Towards Constant Bandwidth Overhead Integrity Checking of Untrusted Data

by

Dwaine E. Clarke

S.B., Computer Science, Massachusetts Institute of Technology, 1999
M.Eng., Computer Science, Massachusetts Institute of Technology, 2001

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2005

Certified by .....
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Towards Constant Bandwidth Overhead Integrity Checking of Untrusted Data

by

Dwaine E. Clarke

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

We present a *trace-hash scheme* and an *adaptive tree-trace 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 trace-hash and adaptive tree-trace schemes check sequences of data operations. For each of the schemes, for all programs, as the average number of times the program accesses data between critical operations increases, the scheme's bandwidth overhead approaches a constant bandwidth overhead.

The trace-hash scheme, intuitively, maintains a "write trace" and a "read trace" of the write and read operations on the untrusted data. At runtime, the traces are updated with a minimal constant-sized bandwidth overhead so that the integrity of a sequence of data operations can be verified at a later time. To maintain the traces in a small fixed-sized trusted space, we introduce a new cryptographic tool, *incremental multiset hash functions*, to update the traces. To check a sequence of operations, a separate integrity-check operation is performed using the traces. The integrity-check operation is performed whenever the program executes a critical operation: a critical operation acts as a signal indicating when it is necessary to perform the integrity-check operation. When sequences of operations are checked, the trace-hash scheme significantly outperforms the hash tree.

Though the trace-hash scheme does not incur the logarithmic bandwidth overhead of the hash tree, its integrity-check operation needs to read all of the data that was used since the beginning of the program's execution. When critical operations occur infrequently, the amortized cost over the number of data operations performed of the integrity-check operation is small and the trace-hash scheme performs very well. However, when critical operations occur frequently, the amortized cost of the integrity-check operation becomes prohibitively large; in this case, the performance of the trace-hash scheme is not good and is much worse than that of the hash tree. Thus, though the trace-hash scheme performs very well when checks are infrequent, it cannot be widely-used because its performance is poor when checks are more frequent. To this end, we also introduce an adaptive tree-trace scheme to optimize

the trace-hash scheme and to capture the best features of both the hash tree and trace-hash schemes. The adaptive tree-trace scheme has three features. Firstly, the scheme is adaptive, allowing programs to benefit from its features without any program modification. Secondly, for all programs, the scheme's bandwidth overhead is guaranteed never to be worse than a parameterizable worst-case bound, expressed relative to the bandwidth overhead of the hash tree if the hash tree had been used to check the integrity of the data. Finally, for all programs, as the average number times the program accesses data between critical operations increases, the scheme's bandwidth overhead moves from a logarithmic to a constant bandwidth overhead.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Firstly, many thanks and praises to the Almighty by Whom I have been blessed.

Many thanks to my family who has always been there for me. Mummy, Daddy, Sylvan, Gerren, thank you very much; I really appreciate it.

Many thanks to my thesis advisor, Prof. Srinivas Devadas. I was blessed with a gem of an advisor in Srini. Srini was a pleasure to work with, and he attracted and hired people who were a pleasure to work with. Srini's support enabled me to do the quality of research that he believed that I was capable of doing, and he provided me an unmatched environment to conduct such research. The result was that my research was accepted in top conferences in my fields. I can truly say that my experience with Srini was a very positive one, for which I am very grateful.

Many thanks to the following people, with whom, besides Srini, I worked closely on this research: Ed Suh, Blaise Gassend and Marten van Dijk. Ed always happily answered any questions that I had about computer hardware and his suggestion about cache simulators for the adaptive tree-trace checker reduced the time towards my graduation significantly. Blaise was a phenomenal sounding board for ideas: once I had ideas that survived Blaise's critique, I knew that the ideas were definitely worth the investment of time and energy. Marten was an immense help with the thesis chapter on multiset hash functions and his assistance with the Asiacrypt 2003 paper was a significant boost to this research.

Many thanks to Kirimania Murithi and Ana Isasi for their friendship and support throughout graduate school. I had met Kiri and Ana at International Orientation 1995 (Kiri's from Kenya and Ana's from Uruguay). Their continuing friendship throughout these ten years has been among my most valuable MIT experiences.

Many thanks to all my other friends - and with this statement, you've all made it into the acknowledgements!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Overview

This thesis 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 [20] 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 [5, 10, 19], 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 which an adversary is unable to obtain any information about the program, and in which an adversary cannot tamper with the program's execution without being detected. In such an application, an adversary's job is to try 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 [10, 25], with regard 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 instruc-

tion. 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.

This thesis presents two new schemes, a *trace-hash scheme* [8, 11] and an *adaptive tree-trace scheme* [7]. For each of the schemes, for all programs, as the average number of times the program accesses data between critical operations increases, the scheme's bandwidth overhead approaches a constant bandwidth overhead.

Intuitively, in the trace-hash scheme, the processor maintains a "write trace" and a "read trace" of its write and read operations to the external memory. At runtime, the processor updates the traces 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 traces in a small fixed-sized trusted space in the processor, we introduce a new cryptographic tool, *incremental multiset hash functions* [8], to update the traces. When the processor needs to check a sequence of its operations, it performs a separate *integrity-check* operation using the traces. The integrity-check operation is performed whenever the program executes a critical instruction: a critical instruction acts as a signal indicating when it is necessary to perform the integrity-check operation. When sequences of operations are checked, the trace-hash scheme significantly outperforms the hash tree. (Theoretically, the hash tree checks each memory operation as it is performed. However, in a secure 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 trace-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 over the number of memory operations performed of the integrity-check operation is

very small. The bandwidth overhead of the trace-hash scheme is mainly its constant-sized runtime bandwidth overhead, which is small. This leads the trace-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 performs a small number of memory operations and uses a small subset of the addresses that are protected by the trace-hash scheme between the checks. The amortized cost of the integrity-check operation is large. As a result, the performance of the trace-hash scheme is not good and is much worse than that of the hash tree. Thus, though the trace-hash scheme performs very well when checks are infrequent, it cannot be widely-used because its performance is poor when checks are frequent.

To this end, we also introduce secure tree-trace integrity checking. This hybrid scheme of the hash tree and trace-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 trace-hash scheme. When the trace-hash scheme is used, *only the addresses of the data that have been moved to the trace-hash scheme since the last trace-hash integrity check* need to be read to perform the next trace-hash integrity check, instead of reading all of the addresses that the program used since the beginning of its execution. This optimizes the trace-hash scheme, facilitating much more frequent trace-hash integrity checks, making the trace-hash approach more widely-applicable.

The tree-trace scheme we present has three features. Firstly, the scheme *adaptively* chooses a *tree-trace strategy* for the program that indicates how the program should use the tree-trace scheme when the program is run. This allows programs to be run unmodified and still benefit from the tree-trace 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 never to 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-trace integrity checking moves from *a logarithmic to a constant bandwidth overhead*.

With regard to the second feature, the worst-case bound is a parameter to the adaptive

tree-trace 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-trace 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-trace scheme to be turned on by default in applications. To provide the bound, we use the notion of a *potential* [29, Chapter 18] to determine when data should just be kept in the tree and to regulate the rate at which data is added to the trace-hash scheme. The adaptive tree-trace 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 regard to the third feature, the adaptive tree-trace scheme is able to approach a constant bandwidth data integrity checking overhead because it can use the optimized trace-hash scheme to check sequences of data operations before a critical operation is performed. The longer the sequence, the more data the tree-trace scheme moves from the tree to the trace-hash scheme and the more the overhead approaches the constant-runtime overhead of the trace-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-trace scheme, we will state the feature more precisely for the case without caching in Section 9.2.2, and modify the theoretical claims on the feature for the case with caching in Section 9.2.3.)

The thesis is primarily focused on providing the theoretical foundations for the trace-hash and adaptive tree-trace schemes. For the trace-hash scheme, the thesis will present processor simulation results comparing the trace-hash and hash tree schemes. For the adaptive tree-trace scheme, the thesis will present software experimental results showing that the bandwidth overhead can be significantly reduced when the adaptive tree-trace scheme is used, as compared to when a hash tree is used. In light of the adaptive tree-trace algorithm's

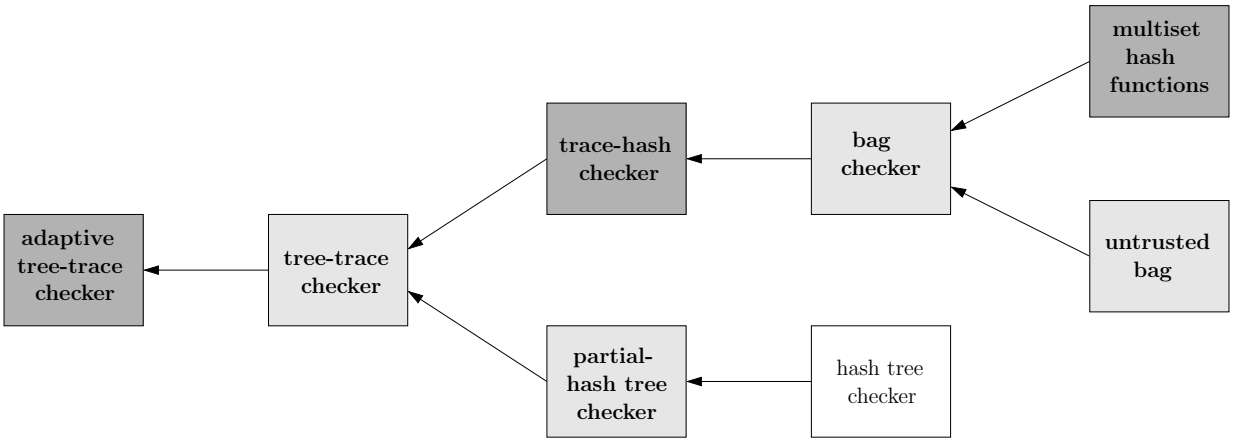


Figure 1-1: Interfaces

features and the results, the thesis will provide a discussion on tradeoffs a system designer may consider making when implementing the scheme in his system.

Hash trees have been implemented in both software and hardware applications. For simplicity, throughout the thesis, we will use secure processors and memory integrity checking as our example application. The trace-hash scheme is well suited for the certified execution secure processor application [3], where the processor signs a certificate at the end of a program's execution certifying the results produced by the program. The adaptive tree-trace algorithm can be implemented anywhere where hash trees are currently being used to check untrusted data. The application can experience a significant benefit if programs can perform sequences of operations before performing a critical operation. The general trend is that the greater the hash tree bandwidth overhead, the greater will be the adaptive tree-trace scheme's improvement when the scheme improves the processor's performance.

1.2 Thesis Organization

Figure 1-1 illustrates the relationships between the different interfaces that will be introduced in the thesis. The thesis is organized as follows:

- Chapter 2 describes related work.

- Chapter 3 presents our model.
- Chapter 4 presents background information on memory integrity checking: it describes the hash tree checker.
- Chapter 5 defines and describes multiset hash functions.
- Chapter 6 demonstrates how to build the secure bag checker from an untrusted bag and the multiset hash functions.
- Chapter 7 demonstrates how to build the trace-hash checker from the bag checker.
- Chapter 8 describes how to construct the partial-hash tree checker from a hash tree checker, and how to build the tree-trace checker from the trace-hash checker and the partial-hash tree checker.
- Chapter 9 describes how to construct the adaptive tree-trace checker from the tree-trace checker.
- Chapter 10 explores additional ideas that showed initial promise, but, upon careful examination, are vulnerable to attacks.
- Chapter 11 concludes the thesis.

