Towards Constant Bandwidth Overhead Integrity Checking of Untrusted Data*

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

May 14, 2005

Abstract

We present an adaptive tree-log scheme to improve the performance of checking the integrity of arbitrarily-large untrusted data, when using only a small fixed-sized trusted state. Currently, hash trees are used to check the data. In many systems that use hash trees, programs perform many data operations before performing a critical operation that exports a result outside of the program's execution environment. The adaptive tree-log scheme we present uses this observation to harness the power of the constant runtime bandwidth overhead of a log-based scheme. For all programs, the adaptive tree-log scheme's bandwidth overhead is guaranteed to never be worse than a parameterizable worst case bound. Furthermore, for all programs, as the average number of times the program accesses data between critical operations increases, the adaptive tree-log scheme's bandwidth overhead moves from a logarithmic to a constant bandwidth overhead.

1. Introduction

This paper studies the problem of checking the integrity of operations performed on an arbitrarily-large amount of untrusted data, when using only a small fixed-sized trusted state. Commonly, hash trees [1] are used to check the integrity of the operations. The hash tree checks data *each* time it is accessed and has a *logarithmic bandwidth overhead* as an extra logarithmic number of hashes must be read each time the data is accessed.

One proposed use of a hash tree is in a single-chip secure processor [7, 9, 11], where it is used to check the integrity of external memory. A secure processor can be used to help license software programs, where it seeks to provide the programs with private, tamper-evident execution environments. In such an application, an adversary's job is to get the processor to unintentionally sign incorrect results or unintentionally reveal private instructions or private data in plaintext. Thus, assuming covert channels are protected by techniques such as memory obfuscation [4, 9], with regards to security, the *critical* instructions are the instructions that export plaintext outside of the program's execution environment, such as the instructions that sign certificates certifying program results and the instructions that export plaintext data to the user's display. It is common for programs to perform millions of instructions, and perform millions of memory accesses, before performing a critical instruction. As long as the sequence of memory operations is checked when the critical instruction is performed, it is not necessary to check each memory operation as it is performed and using a hash tree to check the memory may be causing unnecessary overhead.

In [2, 10], a new scheme, referred to as a *log-hash* scheme, was introduced to securely check memory. Intuitively, the processor maintains a "write log" and a "read log" of its write and read operations to the external memory. At runtime, the processor updates the logs with a minimal *constant-sized bandwidth 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-sized trusted space in the processor, the processor uses incremental multiset hash functions [2] to update the logs. When the processor needs to check a sequence of its operations, it performs a separate *integrity-check* operation using the logs. The integrity-check operation is performed when the program performs a critical instruction: a critical instruction acts as a signal indicating when it is necessary to perform the integrity-check operation. (Theoretically, the hash tree checks each memory operation as it is performed. However, in a secure

* This memo is an extended and updated version of the paper that appears in the *2005 IEEE Symposium on Security and Privacy*, May 2005.

processor implementation, because the latency of verifying values from memory can be large, the processor "speculatively" uses instructions and data that have not yet been verified, performing the integrity verification in the background. Whenever a critical instruction occurs, the processor waits for all of the integrity verification to be completed before performing the critical instruction. Thus, the notion of a critical instruction that acts as signal indicating that a sequence of operations must be verified is already present in secure processor hash tree implementations.)

While the log-hash scheme does not incur the logarithmic bandwidth overhead of the hash tree, its integrity-check operation needs to read all of the memory that was used *since the beginning of the program's execution*. When integrity-checks are infrequent, the number of memory operations performed by the program between checks is large and the amortized cost of the integrity-check operation is very small. The bandwidth overhead of the log-hash scheme is mainly its constant-sized runtime bandwidth overhead, which is small. This leads the log-hash scheme to perform very well and to significantly outperform the hash tree when integrity-checks are infrequent. However, when integrity checks are frequent, the program just uses a small subset of addresses that are protected by the log-hash scheme between the checks. The amortized cost of the integrity-check operation is large. As a result, the performance of the log-hash scheme is not good and is much worse than that of the hash tree. Thus, though the log-hash scheme performs very well when checks are infrequent, it cannot be widely-used because its performance is poor when checks are frequent.

In this paper, we introduce secure *tree-log* integrity checking. This hybrid scheme of the hash tree and log-hash schemes captures the best features of both schemes. The untrusted data is originally protected by the tree, and subsets of it can be optionally and dynamically moved from the tree to the log-hash scheme. When the log-hash scheme is used, *only the addresses of the data that have been moved to the log-hash scheme since the last log-hash integrity check* need to be read to perform the next log-hash integrity check, instead of reading all of the addresses that the program used since the beginning of its execution. This optimizes the log-hash scheme, facilitating much more frequent log-hash integrity checks, making the log-hash approach more widely-applicable.

The tree-log scheme we present has three features. Firstly, the scheme *adaptively* chooses a *tree-log strategy* for the program that indicates how the program should use the tree-log scheme when the program is run. This allows programs to be run unmodified and still benefit from the tree-log scheme's features. Secondly, even though the scheme is adaptive, it is able to provide a guarantee on its worst case performance such that, for all programs, the performance of the scheme is guaranteed to never be worse than a *parameterizable worst case bound*. The third feature is that, for all programs, as the average number of per data program operations (total number of program data operations/total number of data accessed) between critical operations increases, the performance of the tree-log integrity checking moves from *a logarithmic to a constant bandwidth overhead*.

With regards to the second feature, the worst-case bound is a parameter to the adaptive tree-log scheme. The bound is expressed relative to the bandwidth overhead of the hash tree, if the hash tree had been used to check the integrity of the data during the program's execution. For instance, if the bound is set at 10%, then, for all programs, the tree-log bandwidth overhead is guaranteed to be less than 1.1 times the hash tree bandwidth overhead. This feature is important because it allows the adaptive tree-log scheme to be turned on by default in applications. To provide the bound, we introduce the notion of a *reserve* to determine when data should just be kept in the tree and to regulate the rate at which data is added to the log-hash scheme. The adaptive tree-log scheme is able to provide the bound even when no assumptions are made about the program's access patterns and even when the processor uses a cache, about which minimal assumptions are made (the cache only needs to have a deterministic cache replacement policy, such as the least recently used (LRU) policy).

With regards to the third feature, the adaptive tree-log scheme is able to approach a constant bandwidth data integrity checking overhead because it can use the optimized log-hash scheme to check sequences of data operations before a critical operation is performed. The longer the sequence, the more the data that the tree-log scheme moves from the tree to the log-hash scheme and the more the overhead approaches the constant-runtime overhead of the log-hash scheme. As programs typically perform many data operations before performing a critical operation, there are large classes of programs that will be able to take advantage of this feature to improve their data integrity checking performance. (We note that we are actually stating the third feature a bit imprecisely in this section. After we have described the adaptive tree-log scheme, we will state the feature more precisely for the case without caching in Section 6.3, and modify the theoretical claims on the feature for the case with caching in Section 6.4.)

While the paper is primarily focused on providing the theoretical foundation for the adaptive tree-log scheme, we present some experimental results showing that the bandwidth overhead can be significantly reduced when the adaptive tree-log scheme is used, compared to when a hash tree is used. In light of the algorithm's features and the results, we provide a discussion in Appendix E on tradeoffs that a system designer may consider making when implementing the scheme in his system.

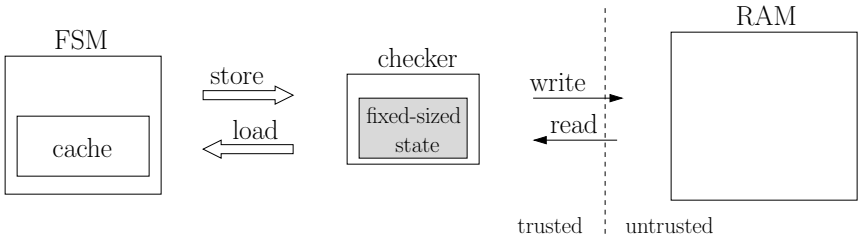


Figure 1. Model

Hash trees have been implemented in both software and hardware applications. For simplicity, throughout this paper, we will use secure processors and memory integrity checking as our example application. However, the adaptive tree-log algorithm can be implemented anywhere hash trees are currently being used to check untrusted data. The application can experience a significant benefit if programs can perform sequences of data operations before performing a critical operation.

The paper is organized as follows. Section 2 describes related work. Section 3 presents our model. Section 4 presents background information on memory integrity checking: it describes the *hash-tree* and *log-hash checkers*. Section 5 details our *tree-log checker*. Section 6 describes our *adaptive tree-log checker*. Section 7 provides an experimental evaluation of the adaptive tree-log checker. Section 8 concludes the paper. The appendices provide various supplemental material.

2. Related Work

The use of a hash tree (also known as a Merkle tree [8]) to check the integrity of untrusted memory was introduced by Blum et al. [1]. The paper also introduced a log-based scheme to check the correctness of memory. The log-based scheme in [1] could detect random errors, but it was not secure against active adversaries. The log-hash scheme that the tree-log scheme uses is secure against an active adversary. It is also more efficient than the log-based scheme in [1] because time stamps can be smaller without increasing the frequency of checks. Log-based schemes, by themselves, are not general enough because they do not perform well when integrity checks are frequent. The tree-log scheme can use the tree when checks are frequent and move data from the tree to the log-hash scheme as sequences of operations are performed to take advantage of the constant runtime bandwidth overhead of the log-hash scheme.

Hall and Jutla [5] propose parallelizable authentication trees. In a standard hash tree, the hash nodes along the path from the leaf to the root can be verified in parallel. Parallelizable authentication trees also allow the nodes to be updated in parallel on store operations. The log-hash scheme could be integrated into these trees in a manner similar to how we integrate it into a standard hash tree. However, the principal point is that trees still incur a logarithmic bandwidth overhead, whereas our tree-log scheme can reduce the overhead to a constant bandwidth overhead.

