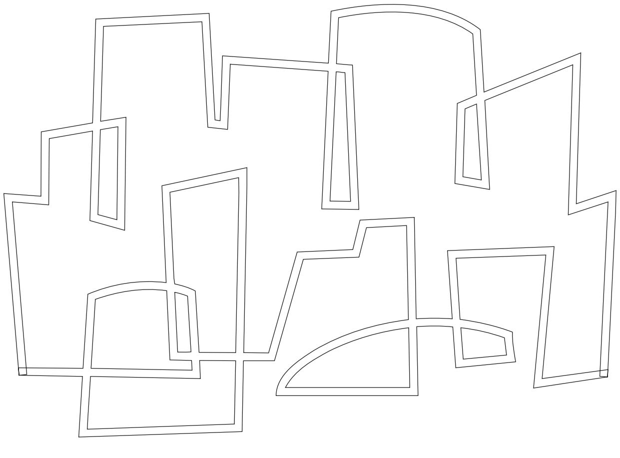
Efficient Memory Integrity Verification and Encryption for Secure Processors

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

To appear in the Proceedings of the 36th Annual International
Symposium on Microarchitecture, December 2003

Computation Structures Group
Memo 465



Efficient Memory Integrity Verification and Encryption for Secure Processors

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

ABSTRACT

Secure processors enable new sets of applications such as commercial grid computing, software copy-protection, and secure mobile agents by providing security from both physical and software attacks. This paper proposes new hardware mechanisms for memory integrity verification and encryption, which are two key primitives required in *single-chip* secure processors. These new mechanisms offer significant performance advantages over existing mechanisms, which directly translates to better performance for security-sensitive applications.

1. INTRODUCTION

Many emerging applications require physical security as well as conventional security against software attacks. For example, in Digital Rights Management (DRM), the owner of a computer system is motivated to break the system security to make illegal copies of protected digital content. Similarly, mobile agent applications [5] require that sensitive electronic transactions be performed on untrusted hosts. In this case, the hosts are under the control of an adversary who is financially motivated to break the system and alter the behavior of the mobile agents. Therefore, physical security is essential for enabling many applications in the Internet era.

Conventional approaches to build physically secure systems [20, 22] are based on building processing systems containing processor and memory elements in a private and tamper-proof environment that is typically implemented using active intrusion detectors. Providing high-grade tamper-resistance can be quite expensive [1]. Moreover, the applications of these systems are limited to performing a small number of security critical operations because system computation power is limited by the components that can be enclosed in a small tamper-proof package. In addition, these processors are not flexible, e.g., their memory or I/O subsystems cannot be upgraded easily.

Just requiring tamper-resistance for a single processor chip would significantly enhance the amount of secure computing power, making possible applications with heavier computation requirements. Secure processors have been recently proposed [12, 21], where only a single processor chip is trusted and the operations of all other components including off-chip memory are verified by the processor.

To enable single-chip secure processors, two main primitives, which prevent an attacker from tampering with the off-chip untrusted memory, have to be developed: memory integrity verification and encryption. Integrity verification prevents an adversary from changing a running program's state. The processor monitors the memory for any form of corruption. If any is detected, then the processor aborts the tasks that were tampered with to avoid producing incorrect results. Encryption ensures the privacy of data stored in the off-chip memory. To be worthwhile, the verification and encryption schemes must not impose too great a performance penalty on the computation.

Given off-chip memory integrity verification, secure processors can provide *tamper-evident (TE)* environments where software processes can run in an authenticated environment, such that any physical tampering or software tampering by an adversary is guaranteed to be detected. TE environments enable applications such as certified execution and commercial grid computing, where computation power can be sold with the guarantee of a compute environment that processes data correctly. The performance overhead of the TE processing largely depends on the performance of the integrity verification [21].

With both integrity verification and encryption, secure processors can provide *private and authenticated tamper-resistant (PTR)* environments where, additionally, an adversary is unable to obtain any information about software and data within the environment by tampering with, or otherwise observing, system operation. PTR environments can enable applications such as trusted third party computation, secure mobile agents, and Digital Rights Management (DRM) applications.

In this paper, we describe new hardware schemes to efficiently verify and encrypt all or a part of untrusted external memory. Our integrity verification scheme maintains incremental multiset hashes of all memory reads and writes at run-time, and verifies a *sequence* of memory operations at a chosen later point of time. Our encryption scheme uses one-time-pad encryption and time stamps, and decouples the decryption computation from the corresponding data access. This enables a processor to overlap decryption computation with data access and hide most of the decryption latency.

We evaluate our new schemes, viewing them as hardware mechanisms in a microprocessor. We compare them to the most efficient existing schemes, namely, hash tree integrity checking and direct block encryption. Simulations show that our integrity verification scheme outperforms the hash tree scheme when sequences of memory operations are verified, as opposed to verifying each memory operation. In these cases, the performance overhead of our scheme is less than 5% in most cases and 15% in the worst case. On the other hand, 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. Therefore, our integrity verification scheme significantly reduces the performance overhead of TE processing. Our new scheme has the added benefit of reducing memory space overhead from 33% for a typical hash-tree scheme to 6.25%.

Simulations also demonstrate that the one-time-pad encryption scheme outperforms the existing direct block encryption scheme in all cases. The one-time-pad scheme incurs about 8% performance overhead on average, and 18% in the worst case. The direct encryption incurs about 13% performance degradation on average, and 25% in the worst case. Combining the new integrity verification and encryption schemes, PTR processing can be done with less than 15% overhead in most cases compared to 40% overhead of the existing schemes.

The assumed model and how integrity verification and encryption are used in secure processors is presented in Section 2. The hash-tree mechanism for memory verification and our new scheme are described in Section 3. The conventional encryption mechanism and our new mechanism based on a one-time-pad and time stamps are discussed in Section 4. Section 5 discusses implementation issues. In section 6 we evaluate the schemes on a superscalar processor simulator. We discuss related work in Section 7 and conclude the paper in Section 8.

2. SECURE COMPUTING MODEL

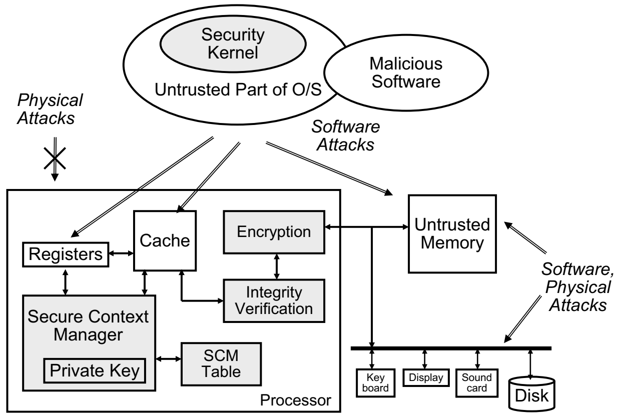


Figure 1: Our secure computing model.

We consider systems that are built around a single processor with external memory and peripherals. We do not consider multiprocessor systems in this paper.

Figure 1 illustrates the model. The processor, implemented on a monolithic integrated circuit (IC), is assumed to be trusted and protected from physical attacks; its internal state cannot be tampered with or observed directly by physical means. The processor can contain secret information that identifies it and allows it to communicate securely with the outside world. This information could be a Physical Random Function [8], or the secret part of a public key pair protected by a tamper-sensing environment [20].