The bag checker is a useful abstraction that we use to prove various theorems about the trace-hash checker. *All of the interfaces in Figure 1-1 are new, except for the hash tree checker.* The main contributions of the thesis are:

1. the incremental multiset hash functions [8],
2. the trace-hash integrity checker [8, 11],
3. and the adaptive tree-trace integrity checker [7].

Chapter 2

Related Work

Bellare, Guérin and Rogaway [23] and Bellare and Micciancio [21] describe incremental hash functions that operate on lists of strings. For the hash functions in [21, 23], the order of the inputs is important: if the order is changed, a different output hash is produced. Our incremental multiset hash functions operate on multisets (or sets) and the order of the inputs is unimportant.

Benaloh and de Mare [18], Barić and Pfitzmann [24] and Camenisch and Lysyanskaya [13] describe cryptographic accumulators. A cryptographic accumulator is an algorithm that hashes a large set of inputs into one small, unique, fixed-sized value, called the accumulator, such that there is a short witness (proof) that a given input was incorporated into the accumulator; it is also infeasible to find a witness for a value that was not incorporated into the accumulator. Cryptographic accumulators are incremental and the order in which the inputs are hashed does not matter. Compared with our multiset hash functions, cryptographic accumulators have the additional property that each value in the accumulator has a short proof of this fact. However, the cryptographic accumulators in [13, 18, 24] are more computationally expensive than our multiset hash functions because they use modular exponentiation operations, whereas the our most expensive multiset hash function makes use of multiplication modulo a large prime.

The use of a hash tree (also known as a Merkle tree [26]) to check the integrity of untrusted memory was introduced by Blum et al. [20]. The paper also introduced an offline scheme to check the correctness of memory. The offline scheme in [20] differs from our trace-hash scheme

in two respects. Firstly, the trace-hash scheme is more efficient than the offline scheme in [20] because time stamps can be smaller without increasing the frequency of checks, which improves the performance of the scheme. Secondly, the offline scheme in [20] is implemented with an ϵ-biased hash function [17]; ϵ-biased hash functions can detect random errors, but are not secure against active adversaries. The trace-hash scheme is secure against an active adversary because it is implemented with multiset-collision resistant multiset hash functions. We note that the offline scheme can be made secure against an active adversary if it used a set-multiset-collision resistant multiset hash function, instead of an ϵ-biased hash function. By themselves, trace-based schemes, such as the trace-hash scheme and the offline scheme in [20], are not general enough because they do not perform well when integrity checks are frequent. Our tree-trace scheme can use the hash tree when checks are frequent and move data from the tree to the trace-hash scheme as sequences of operations are performed to take advantage of the constant runtime bandwidth overhead of the trace-hash scheme.

Hall and Jutla [9] 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 trace-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-trace scheme can reduce the overhead to a constant bandwidth overhead.

Chapter 3

Model

Figure 3-1 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 ex-

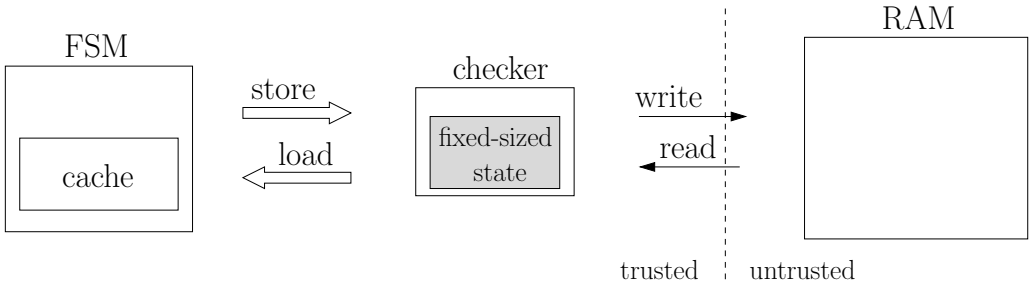


Figure 3-1: Model

ecution of the FSM. The adversary could corrupt the entire contents of the RAM and there is no general way of recovering from tampering 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.

Chapter 4

Background

This chapter describes previous work on memory integrity checking. Chapter 2 describes how our work compares with the work presented in this chapter.

4.1 Hash Tree Checker

The scheme with which we compare our work is integrity checking using hash trees [20]. Figure 4-1 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

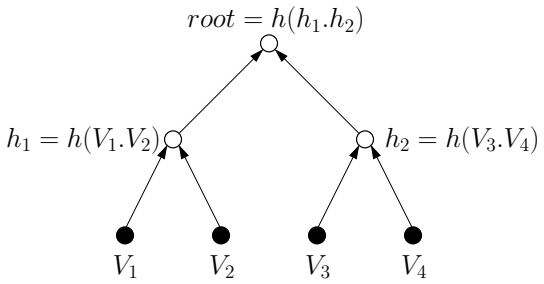


Figure 4-1: A binary hash tree

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 hashes that must be fetched/updated 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.

4.1.1 Caching

A cache can be used to improve the performance of the scheme [3, 30] (the model in Chapter 3 is augmented such that the checker is able to read and write to the trusted 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 Offline Checker

The offline checker [20] intuitively maintains a "trace" of the sequence of its operations on the untrusted RAM. In its fixed-sized trusted state, the checker maintains:

- the description of a hash function h,
- $h(W)$, the hash of a string W that encodes the information in all of the checker's write operations on the untrusted RAM,
- $h(R)$, the hash of a string R that encodes the information in all of the checker's read operations on the untrusted RAM,
- a counter.

The hash function h must be incremental (c.f Section 5.1) because the checker must be able to update $h(W)$ and $h(R)$ quickly.

Whenever the FSM stores a data value v to address a in RAM, the checker:

- reads the data value v' and time stamp t' stored in address a,
- checks that t' is less than or equal to the current value of the counter,
- updates $h(R)$ with the (a, v', t') triple,
- increments the counter,
- writes the new data value v and the current value of the counter t to address a,
- updates $h(W)$ with the (a, v, t) triple.

Whenever the FSM loads a data value from address a in RAM, the checker:

- reads the data value v' and time stamp t' stored in address a,
- checks that t' is less than or equal to the current value of the counter,
- updates $h(R)$ with the (a, v', t') triple,
- increments the counter,
- writes the data value v' and the current value of the counter t to address a,
- updates $h(W)$ with the (a, v', t) triple.

The untrusted RAM is initialized by writing zero data values with time stamps to each address in the RAM, updating the counter and $h(W)$ accordingly. To check the RAM at the end of a sequence of operations, the checker reads all of the RAM, checking the time stamps and updating $h(R)$ accordingly. The RAM has behaved like valid RAM if and only if W is equal to R. The counter can be reset when the RAM is checked.

In [20], an ϵ-biased hash function [17] is proposed for the implementation of h. ϵ-biased hash functions can detect random errors, but are not secure against active adversaries. Because the adversary controls the pairs that are read from the untrusted RAM, the pairs that are used to update $h(R)$ can form a multiset. The checker's counter is incremented each time that the checker writes to the untrusted RAM. Furthermore, the counter is not a function of the pairs that the checker reads from RAM and is solely under the control of the checker. This means that the pairs that are used to update $h(W)$ are guaranteed to form a set and a set-multiset-collision resistant hash function (c.f. Section 5.1) is sufficient for the implementation of h.

Chapter 5

Incremental Multiset Hash Functions

Multiset hash functions [8] are a new cryptographic tool that we develop to help build the trace-hash integrity checker. Unlike standard hash functions which take strings as input, multiset hash functions operate on multisets (or sets). They map multisets of arbitrary finite size to strings (hashes) of fixed length. They are incremental in that, when new members are added to the multiset, the hash can be updated in time proportional to the change. The functions may be multiset-collision resistant in that it is difficult to find two multisets that produce the same hash, or set-multiset-collision resistant in that it is difficult to find a set and a multiset that produce the same hash. Multiset-collision resistant multiset hash functions are used to build our trace-hash memory integrity checker (cf. Chapter 7). Set-multiset-collision resistant multiset hash functions can be used to make the offline checker of Blum et al. [20] (cf. Section 4.2) secure against an active adversary.

5.1 Definition

We work with a countable set of values $\mathcal{V}$. We refer to a *multiset* as a finite unordered collection of elements where an element can occur as a member more than once. All sets are multisets, but a multiset is not a set if an element appears more than once. We shall use $\mathcal{M}$ to denote the set of multisets of elements of $\mathcal{V}$.

Let M be a multiset in $\mathcal{M}$. The number of times $v \in \mathcal{V}$ is in the multiset M is denoted by M_v and is called the *multiplicity* of v in M. The number of elements in M, $\sum_{v \in \mathcal{V}} M_v$, is

called the *cardinality* of M, also denoted as $|M|$.

Multiset union, $\cup_{\mathcal{M}}$, combines two multisets into a multiset in which elements appear with a multiplicity that is the sum of their multiplicities in the initial multisets.

Definition 5.1.1. Let $(\mathcal{H}, \equiv_{\mathcal{H}}, +_{\mathcal{H}})$ be a triple of probabilistic polynomial time (ppt) algorithms. That triple is a *multiset hash function* if it satisfies:

compression: Compression guarantees that we can store hashes in a small bounded amount of memory. $\mathcal{H}$ maps elements of $\mathcal{M}$ into elements of a set with cardinality equal to 2^n, where n is some integer:

$$\forall M \in \mathcal{M} : \ \mathcal{H}(M) \rightarrow \{0,1\}^n.$$

comparability: Since $\mathcal{H}$ can be a probabilistic algorithm, a multiset need not always hash to the same value. Therefore we need $\equiv_{\mathcal{H}}$ to compare hashes. The following relation must hold for comparison to be possible:

$$\forall M \in \mathcal{M} : \ \mathcal{H}(M) \equiv_{\mathcal{H}} \mathcal{H}(M).$$

incrementality: We would like to be able to efficiently compute $\mathcal{H}(M \cup_{\mathcal{M}} M')$ knowing $\mathcal{H}(M)$ and $\mathcal{H}(M')$. The $+_{\mathcal{H}}$ operator makes that possible:

$$\forall M, M' \in \mathcal{M} : \ \mathcal{H}(M \cup_{\mathcal{M}} M') \equiv_{\mathcal{H}} \mathcal{H}(M) +_{\mathcal{H}} \mathcal{H}(M').$$

In particular, knowing only $\mathcal{H}(M)$ and an element $v \in \mathcal{V}$, we can easily compute $\mathcal{H}(M \cup_{\mathcal{M}} \{v\}) = \mathcal{H}(M) +_{\mathcal{H}} \mathcal{H}(\{v\})$.

As it is, this definition is not very useful, because $\mathcal{H}$ could be any constant function. We need to add some kind of collision resistance to have a useful hash function. A multiset hash function is *multiset-collision resistant* if it is computationally infeasible to find a multiset M of $\mathcal{V}$ and a multiset M' of $\mathcal{V}$ such that $M \neq M'$ and $\mathcal{H}(M) \equiv_{\mathcal{H}} \mathcal{H}(M')$. A multiset hash function is *set-multiset-collision resistant* if it is computationally infeasible to find a set S of

$\mathcal{V}$ and a multiset M of $\mathcal{V}$ such that $S \neq M$ and $\mathcal{H}(S) \equiv_{\mathcal{H}} \mathcal{H}(M)$. The following definitions make these notions formal.

Definition 5.1.2. Let F be a family of multiset hash functions; each multiset hash function in F is indexed by its own seed (key). We denote a multiset hash function's seed as k. For $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$ in F, we denote by n the logarithm of the cardinality of the set into which $\mathcal{H}_k$ maps multisets of $\mathcal{V}$, that is n is the number of output bits of $\mathcal{H}_k$. By $\mathcal{A}(\mathcal{H}_k)$ we denote a probabilistic polynomial time, in n, algorithm with oracle access to $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$.

