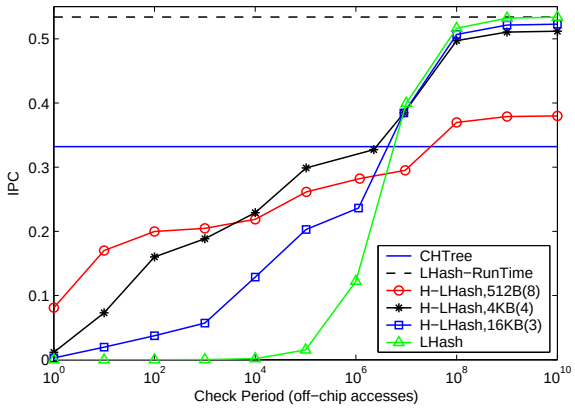


Figure 11: The effect of various subspace sizes for the `H-LHash` scheme. The results are shown for `swim` with a 256-KB L2 cache with 64-B blocks.

Figure 11 demonstrates the trade-off. The performance of the `H-LHash` scheme is shown for different subspace sizes: 512B, 4KB, 16KB, and 4GB [11]. As expected, `H-LHash` with a larger subspace performs better for short checking periods, but worse for long checking periods. Potentially, the hierarchical scheme can be dynamically configured for each program to result in the best performance for program's checking period. This would be possible because the hardware implementations of different subspace sizes are the same except for the configuration parameter that can be easily stored in registers.

Another advantage of the hierarchical scheme (`H-LHash`) over the basic log-hash scheme (`LHash`) is smaller time stamps (not shown in figures). The hierarchical scheme has a timer for each node that increments much slower than the global timer of the `LHash` scheme. Therefore, the hierarchical scheme requires a much smaller timer to execute a given number of memory accesses without a checking operation.

7.5 Design Parameters

This subsection studies the effects of design parameters of the integrity checking schemes on the performance. We focus on the parameters of the log-hash schemes. However, the MAC and hash tree schemes have similar trade-offs and contraints.

7.5.1 Hash Computation Throughput

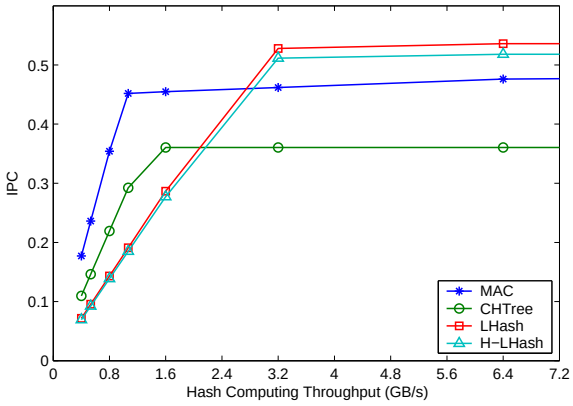


Figure 12: The effect of MAC/hash computation throughput on performance. The results are shown for `swim` with a 1-MB L2 cache with 64-B blocks.

Figure 12 illustrates how the hash computation throughput limits the system performance. To fully utilize the available off-chip memory bandwidth, the hash throughput should be properly balanced with the memory bandwidth. As discussed in Section 6, the `MAC` and `CHTree` schemes require one hash computation of 64B per 64B memory access. Therefore, the hash throughput should match the memory bandwidth (1.6GB/s). The `LHash` and `H-LHash` schemes require more hash computation, and show significant performance degradation for less than 3.2GB/s in the experiment. Therefore, these log-hash scheme would require two to three times more logic overhead to implement a hash computation unit than MACs and hash trees.

7.5.2 Hash Buffer

The hash buffer stores an address, a chunk, and a time stamp while the hash computation unit works on it. The buffer should be large enough to hold all chunks that is being processed in order to fully utilize the throughput of the hash computation unit. For example, Figure 13 illustrates the effect of hash buffer size on the performance. Given the latency of 160 cycles and throughput of one hash per 20 cycles, the figure shows that the buffer should be at least 8 entries large.

7.5.3 Time Stamp Cache

As discribed in Section 4.3.2, we use a small cache for time stamps. Figure 14 shows the performance of the `LHash` scheme for various sizes of the time

[11] We can think `LHash` as a special case of `H-LHash` where there is only one level in the tree.