3. Model

Figure 1 illustrates the model we use. There is a *checker* that keeps and maintains some *small, fixed-sized, trusted state*. The *untrusted RAM* (main memory) is arbitrarily large. The *finite state machine (FSM)* generates loads and stores and the checker updates its trusted state on each FSM load or store to the untrusted RAM. The checker uses its trusted state to verify the integrity of the untrusted RAM. The FSM may also maintain a fixed-sized trusted *cache*. The cache is initially empty, and the FSM stores data that it frequently accesses in the cache. Data that is loaded into the cache is checked by the checker and can be trusted by the FSM.

The FSM is the unmodified processor running a user program. The processor can have an on-chip cache. The checker is special hardware that is added to the processor. The trusted computing base (TCB) consists of the FSM with its cache and the checker with its trusted state.

The problem that this paper addresses is that of checking if the untrusted RAM behaves like valid RAM. *RAM behaves like valid RAM if the data value that the checker reads from a particular address is the same data value that the checker most recently wrote to that address.*

In our model, the untrusted RAM is assumed to be actively controlled by an adversary. The adversary can perform any software or hardware-based attack on the RAM. The untrusted RAM may not behave like valid RAM if the RAM has malfunctioned because of errors, or if the data stored has somehow been altered by the adversary. We are interested in *detecting* whether the RAM has been behaving correctly (like valid RAM) during the execution of the FSM. The adversary could corrupt the entire contents of the RAM and there is no general way of recovering from tampering

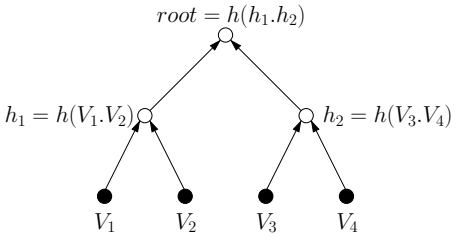


Figure 2. A binary hash tree.

other than restarting the program execution from scratch; thus, we do not consider recovery methods in this paper.

For this problem, a simple approach such as calculating a message authentication code (MAC) of the data value and address, writing the (data value, MAC) pair to the address and using the MAC to check the data value on each read, does not work. The approach does not prevent replay attacks: an adversary can replace the (data value, MAC) pair currently at an address with a different pair that was previously written to the address.

We define a critical operation as one that will break the security of the system if the FSM performs it before the integrity of all the previous operations on the untrusted RAM is verified. The checker must verify whether the RAM has been behaving correctly (like valid RAM) when the FSM performs a critical operation. Thus, the FSM implicitly determines when it is necessary to perform checks based on when it performs a critical operation. It is not necessary to check each FSM memory operation as long as the checker checks the sequence of FSM memory operations when the FSM performs a critical operation.

4. Background

4.1. Hash Tree

The scheme with which we compare our work is integrity checking using hash trees. Figure 2 illustrates a hash tree. The data values are located at the leaves of the tree. Each internal node contains a collision resistant hash of the concatenation of the data that is in each one of its children. The root of the tree is stored in the trusted state in the checker where it cannot be tampered with.

To check the integrity of a node, the checker: 1) reads the node and its siblings, 2) concatenates their data together, 3) hashes the concatenated data and 4) checks that the resultant hash matches the hash in the parent. The steps are repeated on the parent node, and on its parent node, all the way to the root of the tree. To update a node, the checker: 1) checks the integrity of the node's siblings (and the old value of the node) via steps 1-4 described previously, 2) changes the data in the node, hashes the concatenation of this new data with the siblings' data and updates the parent to be the resultant hash. Again, the steps are repeated until the root is updated.

On each FSM load from address a, the checker checks the path from a's data value leaf to the trusted root. On each FSM store of value v to address a, the checker updates the path from a's data value leaf to the trusted root. We refer to these load and store operations as hash-tree-load(a) and hash-tree-store(a, v). The number of accesses to the RAM on each FSM load/store is logarithmic in the number of data values that are being protected.

Given the address of a node, the checker can *calculate* the address of its parent [3, Section 5.6]. Thus, given the address of a leaf node, the checker can calculate the addresses of all of the nodes along the path from the leaf node to the root.

A cache can be used to improve the performance of the scheme (the model in Section 3 is augmented such that the checker is able to read and write to the cache as well as to the untrusted RAM). Instead of just storing recently-used data values, the cache can be used to store both recently-used data values and recently-used hashes. A node and its siblings are organized as a block in the cache and in the untrusted RAM. Thus, whenever the checker fetches and caches a node from the untrusted RAM, it also simultaneously fetches and caches the node's siblings, because they are necessary to check the integrity of the node. Similarly, when the cache evicts a node, it also simultaneously evicts the node's siblings.

The FSM trusts data value blocks stored in the cache and can perform accesses directly on them without any hashing. When the cache brings in a data value block from RAM, the checker checks the path from the block to the root or to the first hash along that path that it finds in the cache. The data value block, along with the hash blocks used in the verification, are stored in the cache. When the cache evicts a data value or hash block, if the block is clean, it is just removed from the cache. If the block is dirty, the checker checks the integrity of the parent block and brings it into the cache, if it is not already in the cache. The checker then updates the parent block in the cache to contain the hash of the evicted block. An invariant of this caching algorithm is that hashes of uncached blocks must be valid whereas hashes of cached blocks can have arbitrary values.

4.2. Log-Hash

The essence of the log-hash scheme [2, 10] is that the checker maintains a "write log" and a "read log" of its write and read operations to the untrusted RAM. Figure 3 shows the basic put and take operations that are used internally in the checker. Figure 4 shows the interface the FSM calls to use the log-hash checker to check the integrity of the RAM.

The checker's fixed-sized state consists of two multiset hashes, WRITEHASH and READHASH, and one counter, TIMER. Initially, the hashes are $\mathcal{H}(\emptyset)$ and the counter is 0.

put(a, v): writes a value v to address a in the untrusted RAM:

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

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

1. Read (v, t) from a in the untrusted RAM.
2. Update READHASH: READHASH $+_{\mathcal{H}}$= hash(a, v, t).
3. TIMER = max(TIMER, $t + 1$).

Figure 3. put **and** get **operations**

log-hash-add(a, v): put(a, v).

log-hash-store(a, v): take(a); put(a, v).

log-hash-load(a): v = take(a); return v to the caller; put(a, v).

log-hash-check$()$: checks if the RAM has behaved like valid RAM (at the end of operation):
take(a) for each address a. If WRITEHASH is equal to READHASH, return true; else, return false.

Figure 4. Log-hash checker for untrusted RAM

In Figure 3, the checker maintains two multiset hashes [2] and a counter. In the untrusted RAM, each data value is accompanied by a time stamp. Each time the checker performs a put operation, it appends the current value of the counter (a time stamp) to the data value, and writes the (data value, time stamp) pair to memory. When the checker performs a take operation, it reads the pair stored at an address and, if necessary, updates the counter so that it is strictly greater than the time stamp that was read. The multiset hashes are updated ($+_{\mathcal{H}}$) with (a, v, t) triples corresponding to the pairs written or read from the RAM.

Figure 4 shows how the checker implements the store-load interface. To initialize an address, the checker calls log-hash-add$(a, 0)$ exactly once on each address that the FSM uses to put an initial value at the address. When the FSM performs a log-hash-store operation, the checker takes the original value at the address, then puts the new value to the address. When the FSM performs a log-hash-load operation, the checker takes the original value at the address and returns this value to the FSM; it then puts the same value back to the address (only the time stamp needs to be written to RAM). To check the RAM at the end of a sequence of FSM operations[1], the checker calls log-hash-check$()$ which takes the value at each address, then compares WRITEHASH and READHASH. If WRITEHASH is equal to READHASH, the checker concludes that the RAM has been behaving correctly, i.e., like valid RAM. Intermediate checks can be performed with a slightly modified log-hash-check operation [2] (also, cf. Section 5).

Because the checker checks that WRITEHASH is equal to READHASH, substitution (the RAM returns a value that is never written to it) and replay (the RAM returns a stale value instead of the one that is most recently written) attacks on the RAM are prevented. The purpose of the time stamps is to prevent reordering attacks in which RAM returns a value that has not yet been written so that it can subsequently return stale data. A formal proof that the scheme is secure is in [2].

A cache can be used to improve the performance of the scheme. The cache contains *just* data value blocks. The RAM contains (value block, time stamp) pairs. When the cache brings in a block from RAM, the checker performs a take operation on the address. When the cache evicts

[1] As a note, the log-hash-check operation can be performed at any time, even after each FSM store/load operation. The log-hash scheme does not require that sequences of operations be performed before log-hash-check operations can be performed. In the model in Section 3, log-hash-checks are performed when critical operations are performed. The log-hash scheme performs very well when critical operations are infrequent, but may not perform well when critical operations are frequent.

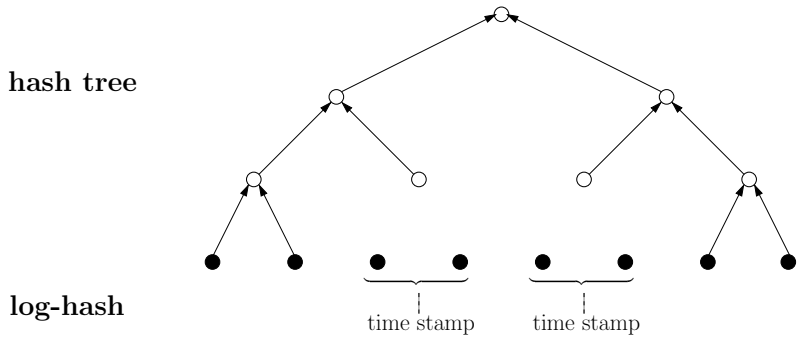


Figure 5. Tree-log checker

a block, the checker performs a put operation on the (address, value block) pair (if the block is clean, only the time stamp is written to RAM). The log-hash-check operation operates as before, except it just has to perform take operations on uncached RAM addresses.