The family F satisfies multiset-collision resistance if, for all ppt algorithms $\mathcal{A}$, any number c,

$$\exists n_0 : \forall n \geq n_0, \text{Prob}\left\{\begin{array}{l} k \xleftarrow{R} \{0,1\}^n, (M, M') \leftarrow \mathcal{A}(\mathcal{H}_k) : \\ M \text{ is a multiset of } \mathcal{V} \text{ and } M' \text{ is a multiset of } \mathcal{V} \\ \text{ and } M \neq M' \text{ and } \mathcal{H}_k(M) \equiv_{\mathcal{H}_k} \mathcal{H}_k(M') \end{array}\right\} < n^{-c}.$$

The probability is taken over a random selection of k in $\{0,1\}^n$ (denoted by $k \xleftarrow{R} \{0,1\}^n$) and over the randomness used in the ppt algorithm $\mathcal{A}(\mathcal{H}_k)$.

The family F satisfies set-multiset-collision resistance if, for all ppt algorithms $\mathcal{A}$, any number c,

$$\exists n_0 : \forall n \geq n_0, \text{Prob}\left\{\begin{array}{l} k \xleftarrow{R} \{0,1\}^n, (S, M) \leftarrow \mathcal{A}(\mathcal{H}_k) : \\ S \text{ is a set of } \mathcal{V} \text{ and } M \text{ is a multiset of } \mathcal{V} \\ \text{ and } S \neq M \text{ and } \mathcal{H}_k(S) \equiv_{\mathcal{H}_k} \mathcal{H}_k(M) \end{array}\right\} < n^{-c}.$$

The Attack Scenario for the Multiset Hash Functions

In accordance with our model (cf. Chapter 3), the outputs of the multiset hash functions are maintained in the checker's small, fixed-sized, trusted state. Typically, we allow the adversary to observe the outputs but the adversary cannot tamper with them. If the multiset hash function uses a secret key, the checker maintains the secret key in a private and authentic manner; for clarity, if the multiset hash function uses a secret key, when the checker updates the multiset hash function outputs, the updates occur privately and authentically, though after the outputs have been updated, the adversary can then observe the outputs.

	collision resistance	computational efficiency	secret key	security based on
`MSet-XOR-Hash`	set-multiset	+	Y	PRF
`MSet-Add-Hash`	multiset	+	Y	PRF
`MSet-Mu-Hash`	multiset	−	N	RO/DL

Table 5.1: Comparison of the multiset hash functions

Because, in our trace-hash application, the checker adds elements to multisets one by one, in order for the adversary to exploit a collision in the multiset hash function, the multisets must be polynomial sized in n. An adversary may be able to compute, in polynomial time, collisions using exponentially-sized multisets by, for example, repeatedly applying the $+_{\mathcal{H}_k}$ operation on a multiset hash function output (i.e., the adversary adds the output to itself, adds the result to itself, and so on). However, these collisions cannot be used by an adversary in our particular application because the checker can only compute hashes of multisets polynomial sized in n. Thus, in our attack model, we require that an adversary can only construct a collision using multisets whose size are each polynomial in n.

The adversary is a probabilistic polynomial time, in n, algorithm with oracle access to $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$. The adversary has oracle access to $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$ because the adversary can tamper with the untrusted RAM to have the checker compute $\mathcal{H}_k$, $\equiv_{\mathcal{H}_k}$ or $+_{\mathcal{H}_k}$ on multisets of it choosing. The attack scenario is for the adversary to adaptively make oracle queries to gain knowledge of at most a polynomial number of tuples $[M_i \ ; \ \mathcal{H}_k(M_i)]$. With the aid of $\equiv_{\mathcal{H}_k}$ and $+_{\mathcal{H}_k}$, the adversary's goal is to find a collision (M, M'), $M \neq M'$ and $\mathcal{H}_k(M) \equiv_{\mathcal{H}_k} \mathcal{H}_k(M')$, where M is a (multi)set of elements of $\mathcal{V}$ and M' is a multiset of elements of $\mathcal{V}$. For our trace-hash application, for a collision to be useful to the adversary, M and M' must each be polynomial sized in n.

Comparison of the Multiset Hash Functions

We introduce three multiset hash functions: `MSet-XOR-Hash`, which is set-multiset-collision resistant, and `MSet-Add-Hash` and `MSet-Mu-Hash`, which are multiset-collision resistant. (In `MSet-XOR-Hash` and `MSet-Add-Hash`, the seed (key) k is selected uniformly from $\{0,1\}^n$ and is secret; in `MSet-Mu-Hash`, the seed (key) is selected uniformly from the set of n-bit

primes and is public.) `MSet-XOR-Hash` and `MSet-Add-Hash` are efficient because they use addition operations. The advantage of `MSet-Mu-Hash` is that it does not require a secret key; however it relies on multiplication modulo a large prime, which makes it too costly for some applications. Table 5.1 summarizes our comparison of the multiset hash functions. In the table, we indicate whether the security is based on assuming a pseudorandom family of hash functions (PRF), or the random oracle model (RO) and the discrete log assumption (DL).

5.2 `MSet-XOR-Hash`

Definition 5.2.1. Let H_k be a pseudorandom function keyed with a seed (key) k. $H_k : \{0,1\}^l \to \{0,1\}^m$. (In practice, one could use the HMAC method [12] to construct such an H_k). Let $r \xleftarrow{R} \{0,1\}^m$ denote uniform random selection of r from $\{0,1\}^m$. Then define[1] `MSet-XOR-Hash` by:

$$\mathcal{H}_k(M) = \left(\left(H_k(0,r) \oplus \bigoplus_{v \in \mathcal{V}} M_v H_k(1,v) \right); |M| \bmod 2^m; r \right) \text{ where } r \xleftarrow{R} \{0,1\}^m$$

$$(h,c,r) \equiv_{\mathcal{H}_k} (h',c',r') = \left(h \oplus H_k(0,r) = h' \oplus H_k(0,r') \wedge c = c' \right)$$

$$(h,c,r) +_{\mathcal{H}_k} (h',c',r') =$$
$$\left(H_k(0,r'') \oplus (h \oplus H_k(0,r)) \oplus (h' \oplus H_k(0,r')); c + c' \bmod 2^m; r'' \right) \text{ where } r'' \xleftarrow{R} \{0,1\}^m$$

Theorem 5.2.1. *As long as the key k remains secret (i.e., is only accessible by the checker),* ***`MSet-XOR-Hash`*** *is a set-multiset-collision resistant multiset hash function.*

Proof. Since the algorithms clearly run in polynomial time, we simply verify the different conditions.

[1] In a real implementation, the three parts of the hash need not be assigned the same number of bits. However the size of each part is a security parameter. Also, for accuracy, $H_k(0,r)$ is actually $H_k(0^{l-m},r)$, where $m+1 \leq l$.

Compression: The output of `MSet-XOR-Hash` is $n = 3m$ bits long by construction.

Comparability and Incrementality: Follows from the definitions of $\mathcal{H}_k$, $\equiv_{\mathcal{H}_k}$, and $+_{\mathcal{H}_k}$.

Set-multiset-collision resistance: We prove the set-multiset-collision resistance of `MSet-XOR-Hash` in Section 5.5.1. We prove the following theorem:

Theorem 5.2.2. *If $\mathcal{H}_k$ is modelled as a random function, then the family of `MSet-XOR-Hash` hash functions is set-multiset-collision resistant (cf. Definition 5.1.2).*

Remark. Theorem 5.2.2 also holds if $\mathcal{H}_k$ is from a pseudorandom family of hash functions.

□

5.2.1 Set-Multiset-Collision Resistance of `MSet-XOR-Hash`

`MSet-XOR-Hash` is set-multiset-collision resistant but not multiset-collision resistant. For instance, though $\mathcal{H}_k(\{1,2\}) \neq \mathcal{H}_k(\{2,2\})$, $\mathcal{H}_k(\{1,1\}) = \mathcal{H}_k(\{2,2\})$. In this section, we describe the various components of `MSet-XOR-Hash` and show that they are necessary. The essential notion for `MSet-XOR-Hash` is that $H_k(0,r)$ conceals $\bigoplus_{v\in\mathcal{V}} M_v H_k(1,v)$, preventing an adversary (without access to key k) from gaining any information on $\bigoplus_{v\in\mathcal{V}} M_v H_k(1,v)$.

Cardinality Counter

$\mathcal{H}_k(M)$ includes the cardinality of M. If the cardinality counter were removed, we would have:

$$\mathcal{H}'_k(M) = \left(\left(H_k(0,r) \oplus \bigoplus_{v\in\mathcal{V}} M_v H_k(1,v)\right); r\right).$$

In this case, $\mathcal{H}'_k(S)$ and $\mathcal{H}'_k(M)$ are equivalent for any set, S, and multiset, M, with $S_v = M_v \bmod 2$ for all $v \in \mathcal{V}$. For instance, $\mathcal{H}'_k(\{1\}) = \mathcal{H}'_k(\{1,1,1\})$. This contradicts set-multiset-collision resistance.

For semantic reasons, because the output of a hash function must be of fixed size, we express the cardinality counter modulo 2^m. However, we note that the cardinality counter

must not be allowed to overflow, else the hash function is not set-multiset-collision resistant. For multisets polynomial-sized in m, the cardinality counter will not overflow for large enough m. We note that, if necessary, the number of bits used for the cardinality counter can be larger than the number of bits that are used for the other parts of the output hash.

Randomly-Chosen Nonce

Notice that $r \xleftarrow{R} \{0,1\}^m$ is randomly chosen. If r was a fixed constant, $\overline{r}$, we would have:

$$\mathcal{H}'_k(M) = \left(\left(H_k(0,\overline{r}) \oplus \bigoplus_{v \in \mathcal{V}} M_v H_k(1,v) \right); |M| \bmod 2^m; \overline{r} \right).$$

Given oracle access to $(\mathcal{H}'_k, \equiv_{\mathcal{H}'_k}, +_{\mathcal{H}'_k})$, $H_k(0,\overline{r})$ can be easily determined: pick a random $v_a \in \mathcal{V}$ and compute $(H_k(0,\overline{r}), 2, \overline{r}) = \mathcal{H}'_k(\{v_a, v_a\})$. Thus, knowledge of t tuples $[M_i \ ; \ \mathcal{H}'_k(M_i)]$ reveals t vectors:

$$\bigoplus_{v \in \mathcal{V}} M_{i_v} H_k(1,v) \in \{0,1\}^m.$$

After a polynomial, in m, number of randomly-chosen vectors, with high probability these t vectors will span the vector space $\mathbb{Z}_2^m$ (the set of vectors of length m and entries in $\mathbb{Z}_2$). This means that *any* vector in $\mathbb{Z}_2^m$ can be constructed as a linear combination of these t vectors:

$$\bigoplus_{i=1}^{t} a_i \cdot \left(\bigoplus_{v \in \mathcal{V}} M_{i_v} H_k(1,v) \right) = \bigoplus_{v \in \mathcal{V}} \left(\bigoplus_{i=1}^{t} a_i M_{i_v} \right) H_k(1,v).$$

Hence, for any polynomial-sized set, S, an adversary can construct a polynomial-sized multiset that is a collision for S using $M_1, M_2, \ldots, M_t$. This contradicts set-multiset-collision resistance.