The trusted computing base (TCB) consists of the processor chip and optionally[1] some core parts of the operating system that plays the part of the Nexus in Palladium [4] or the security kernel in AEGIS [21]. The processor is used in a multitasking environment, which uses virtual memory, and runs mutually mistrusting processes. External memory and peripherals are assumed to be untrusted; they may be observed and tampered with at will by an adversary.

The system provides programs with two secure execution environments: tamper evident (TE) and private tamper resistant (PTR). In the TE environment, the integrity of a program's execution is guaranteed. The PTR environment ensures the privacy of instructions and data in addition to integrity. Once a program has entered a secure execution environment using a special instruction, the TCB protects it and provides it with an additional instruction to sign messages with the processor's private key. The resulting signature is used to prove to a user that he is seeing the results of a correct execution of his program.

Since the adversary can attack off-chip memory, 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 has stored to that address.* We therefore require memory integrity verification. The TCB needs to ensure the integrity of memory accesses before it performs a signing operation or stores data into non-private memory space.

For PTR environments, we additionally have to encrypt data values stored in off-chip memory. The encryption and decryption of data values can be done by a hardware engine placed between the integrity checker and the off-chip memory bus, as in AEGIS.

We assume that programs are well-written and do not leak secrets via their memory access patterns. In particular, we do not handle security issues caused by bugs in an application program.

3. INTEGRITY VERIFICATION

This section presents a new memory integrity verification scheme, which has a significant performance advantage over existing methods. We first briefly summarize existing hash-tree schemes, and then introduce the new scheme.

For memory integrity verification, a simple solution based on message authentication codes (MACs) does not work. XOM [12] uses a MAC of the data and address for each memory block for authentication. Unfortunately, this approach does not prevent replay attacks; valid MACs guarantee that a block is stored by the processor, but do not guarantee that it is the most recent copy. Therefore, we exclude the simple MAC scheme.

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

3.1 Cached Hash Tree: `CHTree`

---

[1] In some models, the operating system may be entirely untrusted.

Hash trees (or Merkle trees) are often used to verify the integrity of dynamic data in untrusted storage [14]. Figure 2 illustrates a binary hash tree. Data (V_1, V_2, etc) is located at the leaves of a tree. Each internal node contains the hash of the concatenation of its children. The root of the tree is stored in secure on-chip memory where it cannot be tampered with.

A chunk consists of children of one node that are covered by the same hash. In the figure, one chunk contains two hashes or the same amount of data. For simplicity, we make the chunks the same as the L2 cache blocks. As a result, a tree with higher arity requires larger chunks and larger L2 blocks.

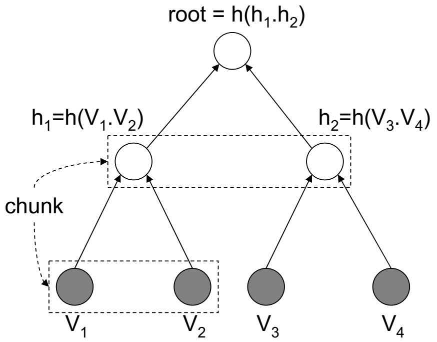


Figure 2: A binary hash tree. Each internal node is a hash of the concatenation of its children.

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.

With a balanced m-ary tree, the number of chunks to check on each memory access is $log_m(N)$, where N is the number of chunks in the verified memory space. The logarithmic overhead of using the hash tree can be significant. For example, [9] 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.

The performance overhead of using a hash tree can be dramatically reduced by caching the internal hash chunks 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 is evicted from the cache, the processor brings its parent into the cache (if it is not already there), and updates the parent in the cache. We refer to this optimized hash tree scheme as `CHTree`.

Previous work [9] showed that, in all cases, `CHTree` clearly outperforms the basic hash tree scheme in which just regular data alone is stored in the cache. In this paper, we use `CHTree` for comparison. (For more details and variants of `CHTree`, see [9].)

3.2 Log Hash Integrity Checking

`CHTree` has to check 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 a certified execution application, a processor only needs to check the integrity of previous operations before signing the results.

We introduce a new approach of verifying memory integrity with low run-time overhead. 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 incremental multiset hash functions [6]. When the processor needs to check its operations, it performs a separate *integrity-check* operation using the trusted state.

Since the multiset hash functions are used to maintain logs, we refer to our scheme as a log-hash scheme. The particular function we use is `MSet-Add-Hash` based on the hash function MD5 [17]. `MSet-Add-Hash` requires one MD5 operation using a secret key in the processor, and one addition operation over a fixed number of bits to update the multiset hash incrementally. The details and formal proofs of the security of `MSet-Add-Hash` and the log-hash memory integrity checking scheme are in [6]. In this paper, we extend the scheme to work with trusted caches and on-demand memory allocation, give a brief overview of why it works, and evaluate the scheme's performance.

3.2.1 Algorithm: `LHash`

Figure 3 shows the steps of the Log Hash (`LHash`) integrity checking scheme. We describe the operations assuming that the chunk is the same as an L2 cache block and the cache is write-allocate. To verify a sequence of memory operations, the processor keeps two multiset hashes (READHASH and WRITEHASH) and a counter (TIMER) in trusted on-chip storage. We denote the (READHASH, WRITEHASH, TIMER) triple by $\mathcal{T}$.

To initialize $\mathcal{T}$, add-chunk is called on each of the chunks that need to have their integrity be verified. This operation effectively remembers the initial value of the chunks in WRITEHASH.

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 removed (evicted or invalidated) from the cache, the processor logs the chunk's value by calling write-chunk. 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 into the cache. READHASH is updated with the hash of the address-chunk-time stamp triple that is read from the off-chip memory. TIMER is then increased to be strictly greater than the time stamp that was read from memory.

```
Initialization Operation
add-chunk($\mathcal{T}$, Address, Chunk):
  1. TimeStamp = $\mathcal{T}$.Timer.
     Update $\mathcal{T}$.WriteHash with the hash of
     (Address·Chunk·TimeStamp).
  2. Write (Chunk, TimeStamp) to address, Address, in
     memory.
```

```
Run-Time Operations
  • For a cache eviction
    write-chunk($\mathcal{T}$, Address, Chunk):
      1. TimeStamp = $\mathcal{T}$.Timer.
         Update $\mathcal{T}$.WriteHash with the hash of
         (Address·Chunk·TimeStamp).
      2. If a block is dirty, write (Chunk, TimeStamp)
         back to memory. If the block is clean, only write
         TimeStamp 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.
      2. Update $\mathcal{T}$.ReadHash with the hash of
         (Address·Chunk·TimeStamp).
      3. Increase $\mathcal{T}$.Timer:
         $\mathcal{T}$.Timer = max($\mathcal{T}$.Timer, TimeStamp+1)
    and store Chunk in the 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-chunk(New$\mathcal{T}$, address, chunk), where chunk
           is the chunk read from memory in Step 2a.
  3. Compare ReadHash and WriteHash. If different,
     there is a read that does not match its most recent
     write, therefore raise an integrity exception.
  4. If the check passes, $\mathcal{T}$ = New$\mathcal{T}$.
```

Figure 3: `LHash` Integrity Checking Algorithm.