5. Tree-Log Checker

Figure 5 illustrates the tree-log checker and Figure 6 shows the interface that the FSM calls to use the tree-log checker to check the integrity of the untrusted RAM.

tree-log-store calls hash-tree-store(a, v) if address a is in the tree or calls log-hash-store(a, v) if a is in the log-hash scheme. tree-log-load operates similarly.

tree-log-moveToLogHash first calls hash-tree-load(a) to check the integrity of the value v at address a in the RAM. The hash-tree-updateParent operation checks the integrity of the parent node of the specified address and updates the parent node to contain a hash of the specified value (the operation propagates the check and the update to the root of the tree). The `NULL` value is a value that address a cannot have, such as a value greater than the maximum possible value for address a. (Though it updates the parent node of the address, hash-tree-updateParent does not actually write a new value for the address). hash-tree-updateParent(a, `NULL`) is called to remove a from the tree. log-hash-add(a, v) is then called to add a with value v to the log-hash scheme[2].

tree-log-check checks the integrity of the RAM that is currently protected by the log-hash scheme by calling log-hash-check. tree-log-check takes an argument Y, representing a set of addresses. Each address in Y is moved back to the hash tree as the log-hash-check operation is performed. Addresses that are not in Y but are in the log-hash scheme remain in the log-hash scheme. The proof that the tree-log scheme is secure is in Appendix B.

In the event the log-hash TIMER becomes close to its maximum value before the FSM calls tree-log-check, the checker can perform tree-log-check($\emptyset$) to reset it. tree-log-check($\emptyset$) essentially performs an intermediate log-hash check on the addresses in the log-hash scheme.

The tree-log scheme allows for optimization of the log-hash scheme. All of the addresses are initially in the tree. We call the period between intermediate tree-log-check operations a *check period*. During a check period, the checker can move an arbitrary set of addresses to the log-hash scheme, where the FSM can perform store and load operations on them in the log hash scheme. When a tree-log-check operation is performed, all of the addresses in the log-hash scheme can be moved back to the tree, where their values will be remembered by the tree. During a subsequent check period, a different arbitrary set of addresses can be moved to the log-hash scheme to be used by the FSM in the log hash scheme. The benefit is that, whenever a tree-log-check operation is performed, *only the addresses of the data that have been moved to the log-hash scheme since the last tree-log-check operation* need to be read to perform the check, as opposed to reading the entire set of addresses that the FSM had used since the beginning of its execution. If the tree-log-check operation needs to read addresses that are protected by the log-hash scheme, but were not used during the check period, then the log-hash scheme is not optimal. Thus, the ability of the tree-log scheme to move the set of addresses that are accessed during a check period into the log-hash scheme and move them back into the tree on a tree-log-check operation so that a different set of addresses can be moved to the log-hash scheme during a subsequent check period, helps to optimize the bandwidth overhead of the log-hash scheme.

[2]Because of the organization of the tree, whenever an address is moved to the log-hash scheme, the block consisting of the address's data value node and its siblings is log-hash-added to the log-hash scheme (the block's address is the address of the first node in the block). If the tree was organized such that the data values are hashed first, then the tree is created over the hashes of the data values, individual data value nodes could be moved to log-hash scheme.

tree-log-store(a, v): stores v at address a:

1. If a is protected by the tree, hash-tree-store(a, v). Else, log-hash-store(a, v).

tree-log-load(a): loads the data value at address a:

1. If a is protected by the tree, hash-tree-load(a). Else, log-hash-load(a).

tree-log-moveToLogHash(a): move address a from the tree to the log-hash scheme:

1. v = hash-tree-load(a).
2. hash-tree-updateParent$(a,$ NULL$)$.
3. log-hash-add(a, v).

tree-log-check(Y): checks if the RAM (currently being protected by the log-hash scheme) has behaved like valid RAM; each of the addresses in set Y is moved from the log-hash scheme to the hash tree:

1. log-hash-check$()$.
   Also, create a new TIMER′ and WRITEHASH′. As the untrusted RAM is read to perform the log-hash-check, for each address a that is read, where v is the data value of a:
   (a) if $a \in Y$, call hash-tree-updateParent(a, v); else call log-hash-add(a, v), using TIMER′ and WRITEHASH′, to reset the time stamps in RAM and update WRITEHASH′.

   Set TIMER and WRITEHASH to TIMER′ and WRITEHASH′; reset READHASH to 0.

Figure 6. Tree-log checker for untrusted RAM

5.1. Caching

Caching is easily integrated into the tree-log scheme using the approaches described in Sections 4.1 and 4.2. If the block's address is protected by the tree, when a data value block is brought into the cache or evicted from the cache, the caching approach in Section 4.1 is used. If the block's address is protected by the log-hash scheme, the caching approach in Section 4.2 is used. tree-log-moveToLogHash brings the block and/or the block's parent into the cache if they are not already in the cache, using the approach in Section 4.1. The parent is then updated in the cache. The tree-log-check uses an approach similar to that in Section 4.2 when performing the log-hash-check operation. If the block's address is in Y, the block's parent is brought into the cache as described in Section 4.1 and updated in the cache.

5.2. Bookkeeping

In Appendix B, we prove that, with regards to security, the data structures that the checker uses to determine if an address is protected by the hash tree or if it is protected by the log-hash scheme, and to determine which addresses to read to perform a tree-log-check operation, do not have to be protected. The necessary information is already implicitly encoded in the hash tree and log hash schemes. The data structures are strictly used for bookkeeping and a system designer is free to choose any data structures that allow the checker to most efficiently perform these functions.

In our experiments in Section 7, a range of addresses is moved to the log-hash scheme when the log-hash scheme is used. The checker maintains the highest and lowest address of the range in its fixed-sized trusted state. When the checker performs a tree-log-check operation, it moves all of the addresses in the log-hash scheme to the tree, so that a separate range of addresses can be moved to the log hash scheme during a subsequent check period.

Maintaining a range of addresses in the log-hash scheme is very effective when the FSM exhibits spatial locality in its accesses, which is common for software programs. However, instead of using a range, a more general data structure would be to use a bitmap stored unprotected in RAM. Optionally, some of the bitmap could be cached. With the bitmap implementation, the checker may also maintain a flag in its trusted state. If the flag is true, the checker knows that all of the data is in the tree and it does not use the bitmap; its stores/loads perform exactly as hash tree store/loads. If the flag is false, the checker then uses the bitmap.

Table 1. ΔR if a range is used for bookkeeping (cf. Section 5.2). In the table, b_t is the number of bits in a time stamp, b_b is the number of bits in a data value/hash block and h is the height of the hash tree (the length of the path from the root to the leaf in the tree).

tree-log-store hash-tree-store	$\Delta R = \omega * (2hb_b - b_b)$
tree-log-load hash-tree-load	$\Delta R = \omega * (h-1)b_b$
tree-log-store log-hash-store	$\Delta R = 2hb_b - (2(b_b + b_t)) + \omega * (2hb_b - b_b)$
tree-log-load log-hash-load	$\Delta R = hb_b - (b_b + 2b_t) + \omega * (h-1)b_b$
tree-log-moveToLogHash	$\Delta R = -(hb_b + (h-1)b_b + b_t)$
tree-log-check	$\Delta R = -n_{lh}((b_b + b_t) + 2(h-1)b_b)$

6. Adaptive Tree-Log Checker

Section 6.1 gives an overview of the interface the FSM calls to use the adaptive tree-log checker to check the integrity of the RAM. Sections 6.2 and 6.3 examine the checker in the case where the FSM does not use a cache. They describe the approach we use to guarantee a worst-case bound on the bandwidth overhead of the checker and the tree-log strategy we adopt. Section 6.4 extends the methodology to caching. Throughout the discussion in this section, we will assume that the checker uses a range for its bookkeeping (cf. Section 5.2). In Appendix D, we extend the discussion to when the checker uses a more general data structure, such as a bitmap, for its bookkeeping.

6.1. Interface Overview

The adaptive tree-log interface consists of just three operations: adaptive-tree-log-store(a, v), adaptive-tree-log-load(a) and adaptive-tree-log-check$()$; these operations call their respective tree-log operations (cf. Section 5). During the FSM's execution, the FSM calls adaptive-tree-log-store and adaptive-tree-log-load to access the untrusted RAM. The adaptive-tree-log-check is called whenever the FSM executes a critical operation (cf. Section 1 and Section 3).

The checker has as a parameter, a worst-case bound. The bound is expressed relative to the bandwidth overhead of the hash tree, if the hash tree had been used to check the integrity of the RAM during the FSM's execution. For instance, if the bound is set at 10%, then, for all FSMs, the tree-log bandwidth overhead is guaranteed to be less than 1.1 times the hash tree bandwidth overhead. (The bandwidth overhead is defined as the additional bandwidth consumed during the program's execution by the integrity checking scheme compared to the bandwidth the program would have consumed without any integrity checking.)

During the FSM's execution, the checker monitors its bandwidth overhead, and it moves addresses to the log-hash scheme based on its bandwidth overhead. Whenever an adaptive-tree-log-check operation occurs, the checker moves all of the addresses in the log-hash scheme back to the tree to optimize the log-hash scheme; the operation does not need any arguments from the FSM because the checker moves all of the addresses in the log-hash scheme to the tree. The adaptive-tree-log-check operation can be performed at anytime; whenever it is performed, the bandwidth overhead of the checker is guaranteed never to be worse than the parameterizable worst-case bound.

6.2. Without Caching: Worst-case Bound

First, we consider the case where the FSM does not use a cache. We make no assumptions about the FSM's access patterns. The naïve approach would be for the checker to just move addresses to the log-hash scheme each time it accesses an address that is in the tree. The naïve approach is a valid approach of using the tree-log scheme. However, the bandwidth overhead of the approach could potentially be more than twice that of the hash tree during short check periods (primarily because of the extra cost of the tree-log-check operation). Thus, to provide the parameterizable worst case bound, the checker needs to regulate the rate at which addresses are added to the log-hash scheme.