Prefixing 0 to the nonce and 1 to the elements of $\mathcal{V}$

Notice that 0 is prefixed to the random nonce and 1 is prefixed to the elements of $\mathcal{V}$. If this were not the case, we would have:

$$\mathcal{H}'_k(M) = \left(\left(H_k(r) \oplus \bigoplus_{v \in \mathcal{V}} M_v H_k(v) \right); |M| \bmod 2^m; r \right).$$

A linear combination attack can be conducted, similar to the linear combination attack that is used when the nonce is fixed. Knowledge of t tuples $[M_i \ ; \ \mathcal{H}'_k(M_i)]$ reveals t vectors:

$$\left(H_k(r_i) \oplus \bigoplus_{v \in \mathcal{V}} M_{i_v} H_k(v) \right) \in \{0,1\}^m.$$

After a polynomial, in m, number of such vectors, with high probability these t vectors will span the vector space $\mathbb{Z}_2^m$. This means that *any* vector in $\mathbb{Z}_2^m$ can be constructed as a linear combination of these t vectors:

$$\bigoplus_{i=1}^{t} a_i \cdot \left(H_k(r_i) \oplus \bigoplus_{v \in \mathcal{V}} M_{i_v} H_k(v) \right) = \bigoplus_{i=1}^{t} a_i H_k(r_i) \oplus \bigoplus_{v \in \mathcal{V}} \left(\bigoplus_{i=1}^{t} a_i M_{i_v} \right) H_k(v).$$

The nonces in the linear combination are *indistinguishable* from the elements of $\mathcal{V}$. For any polynomial-sized set, S, an adversary can construct a polynomial-sized multiset that is a collision for S using $M_1 \cup \{r_1\}, M_2 \cup \{r_2\}, \ldots, M_t \cup \{r_t\}$. This contradicts set-multiset-collision resistance. When a 0 is prefixed to the nonce and a 1 to the elements of $\mathcal{M}$, such attacks fail with high probability because, then, the nonces are distinct from the elements of $\mathcal{M}$ and thus cannot be used as members of collisions.

Secret key k

The pseudorandom function used for the evaluation of the nonce and the evaluation of elements in $\mathcal{V}$ is keyed. If the key were removed in the evaluation of the nonce, we would have:

$$\mathcal{H}'_k(M) = \left(\left(H(0,r) \oplus \bigoplus_{v \in \mathcal{V}} M_v H_k(1,v) \right); |M| \bmod 2^m; r \right).$$

In this case, the adversary can evaluate $H(0,r)$ himself and obtain the vector

$\bigoplus_{v\in\mathcal{V}} M_v H_k(1,v) \in \{0,1\}^m$. After a polynomial number of such vectors, he can perform a linear combination attack that is similar to the attack that is used when the nonce is fixed.

If the key were removed in the evaluation of the elements of $\mathcal{V}$, we would have:

$$\mathcal{H}'_k(M) = \left(\left(H_k(0,r) \oplus \bigoplus_{v\in\mathcal{V}} M_v H(1,v)\right); |M| \bmod 2^m; r\right).$$

In this case, the adversary can evaluate the vector $\bigoplus_{v\in\mathcal{V}} M_v H(1,v) \in \{0,1\}^m$ himself. After a polynomial number of such vectors, he can perform a linear combination attack that is similar to the attack that is used when the nonce is fixed.

Thus, if a secret key is not used, set-multiset-collision resistance is contradicted.

5.2.2 Variants of `MSet-XOR-Hash`

There are a couple of interesting variants of `MSet-XOR-Hash`. The security of these variants is also proven in Section 5.5.2.

`Counter-based-MSet-XOR-Hash`

It is possible to replace the random nonce r by an m-bit counter, COUNTER, that gets incremented before each use of $\mathcal{H}_k$ and $+_{\mathcal{H}_k}$. COUNTER is initialized at 0. Define `Counter-based-MSet-XOR-Hash` by:

$$\mathcal{H}_k(M) = \left(\left(H_k(0,s) \oplus \bigoplus_{v\in\mathcal{V}} M_v H_k(1,v)\right); |M| \bmod 2^m; s\right)$$

where s is the value of COUNTER after it has been incremented.

$$(h,c,s) \equiv_{\mathcal{H}_k} (h',c',s') = \Big(h \oplus H_k(0,s) = h' \oplus H_k(0,s') \wedge c = c'\Big)$$

$$(h,c,s) +_{\mathcal{H}_k} (h',c',s') =$$

$$\Big(H_k(0,s'') \oplus (h \oplus H_k(0,s)) \oplus (h' \oplus H_k(0,s')); c + c' \bmod 2^m; s''\Big)$$

where s'' is the value of COUNTER after it has been incremented.

The advantages of using the counter, COUNTER, are:

- the security of the scheme is stronger (cf. Section 5.5.2).
- it removes the need for a random number generator from the scheme.
- shorter values can be used for COUNTER without decreasing the security of the scheme, as long as the secret key is changed when the counter overflows. This reduces the size of the hash.

The disadvantage of using the counter is that more state needs to be maintained. When a random number is used, k needs to be maintained in a secret and authentic manner. When a counter is used, k need to be maintained in a secret and authentic manner, and COUNTER in an authentic manner.

`Obscured-MSet-XOR-Hash`

The outputs of the multiset hash functions are maintained in the checker's small, fixed-sized, trusted state. We typically allow the adversary to observe the outputs of the multiset hash functions, though the adversary cannot tamper with the outputs (cf. Section 5.1). For the case of `Obscured-MSet-XOR-Hash`, we do not allow the adversary to observe the multiset hash function outputs, i.e., the checker's trusted state is both authentic and private. If the xors of the hashes of the elements of M are completely hidden from the adversary (the adversary knows which M is being hashed, but not the value of the xors of the hashes of the elements of M), then the nonce/counter can be removed from the scheme altogether. In particular:

$$\mathcal{H}_k(M) = \left(\left(\bigoplus_{v \in \mathcal{V}} M_v H_k(v) \right); |M| \bmod 2^m \right)$$
$$(h, c) \equiv_{\mathcal{H}_k} (h', c') = \Big(h = h' \wedge c = c' \Big)$$
$$(h, c) +_{\mathcal{H}_k} (h', c') = \Big(h \oplus h'; c + c' \bmod 2^m \Big)$$

This variant is the simplest, and its security is equivalent to that of `Counter-based-MSet-XOR-Hash`. However, it does require that the output hashes be secret.

`Encrypted-MSet-XOR-Hash`

`Encrypted-MSet-XOR-Hash` is an extension of `Obscured-MSet-XOR-Hash`, where, instead of relying on the application to keep the xor of the hashes of the elements of M hidden, the multiset hash function encrypts these values with a pseudorandom permutation. Let $P_{k'}$ be a pseudorandom permutation keyed with seed (key) k'. $P_{k'} : \{0,1\}^m \to \{0,1\}^m$. Let $P_{k'}^{-1}$ be the inverse of $P_{k'}$.

$$\mathcal{H}_k(M) = \left(P_{k'}\left(\bigoplus_{v \in \mathcal{V}} M_v H_k(v) \right); |M| \bmod 2^m \right)$$
$$(h,c) \equiv_{\mathcal{H}_k} (h',c') = \left(P_{k'}^{-1}(h) = P_{k'}^{-1}(h') \wedge c = c' \right)$$
$$(h,c) +_{\mathcal{H}_k} (h',c') = \left(P_{k'}(P_{k'}^{-1}(h) \oplus P_{k'}^{-1}(h')); c + c' \bmod 2^m \right)$$

Again, the security of `Encrypted-MSet-XOR-Hash` is equivalent to that of `Counter-based-MSet-XOR-Hash`. Compared to `Obscured-MSet-XOR-Hash`, the output hashes no longer need to be kept secret, but there is a second key that needs to be maintained in a secret and authentic manner.

5.3 `MSet-Add-Hash`

Definition 5.3.1. Let H_k be a pseudorandom function keyed with a seed (key) k. $H_k : \{0,1\}^l \to \{0,1\}^m$. (In practice, one could use the HMAC method [12] to construct such an H_k). Let[2] $r \xleftarrow{R} \{0,1\}^m$ denote uniform random selection of r from $\{0,1\}^m$. Then define `MSet-Add-Hash` by:

[2]In a real implementation, the two parts of the hash need not be assigned the same number of bits. However the size of each part is a security parameter.

$$\mathcal{H}_k(M) = \left(H_k(0,r) + \sum_{v \in \mathcal{V}} M_v H_k(1,v) \bmod 2^m \ ; \ r \right) \text{ where } r \xleftarrow{R} \{0,1\}^m$$

$$(h,r) \equiv_{\mathcal{H}_k} (h',r') = \Big((h - H_k(0,r)) = (h' - H_k(0,r')) \bmod 2^m \Big)$$

$$(h,r) +_{\mathcal{H}_k} (h',r') =$$

$$\Big(H_k(0,r'') + (h - H_k(0,r)) + (h' - H_k(0,r')) \bmod 2^m \ ; \ r'' \Big) \text{ where } r'' \xleftarrow{R} \{0,1\}^m$$

Theorem 5.3.1. *As long as the key k remains secret (i.e., is only accessible by the checker),* ***MSet-Add-Hash*** *is a multiset-collision resistant multiset hash function.*

Proof. Since the algorithms clearly run in polynomial time, we simply verify the different conditions.

Compression: The output of `MSet-Add-Hash` is $n = 2m$ bits long by construction.

Comparability and Incrementality: Follows from the definitions of $\mathcal{H}_k$, $\equiv_{\mathcal{H}_k}$, and $+_{\mathcal{H}_k}$.

Multiset-collision resistance: We prove the multiset-collision resistance of `MSet-Add-Hash` in Section 5.6.1. We prove the following theorem:

Theorem 5.3.2. *If $\mathcal{H}_k$ is modelled as a random function, then the family of* ***MSet-Add-Hash*** *hash functions is multiset-collision resistant (cf. Definition 5.1.2).*

Remark. Theorem 5.2.2 also holds if $\mathcal{H}_k$ is from a pseudorandom family of hash functions.

□

5.3.1 Multiset-Collision Resistance of `MSet-Add-Hash`

`MSet-Add-Hash` is multiset-collision resistant. Note that a cardinality counter is not necessary: because the addition is being computed modulo 2^m and because, for our trace-hash application, for a collision to be useful to the adversary, the multisets in the collision must each be polynomial sized in $n = 2m$, values in the multiset cannot cancel each other out when $\sum_{v \in \mathcal{V}} M_v H_k(1,v) \bmod 2^m$ is computed.

5.3.2 Variants of MSet-Add-Hash

MSet-Add-Hash can be modified similar to the manner in which MSet-XOR-Hash was modified to create Counter-based-MSet-Add-Hash, Obscured-MSet-Add-Hash and Encrypted-MSet-Add-Hash (cf. Section 5.2.2). The security of these schemes is stronger (cf. Section 5.6.2).

5.4 MSet-Mu-Hash

Definition 5.4.1. Let H be a pseudorandom function. $H : \{0,1\}^l \to \{0,1\}^m$. Let p be an m-bit prime. Then define MSet-Mu-Hash by:

$$\mathcal{H}(M) = \Big(\prod_{v \in \mathcal{V}} H(v)^{M_v} \bmod p \Big)$$
$$(h) \equiv_{\mathcal{H}} (h') = \Big(h = h' \Big)$$
$$(h) +_{\mathcal{H}} (h') = \Big(h \times h' \bmod p \Big)$$

Theorem 5.4.1. *MSet-Mu-Hash is a multiset-collision resistant multiset hash function.*

Proof. Since the algorithms clearly run in polynomial time, we simply verify the different conditions.

Compression: Because the multiplication operations are performed modulo an m-bit prime, p, the output of MSet-Mu-Hash is $n = m$ bits long.

Comparability: $\mathcal{H}$ and $+_{\mathcal{H}}$ are deterministic, so simple equality suffices to compare hashes.