In order to check memory, all chunks that are not in the cache are read so that each chunk has been added to ReadHash and WriteHash the same number of times. If ReadHash is equal to WriteHash, then the memory was behaving correctly (like valid memory, c.f. Section 2) during the sequence of memory operations since the last integrity checking operation. This checking is done in the integrity-check operation.

The WriteHash logs information on the chunks that, according to the processor, should be in memory at any given point in time. The ReadHash logs information on the chunks the processor reads from memory. Because the checker checks that WriteHash is equal to ReadHash, substitution (the RAM returns a value that is never written to it) and replay (the RAM returns a stale value instead of the one that is most recently written) attacks on the RAM are prevented. The purpose of the time stamps is to prevent reordering attacks in which RAM returns a value that has not been written yet so that it can subsequently return stale data.

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's 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.

In the case where we do not know at initialization how much memory the processor will use, new addresses can be added to the protected region by calling add-chunk on them on demand and using a table to maintain the list of chunks that have been 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-chunk 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 skips some chunks during the check operation, the integrity check will fail in that ReadHash will not be equal to WriteHash (as long as each chunk is read only once during the check).

4. MEMORY ENCRYPTION

Encryption of off-chip memory is essential for providing privacy to programs. Without encryption, physical attackers can simply read confidential information from off-chip memory. On the other hand, encrypting off-chip memory directly impacts the memory latency because encrypted data can be used only after decryption is done. This section discusses issues with conventional encryption mechanisms and proposes a new mechanism that can hide the encryption latency by *decoupling computations for decryption from off-chip data accesses*.

We encrypt and decrypt off-chip memory on an L2 cache block granularity using a symmetric key encryption algorithm because memory accesses are carried out for each cache block. Encrypting multiple cache blocks together will require accessing all those multiple blocks for decrypting any part of them.

4.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) [16] is an approved symmetric algorithm by the National Institute of Standards and Technology (NIST), which we use as a representative algorithm.

AES can process data blocks of *128 bits* using cipher keys with lengths of *128*, *192*, and *256 bits*. The encryption and decryption consist of 10 to 16 rounds of four transformations. The critical path of one round consists of one S-box look-up, two shifts, 6-7 XOR operations, and one 2-to-1 MUX. This critical path will take 2-4 ns with the current 0.13μ technology depending on the implementation of the S-box look-up table. Therefore, encrypting or decrypting one 128-bit data block will take about 20-64 ns depending on the implementation and the key length. Thus, decryption can add a significant overhead to memory latency.

When the difference in technology is considered, this latency is in good agreement with one custom ASIC implementation of Rijndael in 0.18μ technology [11, 19]. It reported that the critical path of encryption is 6 ns and the critical path of key expansion is 10 ns per round with 1.89 ns latency for the S-box. Their key expansion is identical to two rounds of the AES key expansion because they support 256-bit data blocks. Therefore, the AES implementation will take 5 ns per round for key expansion, which results in a 6 ns cycle per round, for a total of 60-96 ns, depending on the number of rounds.

4.2 Direct Block Encryption

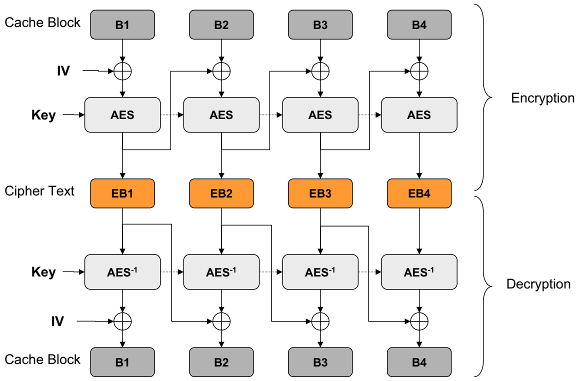


Figure 4: Encryption mechanism that directly encrypts cache blocks with the AES algorithm.

The most straightforward approach is to use an L2 cache block as an input data block of the AES algorithm. For example, a 64-B cache block `B` is broken into 128-bit chunks (`B[1]`, `B[2]`, `B[3]` and `B[4]`), and encrypted by the AES algorithm. Figure 4 illustrates this mechanism with Cipher Block Chaining (CBC) mode. In this case, the encrypted cache block `EB = (EB[1], EB[2], EB[3], EB[4])` is generated by `EB[i]` $= AES_K($`B[i]` $\oplus$ `EB[i-1]`$)$, where `EB[0]` is an initial vector `IV`. To prevent adversaries from comparing whether two cache blocks are the same or not, the address of a block is included in `IV`, and a part of `IV` is randomized and stored in off-chip memory along with data.

This scheme serves our purpose in terms of security, however, it has a major disadvantage for performance. Since decryption can start only after reading data from off-chip memory, the decryption latency is directly added to the memory latency and delays the processing (See Figure 7 (a)). For example, if the memory latency is 120 ns and the decryption latency is 40 ns, the processor will see a load latency of 160 ns.

4.3 One-Time-Pad Encryption

The main problem of the direct encryption scheme is that most of the AES decryption latency cannot be overlapped with the memory access. We therefore adopt a different encryption mechanism that decouples the AES computation from the corresponding data access using one-time-pad encryption [1] and time stamps.

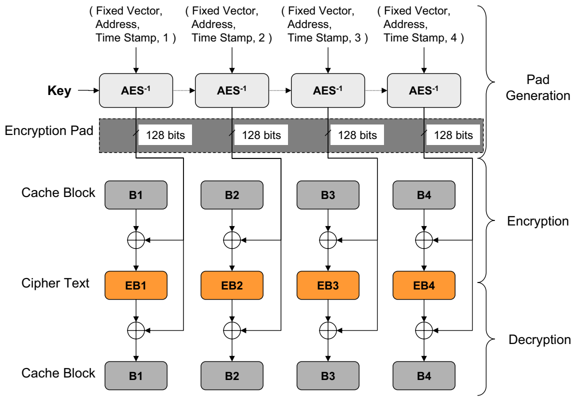


Figure 5: Encryption mechanism that uses one-time-pads from the AES algorithm with time stamps.

```
• For an L2 cache write-back
  write-back-block(Address, B):
    1. Increment TIMER. TS = TIMER.
    2. For each $1 \le i \le 4$
       (a) OTP[i] = $AES_K^{-1}$(V, Address, TS, $i$).
       (b) EB[i] = B[i] $\oplus$ OTP[i].
    3. Write TS and EB to memory.

• For an L2 cache miss
  read-block(Address):
    1. Read the time stamp (TS) from memory.
    2. For each $1 \le i \le 4$
       (a) Start OTP[i] = $AES_K^{-1}$(V, Address, TS, $i$).
    3. Read EB from Address in memory.
    4. For each $1 \le i \le 4$
       (a) B[i] = EB[i] $\oplus$ OTP[i].
    5. Cache B.
```

Figure 6: One-Time-Pad Encryption Algorithm.

Figure 5 illustrates the scheme. A cache block, `B`, consists of four chunks, `B[1]`, `B[2]`, `B[3]`, and `B[4]`. Each chunk is XOR'ed with an encryption pad that is generated by the AES algorithm with a time stamp. The resulting encrypted cache block, `EB`, and a time stamp are stored in off-chip memory. To decrypt the block, the encrypted cache block, `EB`, is XOR'ed with the same encryption pad.