Let ω be the parameterizable worst case bound (e.g., if the bound is 10%, $\omega = 0.1$). While the FSM is running, the adaptive tree-log checker maintains two statistics: (1) its current total *reserve*, R, and (2) the number of data value blocks currently in the log-hash scheme, n_{lh}. $R = (1 + \omega)B_{ht} - B_{tl}$ where B_{tl} is the current total tree-log bandwidth overhead and B_{ht} is the current total hash tree bandwidth overhead, if the hash tree had been used to check the RAM. Intuitively, R is how many bits ahead the tree-log checker is of the parameterizable worst-case bound. B_{ht} is easily determined given the height of the tree, the size of a hash and its siblings, and the total number of FSM operations performed thus far. R and n_{lh} are also maintained in the checker's fixed-sized trusted state. n_{lh} is incremented whenever an address is moved from the tree to the log-hash

adaptive-tree-log-store(a, v):

1. If a is in the tree, and $R_{\text{cp}} > C_{\text{tl-mv-to-lh}} + C_{\text{tl-chk}}(n_{lh}+1)$, then tree-log-moveToLogHash(a).
2. tree-log-store(a, v).

adaptive-tree-log-load(a):

1. If a is in the tree, and $R_{\text{cp}} > C_{\text{tl-mv-to-lh}} + C_{\text{tl-chk}}(n_{lh}+1)$, then tree-log-moveToLogHash(a).
2. tree-log-load(a).

adaptive-tree-log-check():

1. Let Z be the set of addresses in the log-hash scheme. tree-log-check(Z).

Figure 7. Adaptive tree-log checker for untrusted RAM, without caching

scheme, and reset to zero on a tree-log-check operation after the operation has moved the addresses back to the tree. R is updated on each checker operation.

We itemize how R changes on each tree-log operation:

- tree-log-store/tree-log-load: R increases with each operation.
- tree-log-moveToLogHash: R decreases with each operation. Let $C_{\text{tl-mv-to-lh}}$ be the b/w consumed by the tree-log-moveToLogHash operation. Then, $\Delta R = -C_{\text{tl-mv-to-lh}}$.
- tree-log-check: R decreases with each operation. Let $C_{\text{tl-chk}}(n_{lh})$ be the bandwidth consumed by the tree-log-check operation; $C_{\text{tl-chk}}(n_{lh})$ increases with n_{lh}. $\Delta R = -C_{\text{tl-chk}}(n_{lh})$.

Table 1 details the amounts by which R changes when a range is used for bookkeeping (cf. Section 5.2). The reserve increases on each tree-log-store/tree-log-load operation. The essential idea of how we bound the worst case tree-log bandwidth overhead is to have the checker build up enough reserve to cover the cost of the tree-log-moveToLogHash operation plus the increased cost of the tree-log-check operation before the checker moves an address from the tree to the log-hash scheme. Whenever the checker wants to move an address to the log-hash scheme, it performs a test to determine if it has enough reserve to do so. For the test, the checker checks that the address is in the tree and that $R > C_{\text{tl-mv-to-lh}} + C_{\text{tl-chk}}(n_{lh}+1)$. If these checks pass, the test returns true; otherwise, the test returns false. If the test returns true, the checker has enough reserve to be able to move the address to the log-hash scheme. Otherwise, the checker cannot move the address to the log-hash scheme. Whenever an address is moved to the log-hash scheme, n_{lh} is incremented.

The mechanism described in this section is a safety mechanism for the adaptive tree-log scheme: whenever an adaptive-tree-log-check operation occurs, the bandwidth overhead of the checker is guaranteed never to be larger than $(1+\omega)B_{ht}$. As can be seen from the expression for R, the larger ω is, the sooner the checker will be able to move addresses to the log-hash scheme. Also, the larger ω is, the larger could be the potential loss in the case that the tree-log scheme has to perform an adaptive-tree-log-check soon after it has started moving addresses to the log-hash scheme; however, in the case that the performance of the tree-log scheme improves when the log-hash scheme is used, which is the case we expect, the larger ω is, the greater the rate at which addresses can be added to the log-hash scheme and the greater can be the performance benefit of using the log-hash scheme. Also from the expression for R, the smaller the tree-log bandwidth overhead compared to the hash tree bandwidth overhead, the better the tree-log scheme performs and the greater the rate at which addresses can be added to the log-hash scheme. This helps the checker adapt to the performance of the scheme, while still guaranteeing the worst-case bound.

6.3. Without Caching: Tree-Log Strategy

Section 6.2 describes the minimal requirements that are needed to guarantee the bound on the worst-case bandwidth overhead of the tree-log checker. The approach described in Section 6.2 can be applied as a greedy algorithm in which addresses are moved to the log-hash scheme whenever R is sufficiently high. However, it is common for programs to have a long check period during which they process data, then have a sequence of short check periods as they perform critical instructions to display or sign the results. If the checker simply moved addresses to the log-hash scheme as long as R was large enough, for short check periods, the

checker might move a lot of data into the log-hash scheme and incur a costly penalty during that check period when the adaptive-tree-log-check operation occurs. We do not want to risk gains from one check period in subsequent check periods. Thus, instead of using R, we use R_{cp}, the reserve that the checker has gained during the current check period, to control the rate at which addresses are added to the log-hash scheme when we adopt the greedy approach. By using R_{cp} instead of R, for short check periods, it is more likely that the checker will just keep addresses in the tree, instead of moving addresses to the log-hash scheme. If we let $R_{\text{cp-start}}$ be the value of R at the beginning of the check period, then $R_{\text{cp}} = R - R_{\text{cp-start}}$. R_{cp} regulates the rate at which addresses are added to the log-hash scheme during the current check period.

Figure 7 shows the interface that the FSM uses to call the adaptive tree-log checker. The strategy we use is a simple strategy and more sophisticated strategies for moving addresses from the tree to the log-hash scheme can be developed in the future. Nevertheless, the principal point is that whatever strategy the checker uses can be *layered* over the safety mechanism in Section 6.2 to ensure that the strategy's bandwidth overhead is never worse than the parameterizable worst-case bound.

At this point, we precisely describe the three features of the adaptive tree-log checker. Firstly, the checker *adaptively* chooses a tree-log strategy for the FSM when the FSM is executed. This allows FSMs to be run unmodified, yet still be able to benefit from the checker's features. Secondly, even though the checker is adaptive, it is able to provide a guarantee on its worst case performance, such that, for all FSMs, the performance of the checker is guaranteed to never be worse than the *parameterizable worst case bound*. This feature allows the adaptive tree-log checker to be turned on by default in systems. The third feature is that, for all FSMs, as the average number of per data FSM operations (total number of FSM data operations/total number of data accessed) during a check period increases, the checker moves from a *logarithmic bandwidth overhead* to a *constant bandwidth overhead*, ignoring the bandwidth consumption of intermediate log-hash integrity checks. This feature allows large classes of FSMs to take advantage of the constant runtime bandwidth overhead of the optimized log-hash scheme to improve their integrity checking performance, because FSMs typically perform many data operations before performing a critical operation.

Let $|\text{TIMER}|$ be the bit-length of the log-hash TIMER counter. In the third feature, we exclude intermediate log-hash integrity checks because they become insignificant for sufficiently large $|\text{TIMER}|$. Whenever the TIMER reaches its maximum value during a check period, an intermediate log-hash check is performed (cf. Section 5). However, by using a large enough $|\text{TIMER}|$, intermediate checks occur so infrequently that the amortized bandwidth cost of the check is very small, and the principal overhead is the *constant runtime bandwidth overhead* of the time stamps.

6.4. With Caching

We now consider the case where the FSM uses a cache. The only assumption that we make about the cache is that it uses a deterministic cache replacement policy, such as the popular LRU (least recently-used) policy. There are two main extensions that are made to the methodology in Sections 6.2 and 6.3. Firstly, to accurately calculate the reserves, the checker will need to be equipped with *cache simulators*. Secondly, with a cache, the hash tree may perform very well. There can exist FSMs for which the reserve can decrease on tree-log-store and tree-log-load operations. To handle this situation, the adaptive checker will need an additional tree-log operation that allows it to *backoff*, and will need to perform an additional test to determine whether it will need to backoff. We describe the extensions.

Cache performance is very difficult to predict. Thus, to help determine B_{tl} and B_{ht}, the checker maintains a hash tree cache simulator and a base cache simulator. The hash tree simulator simulates the hash tree and gives the hash tree bandwidth consumption. The base cache simulator simulates the FSM with no memory integrity checking and gives the base bandwidth consumption, from which the bandwidth overheads can be calculated. The checker also maintains a tree-log simulator that can be used to determine the cost of a particular tree-log operation before the checker actually executes the operation. It is important to note that each simulator only needs the cache status bits (e.g., the dirty bits and the valid bits) and the cache addresses, in particular the cache address tags [6]; the data values are not needed. The tag RAM is a small percentage of the cache [6]. Thus, each simulator is small and of a fixed size (because the cache is of a fixed size) and can, in accordance with our model in Section 3, be maintained in the checker. The simulators do not make any accesses to the untrusted RAM. The simulators are being used to help guarantee the worst-case bound when the FSM uses a cache and, in Appendix E, we discuss how they can be dropped if the strictness of the bound is relaxed.

We expect tree-log-store and tree-log-load operations to generally perform better than the corresponding hash tree operations because the log-hash scheme does not pollute the cache with hashes and because the runtime overhead of the log-hash scheme is constant-sized instead of logarithmic. However, unlike a cacheless hash tree, a hash tree with a cache may perform very well. Furthermore, in the tree-log scheme, because the log-hash scheme does not cache hashes, when the hash tree is used, the tree's cost may be more expensive on average. Also, the tree-log and hash tree

adaptive-tree-log-store(a, v):

1. If a is in the tree, and
   $R'_{\text{cp}} > C_{\text{tl-mv-to-lh}} + C^{\text{marg}}_{\text{tl-chk}}(n_{lh}+1) + C_{\text{buffer}}(n_{lh}+1)$,
   then tree-log-moveToLogHash(a).
2. If the tree-log and hash tree caches are not synchronized, if $R + \Delta R_{\text{tl-op}} < C_{\text{bkoff}}(n_{lh})$,
   then tree-log-bkoff.
3. tree-log-store(a, v).