Incrementality: Follows from the definitions of $\mathcal{H}$, $\equiv_{\mathcal{H}}$, and $+_{\mathcal{H}}$.

Multiset-collision resistance: We prove the multiset-collision resistance of MSet-Mu-Hash in Section 5.7.1. We prove the following theorem:

Theorem 5.4.2. *If $\mathcal{H}$ is modelled as a random function, then the family of* `MSet-Mu-Hash` *hash functions is multiset-collision resistant (cf. Definition 5.1.2).*

Remark. Theorem 5.2.2 also holds if $\mathcal{H}_k$ is from a pseudorandom family of hash functions.

□

5.5 `MSet-XOR-Hash` Proofs

5.5.1 Proof of Set-Multiset-Collision Resistance of `MSet-XOR-Hash`

In this section, we prove Theorem 5.2.2 (cf. Section 5.2). We will first prove Theorem 5.5.1 and Theorem 5.5.7, and use these theorems to formulate a proof for Theorem 5.2.2.

Recall that $H_k : \{0,1\}^l \rightarrow \{0,1\}^m$. (For accuracy, $H_k(0,r)$ is actually $H_k(0^{l-m}, r)$, where $m+1 \leq l$.) We first model H_k as a random function and prove that, if H_k is a random function, then `MSet-XOR-Hash` is set-multiset-collision resistant. In practice, H_k is a pseudorandom function; after proving the theorems modelling H_k as a random function, we then show how to extend them when H_k is a pseudorandom function.

Let $\mathcal{R}$ be the family of random functions represented as matrices with 2^l rows, m columns, and entries in $\mathbb{Z}_2$. Let H_k be a randomly-chosen matrix in $\mathcal{R} = \{H_1, H_2, H_3, \ldots, H_{2^{m2^l}}\}$. We assume that this matrix is secret (i.e., is only accessible by the checker). The family of matrices $\mathcal{R}$ from which H_k is selected is publicly known.

The rows of H_k are labelled by $x \in \{0,1\}^l$ and denoted by $H_k(x)$. The matrix represents H_k as a random function from $x \in \{0,1\}^l$ to $\mathbb{Z}_2^m$, the set of vectors with length m and entries in $\mathbb{Z}_2$. We note that, because H_k is a random function, each row in H_k is uniformly distributed in $\mathbb{Z}_2^m$.

Theorem 5.5.1 is about the probability that an adversary, given oracle access to $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$, finds two different multisets, M and M', such that $\bigoplus_{v \in \mathcal{V}} M_v H_k(1,v) = \bigoplus_{v \in \mathcal{V}} M'_v H_k(1,v)$. The probability is taken over the random matrices H_k in $\mathcal{R}$, the randomness of the random nonce used in $\mathcal{H}_k$ and the randomness used in the probabilistic polynomial time adversary.

Theorem 5.5.1. *Suppose that an adversary, given oracle access to $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$, gains knowledge of t tuples $[M_i\ ;\ \mathcal{H}_k(M_i)]$. Let M and M' be different multisets of elements of $\mathcal{V}$. Let g be the greatest common divisor of 2 (because the xor operation is addition modulo 2) and each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$. (Since $M, M' \neq \emptyset$, $g = 1$ or $g = 2$.) Given knowledge of t tuples $[M_i\ ;\ \mathcal{H}_k(M_i)]$, the probability that the adversary finds an M and an M', $M \neq M'$, such that $\bigoplus_{v\in\mathcal{V}} M_v H_k(1, v) = \bigoplus_{v\in\mathcal{V}} M'_v H_k(1, v)$ is at most $t^2/2^m + (g/2)^m$.*

Proof. Let $N_i \in \{0,1\}^m$ denote the random variable whose value is the random nonce chosen by the multiset hash when creating $\mathcal{H}_k(M_i)$. Let `Distinct` be the event that $N_1, N_2, \ldots, N_t$ are all distinct and `Succ` be the event that $\bigoplus_{v\in\mathcal{V}} M_v H_k(1, v) = \bigoplus_{v\in\mathcal{V}} M'_v H_k(1, v)$.

We observe that:

$$\text{Prob}(\texttt{Succ}) = \text{Prob}(\texttt{Succ} \cap \texttt{Distinct}) + \text{Prob}(\texttt{Succ} \cap \neg\texttt{Distinct}),$$

$$\begin{aligned}\text{Prob}(\texttt{Distinct}) \leq 1 \Rightarrow \text{Prob}(\texttt{Succ} \cap \texttt{Distinct}) &\leq \frac{\text{Prob}(\texttt{Succ} \cap \texttt{Distinct})}{\text{Prob}(\texttt{Distinct})} \\ &\Rightarrow \text{Prob}(\texttt{Succ} \cap \texttt{Distinct}) \leq \text{Prob}(\texttt{Succ} \mid \texttt{Distinct}),\end{aligned}$$

$$\text{Prob}(\texttt{Succ} \cap \neg\texttt{Distinct}) \leq \text{Prob}(\neg\texttt{Distinct})$$

$$\text{thus Prob}(\texttt{Succ}) \leq \text{Prob}(\texttt{Succ} \mid \texttt{Distinct}) + \text{Prob}(\neg\texttt{Distinct}).$$

Fact 5.5.2. *The probability of at least one collision (i.e., two balls in the same bin) in the experiment of throwing a balls, independently at random, into b bins is $\leq \frac{a^2}{b}$ [23].*

Using Fact 5.5.2, $\text{Prob}(\neg\texttt{Distinct}) \leq t^2/2^m$.

Thus, we now need to show that

$$\text{Prob}(\texttt{Succ} \mid \texttt{Distinct}) \leq (g/2)^m \tag{5.1}$$

This follows.

Assume $N_1, N_2, \ldots, N_t$ are all distinct. Let us introduce some notation. Let $e(r, M)$ be a vector of integers of length 2^l. Its entries are indexed by all l-bit strings in lexicographic order. Let $e(r, M)_{(i)}$ denote the i^{th} entry of $e(r, M)$. Then, we define $e(r, M)$ by

$$e(r, M)_{(0v)} = 1 \text{ if and only if } v = r$$

and

$$e(r, M)_{(1v)} = M_v.$$

$e(r, M)$ encodes the multiset M and the nonce r that is used when $\mathcal{H}_k(M)$ is created.

Similarly, let $e(M)$ be a 2^l-bit vector defined by

$$e(M)_{(0v)} = 0$$

and

$$e(M)_{(1v)} = M_v.$$

$e(M)$ encodes the multiset M.

Lemma 5.5.3. *(i) Knowing* $[M \ ; \ \mathcal{H}_k(M)]$ *is equivalent to knowing*

$$[e(r, M) \ ; \ e(r, M)H_k \mod 2].$$

(ii) $\mathcal{H}_k(M) \equiv_{\mathcal{H}_k} \mathcal{H}_k(M')$ *if and only if* $e(M)H_k = e(M')H_k \mod 2$ *and* $\sum_{v \in \mathcal{V}} M_v = \sum_{v \in \mathcal{V}} M'_v \mod 2^m$.

Proof. Notice that $e(r, M)$ encodes r, M, and, hence, the cardinality $\sum_{v \in \mathcal{V}} M_v \mod 2^m$ of M, and notice that

$$\mathcal{H}_k(M) = \left[e(r, M)H_k \mod 2 \ ; \ \sum_{v \in \mathcal{V}} M_v \mod 2^m \ ; \ r \right].$$

The lemma follows immediately from these observations. □

Suppose that an adversary learns t tuples $[M_i \ ; \ \mathcal{H}_k(M_i)]$ or, according to Lemma 5.5.3(i),

t vectors $e(r_i, M_i)$ together with the corresponding $e(r_i, M_i)H_k \bmod 2$. Let E be the $t \times 2^l$ matrix with rows $e(r_i, M_i)$. Because, for this part of the proof, we have assumed that the r_i's are all distinct, *matrix E has full row rank.*

Lemma 5.5.4. *Let M and M' be different multisets of elements of $\mathcal{V}$. The probability that H_k satisfies $e(M)H_k = e(M')H_k \bmod 2$ is statistically independent of the knowledge of a full row rank matrix E and the knowledge $\mathtt{K}= EH_k \bmod 2$.*

Proof. Without loss of generality (after reordering the first 2^{l-1} columns of E) matrix E has the form $E = (I\ E_1)$, where I is the $t \times t$ identity matrix. Denote the top t rows of H_k by H_{k_0} and let H_{k_1} be such that

$$H_k = \begin{pmatrix} H_{k_0} \\ H_{k_1} \end{pmatrix}.$$

Clearly, $\mathtt{K}= EH_k \bmod 2$ is equivalent to

$$\mathtt{K} = H_{k_0} + E_1 H_{k_1} \mod 2. \tag{5.2}$$

$e(M)H_k = e(M')H_k \bmod 2 \Rightarrow 0 = (e(M) - e(M'))H_k \bmod 2$. $(e(M) - e(M'))$ has the form $(0\ e_1)$, where 0 is the all zero vector of length 2^{l-1}.

The equation $0 = (e(M) - e(M'))H_k \bmod 2$ is equivalent to

$$0 = e_1 H_{k_1} \mod 2. \tag{5.3}$$

$\text{Prob}((5.3)|(5.2)) = \frac{\text{Prob}((5.3)\cap(5.2))}{\text{Prob}((5.2))} = \frac{\#H_{k_1} \text{ such that } ((5.3)\cap(5.2))}{\text{total } \#H_{k_1}} / \frac{\#H_{k_1} \text{ such that } (5.2)}{\text{total } \#H_{k_1}} = \frac{\#H_{k_1} \text{ such that } ((5.3)\cap(5.2))}{\#H_{k_1} \text{ such that } (5.2)}$. Because, for each H_{k_1}, there exists a unique H_{k_0} such that (5.2) holds, $\frac{\#H_{k_1} \text{ such that } ((5.3)\cap(5.2))}{\#H_{k_1} \text{ such that } (5.2)} = \frac{\#H_{k_1} \text{ such that } (5.3)}{\text{total } \#H_{k_1}} = \text{Prob}((5.3))$.

Thus, the probability that H_k satisfies $e(M)H_k = e(M')H_k \bmod 2$ is statistically independent of the knowledge of a full row rank matrix E and the knowledge $\mathtt{K}= EH_k \bmod 2$. □

Lemma 5.5.5. *Let M and M' be different multisets of elements of $\mathcal{V}$. Let g be the greatest common divisor of 2 and each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$. ($g = 1$ or $g = 2$.) Then $(e(M) - e(M'))H_k \bmod 2$ is uniformly distributed in $g\mathbb{Z}_2^m$.*

Proof. To prove this lemma, we show that each entry of $(e(M) - e(M'))H_k \mod 2$ is uniformly distributed in $g\mathbb{Z}_2$. (For clarity, $g\mathbb{Z}_d = \{0, g, 2g, \ldots, (\frac{d}{\gcd(d,g)} - 1)g\}$. Thus, if $g|d$, then $g\mathbb{Z}_d = \{0, g, 2g, \ldots, (\frac{d}{g} - 1)g\}$. Thus, $\mathbb{Z}_2 = \{0, 1\}$ and $2\mathbb{Z}_2 = \{0\}$. Also $g\mathbb{Z}_2^m$ is the set of vectors with length m and entries in $g\mathbb{Z}_2$.) Let y represent one of the possible columns of H_k.