To generate an encryption pad for the 128-bit chunk, `B[i]`, of a cache block, `B`, the processor decrypts (`V`, `Address`, `TS`, i) with a secret key K. `V` is a fixed bit vector that makes the input 128 bits, and can be randomly selected by the processor at the start of program execution. `TS` is a time stamp that is the current value of TIMER, which is a counter stored on-chip where it cannot be tampered with. The processor increments TIMER for every write-back of a cache block. As (`Address`, `TS`) is unique for each write-back to memory, the encryption pads are used only once.

Figure 6 details the scheme. write-back-block is used to encrypt and write dirty cache blocks to memory[2]. The TIMER is increased, and the block is encrypted using a one-time pad. The encrypted block and the time stamp are stored in off-chip memory. If `LHash` is used for integrity verification, two independent TIMERS should be used for integrity checking and encryption because the TIMERS are increased at different paces.

To read an encrypted block from memory, the processor uses the `read-block` operation. First, the processor reads the time stamp of `Address` from memory. To improve performance, it is also possible to cache time stamps on-chip. Once the time stamp is retrieved, we immediately start with the generation of the `OTP` using AES in step 2. *The pad is generated while `EB` is fetched from memory in step 3.* Once the pad has been generated and `EB` has been retrieved from memory, `EB` is decrypted in step 4.

When the TIMER reaches its maximum value, the processor changes the secret key and re-encrypts blocks in the memory. The re-encryption is very infrequent given an appropriate size for the time stamp (32 bits for example), and given that the timer is only incremented when dirty cache blocks are evicted from the cache. We do not need to increment `TS` during re-encryption, because `Address` is included as an argument to AES_K^{-1}, thus guaranteeing the unicity of the one-time-pads.

Security of the Scheme. The conventional one-time-pad scheme is proven to be secure [1]. Our scheme using one-time-pads and time stamps is an instantiation of a counter-mode encryption scheme. This can easily be proven to be secure, given a good symmetric encryption algorithm that is non-malleable [13].

Hiding Latency. Unlike the direct encryption scheme, the data access and the AES computation are independent in our new scheme. Therefore, the encryption latency can be hidden from the processor by overlapping AES computations with data accesses.

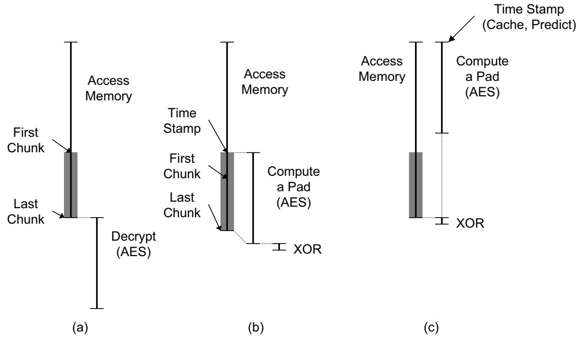


Figure 7: Impact of encryption mechanisms on memory latency.

Computing an encryption pad requires the time stamp for the cache block. In the worst case, without caching or speculation, the AES computation for generating an encryption pad starts after the time stamp comes back from the off-chip memory as shown in Figure 7 (b). This computation is overlapped with the following bus accesses for the cache block. Once the entire cache block is read and the pad computation is done, an XOR operation is performed for decryption. Although we may not hide the entire AES latency, our scheme can hide a significant portion of the latency. For example, if it takes 80 ns for reading the first chunk and 40 ns for the rest of the chunks in a cache block, we can hide 40 ns of the AES latency.

When overlapping the AES computation with data bus accesses is not sufficient to hide the entire latency, the time stamp can be cached on-chip or speculated based on recent accesses. In this case, the AES computation can start as soon as the memory access is requested as in Figure 7 (c), and completely hides the encryption latency.

The ability to hide the encryption latency obviously improves processor performance. It also enables a less aggressive implementation of the AES algorithm.

5. IMPLEMENTATION ISSUES

5.1 Memory Layout

To implement the memory checking and the encryption schemes, the layout of data and time stamps should be determined. The layout should be simple enough for hardware to easily compute the address of the corresponding time stamp from the address of a data chunk. We give an example layout for the `LHash` scheme where we use the beginning of the memory space for data and the end of the space for time stamps.

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

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

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

For the encryption scheme, we use the same linear layout with a different base address to determine the address of time stamps or random initial vectors.

5.2 Checking Virtual Memory

In this section, we describe the support needed to check the integrity of a process's virtual memory space.

As the checker algorithms are authenticating virtual memory and operate at the level of the L2 cache, one problem that must be addressed is determining the physical addresses of time stamps for cache blocks. To address this problem, the L2 cache should contain virtual addresses. To compute the virtual address of the corresponding meta-data, the processor uses the mappings described in Section 5.1. The processor also needs to convert virtual addresses of meta-data into physical addresses. For this we use a TLB; in practice, we should not use the processor core's standard TLB and should use a second TLB to avoid increasing the latency of the standard TLB. The second TLB is also tagged with process identifier bits which are combined with virtual addresses to translate to physical addresses.

In many programs virtual memory is sparsely filled, with the stack at high addresses, and code and heap at low ad-

---

[2] If the block that is being evicted is clean, it is simply evicted from the cache, and not written back to memory. This avoids incrementing TIMER in the processor and updating `TS` in memory; this implies that we do not need to update `EB` by decrypting and re-encrypting with a new time stamp.

dresses. Note that our method does not require actual physical space for time stamps of unused virtual memory space. Physical memory for time stamps can be allocated on demand.

5.3 Untrusted I/O

Our integrity verification and encryption allow only the primary processor to access off-chip memory. For untrusted I/O such as Direct Memory Access (DMA), a part of memory is set aside as an unprotected and unencrypted area. When the transfer is done into this area, a trusted application or OS copies the data into protected memory space, and checks/decrypts it using a scheme of its choosing.

6. EVALUATION

This section evaluates our integrity verification and encryption schemes compared to the existing schemes through analysis and detailed simulations.

6.1 Space Overhead

We first evaluate the memory space overhead and logic overhead of the integrity verification and encryption schemes.

6.1.1 Integrity Verification

Integrity checking schemes need memory space in addition to the data they verify for hashes or time stamps. The additional memory space compared to data chunks is approximately $1/(m_{CHTree} - 1)$ for `CHTree` with a m_{CHTree}-ary hash tree and B_{TS}/B_{Chunk} for `LHash`. For typical values ($m_{CHTree} = 4$, $B_{TS} = 4$ Bytes, $B_{Chunk} = 64$ Bytes), the overheads are 33% for `CHTree` and 6.25% for `LHash`, respectively. Therefore, `LHash` has significantly less memory space overhead compared to the `CHTree` scheme. Note that increasing the arity of the hash tree for less space overhead is usually not viable; it implies a larger L2 cache block, which often degrades the baseline performance without integrity verification.

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 [17] and SHA-1 [7] hashing algorithms. The core of each algorithm is an operation that takes a 512-bit block, and produces a 128-bit or 160-bit (for SHA-1) digest. In each case, simple 32-bit operations are performed over 80 rounds, which requires on the order of 625 1-bit gates per round. The logic overhead depends on how many rounds need to be implemented in parallel to meet the required throughput.