Figure 8. adaptive-tree-log-store**, with caching**

cache access patterns are different, and the tree-log cache performance could be worse than the hash tree cache performance. Reserve can sometimes decrease on tree-log-store and tree-log-load operations. Thus, because the FSM uses a cache, the adaptive checker needs to have an additional *backoff procedure* that reverts it to the vanilla hash tree if the reserve gets dangerously low.

The backoff procedure consists of performing a tree-log-check operation and synchronizing the FSM's cache by putting the cache into the exact state in which it would have been in the hash-tree scheme. This is done by writing back dirty tree nodes that are in the cache and updating them in the tree in RAM, then checking and bringing into the cache, blocks from RAM that are in the hash tree cache simulator that are not in the FSM's cache[3]. We refer to the backoff procedure as tree-log-bkoff. Let C_{sync} be the cost of synchronizing the cache (it is independent of n_{lh}). Then the bandwidth consumed by tree-log-bkoff is $C_{\text{bkoff}}(n_{lh}) = C_{\text{tl-chk}}(n_{lh}) + C_{\text{sync}}$. Whenever the checker backs off, it continues execution just using the tree alone, until it has enough reserve to try moving addresses to the log-hash scheme again.

Again, we indicate how R changes with each tree-log operation:

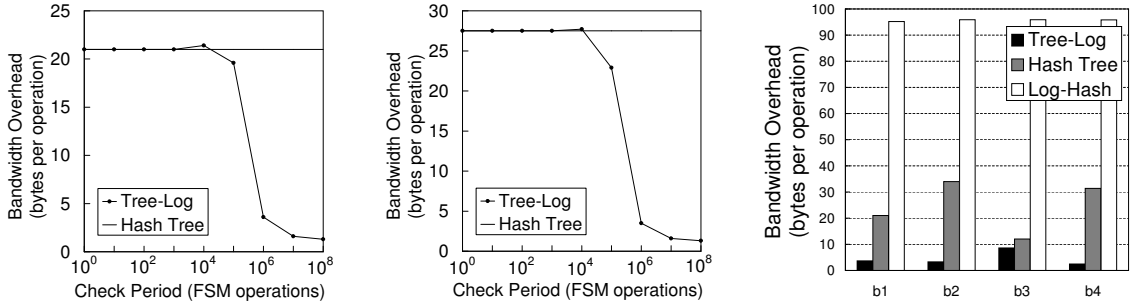
- tree-log-store/tree-log-load: With each operation, R usually increases; however it can decrease. Let $\Delta R_{\text{tl-op}}$ be the change in R that occurs when the store/load operation is performed; $\Delta R_{\text{tl-op}}$ can be positive or negative (and is different for each store/load operation). $\Delta R = \Delta R_{\text{tl-op}}$.
- tree-log-moveToLogHash: R decreases with each operation. $\Delta R = -C_{\text{tl-mv-to-lh}}$.
- tree-log-check: R decreases with each operation. $\Delta R \geq -C_{\text{tl-chk}}(n_{lh})$.
- tree-log-bkoff: R decreases with each operation. $\Delta R \geq -C_{\text{bkoff}}(n_{lh})$.

Figure 8 shows the adaptive-tree-log-store operation when the FSM uses a cache. The adaptive-tree-log-load operation is similarly modified. adaptive-tree-log-check is similar to the operation in Figure 7. The actual costs of $\Delta R_{\text{tl-op}}$ and $C_{\text{tl-mv-to-lh}}$ are obtained at runtime from the simulators. The worst-case costs of $C_{\text{tl-chk}}(n_{lh})$ and $C_{\text{bkoff}}(n_{lh})$ can be calculated; on the tree-log-check and tree-log-bkoff operations, R decreases by an amount that is guaranteed to be smaller than these worst-case costs. We show how to calculate these worst-case costs in Appendix C. The worst-case cost of the tree-log-check operation can be expressed as $C_{\text{tl-chk}}(n_{lh}) = C_{\text{tl-chk}}(0) + C^{\text{marg}}_{\text{tl-chk}}(n_{lh})$, where $C_{\text{tl-chk}}(0)$ is a fixed cost, a cost that independent of n_{lh}, and $C^{\text{marg}}_{\text{tl-chk}}(n_{lh})$ is a marginal cost, a cost that is dependent on n_{lh}. Recall that $C_{\text{bkoff}}(n_{lh}) = C_{\text{tl-chk}}(n_{lh}) + C_{\text{sync}}$.

In Figure 8, the first test in step 1 is similar to the test in Section 6.3. However, in this case, the reserve that the checker uses to regulate the rate at which addresses are added to the log-hash scheme is R'_{cp}, where R'_{cp} is the reserve that the checker has gained after $R > C_{\text{bkoff}}(0)$ during the current check period (or, if the checker has backed off, the reserve that the checker has gained after $R > C_{\text{bkoff}}(0)$ since backing off). Thus, $R'_{\text{cp}} = R - \max(C_{\text{bkoff}}(0), R_{\text{cp-start}})$ (where $R_{\text{cp-start}}$ is the value of R at the beginning of the check period, or, if the checker has backed off, the value of R after the checker has completed backing off). R'_{cp} only begins recording reserve after R has become greater than $C_{\text{bkoff}}(0)$ because otherwise, the checker would not have enough reserve to be able to backoff if it needed to. The test also gives a small reserve buffer per address in the log-scheme, $C_{\text{buffer}}(n_{lh})$, for the tree-log scheme to start outperforming the hash tree.

The test in step 2 determines whether the checker needs to backoff. The tree-log and hash tree caches are unsynchronized if the log-hash scheme has been used (since the begin-

[3] In the synchronized cache, the hashes of cached nodes may not be the same as they would have been if the hash tree had been used. However, the values of these hashes are not important (cf. the invariant in Section 4.1).



(a) b1, cache size = 16 blocks (b) b1, cache size = 12 blocks (c)

Figure 9. Experimental results

ning of the FSM's execution or since the checker last backed off). It is only necessary to perform the test if the tree-log and hash tree caches are unsynchronized. From the expression for R'_{cp}, $R > R'_{\text{cp}} + C_{\text{bkoff}}(0)$. From the first test, to successfully move an address to the log-hash scheme, $R'_{\text{cp}} > (C_{\text{tl-mv-to-lh}} + C^{\text{marg}}_{\text{tl-chk}}(n_{lh}+1) + C_{\text{buffer}}(n_{lh}+1))$. Thus, $R > (C_{\text{tl-mv-to-lh}} + C^{\text{marg}}_{\text{tl-chk}}(n_{lh}+1) + C_{\text{buffer}}(n_{lh}+1) + C_{\text{bkoff}}(0))$. From the expression for $C_{\text{bkoff}}(n_{lh})$, $R > (C_{\text{tl-mv-to-lh}} + C^{\text{marg}}_{\text{tl-chk}}(n_{lh}+1) + C_{\text{buffer}}(n_{lh}+1) + C_{\text{tl-chk}}(0) + C_{\text{sync}}) > (C_{\text{tl-mv-to-lh}} + C_{\text{bkoff}}(n_{lh}+1) + C_{\text{buffer}}(n_{lh}+1))$. Thus, $R > (C_{\text{tl-mv-to-lh}} + C_{\text{bkoff}}(n_{lh}+1) + C_{\text{buffer}}(n_{lh}+1))$. This means that, whenever an address is successfully moved to the log-hash scheme, $R > (C_{\text{bkoff}}(n_{lh}) + C_{\text{buffer}}(n_{lh}))$, (recall that n_{lh} is incremented when an address is successfully moved to the log-hash scheme). If the log-hash scheme has been used, the second test uses $\Delta R_{\text{tl-op}}$, obtained from the simulators, to determine if performing the store operation would result in its reserve dropping below $C_{\text{bkoff}}(n_{lh})$. If it does, the checker backs off, then performs the operation. Otherwise, it just performs the operation in its current state.

With regards to the theoretical claims on the tree-log algorithm in Section 6.3, the first two features on being adaptive and providing a parameterizable worst case bound remain the same. (With the second feature, it is implicit that if, for a particular FSM, the hash tree performs well, then the tree-log scheme will also perform well, because the tree-log bandwidth overhead will be, at most, the parameterizable worst case bound more than the hash tree bandwidth overhead.) With regards to the third feature, with a cache, the adaptive tree-log checker will not improve over the hash tree for some FSMs. FSMs whose runtime performance improves when the log-hash scheme is used experience the asymptotic behavior. From the expression for calculating reserve (and from our experiences with experiments), we see that *the general trend is that the greater the hash tree bandwidth overhead, the less likely it is for the checker to backoff and the greater the tree-log scheme's improvement will be when it improves the checker's performance*. Thus, if the hash tree is expensive for a particular FSM, the adaptive tree-log scheme will, when it has built up sufficient reserve, automatically start using the log-hash scheme to try to reduce the integrity checking bandwidth overhead.

7. Experiments

We present some experimental evidence to support the theoretical claims on the adaptive tree-log algorithm. In the experiments, a 4-ary tree of height 10 was used; the data value/hash block size was 64 bytes and the time stamp size was 32 bits. ω was set at 10%. The benchmarks are synthetic and give the access patterns of stores and loads. The size of the working set, the amount of data accessed by the benchmarks, is about 2^{14} bytes. ($C_{\text{buffer}}(n_{lh})$ was about $4hb_bn_{lh} = (4 * 10 * 64 * n_{lh})$ bytes.) Figure 9(a) shows the bandwidth overhead for different check periods for a particular benchmark, b1. The cache size was 16 blocks. The tree-log scheme has exactly the same bandwidth overhead as the hash tree for check periods of 10^3 FSM store/load operations and less. Around check periods of 10^4 operations, there is a slight degradation (tree-log: 21.4 bytes per operation, hash tree: 21.0 bytes per operation), though not worse than the 10% bound. Thereafter, the bandwidth overhead of the tree-log scheme becomes significantly smaller. By check periods of 10^7 operations, the tree-log scheme consumes 1.6 bytes per operation, a 92.4% reduction in the bandwidth overhead compared to that of the hash tree. (We do not show the results for the log-hash scheme for this experiment because its bandwidth overhead is prohibitively large when check periods are small.) In Figure 9(b), the

cache size is reduced to 12 blocks, making the hash tree more expensive. The figure shows a greater tree-log scheme improvement over the hash tree bandwidth overhead when the tree-log scheme improves on the hash tree. Figure 9(c) shows the results for different benchmarks of an access pattern that checked after a check period of 10^6 operations, then after each of five check periods of 10^3 operations. The experiment demonstrates a simple access pattern for which the tree-log scheme outperforms both the hash tree and log-hash schemes.