We first show that $\text{Prob}((e(M) - e(M'))y \mod 2 \notin g\mathbb{Z}_2) = 0$. We see that $g = \gcd(2, |M_v - M'_v| \text{ such that } v \in \mathcal{V}) \Rightarrow g|2$ and, $\forall i : 0 \leq i \leq 2^l,\ g|(e(M) - e(M'))_{(i)}$. Since, $\forall i : 0 \leq i \leq 2^l,\ g|(e(M) - e(M'))_{(i)}$, then $g|(e(M) - e(M'))y$. Let $(e(M) - e(M'))y = a \mod 2$. Then $(e(M) - e(M'))y = a + q2$ for some integer q, and $0 \leq a < 2$. Since $g|(e(M) - e(M'))y$ and $g|2$, then $g|a$. Since $0 \leq a < 2$ and $g|a$, $a \in g\mathbb{Z}_2$.

Secondly, we show that $\forall z_1, z_2 \in g\mathbb{Z}_2$, $\text{Prob}((e(M) - e(M'))y = z_1 \mod 2) = \text{Prob}((e(M) - e(M'))y = z_2 \mod 2)$. Define for $\beta \in g\mathbb{Z}_2$, the set

$$\mathcal{C}_\beta = \{y : (e(M) - e(M'))y = \beta \mod 2\}.$$

For a fixed column $y' \in \mathcal{C}_\beta$, the mapping $y \in \mathcal{C}_\beta \rightarrow y - y' \in \mathcal{C}_0$ is a bijection. Hence, all of the sets $\mathcal{C}_\beta$ have equal cardinality. $\text{Prob}((e(M) - e(M'))y = \beta \mod 2) = \frac{|\mathcal{C}_\beta|}{2^{2^l}}$. Since all of the sets $\mathcal{C}_\beta$ have equal cardinality, $\forall z_1, z_2 \in g\mathbb{Z}_2$, $\text{Prob}((e(M) - e(M'))y = z_1 \mod 2) = \text{Prob}((e(M) - e(M'))y = z_2 \mod 2)$.

We conclude that each entry of $(e(M) - e(M'))H_k \mod 2$ is uniformly distributed in $g\mathbb{Z}_2$. Thus, $(e(M) - e(M'))H_k \mod 2$ is uniformly distributed in $g\mathbb{Z}_2^m$.

□

Lemma 5.5.6. *Let M and M' be different multisets of elements of $\mathcal{V}$. Let g be the greatest common divisor of 2 and each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$. ($g = 1$ or $g = 2$.) Given the knowledge of a full row rank matrix E and given the knowledge $\mathtt{K}= EH_k \mod 2$, the probability that $(e(M) - e(M'))H_k = 0 \mod 2$ is equal to $(g/2)^m$.*

Proof. By Lemma 5.5.4, the probability that H_k satisfies $e(M)H_k = e(M')H_k \mod 2$ is statistically independent of the knowledge of a full row rank matrix E and the knowledge $\mathtt{K}= EH_k \mod 2$. By Lemma 5.5.5, $(e(M) - e(M'))H_k \mod 2$ is uniformly distributed in $g\mathbb{Z}_2^m$. Thus, the probability that $(e(M) - e(M'))H_k = 0 \mod 2$ is equal to one divided

by the cardinality of $g\mathbb{Z}_2^m$. When $g|d$, $|g\mathbb{Z}_d| = \frac{d}{g}$. Thus, Prob$((e(M) - e(M'))H_k = 0 \mod 2) = (\frac{g}{d})^m = (\frac{g}{2})^m$. □

Lemma 5.5.6 proves that Prob(`Succ` | `Distinct`) $= (g/2)^m$. This concludes the proof of Theorem 5.5.1.

□

Theorem 5.5.7 shows the necessity of including the cardinality of the multiset M that is being hashed in $\mathcal{H}_k(M)$ and shows why m bits are sufficient to encode the cardinality.

Theorem 5.5.7. *Let M and M' be different multisets of elements of $\mathcal{V}$. Suppose that each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$ is equal to $0 \mod 2$, and that the multiplicities of M are < 2 (i.e., M is a set), and that $\sum_{v\in\mathcal{V}} M_v = \sum_{v\in\mathcal{V}} M'_v \mod 2^m$ (i.e., $|M| = |M'| \mod 2^m$), and that the cardinalities of M and M' are $< 2^m$. Then $M = M'$.*

Proof. If the cardinalities of M and M' are equal modulo 2^m and $< 2^m$, then

$$\sum_{v\in\mathcal{V}} M_v = \sum_{v\in\mathcal{V}} M'_v. \tag{5.4}$$

If each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$ is equal to $0 \mod 2$, then, for $v \in \mathcal{V}$,

$$\exists \alpha_v \in \mathbb{Z} \text{ such that } M'_v = M_v + 2\alpha_v. \tag{5.5}$$

If all of the multiplicities of M are < 2, then $\forall v \in \mathcal{V}$, $\alpha_v \geq 0$.
If, $\forall v \in \mathcal{V}$, $\alpha_v \geq 0$, and $\sum_{v\in\mathcal{V}} M_v = \sum_{v\in\mathcal{V}} M'_v$, then, $\forall v \in \mathcal{V}$, $\alpha_v = 0$.
If, $\forall v \in \mathcal{V}$, $\alpha_v = 0$, then, $\forall v \in \mathcal{V}$, $M'_v = M_v$. We conclude that $M = M'$. □

With Theorem 5.5.1 and Theorem 5.5.7, we are now ready to prove Theorem 5.2.2.

Proof. Let $\mathcal{A}(\mathcal{H}_k)$ denote a probabilistic polynomial time, in n, algorithm with oracle access to $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$. Then $\mathcal{A}(\mathcal{H}_k)$ can gain knowledge of at most a polynomial number $t(n)$ tuples $[M_i\ ;\ \mathcal{H}_k(M_i)]$ (here $t(n)$ denotes a polynomial in n). Furthermore, $\mathcal{A}(\mathcal{H}_k)$ can search for a collision among at most a polynomial number $u(n)$ of pairs (S, M), $S \neq M$, where S is a set of elements of $\mathcal{V}$ and M is a multiset of elements of $\mathcal{V}$.

According to Theorem 5.5.1, the probability that $\mathcal{A}(\mathcal{H}_k)$ finds an (S, M), $S \neq M$, such that $\bigoplus_{v \in \mathcal{V}} S_v H_k(1, v) = \bigoplus_{v \in \mathcal{V}} M_v H_k(1, v)$ is at most $u(n)(t(n)^2/2^m + (g/2)^m)$.

Let us consider one of these pairs (S, M), $S \neq M$ where $\bigoplus_{v \in \mathcal{V}} S_v H_k(1, v) = \bigoplus_{v \in \mathcal{V}} M_v H_k(1, v)$. Since, for our trace-hash application, for a collision to be useful to the adversary, the multisets in the collision must each be of polynomial size in $n = 3m$, the cardinality of S and M are $< 2^m$. Since $S \neq M$, S is a set, the cardinalities of S and M are $< 2^m$, *if* $\sum_{v \in \mathcal{V}} S_v = \sum_{v \in \mathcal{V}} M_v \mod 2^m$, then, by Theorem 5.5.7, *there is at least one difference* $|S_v - M_v|$, $v \in \mathcal{V}$ *that is not equal to* $0 \mod 2$. Hence, the greatest common divisor, g, of 2 and each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$, is equal to 1.

This means that, by Theorem 5.5.1 and 5.5.7, the probability that $\mathcal{A}(\mathcal{H}_k)$ finds a collision (S, M), $S \neq M$, such that $\bigoplus_{v \in \mathcal{V}} S_v H_k(1, v) = \bigoplus_{v \in \mathcal{V}} M_v H_k(1, v)$ and $\sum_{v \in \mathcal{V}} S_v = \sum_{v \in \mathcal{V}} M_v \mod 2^m$ is at most

$$u(n)\left(\frac{t(n)^2}{2^m} + \frac{1}{2^m}\right).$$

Thus, for any number c, for sufficiently-large m, the probability that $\mathcal{A}(\mathcal{H}_k)$ finds a collision (S, M), $S \neq M$, such that $\mathcal{H}_k(S) \equiv_{\mathcal{H}_k} \mathcal{H}_k(M)$ is $\leq m^{-c}$. $n = 3m$. Thus, for all ppt algorithms $\mathcal{A}$, any number c,

$$\exists n_0 : \forall n \geq n_0, \text{Prob}\left\{ \begin{array}{l} k \xleftarrow{R} \{0,1\}^n, (S, M) \leftarrow \mathcal{A}(\mathcal{H}_k) : \\ S \text{ is a set of } \mathcal{V} \text{ and } M \text{ is a multiset of } \mathcal{V} \\ \text{ and } S \neq M \text{ and } \mathcal{H}_k(S) \equiv_{\mathcal{H}_k} \mathcal{H}_k(M) \end{array} \right\} < n^{-c}.$$

This concludes the proof of Theorem 5.2.2.

□

Remark. Theorem 5.2.2 not only holds when H_k is random function, but also holds when H_k is pseudorandom function. Suppose that there exists an adversary, $\mathcal{A}(\mathcal{H}_k)$, that can compute a collision of the multiset hash function $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$ with a significant probability of success, τ_p, in the case when H_k is a pseudorandom function. Let $\tau_r = u(n)(\frac{t(n)^2}{2^m} + \frac{1}{2^m})$ be the maximum probability with which an adversary can compute a collision of the multiset hash function when H_k is a random function. Let H_k be chosen at random from either a random function family or a pseudorandom function family. We build an adversary, $\mathcal{B}(H_k)$,

that can distinguish whether H_k is a random function or a pseudorandom function with significant probability, contradicting the assumption that H_k is pseudorandom. Let τ_D be the probability that $\mathcal{B}(H_k)$ distinguishes whether H_k is a random function or a pseudorandom function. $\mathcal{B}(H_k)$ runs $\mathcal{A}(\mathcal{H}_k)$, using H_k to create the multiset hashes to respond to $\mathcal{A}(\mathcal{H}_k)$'s multiset hash requests. If $\mathcal{A}(\mathcal{H}_k)$ successfully produces a collision in the multiset hash function, $\mathcal{B}(H_k)$ says that H_k is pseudorandom; otherwise, $\mathcal{B}(H_k)$ says that H_k is random. The probability that $\mathcal{B}(H_k)$ says that H_k is pseudorandom when H_k is pseudorandom is the probability that $\mathcal{A}(\mathcal{H}_k)$ successfully produces a collision in the multiset hash function when H_k is pseudorandom: τ_p. The probability that $\mathcal{B}(H_k)$ says that H_k is pseudorandom when H_k is random is the probability that $\mathcal{A}(\mathcal{H}_k)$ successfully produces a collision in the multiset hash function when H_k is random: τ_r. Thus, $\tau_D = \tau_p - \tau_r$. Because, as we have proven, τ_r is negligible, τ_D is a significant probability. Thus, $\mathcal{B}(H_k)$ is able to distinguish whether H_k is a random function or a pseudorandom function with significant probability. This contradicts the assumption that H_k is pseudorandom. Thus, theorem 5.2.2 also holds when H_k is a pseudorandom function: in particular, the probability of an adversary finding collisions in $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$ when H_k is a pseudorandom function is $\leq u(n)(\frac{t(n)^2}{2^m} + \frac{1}{2^m}) + \tau_D$. (We refer the reader to [23] for a detailed proof of a similar result.)

5.5.2 Proof of Set-Multiset-Collision Resistance of Variants of `MSet-XOR-Hash`

In this section, we show how the proof of the set-multiset-collision resistance of `MSet-XOR-Hash` in Section 5.5.1 is revised to prove the set-multiset-collision resistance of each of the variants in Section 5.2.2.

`Counter-based-MSet-XOR-Hash`

Since the counter never repeats itself (in polynomial time), in terms of the proof of Theorem 5.5.1, Prob(`Distinct`) = 1. Equation 5.1 is argued as before. In terms of Theorem 5.5.1, the bound is $(g/2)^m$.