For `CHTree`, the hash of B_{Chunk} (typically 64 Bytes) needs to be computed for each memory read/write. For `LHash`, two hashes of $B_{Address} + B_{Chunk} + B_{TS}$ (typically 72 Bytes) for each memory read. Therefore, `LHash` would require about 2-3 times of hash throughput and would have about 2-3 times more logic overhead compared to `CHTree`. For a memory throughput of 1.6GB/s, the circuit size will be around 5,000 1-bit gates for `CHTree` and 10,000 to 15,000 1-bit gates for `LHash`.

6.1.2 Encryption

Both direct block encryption and one-time-pad encryption can use the same size random initial vectors and time stamps. Therefore, the memory space overhead will be the same for both schemes. For typical 4-B random vectors and time stamps with 64-B cache block, the overhead is 6.25%.

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
AES latency	40 cycles
AES throughput	3.2 GB/s
Hash latency	160 cycles
Hash throughput	3.2 GB/s
Hash length	128 bits
Time stamps	32 bits
Time stamp buffer	32 8-B entries

Table 1: Architectural parameters.

The main area overhead of the encryption schemes is the logic required by the AES algorithm. Given the gate counts of the AES implementation of [19], a 128-bit AES encryption without pipelining costs approximately 75,000 gates. For 1.6GB/s throughput, the module needs to be duplicated four times, which corresponds to the order of 300,000 gates. Using a simpler encryption algorithm such as RC5 [18] can substantially decrease the gate count, at the cost of decreased security. Both direct encryption and one-time-pad encryption will have the same logic overhead because they require the same encryption throughput.

6.2 Simulation Framework

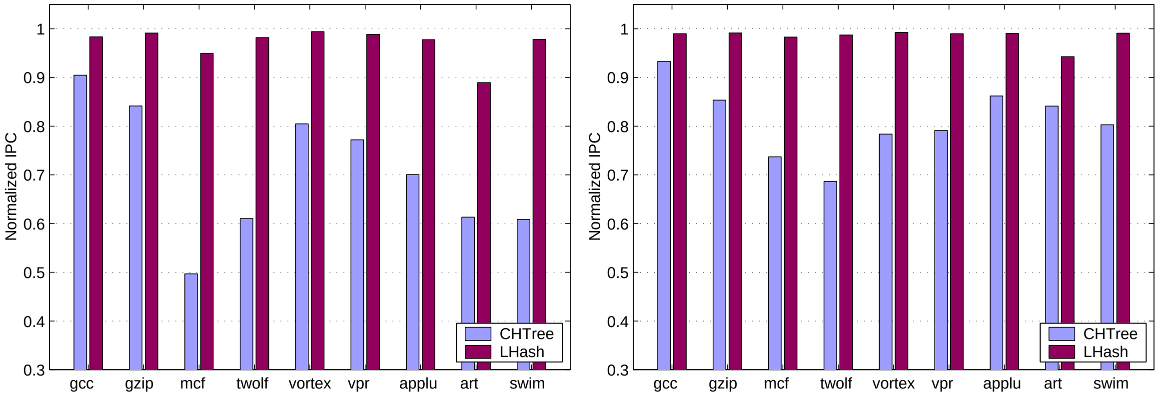
Our simulation framework is based on the SimpleScalar tool set [3]. The simulator models speculative out-of-order processors with separate address and data buses. All structures that access the main memory including an L2 cache, the integrity checking unit, and the encryption unit share the same bus. 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. We used a small buffer for time stamps to exploit spatial locality because time stamps are only 4 B while the memory bus is 8-B wide.

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.

6.3 Baseline Characteristics

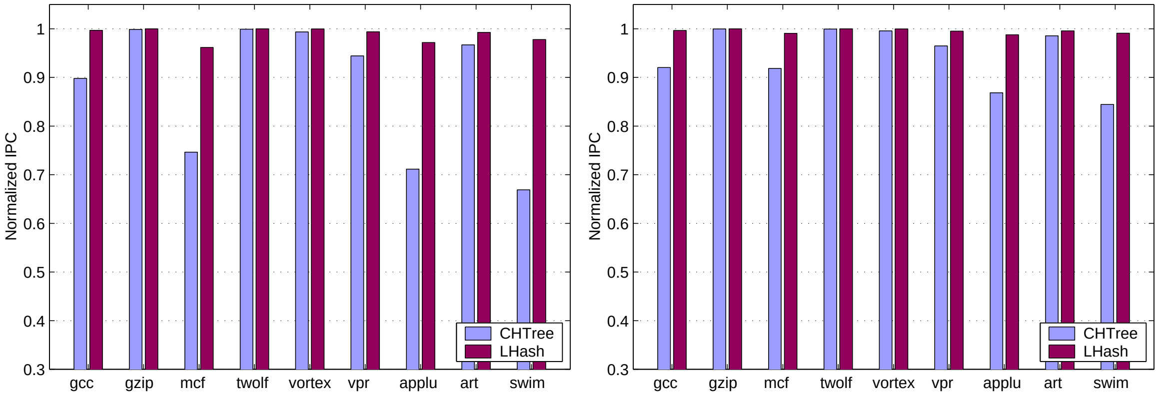
For all the experiments in this section, nine SPEC2000 CPU benchmarks [10] are used as representative applications. The baseline characteristics of these benchmarks are illustrated in Figure 8. Benchmarks `mcf`, `applu`, and `swim` show poor L2 cache performance, and heavily utilize the off-chip memory bandwidth (`bandwidth-sensitive`). The other benchmarks are sensitive to cache sizes, and do not require high off-chip bandwidth (`cache-sensitive`).

6.4 Integrity Verification



(a) 256KB, 64B

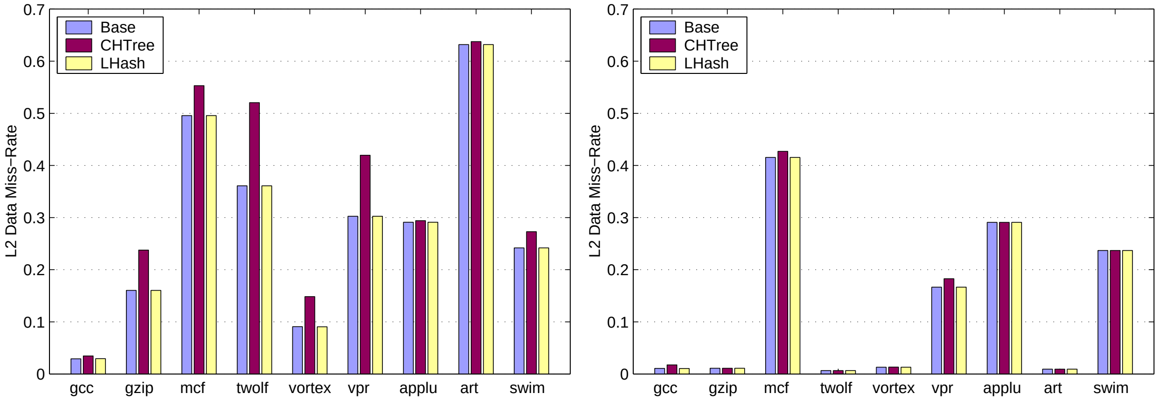
(b) 256KB, 128B



(c) 4MB, 64B

(d) 4MB, 128B

Figure 9: Run-time performance overhead of memory integrity checking: cached hash trees (`CHTree`) and log-hashes (`LHash`). Results are shown for two different cache sizes (256KB, 4MB) with cache block size of 64B and 128B. 32-bit time stamps and 128-bit hashes are used.