8. Conclusion

We have introduced an adaptive tree-log scheme as a general-purpose integrity checker. We have provided a theoretical foundation for the checker and the methodology that can be used to provide the guarantees and the asymptotic behavior of the checker's performance. The adaptive tree-log algorithm can be implemented anywhere hash trees are currently being used to check the integrity of untrusted data. The application can experience a significant benefit if programs can perform sequences of data operations before performing a critical operation.

References

[1] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the Correctness of Memories. In *Algorithmica*, volume 12, pages 225–244, 1994.

[2] 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 *Advances in Cryptology - Asiacrypt 2003 Proceedings*, volume 2894 of *LNCS*, pages 188–207. Springer-Verlag, November 2003.

[3] 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*, pages 295–306, February 2003.

[4] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[5] E. Hall and C. S. Jutla. Parallelizable Authentication Trees. In *Cryptology ePrint Archive*, December 2002.

[6] J. L. Hennessy and D. A. Patterson. *Computer Organization and Design*. Morgan Kaufmann Publishers, Inc., 1997.

[7] D. Lie. *Architectural Support for Copy and Tamper-Resistant Software*. PhD thesis, Stanford University, December 2003.

[8] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, June 1979.

[9] 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*, pages 160–171, June 2003.

[10] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings of the* 36^{th} *Int'l Symposium on Microarchitecture*, pages 339–350, December 2003.

[11] J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *Proceedings of the* 36^{th} *Int'l Symposium on Microarchitecture*, pages 351–360, December 2003.

A. Detailed Comparison of Hash Tree and Log-Hash Schemes

In this appendix, we present a detailed comparison of the hash tree and log-hash schemes when caches are used (Section 4.1 and Section 4.2). For the log-hash scheme, let b_t be the number of bits in a time stamp, b_b be the number of bits in a cache block, and n_{lh} be the number of data value blocks protected by the log-hash scheme, i.e. the number of blocks to which a time stamp is appended when the block is in the untrusted RAM. Let t be the number of FSM stores and loads that the FSM performs before the checker performs a log-hash-check operation. C is the number of blocks that can be stored in the cache, and h_v is the data value cache hit rate - the fraction of FSM load and store operations that find their data in the cache. The fraction of checker accesses to the untrusted RAM is $(1 - h_v)$. On each of these accesses, the checker reads a time stamp and writes a time stamp as it brings in a block and evicts a block from the cache. Thus, the cost of writing and reading time stamps is $t(1-h_v)(2b_t)$. This is the bound on the runtime bandwidth overhead of the log-hash scheme. During a log-hash-check operation, the blocks that are not in the cache are read, with their associated time stamps; if the log-hash-check operation is an intermediate check, time stamps are also written back to memory to reset them. Thus, the bandwidth consumption of the log-hash-check operation is $(n_{lh} - C)(b_b + 2b_t)$. Therefore, the total bandwidth overhead of the log-hash scheme is $t(1 - h_v)(2b_t) + (n_{lh} - C)(b_b + 2b_t)$.

For the hash tree scheme, we additionally let δh_v, where $0 \leq \delta \leq 1$, be the cache hit rate of data values in the hash tree scheme (this is less than the data value cache hit rate of the log hash scheme because both hashes and data values must be stored in the cache for the hash tree to perform well). z is the average number of data value and hash blocks that the checker fetches and writes back to the untrusted RAM on each FSM store and load operation that misses in the cache. The common case is for evicted blocks to be clean, in which case the total bandwidth consumed by the FSM without integrity checking is $t(1 - h_v)b_b$. Thus, the total bandwidth overhead of the hash tree scheme is $t(1 - \delta h_v)zb_b - t(1 - h_v)b_b = t((1 - \delta h_v)zb_b - (1 - h_v)b_b)$.

We see that, if $t > \frac{(n_{lh}-C)(b_b+2b_t)}{((1-\delta h_v)zb_b-(1-h_v)b_b) - (1-h_v)(2b_t)}$,

the overhead of fetching and caching hashes in the hash tree scheme will exceed the cost of reading the addresses the FSM used to perform a log-hash-check operation in the log-hash scheme, and the log-hash scheme will consume less bandwidth and perform better than the hash tree scheme. Importantly, we can observe that, the smaller n_{lh} is, the smaller t will be before the log-hash scheme will perform better than the hash tree scheme. The tree-log scheme in Section 5 uses this observation to optimize the log-hash scheme.

When the number of loads and stores performed by the FSM is large, the amortized bandwidth consumption of reading the untrusted RAM to perform the log-hash-check is very small and the principal overhead is the constant-sized runtime overhead of reading and writing time stamps, which are also very small (32 bits). The log-hash scheme thus performs very well and its bandwidth overhead is very small. Computer simulations [10] have shown, that, when log-hash-checks are infrequent, the log-hash scheme outperforms the hash tree scheme.

When log-hash-checks are more frequent, the FSM performs a small number of memory operations and uses a small subset of the addresses that are protected by the log-hash scheme between the checks. In this case, the amortized bandwidth consumption of reading the entire set of addresses protected by the log-hash scheme, which is the entire set of addresses that the FSM has used since the beginning of its execution, is more costly, and the hash tree scheme performs better than the log-hash scheme. (When check periods are small, the log-hash scheme is not optimal because the log-hash-check operation has to read all of the addresses that are protected by the log-hash scheme to perform the check, instead of just the addresses that are used by the FSM during that check period.)

As a note, the log-hash-check operation can be performed at any time, even after each FSM store/load operation. The log-hash scheme does not require that sequences of operations be performed before log-hash-check operations can be performed. In the model in Section 3, log-hash-checks are performed when critical operations are performed. The log-hash scheme performs very well when critical operations are infrequent, but poorly when critical operations are frequent.

B. Proof of Tree-Log Checker

In this appendix, we prove the security of the tree-log checker in Section 5. We refer to a multiset as a finite unordered group of elements where an element can occur as a member more than once. Recall from Section 3 that we say that RAM behaves like valid RAM if the data value that the checker reads from a particular address is the same data value that the checker most recently wrote to that address.

The simplified definition of valid RAM in Section 3 does not specify what happens in the log-hash scheme (cf. Section 4.2) if a store or load operation is done on an address that has not been added to the RAM. For clarity, if log-hash-store or log-hash-load is called on an address before log-hash-add is called to add the address to the log-hash scheme, then the RAM has not behaved like valid RAM.

We first prove the security of the log-hash scheme in Section 4.2.

Lemma B.1 *Denote the addresses on which log-hash-add has been called on as the multiset, $M_{\text{lh-add}}$. If $M_{\text{lh-add}}$ is a set (log-hash-add has been called exactly once on each address), the log-hash-check operation returns true if and only if the untrusted RAM has behaved like valid RAM and the log-hash-check operation has read exactly the set of addresses in $M_{\text{lh-add}}$.*

Proof Let W be the multiset of triples written to memory and let R be the multiset of triples read from memory. That is, W hashes to WRITEHASH and R hashes to READHASH.

If the untrusted RAM has behaved like valid RAM and the log-hash-check operation has read exactly the set of addresses in $M_{\text{lh-add}}$, it is easy to verify that the log-hash-check operation returns true. Suppose the RAM does not behave like valid RAM (i.e., the data value that the checker reads from an address is not the same data value that the checker had most recently written to that address). We will prove that $W \neq R$.

Consider the put and take operations that occur on an address as occurring on a timeline. To avoid confusion with the values of TIMER, we express this timeline in terms of processor cycles. Let x_1 be the cycle of the first incorrect take operation. Suppose the checker reads the pair (v_1, t_1) from address a at x_1. If there does not exist a cycle at which the checker writes the pair (v_1, t_1) to address a, then $W \neq R$ and we are done.

Suppose there is a cycle x_2 when the checker first writes (v_1, t_1) to address a. Because of line 3 in the take operation, the values of time stamps of all of the writes to a after x_1 are strictly greater than t_1. Because the time stamps at x_1 and x_2 are the same and since put operations and take operations do not occur on the same cycle, x_2 occurs before x_1 ($x_2 < x_1$). Let x_3 be the cycle of the first read from a after x_2. Notice that x_1 is a read after x_2, so $x_1 \geq x_3$. If x_1 were equal to x_3, then the data value most recently written to a, i.e. v_1, would be read at x_1. This contradicts the assumption that x_1 is an incorrect read. Therefore, $x_1 > x_3$.

Because the read at cycle x_1 is the first incorrect read, the read at cycle x_3 is a correct read. So the read at x_3 reads the same pair that was written at x_2. Again, because of line 3 in the take operation, the values of time stamps of all the writes to a after x_3 are strictly greater than t_1. Therefore, (v_1, t_1) cannot be written after x_3. Because x_2 is the first

Table 2. Worst-case costs of tree-log-check **and** tree-log-bkoff**, with caching, when a range is used for bookkeeping (cf. Section 5.2). In the table, b_t is the number of bits in a time stamp, b_b is the number of bits in a data value/hash block, h is the height of the hash tree (the length of the path from the root to the leaf in the tree) and C is the number of blocks that can be stored in the cache.**

$C_{\text{tl-chk}}(n_{lh})$	$2Chb_b + n_{lh}((b_b + b_t) \ + \ 2(h-1)b_b)$
$C_{\text{bkoff}}(n_{lh})$	$C_{\text{tl-chk}}(n_{lh}) + 2Chb_b + Chb_b$

cycle on which (v_1, t_1) is written to a, (v_1, t_1) cannot be written before x_2. Because $M_{\text{lh-add}}$ is a set, two writes to an address always have a read from that address between them. Because x_3 is the first read from a after x_2, and two writes to an address always have a read from that address between them, (v_1, t_1) cannot be written between x_2 and x_3. Therefore, the pair (v_1, t_1) is written only once, but it is read at x_1 and x_3. Therefore, $W \neq R$.