Theorem 5.5.1 and Theorem 5.5.7 can then be combined as before to prove Theorem 5.2.2, for which the probability that $\mathcal{A}(\mathcal{H}_k)$ finds an (S, M), $S \neq M$, such that

$\bigoplus_{v\in\mathcal{V}} S_v H_k(1,v) = \bigoplus_{v\in\mathcal{V}} M_v H_k(1,v)$ and $\sum_{v\in\mathcal{V}} S_v = \sum_{v\in\mathcal{V}} M_v \mod 2^m$ is at most:

$$u(n)\left(\frac{1}{2^m}\right).$$

`Obscured-MSet-XOR-Hash`

In terms of the proof of Theorem 5.5.1, there is no `Distinct` event. When the output hashes are completely hidden from the adversary, the adversary learns no tuples $[M_i\ ;\ \mathcal{H}_k(M_i)]$. (The adversary is, in essence, restricted to searching for a collision among at most a polynomial number $u(n)$ of pairs (S, M), $S \neq M$, where S is a set of elements of $\mathcal{V}$ and M is a multiset of elements of $\mathcal{V}$.)

Lemma 5.5.6 and its proof are easily revised to be as follows:

Lemma 5.5.6 *Let M and M' be different multisets of elements of $\mathcal{V}$. Let g be the greatest common divisor of 2 and each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$. ($g = 1$ or $g = 2$.) The probability that $(e(M) - e(M'))H_k = 0 \mod 2$ is equal to $(g/2)^m$.*

Proof. By Lemma 5.5.5, $(e(M) - e(M'))H_k \mod 2$ is uniformly distributed in $g\mathbb{Z}_2^m$. Thus, the probability that $(e(M) - e(M'))H_k = 0 \mod 2$ is equal to one divided by the cardinality of $g\mathbb{Z}_2^m$. When $g|d$, $|g\mathbb{Z}_d| = \frac{d}{g}$. Thus, $\text{Prob}((e(M) - e(M'))H_k = 0 \mod 2) = (\frac{g}{d})^m = (\frac{g}{2})^m$. □

Lemma 5.5.6 proves that $\text{Prob}(\texttt{Succ}) = (g/2)^m$; this is the bound for Theorem 5.5.1.

Theorem 5.5.1 and Theorem 5.5.7 can then be combined as before to prove Theorem 5.2.2, for which the adversary will have the same probability of finding a collision as that of `Counter-based-MSet-XOR-Hash`.

`Encrypted-MSet-XOR-Hash`

Let $\mathcal{P}$ be the family of random permutations represented as matrices with m rows, m columns, and entries in $\mathbb{Z}_2$. Let $P_{k'}$ be a randomly-chosen matrix in $\mathcal{P} = \{P_1, P_2, P_3, \ldots, P_{2^{m^2}}\}$. We assume that this matrix is secret (i.e., is only accessible by the checker). The family of matrices $\mathcal{P}$ from which $P_{k'}$ is selected is publicly known.

In terms of the proof of Theorem 5.5.1, there is no `Distinct` event. Lemma 5.5.3 and Lemma 5.5.4 and their proofs are revised as follows:

Lemma 5.5.3 (i) Knowing $[M \ ; \ P_{k'}(\bigoplus_{v\in\mathcal{V}} M_v H_k(v))]$ is equivalent to knowing

$$[e(M) \ ; \ e(M)H_kP_{k'} \mod 2].$$

(ii) $\mathcal{H}_k(M) \equiv_{\mathcal{H}_k} \mathcal{H}_k(M')$ if and only if $e(M)H_kP_{k'} = e(M')H_kP_{k'} \mod 2$ and $\sum_{v\in\mathcal{V}} M_v = \sum_{v\in\mathcal{V}} M'_v \mod 2^m$.

Proof. Notice that

$$\mathcal{H}_k(M) = \left[e(M)H_kP_{k'} \mod 2 \ ; \ \sum_{v\in\mathcal{V}} M_v \mod 2^m\right].$$

The lemma follows immediately from this observation. □

Suppose that an adversary learns t tuples $[M_i \ ; \ \mathcal{H}_k(M_i)]$ or, according to Lemma 5.5.3(i), t vectors $e(M_i)$ together with the corresponding $e(M_i)H_kP_{k'} \bmod 2$. Let E be the $t \times 2^l$ matrix with rows $e(M_i)$.

Lemma 5.5.4 Let M and M' be different multisets of elements of $\mathcal{V}$. The probability that H_k satisfies $e(M)H_k = e(M')H_k \mod 2$ is statistically independent of the knowledge of matrix E and the knowledge $\mathtt{K}= EH_kP_{k'} \mod 2$.

Proof. Observe that $\mathtt{K} = EH_kP_{k'} \mod 2$ means that $\mathtt{K}P_{k'}^{-1} = EH_k \mod 2$.

The proof is as before, with equation (5.2) being

$$\mathtt{K}P_{k'}^{-1} = H_{k_0} + E_1H_{k_1} \mod 2.$$

Thus, the probability that H_k satisfies $e(M)H_k = e(M')H_k \mod 2$ is statistically independent of the knowledge of matrix E and the knowledge $\mathtt{K}= EH_kP_{k'} \mod 2$. □

The rest of the proof of Theorem 5.5.1 is similar to that of the original proof in Section 5.5.1, with the bound on the probability in Theorem 5.5.1 being at most $(g/2)^m$.

Theorem 5.5.1 and Theorem 5.5.7 can then be combined as before to prove Theorem 5.2.2, for which the adversary will have the same probability of finding a collision as that of `Counter-based-MSet-XOR-Hash`.

This result also holds for a pseudorandom family of permutations $P_{k'}$ (cf. the remark at the end of the proof of Theorem 5.2.2 in Section 5.5.1).

5.5.3 `Set-XOR-Hash`

In Definition 5.1.2, a multiset hash function is set-multiset-collision resistant if it is computationally infeasible to find a set S of $\mathcal{V}$ and a multiset M of $\mathcal{V}$ such that $S \neq M$ and $\mathcal{H}(S) \equiv_{\mathcal{H}} \mathcal{H}(M)$. In this section, we give a definition for set-collision resistance: a multiset hash function is set-collision resistant if it is computationally infeasible to find sets S and S' of $\mathcal{V}$ such that $S \neq S'$ and $\mathcal{H}(S) \equiv_{\mathcal{H}} \mathcal{H}(S')$. Let `Set-XOR-Hash` be `MSet-XOR-Hash` (cf. Section 5.2) without the cardinality counter. We note that `Set-XOR-Hash` is set-collision resistant.

Definition 5.5.1. Let H_k be a pseudorandom function keyed with a seed (key) k. $H_k : \{0,1\}^l \rightarrow \{0,1\}^m$. (In practice, one would use the HMAC method [12] to construct such an H_k). Let[3] $r \xleftarrow{R} \{0,1\}^m$ denote uniform random selection of r from $\{0,1\}^m$. Then define `Set-XOR-Hash` by:

$$\mathcal{H}_k(S) = \left(\left(H_k(0,r) \oplus \bigoplus_{v \in \mathcal{V}} S_v H_k(1,v) \right) \ ; \ r \right) \text{ where } r \xleftarrow{R} \{0,1\}^m$$

$$(h,r) \equiv_{\mathcal{H}_k} (h',r') = \Big(h \oplus H_k(0,r) = h' \oplus H_k(0,r') \Big)$$

$$(h,c,r) +_{\mathcal{H}_k} (h',c',r') =$$

$$\Big(H_k(0,r'') \oplus (h \oplus H_k(0,r)) \oplus (h' \oplus H_k(0,r')) \ ; \ r'' \Big) \text{ where } r'' \xleftarrow{R} \{0,1\}^m$$

In terms of the proof of Theorem 5.2.2, Theorem 5.5.1 is proven as before, but now Theorem 5.5.7 is not necessary. Since S and S' are sets, $\forall v \in \mathcal{V}, S_v \leq 1$, and $S'_v \leq 1$. Because

[3] In a real implementation, the two parts of the hash need not be assigned the same number of bits. However the size of each part is a security parameter.

$S \neq S'$, there exists a v such that $S_v \neq S'_v$. Thus g, the greatest common divisor of 2 and each of the differences $|S_v - S'_v|$, $v \in \mathcal{V}$, is equal to 1. Thus, by Theorem 5.5.1, the probability that $\mathcal{A}(\mathcal{H}_k)$ finds an (S, S'), $S \neq S'$, such that $\bigoplus_{v \in \mathcal{V}} S_v H_k(1, v) = \bigoplus_{v \in \mathcal{V}} S'_v H_k(1, v)$ (i.e., $\mathcal{H}_k(S) \equiv_{\mathcal{H}_k} \mathcal{H}_k(S')$) is at most:

$$u(n)\left(\frac{t(n)^2}{2^m} + \frac{1}{2^m}\right) .$$

`Set-XOR-Hash` can be modified similar to the manner in which `MSet-XOR-Hash` was modified to create `Counter-based-Set-XOR-Hash`, `Obscured-Set-XOR-Hash` and `Encrypted-Set-XOR-Hash` (cf. Section 5.2.2). By Theorem 5.5.1, the probability that $\mathcal{A}(\mathcal{H}_k)$ finds an (S, S'), $S \neq S'$, such that $\mathcal{H}_k(S) \equiv_{\mathcal{H}_k} \mathcal{H}_k(S')$ is at most:

$$u(n)\left(\frac{1}{2^m}\right).$$

5.6 `MSet-Add-Hash` Proofs

5.6.1 Proof of Multiset-Collision Resistance of `MSet-Add-Hash`

In this section, we prove Theorem 5.3.2 (cf. Section 5.3). We will first prove Theorem 5.6.1, then use this theorem to formulate a proof for Theorem 5.3.2.

Recall that $H_k : \{0,1\}^l \rightarrow \{0,1\}^m$. (For accuracy, $H_k(0, r)$ is actually $H_k(0^{l-m}, r)$, where $m+1 \leq l$.) We first model H_k as a random function and prove that, if H_k is a random function, then `MSet-Add-Hash` is multiset-collision resistant. In practice, H_k is a pseudorandom function; after proving the theorems modelling H_k as a random function, we then show how to extend them when H_k is a pseudorandom function.

Let $\mathcal{R}$ be the family of random functions represented as matrices with 2^l rows, one column, and entries in $\mathbb{Z}_{2^m}$. Let H_k be a randomly-chosen matrix in $\mathcal{R} = \{H_1, H_2, H_3, \ldots, H_{2^{m2^l}}\}$. We assume that this matrix is secret (i.e., is only accessible by the checker). The family of matrices $\mathcal{R}$ from which H_k is selected is publicly known.

The rows of H_k are labelled by $x \in \{0,1\}^l$ and denoted by $H_k(x)$. The matrix represents H_k as a random function from $x \in \{0,1\}^l$ to $\mathbb{Z}_{2^m}$, the set of vectors with length one and entries in $\mathbb{Z}_{2^m}$. We note that, because H_k is a random function, each row in H_k is uniformly distributed in $\mathbb{Z}_{2^m}$.

Theorem 5.6.1 is about the probability that an adversary, given oracle access to $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$, finds two different multisets, M and M', such that $\sum_{v \in \mathcal{V}} M_v H_k(1,v) = \sum_{v \in \mathcal{V}} M'_v H_k(1,v) \bmod 2^m$. The probability is taken over random matrices H_k in $\mathcal{R}$, the randomness of the random nonce used in $\mathcal{H}_k$ and the randomness used in the probabilistic polynomial time adversary.