(a) 256KB, 64B

(b) 4MB, 64B

Figure 10: L2 cache miss-rates of program data for a standard processor (`Base`) and the ones with memory verification schemes (`CHTree` and `LHash`). The results are shown for 256-KB and 4-MB caches with 64-B cache blocks.

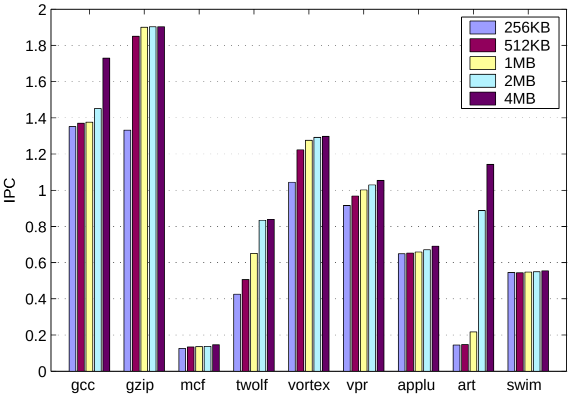


Figure 8: Baseline performance of simulated benchmarks. Results for five different L2 caches with 64-B blocks are shown.

This subsection evaluates the log hash integrity checking scheme `LHash` and compares it to the hash tree scheme `CHTree`. For all integrity checking schemes, we used chunks that are the same as L2 cache blocks.

6.4.1 Run-Time Performance

We first investigate the performance of the `LHash` scheme ignoring the overhead of the integrity-check operation. Most applications only need to verify their operations at the end of execution or relatively infrequently when they export results. In this case, the overhead of the integrity-check operation is negligible and the results in this section represent the overall performance.

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 effect of frequent integrity checking is studied in the following subsection.

Figure 9 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 cached hash trees (`CHTree`) and log-hashes (`LHash`) are shown. The IPCs are normalized to the baseline performance with the same configuration.

The experimental results clearly demonstrate the advantage of the log-hash scheme (`LHash`) over the conventional hash tree scheme when we can ignore the integrity-check overhead. For all cases we simulated, `LHash` outperforms `CHTree`. The performance overhead of the `LHash` scheme is often less than 5% and less than 15% even for the worst case. On the other hand, the cached hash tree `CHTree` has as much as 50% overhead in the worst case and 20-30% in general.

The figure also demonstrates the general effects of cache configuration on the memory integrity verification performance. The overhead of integrity checking decreases as we increase either cache size or cache block size. Larger caches result 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. However, we note that increasing the cache block size beyond an optimal point degrades the baseline performance.

Memory integrity checking impacts the run-time performance in two ways: *cache pollution* and *bandwidth consumption.*

Cache Pollution. Figure 10 illustrates the effects of integrity checking on cache miss-rates. Since `LHash` does not store hashes in the cache, it does not affect the L2 miss-rate. However, `CHTree` can significantly increase miss-rates for small caches since it stores its hash nodes in the L2 cache with program data. 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 9) 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 10 (b)).

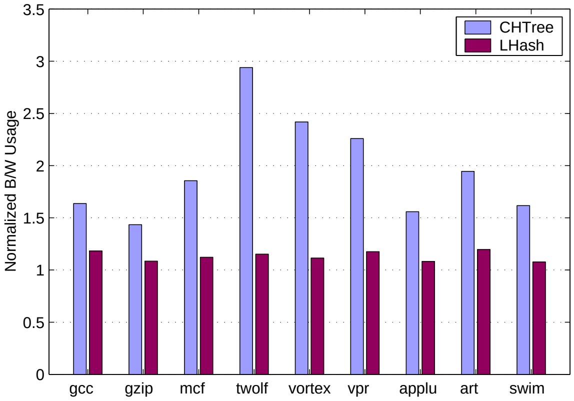
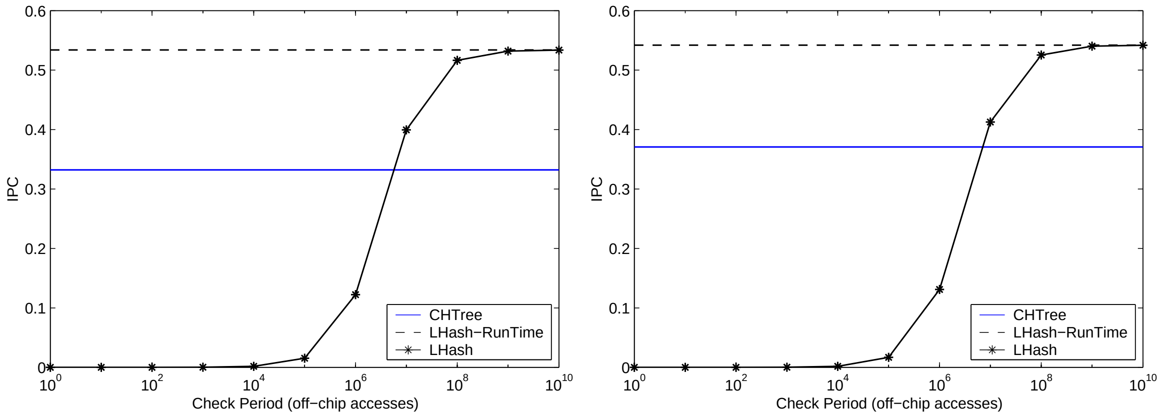


Figure 11: Off-chip bandwidth consumption of memory verification schemes (`CHTree` and `LHash`). The L2 cache is 1 MB with 64-B cache blocks. The bandwidth consumption is normalized to the baseline case.

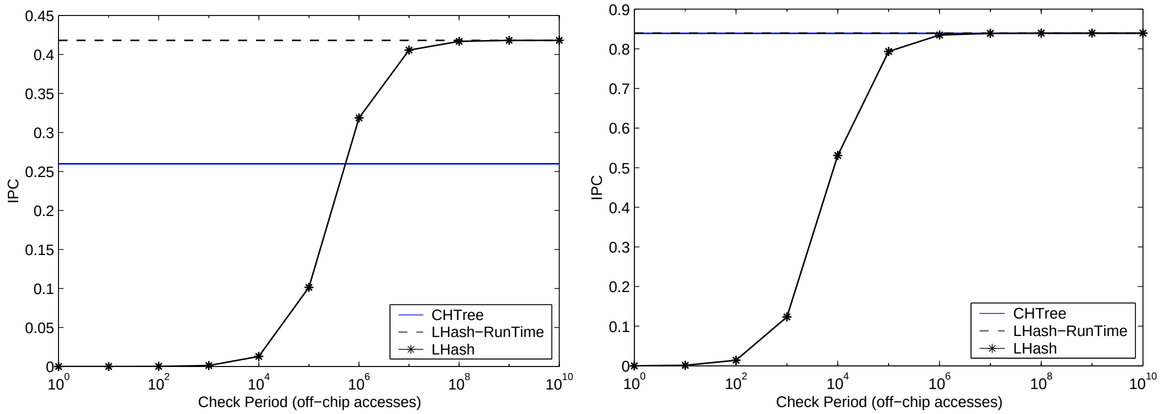
Bandwidth Consumption. The bandwidth consumptions of the integrity checking schemes are shown in Figure 11. The `LHash` scheme theoretically consumes 6.25% to 12.5% of additional bandwidth compared to the baseline. In our processor implementation, however, it consumed more (8.5% to 20%) because our bus width is 8B while the time stamps are only 4B. The `CHTree` scheme consumes additional bandwidth depending on the L2 cache performance on hashes. Because `CHTree` needs a deep tree, it has significant bandwidth overhead. 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.