Suppose the log-hash-check operation has not read exactly the set of the addresses in $M_{\text{lh-add}}$. Then, there is a triple in W that is not in R, or a triple in R that is not in W. Therefore, $W \neq R$.

$W \neq R$ implies that WRITEHASH is not equal to READHASH i.e. the log-hash-check operation fails, or that a collision has been found in the multiset hash function. ■

We now prove the security of the tree-log scheme in Section 5.

Theorem B.2 *The untrusted RAM has behaved like valid RAM if and only if the tree-log integrity checks (using the hash tree and the* tree-log-check *operation) return true.*

Proof The validity condition, that if the RAM has behaved like RAM, then the tree-log integrity checks return true, is easy to verify. We present an argument for the safety condition: if the tree-log integrity checks return true, then the RAM has behaved like valid RAM.

We assume that the bookkeeping data structures (cf. Section 5.2) are not protected. The adversary can tamper with the data structures, data values and time stamps at will. We will assume that all of the hash tree integrity checks and tree-log-check integrity checks return true. We will prove that an adversary is unable to affect the validity of the RAM.

First we show that $M_{\text{lh-add}}$, the multiset of addresses on which log-hash-add has been called, is a set. Suppose tree-log-moveToLogHash is called on an address that has already been added to the log-hash scheme. When the checker first called tree-log-moveToLogHash on the address in the tree to add it to the log-hash scheme, hash-tree-updateParent(a, NULL) was called to update, in the tree, the parent node of the address with a value that the address can never have. If the checker subsequently calls the tree-log-moveToLogHash operation on the address again during the same check period, the operation first checks the integrity of the old value of the node and its siblings in the hash tree. The hash tree integrity check will not pass. Thus, we infer that if all of the integrity checks pass, then $M_{\text{lh-add}}$ is a set and the results of Lemma B.1 apply.

We now show that the adversary cannot tamper with the bookkeeping data structures without the checker detecting the tampering. If the adversary did tamper with the bookkeeping data structures, then either the tree-log-check operation would not read exactly the set of address in $M_{\text{lh-add}}$, or a hash tree store or load operation would be performed on an address that is in the log-hash scheme, or a log-hash store or load operation would be performed on an address that is in the hash tree. Suppose that the tree-log-check operation does not read exactly the set of addresses in $M_{\text{lh-add}}$. This means that the log-hash-check operation does not read exactly the set of addresses in $M_{\text{lh-add}}$. By Lemma B.1, the tree-log-check operation will not pass. Suppose that a hash tree store or load operation is performed on an address that is in the log-hash scheme. Because the NULL value was recorded in the address's parent in the tree when the address was first moved to the log-hash scheme and because hash-tree-store and hash-tree-load each check the integrity of the data value read from the RAM (recall that hash-tree-store checks the integrity of the old value of node and its siblings before updating the node), the hash tree integrity check will not pass. Suppose that a log-hash store or load operation is performed on an address that is in the hash tree. log-hash-store or log-hash-load is then called on the address before log-hash-add is called to add the address to the log-hash scheme. By Lemma B.1, the tree-log-check operation will not pass. Thus, if the adversary tampers with the bookkeeping data structures, the checker will detect the tampering.

Finally, we show that the adversary cannot tamper with the data values (or time stamps) without the checker detecting the tampering. Suppose the adversary tampers with the data value of an address that is protected by the tree. tree-log-moveToLogHash, tree-log-store and tree-log-load each check the integrity of the data value read from the untrusted RAM. If the data value is tampered with, the hash

Table 3. ΔR if a bitmap is used for bookkeeping (cf. Section 5.2). In the table, b_t is the number of bits in a time stamp, b_b is the number of bits in a data value/hash block, h is the height of the hash tree (the length of the path from the root to the leaf in the tree), $C_{\text{read-bmap}}$ is the number of bits in a bitmap segment and N_{bmap} is the number of bits in the bitmap. If `FLAG` is true, all of the data is in the tree; if `FLAG` is false, there is data in the log-hash scheme.

tree-log-store hash-tree-store when `FLAG` = true	$\Delta R = \omega * (2hb_b - b_b)$
tree-log-load hash-tree-load when `FLAG` = true	$\Delta R = \omega * (h-1)b_b$
tree-log-store hash-tree-store when `FLAG` = false	$\Delta R = \omega * (2hb_b - b_b) - C_{\text{read-bmap}}$
tree-log-load hash-tree-load when `FLAG` = false	$\Delta R = \omega * (h-1)b_b - C_{\text{read-bmap}}$
tree-log-store log-hash-store	$\Delta R = 2hb_b - (2(b_b + b_t)) + \omega * (2hb_b - b_b) - C_{\text{read-bmap}}$
tree-log-load log-hash-load	$\Delta R = hb_b - (b_b + 2b_t) + \omega * (h-1)b_b - C_{\text{read-bmap}}$
tree-log-moveToLogHash	$\Delta R = -(hb_b + (h-1)b_b + b_t) - 2C_{\text{read-bmap}}$
tree-log-check	$\Delta R = -n_{lh}((b_b + b_t) + 2(h-1)b_b) - 2N_{\text{bmap}}$

tree integrity check will not pass. Suppose the adversary tampers with the data value (or time stamp) of an address that is protected by the log-hash scheme. By Lemma B.1, the tree-log-check operation will not pass. Thus, if the adversary tampers with the data values (or time stamps), the checker will detect the tampering.

Thus, if all of the hash tree integrity checks and tree-log-check integrity checks return true, then the RAM has behaved like valid RAM. This concludes the proof of Theorem B.2.■

The proof demonstrates that, with regards to security, the bookkeeping data structures do not have to be protected.

C. Worst-case Costs of tree-log-check and tree-log-bkoff, with caching

In this appendix, we give the worst-case costs of $C_{\text{tl-chk}}(n_{lh})$ and $C_{\text{bkoff}}(n_{lh})$ for the adaptive checker in the case that the FSM uses a cache (cf. Section 6.4). For this analysis, we assume a range is used for bookkeeping (cf. Section 5.2). Table 2 summarizes the costs.

The worst-case bandwidth consumption of the tree-log-check operation is $2Chb_b + n_{lh}((b_b + b_t) + 2(h-1)b_b)$, where $2Chb_b$ is the cost of evicting dirty tree nodes that are in the cache and updating them in the tree in RAM, and $n_{lh}((b_b + b_t) + 2(h-1)b_b)$ is the cost of reading the addresses in the log-hash scheme and moving them to the tree.

The worst-case bandwidth consumption of the tree-log-bkoff operation is $C_{\text{tl-chk}}(n_{lh}) + 2Chb_b + Chb_b$, where $C_{\text{tl-chk}}(n_{lh})$ is the worst-case cost of the tree-log-check operation and $2Chb_b + Chb_b = 3Chb_b$ is the cost of synchronizing the cache (C_{sync}). $C_{\text{bkoff}}(n_{lh})$ covers the cost of the backing off in both the case where the cache is unsynchronized and all of the addresses are in the tree, and in the case the cache is unsynchronized and some of the addresses are in the log-hash scheme.

These bounds on the worst-case costs are actually the costs of the operations if the cache is not used for the operations. The checker could simulate the operation using the tree-log simulator to determine the actual costs when caching is used when the operation is called. If this cost is less than the bound, caching is used for the operation; otherwise caching is not used for the operation.

With reference to Section 6.4, $C_{\text{tl-chk}}(0) = 2Chb_b$ and $C^{\text{marg}}_{\text{tl-chk}}(n_{lh}) = n_{lh}((b_b + b_t) + 2(h-1)b_b)$. $C_{\text{bkoff}}(0) = 5Chb_b$.

D. Adaptive Tree-Log Checker with a Bitmap

In this section, we describe the modifications that are made to the adaptive checker in Section 6 when the checker uses a bitmap for bookkeeping (cf. Section 5.2). Appendix D.1 describes the modifications when the FSM does

adaptive-tree-log-store(a, v):

1. If a is in the tree, and $R_{\text{cp}} > C_{\text{tl-mv-to-lh}} + C^{\text{marg}}_{\text{tl-chk}}(n_{lh}+1)$, then tree-log-moveToLogHash(a).
2. tree-log-store(a, v).

adaptive-tree-log-load(a):

1. If a is in the tree, and $R_{\text{cp}} > C_{\text{tl-mv-to-lh}} + C^{\text{marg}}_{\text{tl-chk}}(n_{lh}+1)$, then tree-log-moveToLogHash(a).
2. tree-log-load(a).

adaptive-tree-log-check():

1. Let Z be the set of addresses in the log-hash scheme. tree-log-check(Z).

Figure 10. Adaptive tree-log checker, without caching, when a bitmap is used for bookkeeping

not use a cache. Appendix D.2 describes the modifications when the FSM does use a cache.

D.1. Without Caching

We first examine the case when the checker uses a bitmap for its bookkeeping and the FSM does not use a cache. We assume that the bitmap is read from RAM with a granularity the size of the smallest granularity that can be read from RAM (64 bits, for example). We denote this granularity as a segment. We denote the cost of reading a bitmap segment from the untrusted RAM as $C_{\text{read-bmap}}$ (if the size of a bitmap segment is 64 bits, then $C_{\text{read-bmap}} = 64$ bits).