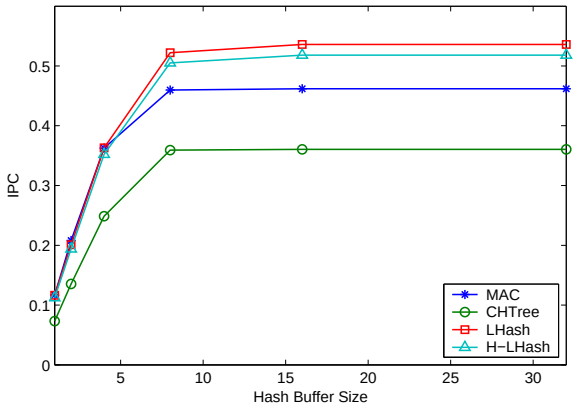


Figure 13: The effect of MAC/hash buffer size on performance. The results are shown for `swim` with a 1-MB L2 cache with 64-B blocks.

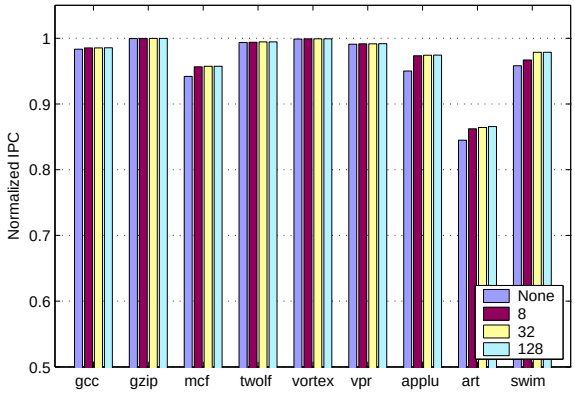


Figure 14: The effect of time stamp cache size (the number of cache blocks) on performance. The results are shown for `swim` with a 1-MB L2 cache with 64-B blocks.

stamp cache. The size of the cache does not matter for applications where bandwidth consumption is not a major issue. However, the cache improves the performance by a few percent by reducing the bandwidth overhead for *bandwidth-sensitive* benchmarks such as `mcf`, `applu`, and `swim`. About 32-block caches are large enough because the cache only exploits the spatial locality of time stamp accesses.

8 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 a hash tree rooted in trusted memory [1]. This scheme has a $O(\log(N))$ cost for each memory access. They also proposed offline schemes to check the correctness of RAM after a sequence of operations have been performed on RAM. These schemes compute a running hash of memory reads and writes. Their implementation of offline checkers uses ϵ-biased hash functions [13]; these hash functions can be used to detect random errors, but are not cryptographically secure. We have used incremental multiset hashes and their offline scheme as the basis to build log-hash-based RAM integrity checkers secure against active adversaries. We have described how an on-chip trusted cache can be used to significantly improve the efficiency of the log-hash scheme, and also described how the scheme can be generalized to a hierarchical scheme for further efficiency improvement.

Recent papers, [7], [5], [11], [10], describe systems in which a trusted program, running in a trusted computing base (TCB), uses hash trees to maintain the integrity of data stored on an untrusted storage. The untrusted storage is typically some arbitrarily large, easily accessible, bulk store in which the program regularly stores and loads data which does not fit in a cache in the TCB. In [7], the program runs on a trusted processor; the untrusted storage is the external random access memory (RAM). [11] and [10] describe a file system and database respectively, in which the program runs on a protected client and the data is maintained on an untrusted server. We have compared our log-hash and hierarchical log-hash schemes with the hash-tree-based schemes of [7] and a MAC scheme and shown that the log-hash schemes can be more efficient, in the case where integrity checking is infrequent. For the same performance overhead, the hierarchical log-hash scheme allows for more frequent integrity checking than the log-hash scheme.

9 Conclusion

We have described and compared various memory integrity verification schemes that can be used to build high performance secure computing platforms out of slightly modified general-purpose processors. Our conclusion is that log-hash schemes provide the best tradeoff considering logic complexity, memory space overhead and performance.

Ongoing work includes the investigation of adaptive log-hash schemes and the development of hybrid hash-tree and log-hash schemes.

References

[1] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of mem-

ories. In *IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.

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

[3] A. Carroll, J. Polk, and T. Leininger. Microsoft Palladium: A Business Overview. Technical report, Microsoft Corporation, 2002. http://www.neowin.net/staff/users/Voodoo/Palladium_White_Paper_final.pdf.

[4] D. Clarke, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas. Offline integrity checking of untrusted storage. Technical report, MIT LCS TR-871, November 2002.

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

[6] D. Eastlake and P. Jones. RFC 3174: US secure hashing algorithm 1 (SHA1), Sept. 2001. Status: INFORMATIONAL.

[7] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, February 2003.

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

[9] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. 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.

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

[11] D. Mazières and D. Shasha. Don't trust your file server. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

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

[13] J. Naor and M. Naor. Small-bias probability spaces: efficient constructions and applications. In *22nd ACM Symposium on Theory of Computing*, pages 213–223, 1990.

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

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

[16] K. H. Rosen. *Discrete Mathematics and Its Applications*. WCB McGraw-Hill, 1999.

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

[18] 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.

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