6.4.2 Overall Performance

The last subsection clearly demonstrated that the `LHash` scheme outperforms the hash tree scheme 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 com-



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



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

Figure 12: Performance comparison between `LHash` and `CHTree` for various checking periods. `LHash-RunTime` indicates the performance of the `LHash` scheme without checking overhead. 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.

pare the integrity checking schemes including the overhead of periodic integrity-check operations.

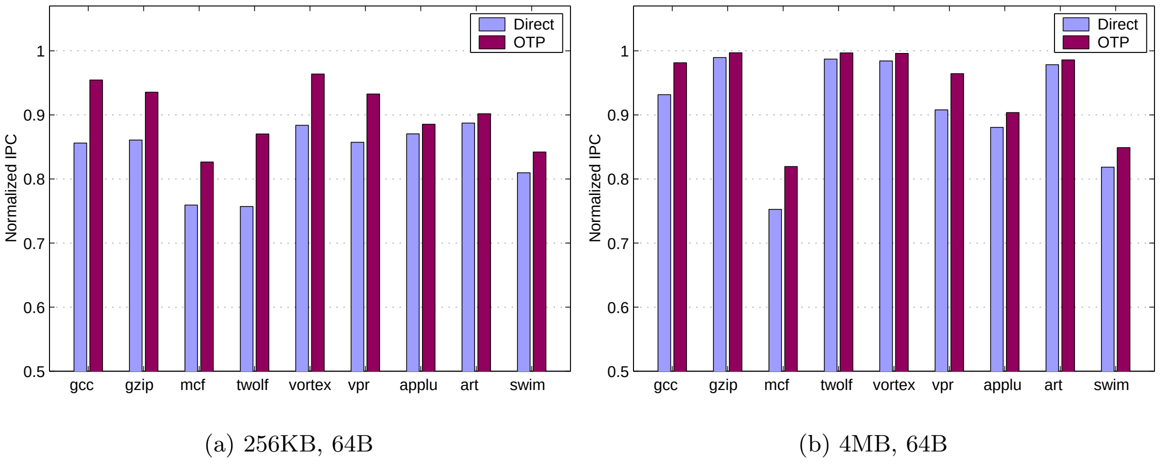
We assume that the log-hash schemes check memory integrity every T memory accesses. A processor executes a program until it makes T main memory accesses, then checks the integrity of the T accesses by performing an integrity-check operation. Obviously, the overhead of the checking heavily depends on 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 12 compares the performance of the memory integrity checking schemes for varying check periods. The performance of the conventional `CHTree` scheme is indifferent to the checking period since it has no choice but to check the integrity after each access. Effectively, this scheme always has a checking period of one memory access.

On the other hand, the performance of the log-hash scheme (`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. 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, `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 than `mcf` because it needs to read less amount of data per check. The break-even point can be reduced by an order of magnitude by making the `LHash` scheme hierarchical. However, we will not detail this scheme for space reasons.

6.5 Encryption Performance

Figure 13 compares the direct encryption mechanism with the one-time-pad encryption mechanism. The instructions



(a) 256KB, 64B

(b) 4MB, 64B

Figure 13: The performance overhead of direct encryption and one-time-pad encryption.

per cycle (IPC) of each benchmark is normalized by the IPC of standard processor without encryption. In the experiments, we simulated the case when *all instructions and data are encrypted in the memory*. Both encryption mechanisms degrade the processor performance by consuming additional memory bandwidth for either time stamps or initial vectors, and by delaying the data delivery for decryption.

As shown in the figure, the memory encryption for these configurations results in up to 18% performance degradation for the one-time-pad encryption, and 25% degradation for the direct encryption. On average, the one-time-pad scheme reduces the overhead of the direct encryption by 43%. Our scheme is particularly effective when the decryption latency is the major concern. For applications with low bandwidth usage such as `gcc`, `gzip`, `twolf`, `vortex`, and `vpr`, the performance degradation mainly comes from the decryption latency. In these applications, our scheme reduces the overhead of the conventional scheme by more than one half.

6.5.1 *Impact of Memory Bandwidth*

Our base configuration assumes the memory bandwidth of 1.6GB/s, which corresponds to 5 processor cycles per 8-B memory transfer in our case. Modern microprocessors are beginning to have higher bandwidth with the development of new memory and interconnect technologies.

First, Figure 14 (a) shows the impact of higher bandwidth on the direct encryption overhead. For applications with low bandwidth usage, higher off-chip bandwidth reduces the performance degradation of direct encryption because accessing time stamps and initial vectors incurs relatively less overhead. However, for bandwidth-sensitive applications, higher bandwidth can significantly increase the overhead of direct encryption. With high bandwidth, the performance is more sensitive to the memory latency because it is not limited by the bandwidth anymore. Therefore, the encryption latency becomes more significant portion of memory latency.

On the other hand, the overhead of the one-time-pad encryption slightly increases as the bandwidth increases (see Figure 14 (b)). With higher bandwidth, it takes less time to transfer a cache block from off-chip memory after reading a time stamp. As a result, the scheme can overlap less of the encryption computation with the memory access. Larger caches for time stamps are required to solve this problem.

For low-bandwidth applications, the benefit of our one-time-pad scheme over the direct encryption scheme slightly decreases with higher bandwidth. However, the benefit of our scheme for high-bandwidth applications significantly increases with higher bandwidth available. For example, with 4GB/s bandwidth, the one-time-pad scheme has 12% less overhead than the direct encryption for `swim` while it was only 3% better with 1.6GB/s bandwidth.

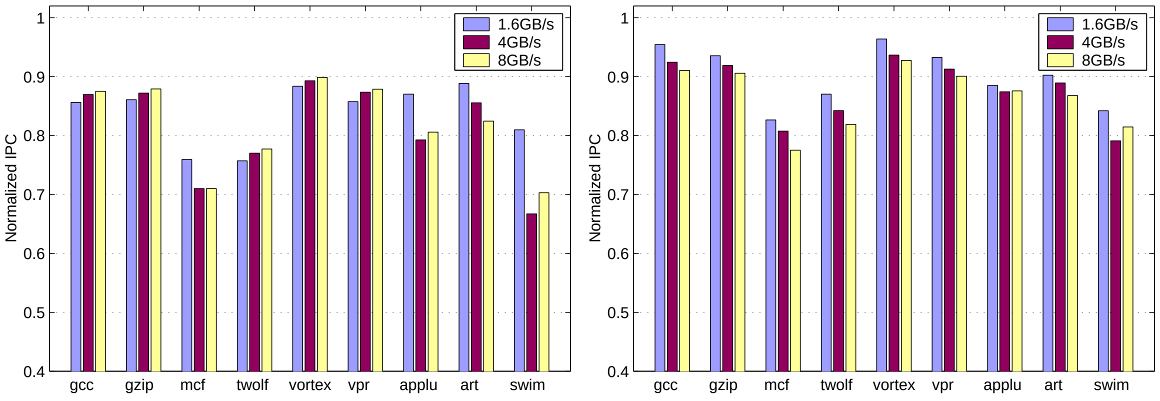
6.5.2 *Re-Encryption Period*

As noted in Section 4, the one-time-pad encryption mechanism requires re-encrypting the memory when the global time stamp reaches its maximum value. Because the re-encryption operation is rather expensive, the time stamp should be large enough to either amortize the re-encryption overhead or avoid the re-encryption itself.