Because the bitmap is stored in the untrusted RAM, there are two main changes that are made to the descriptions in Section 6.2 and Section 6.3. Firstly, the amounts by which R changes on the various tree-log operations are slightly different. Secondly, because $C_{\text{tl-chk}}(0)$, the fixed-cost of the tree-log-check operation, is no longer zero when a bitmap is used for bookkeeping, the test that the checker performs to determine whether to move an address from the hash tree to the log-hash scheme needs to be slightly modified.

Table 3 details the new amounts by which R changes on the various tree-log operations. For simplicity, we assume the bitmap is on a per data value block granularity; it is simple to extend the analysis to bitmaps with larger granularities. If `FLAG` is true, the checker knows that all of the data is in the hash tree and it does not use the bitmap; its stores/loads perform exactly as hash tree store/loads. If `FLAG` is false, there is data in the log-hash scheme and the checker then uses the bitmap. (A simple optimization is for the checker to maintain the highest and lowest addresses that have been moved to the log-hash scheme during a check period in its fixed-sized trusted state. On tree-log-store and tree-log-load operations, if `FLAG` is false and if the address falls within the range maintained in the checker, then the appropriate bitmap segment is read from the untrusted RAM. The tree-log-check operation only needs to read the bitmap bits corresponding to the addresses within the range maintained in the checker.)

With the bitmap, when `FLAG` is false, each hash-tree-store and hash-tree-load costs an extra $C_{\text{read-bmap}}$ bits. When $h \geq 0$, $(2hb_b - b_b) \geq (h-1)b_b$. Thus, assuming $\omega * (h-1)b_b > C_{\text{read-bmap}}$, the reserve, R, still increases on each tree-log-store/tree-log-load operation. As an example, for $\omega = 0.1$, a block size of 64 bytes ($b_b = 2^9$) and a tree of $h = 12$, $\lfloor \omega * (h-1)b_b \rfloor = 563$ bits $\gg 64$ bits and the reserve increases on each tree-log-store/tree-log-load operation.

The worst-case cost of the tree-log-check operation can be expressed as $C_{\text{tl-chk}}(n_{lh}) = C_{\text{tl-chk}}(0) + C^{\text{marg}}_{\text{tl-chk}}(n_{lh})$, where $C_{\text{tl-chk}}(0)$ is a fixed cost, a cost that independent of n_{lh}, and $C^{\text{marg}}_{\text{tl-chk}}(n_{lh})$ is a marginal cost, a cost that is dependent on n_{lh}. From Table 3, $C_{\text{tl-chk}}(0) = 2N_{\text{bmap}}$ and $C^{\text{marg}}_{\text{tl-chk}}(n_{lh}) = n_{lh}((b_b + b_t) + 2(h-1)b_b)$.

Figure 10 shows the interface that the FSM uses to call the adaptive tree-trace checker. Compared with Figure 7 (c.f. Section 6.3), R_{cp} is the reserve that the checker has gained after $R > C_{\text{tl-chk}}(0)$ during the current check period. Thus, $R_{\text{cp}} = R - \max(C_{\text{tl-chk}}(0), R_{\text{cp-start}})$ (where $R_{\text{cp-start}}$ is the value of R at the beginning of the check period).

Table 4. Worst-case costs of tree-log-check **and** tree-log-bkoff**, with caching, when a bitmap is used for bookkeeping. In the table, b_t is the number of bits in a time stamp, b_b is the number of bits in a data value/hash block, h is the height of the hash tree (the length of the path from the root to the leaf in the tree), C is the number of blocks that can be stored in the cache and N_{bmap} is the number of bits in the bitmap.**

$C_{\text{tl-chk}}(n_{lh})$	$2Chb_b + n_{lh}((b_b + b_t) + 2(h-1)b_b) + 3N_{\text{bmap}}$
$C_{\text{bkoff}}(n_{lh})$	$C_{\text{tl-chk}}(n_{lh}) + 2Chb_b + Chb_b$

D.2. With Caching

We now examine the case when the checker uses a bitmap for its bookkeeping and the FSM does use a cache. Because the bitmap is stored in RAM, there are two main changes that are made to the descriptions in Section 6.4 and Appendix C. Firstly, an extra term is added to $C_{\text{tl-chk}}(n_{lh})$ to account for using the bitmap. Secondly, bitmap segments may need to be read from RAM in the adaptive-tree-log-store and adaptive-tree-log-load operations. If a bitmap segment needs to be read, the checker needs to perform an additional test to ensure that it has enough reserve to be able to do so. Thus, the adaptive checker's interface in Figure 8 also needs to be slightly modified.

Table 4 gives the worst-case costs of $C_{\text{tl-chk}}(n_{lh})$ and $C_{\text{bkoff}}(n_{lh})$. The tree-log simulator needs to read the bitmap to simulate the check, which costs N_{bmap}. The actual check operation needs to read and update the bitmap, which costs $2N_{\text{bmap}}$. Thus, compared to Table 2, the cost of $C_{\text{tl-chk}}(n_{lh})$ increases by $3N_{\text{bmap}}$. As before, $C_{\text{bkoff}}(n_{lh})$ costs an extra $3Chb_b$ more than the cost of $C_{\text{tl-chk}}(n_{lh})$. $C_{\text{tl-chk}}(0) = 2Chb_b + 3N_{\text{bmap}}$ and $C^{\text{marg}}_{\text{tl-chk}}(n_{lh}) = n_{lh}((b_b + b_t) + 2(h-1)b_b)$. $C_{\text{bkoff}}(0) = 5Chb_b + 3N_{\text{bmap}}$.

If the checker's `FLAG` is true, the bitmap is not used. If it is false, the bitmap is used. Optionally, some of the bitmap could be cached in a small part of the FSM's cache, though, for this discussion, we assume that the bitmap is not cached.

For data value/hash block cache evictions, we have an extra status bit per cache line, indicating whether the data value/hash block is in the tree or in the log-hash scheme. On adaptive-tree-log-store/adaptive-tree-log-load, if the bitmap is being used and the data value block is not already cached (i.e., the checker will need to read the data value block from RAM), the checker will then have to read a bitmap segment from RAM to determine which kind of store/load operation to perform in tree-log-store/tree-log-load. (Let b_{ht} be the average bandwidth overhead on a hash tree operation. When the log-hash scheme is used, the data value cache hit rate tends to improve. Similarly to the case in Appendix D.1, if $\omega * b_{ht} > C_{\text{read-bmap}}$, then, in the calculation of the reserve, R, the extra cost of using the bitmap is covered by the overhead of the hash tree.)

Figure 11 shows the modified adaptive-tree-log-store. The difference is in Step 1, where there is an extra test to determine if the checker has enough reserve to read the bitmap segment if it needs to read the segment. adaptive-tree-log-load is similarly modified. adaptive-tree-log-check is similar to the check when a range is used for the bookkeeping (cf. Section 6.4 and Appendix C).

On adaptive-tree-log-store/adaptive-tree-log-load operations, the tree-log simulator is used to determine the costs of tree-log-moveToLogHash and tree-log-store/tree-log-load. These simulations occur after the checker has determined which scheme the address is in in step 1. The checker can pass this information on to these simulations so that these simulations do not have to make any accesses to the untrusted RAM. Thus, with a bitmap, as is the case when a range is used, the simulations in the adaptive-tree-log-store/adaptive-tree-log-load operations do not make any accesses to the RAM. (In the simulations, if the tree-log-moveToLogHash operation in adaptive-tree-log-store/adaptive-tree-log-load needs to write a bitmap segment from RAM in order to update it, it does not actually update the segment, but simply accounts for the cost of writing the segment.) With regard to the adaptive-tree-log-check operation, with a range, the simulation in the operation does not make any accesses to the RAM; with a bitmap, the simulation reads the bitmap from RAM, but the cost of this simulation is budgeted for in the check, similar to how we budget for the cost of the check itself.

The methodology described in this section shows how to guarantee the parameterizable worst-case bound for the adaptive tree-log checker, with caching, when a bitmap is used for bookkeeping. When check periods are large, the amortized cost of the tree-log-check operation, including reading and updating the bitmap, is small, and the principal overhead is the constant runtime bandwidth overhead of the time stamps and the bitmap segments.

adaptive-tree-log-store(a, v)**:**

1. Determine whether a is in the hash tree or log-hash scheme. If the bitmap segment for a needs to be read from RAM and $R > C_{\text{bkoff}}(n_{lh}) + C_{\text{read-bmap}}$, then read the bitmap segment; otherwise call tree-log-bkoff.
2. If a is in the tree, and
   $R'_{\text{cp}} > C_{\text{tl-mv-to-lh}} + C^{\text{marg}}_{\text{tl-chk}}(n_{lh}+1) + C_{\text{buffer}}(n_{lh}+1)$,
   then tree-log-moveToLogHash(a).
3. If the tree-log and hash tree caches are not synchronized,
   if $R + \Delta R_{\text{tl-op}} < C_{\text{bkoff}}(n_{lh})$, then tree-log-bkoff.
4. tree-log-store(a, v).

Figure 11. adaptive-tree-log-store**, with caching, when a bitmap is used for bookkeeping**

E. Tradeoffs

In this appendix, we discuss some of the tradeoffs a system designer may consider making, particularly with regards to the cache simulators in Section 6.4. The simulators are being used to help guarantee the worst-case bound when the FSM uses a cache. Though they are small, they do consume extra space overhead. Firstly, if the bound was guaranteed on bandwidth consumption, instead of bandwidth overhead, the base simulator could be dropped. Secondly, if the strictness of the bound is relaxed, we could have conservative estimates for the various tree-log operations. Then, the tree-log simulator could be dropped. Finally, we could have an estimate on the hash tree cost, using information on its cost when all of the data is in the tree and information on the current program access patterns. Then, the hash tree simulator could also be dropped. (If the hash tree simulator is not used, the checker will not be able to synchronize the cache if it backs off, but, in practice, the performance of the tree after it has moved all of the addresses back into the tree should soon be about the same with the unsynchronized cache as with a synchronized cache.) Areas of future research are to investigate heuristics for the various estimations, as well as more sophisticated tree-log strategies (cf. Section 6.3), that would work well in practice in different system implementations.