Fortunately, the simulation results for the SPEC benchmarks show that even 32-bit time stamps are large enough. In our experiments, the processor writes back to memory every 4800 cycles when averaged over all the benchmarks, and 131 cycles in the worst case of `swim`. Given the maximum time stamp size of 4 billion, this indicates the re-encryption needs to be done on every 5.35 hours (in our 1 GHz processor) on average, or 35 minutes for `swim`. For our benchmarks, the re-encryption takes less than 300 million cycles even for `swim` that has the largest working set. Therefore, the re-encryption overhead is negligible in practice. If the 32-bit time stamps are not large enough, the encryption period can be increased by having larger time stamps or per-page time stamps.

6.6 PTR Processing

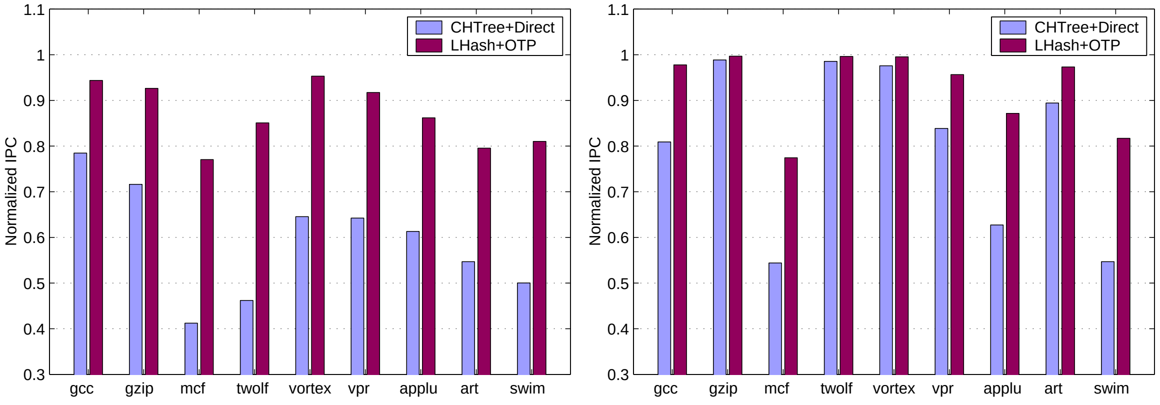
Finally, we study the performance of the PTR processing by simulating integrity verification and encryption together. Previous work [21] has shown that these two mechanisms are the primary concerns for the performance of the PTR processing. We compare the performance using our new schemes with the performance using `CHTree` and direct block encryption. In the new schemes, two separate time stamps



(a) Direct Encryption

(b) One-Time-Pad Encryption

Figure 14: The impact of memory bandwidth on the memory encryption overhead. The results are shown for a 1-MB cache with 64-B blocks.



(a) 256KB, 64B

(b) 4MB, 64B

Figure 15: The performance overhead of PTR processing with the conventional schemes and the new schemes.

are used for integrity verification and encryption.

Figure 15 demonstrates that our new schemes significantly improve the performance of PTR processing over the existing schemes. With existing schemes, PTR processing incurs up to 60% performance degradation in the worst case (`mcf`), and around 40% overhead in most cases. With `LHash` and one-time-pad encryption, PTR processing can be done with 23% overhead even in the worst case, and less than 15% in most cases.

7. RELATED WORK

Blum et al. [2] addressed the problem of securing various data structures in untrusted memory. They proposed using a hash tree rooted in trusted memory to check the integrity of arbitrarily large untrusted RAM. Their approach has a $O(\log(N))$ cost for each memory access. [9] shows how caching of internal nodes of the tree can significantly improve the performance the scheme. The log hash scheme we introduce can perform better than a hash-tree based scheme because it checks sequences of memory operations, rather than checking each operation. Blum et al. [2] also proposed an offline checker to check the correctness of RAM after a sequence of operations have been performed on RAM. Their scheme computes a running hash of memory reads and writes. We have used their offline checker as a basis for designing our log hash checker, though there are key differences between the two checkers. Their checker's implementation uses ϵ-biased hash functions [15]; these hash functions can be used to detect random errors, but are not cryptographically secure. For our log hash checker, we have used incremental multiset hashes [6], which are cryptographically secure. Furthermore, our log hash checker can use smaller time stamps without increasing the frequency of checks, which leads to better performance.

Previous designs of secure processors [12, 21] directly use encryption algorithms such as DES, Triple DES, and AES to encrypt and decrypt memory blocks; this can appreciably increase memory access latency for reads. We have used one-time pads to hide virtually all the decryption latency.

8. CONCLUSION

Memory integrity verification and encryption are key primitives required to implement secure computing systems with trusted processors and untrusted memory components. They are also responsible for almost all of the performance overhead of tamper-evident and private tamper-resistant processing. We have presented a new scheme for memory verification based on maintaining an incremental hash of logs of memory operations, and a new encryption scheme based on one-time-pads and time stamps. The new schemes significantly reduce the performance overhead of secure processors, and make these processors usable over a wider range of applications.

9. REFERENCES

[1] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* John Wiley and Sons, 2001.

[2] 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.

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

[4] A. Carroll, M. Juarez, J. Polk, and T. Leininger. Microsoft "Palladium": A Business Overview. In *Microsoft Content Security Business Unit*, August 2002.

[5] J. Claessens, B. Preneel, and J. Vandewalle. (how) can mobile agents do secure electronic transactions on untrusted hosts? a survey of the security issues and the current solutions. *ACM Transactions on Internet Technology*, 3, Feb. 2003.

[6] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental multiset hash functions and their application to memory integrity checking. In *Technical Report MIT-LCS-TR-899*, May 2003.

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

[8] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Silicon Physical Random Functions . In *Proceedings of the Computer and Communication Security Conference*, May 2002.

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

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

[11] H. Kuo and I. M. Verbauwhede. Architectural optimization for a 1.82 gb/s vlsi implementation of the aes rijndael algorithm. In *Cryptographic Hardware and Embedded Systems 2001 (CHES 2001), LNCS 2162*, 2001.

[12] 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* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 169–177, November 2000.

[13] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES-modes of operations: CTR-mode encryption. In *Symmetric Key Block Cipher Modes of Operation Workshop*, Baltimore, Maryland, USA, 2000.

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

[15] 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.

[16] N. I. of Science and Technology. FIPS PUB 197: Advanced Encryption Standard (AES), November 2001.

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

[18] R. L. Rivest. The RC5 encryption algorithm, from dr. dobb's journal, january, 1995. In *William Stallings, Practical Cryptography for Data Internetworks, IEEE Computer Society Press, 1996.* 1996.

[19] P. R. Schaumont, H. Kuo, and I. M. Verbauwhede. Unlocking the design secrets of a 2.29 gb/s rijndael processor. In *Design Automation Conference 2002*, June 2002.

[20] 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.

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

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