Theorem 5.6.1. *Suppose that an adversary, given oracle access to $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$, gains knowledge of t tuples $[M_i \ ; \ \mathcal{H}_k(M_i)]$. Let M and M' be different multisets of elements of $\mathcal{V}$. Let g be the greatest common divisor of 2^m (because the addition is being computed modulo 2^m) and each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$. (Since $M, M' \neq \emptyset$, $g \in \{1,2,4,8,\ldots,2^m\}$.) Given knowledge of t tuples $[M_i \ ; \ \mathcal{H}_k(M_i)]$, the probability that the adversary finds an M and an M', $M \neq M'$, such that $\sum_{v \in \mathcal{V}} M_v H_k(1,v) = \sum_{v \in \mathcal{V}} M'_v H_k(1,v)$ mod 2^m is at most $t^2/2^m + g/2^m$.*

Proof. The proof is analogous to the proof of Theorem 5.5.1 (cf. Section 5.5.1). □

With Theorem 5.6.1, we are now ready to prove Theorem 5.3.2.

Proof. Let $\mathcal{A}(\mathcal{H}_k)$ denote a probabilistic polynomial time, in n, algorithm with oracle access to $(\mathcal{H}_k, \equiv_{\mathcal{H}_k}, +_{\mathcal{H}_k})$. Then $\mathcal{A}(\mathcal{H}_k)$ can gain knowledge of at most a polynomial number $t(n)$ tuples $[M_i \ ; \ \mathcal{H}_k(M_i)]$ (here $t(n)$ denotes a polynomial in n). Furthermore, $\mathcal{A}(\mathcal{H}_k)$ can search for a collision among at most a polynomial number $u(n)$ of pairs (S, M), $S \neq M$, where S is a set of elements of $\mathcal{V}$ and M is a multiset of elements of $\mathcal{V}$.

According to Theorem 5.6.1, the probability that $\mathcal{A}(\mathcal{H}_k)$ finds an (M, M'), $M \neq M'$, such that $\sum_{v \in \mathcal{V}} M_v H_k(1,v) = \sum_{v \in \mathcal{V}} M'_v H_k(1,v) \bmod 2^m$ is at most $u(n)(2t^2/2^l + g/2^m)$.

Let us consider one of these pairs (M, M'), $M \neq M'$ where
$\sum_{v \in \mathcal{V}} M_v H_k(1,v) = \sum_{v \in \mathcal{V}} M'_v H_k(1,v) \bmod 2^m$. Since, for our trace-hash application, for

a collision to be useful to the adversary, the multisets in the collision must each be of polynomial size in $n = 2m$, each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$ is polynomial sized in $n = 2m$. Since g divides each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$, g is polynomial sized in $n = 2m$. Hence, there exists a number $\gamma > 0$ such that $g \leq m^\gamma$ for m large enough. Again, because each of the differences $|M_v - M'_v|$, $v \in \mathcal{V}$ is polynomial sized in $n = 2m$, there exits a number $\varphi > 0$ such that $m^\varphi \geq$ each of the differences $|M_v - M'_v|, v \in \mathcal{V}$. Because $g \leq$ each of the differences $|M_v - M'_v|, v \in \mathcal{V}$, $m^\gamma \leq m^\varphi$. Let $\varsigma > 0$ be any number. Then

$$\frac{g}{2^m} \leq \frac{m^\gamma}{2^m} \leq \frac{m^\varphi}{2^m} \leq m^{-\varsigma} \text{ for } m \text{ large enough.}$$

Thus, by Theorem 5.6.1, the probability that $\mathcal{A}(\mathcal{H}_k)$ finds a collision (M, M'), $M \neq M'$, such that $\sum_{v \in \mathcal{V}} M_v H_k(1, v) = \sum_{v \in \mathcal{V}} M'_v H_k(1, v) \bmod 2^m$ is at most:

$$u(n)\left(\frac{t(n)^2}{2^m} + \frac{m^\varphi}{2^m}\right)$$

where $m^\varphi \geq$ each of the differences $|M_v - M'_v|, v \in \mathcal{V}$.

Thus, for any number c, for sufficiently-large m, the probability that $\mathcal{A}(\mathcal{H}_k)$ finds a collision (M, M'), $M \neq M'$, such that $\mathcal{H}_k(M) \equiv_{\mathcal{H}_k} \mathcal{H}_k(M)$ is $\leq m^{-c}$. $n = 2m$. Thus, for all ppt algorithms $\mathcal{A}$, any number c,

$$\exists n_0 : \forall n \geq n_0, \text{Prob}\left\{\begin{array}{l} k \xleftarrow{R} \{0,1\}^n, (M, M') \leftarrow \mathcal{A}(\mathcal{H}_k) : \\ M \text{ and } M' \text{ are a multisets of } \mathcal{V} \\ \text{ and } M \neq M' \text{ and } \mathcal{H}_k(M) \equiv_{\mathcal{H}_k} \mathcal{H}_k(M') \end{array}\right\} < n^{-c}.$$

This concludes the proof of Theorem 5.3.2.

□

Remark. Theorem 5.3.2 not only holds when H_k is random function, but also holds when H_k is pseudorandom function. We refer the reader to the remark at the end of Section 5.5.1 for a similar remark.

5.6.2 Proof of Multiset-Collision Resistance of Variants of `MSet-Add-Hash`

To prove the multiset-collision resistance of `Counter-based-MSet-Add-Hash`, `Obscured-MSet-Add-Hash`, and `Encrypted-MSet-Add-Hash`, the proof of `MSet-Add-Hash` in Section 5.6.1 is revised in a similar manner as the proof of `MSet-XOR-Hash` was revised to prove the set-multiset-collision resistance of the variants of `MSet-XOR-Hash` (cf. Section 5.5.2). The probability that $\mathcal{A}(\mathcal{H}_k)$ finds a collision (M, M'), $M \neq M'$, such that $\sum_{v \in \mathcal{V}} M_v H_k(1, v) = \sum_{v \in \mathcal{V}} M'_v H_k(1, v) \bmod 2^m$ is at most:

$$u(n)\left(\frac{m^{\varphi}}{2^m}\right) .$$

5.7 `MSet-Mu-Hash` Proofs

5.7.1 Proof of Multiset-Collision Resistance of `MSet-Mu-Hash`

In this section, we prove Theorem 5.4.2 (cf. Section 5.4). We will first prove Theorem 5.7.1, then use this theorem to formulate a proof for Theorem 5.4.2.

Recall that $H : \{0,1\}^l \rightarrow \{0,1\}^m$. We first model H as a random function and prove that, if H is a random function, then `MSet-Mu-Hash` is multiset-collision resistant. In practice, H is a pseudorandom function; after proving the theorems modelling H as a random function (oracle), we then give references that discuss into what extent the random oracle model can be met in practice.

Let $\mathcal{R}$ be the family of random functions with domain equal to $\{0,1\}^l$ and range $\subseteq \mathbb{Z}_p^*$, where p is an m-bit prime. Let H be a randomly-chosen function in $\mathcal{R} =$ $\{H_1, H_2, H_3, \ldots, H_{2^{m2^l}}\}$. We assume that $H : \{0,1\}^l \rightarrow \mathbb{Z}_p^*$ is public. The family of matrices $\mathcal{R}$ from which H is selected is also publicly known. Let $\rho = |\mathbb{Z}_p^*|$; thus $\rho = p - 1$.

Theorem 5.7.1 is about the probability that an adversary, given a particular algorithm, $\mathcal{B}(.)$, and given a random $y \in \mathbb{Z}_p^*$, finds the discrete log of y in $\mathbb{Z}_p^*$. The probability is taken

over a random choice of the inputs to $\mathcal{B}(.)$, a random choice of $y \in \mathbb{Z}_p^*$ and the randomness used in the probabilistic polynomial time adversary. $\mathcal{B}(.)$ is a probabilistic polynomial time (in m) algorithm that, with probability at least σ, outputs weights $w_1, w_2, \ldots, w_u \in \mathbb{Z}_\rho$ for a polynomial number of random inputs $z_1, z_2, \ldots, z_u \in \mathbb{Z}_p^*$ such that $1 = \prod_i z_i^{w_i} \mod p$. We show that if such an algorithm exists, then we can construct an algorithm, $\mathcal{B}'(.)$ that, given a random $y \in \mathbb{Z}_p^*$, can find the discrete log of y in $\mathbb{Z}_p^*$ with probability at least $\frac{\sigma}{m^e}$.

Theorem 5.7.1. *Let $\mathcal{B}(.)$ be a probabilistic polynomial time (in m) algorithm such that there exists a number e such that for $u \leq m^e$,*

$$Prob\left\{ \begin{array}{l} (z_i \leftarrow \mathbb{Z}_p^*)_{i=1}^u, (w_i \in \mathbb{Z}_\rho)_{i=1}^u \leftarrow \mathcal{B}(z_1, z_2, \ldots, z_u) : \\ 1 = \prod_i z_i^{w_i} \mod p, \quad \exists_i w_i \neq 0, \quad \forall_i |w_i| \leq m^e \end{array} \right\} \geq \sigma \ . \tag{5.6}$$

Let g be a generator of $\mathbb{Z}_p^*$. If algorithm $\mathcal{B}(.)$ exists, then there exists a probabilistic polynomial time (in m) algorithm $\mathcal{B}'(.)$ such that

$$Prob\{y \leftarrow \mathbb{Z}_p^*, \ x \in \mathbb{Z}_\rho \leftarrow \mathcal{B}'(y, p, g) \ : \ y = g^x \mod p\} \geq \frac{\sigma}{m^e} \ .$$

Proof. We show how to construct $\mathcal{B}'$. $\mathcal{B}'$ selects a polynomial number u of random elements $r_1, r_2, \ldots, r_u$ in $\mathbb{Z}_\rho$ and a random $j \in \{1, 2, \ldots, u\}$. $\mathcal{B}'$ computes

$$\begin{aligned} z_j &= y g^{r_j} \mod p, \text{ and} \\ z_i &= g^{r_i} \mod p, \text{ for all } i \neq j. \end{aligned}$$

$\mathcal{B}'$ calls $\mathcal{B}$ to compute $(w_1, w_2, \ldots, w_u) \leftarrow \mathcal{B}(z_1, z_2, \ldots, z_u)$. Since, by construction, u is polynomial-sized in m and the z_i's have been chosen uniformly at random from $\mathbb{Z}_p^*$, we know that, with probability at least σ, the weights $w_1, w_2, \ldots, w_u \in \mathbb{Z}_\rho$ are computed such that they are not all equal to zero, $\forall_i |w_i| \leq m^e$, and

$$\begin{aligned} 1 = \prod_i z_i^{w_i} &= y^{w_j} g^{\sum_i r_i w_i} \mod p \\ y^{w_j} &= g^{-\sum_i r_i w_i} \mod p \ . \end{aligned} \tag{5.7}$$

Because the u inputs are in random order,

$$Prob(w_j \neq 0) \geq \frac{1}{u} \geq \frac{1}{m^e} \ .$$

Recall that $\rho = |\mathbb{Z}_p^*|$ i.e., $\rho = p - 1$. Suppose that $w_j \neq 0$. Let $d = \gcd(w_j, \rho)$. Then, by Euclid's algorithm, $\mathcal{B}'$ can compute d and w_j' such that $w_j w_j' = d \mod \rho$.

Because $|w_j| \leq m^e < \rho$, we know that $d \neq 0 \mod \rho$. From (5.7), we infer that

$$y^d = g^{-w_j' \sum_i r_i w_i} \mod p \ .$$

$\mathcal{B}'$ can compute $s = -w_j' \sum_i r_i w_i \mod \rho$. (If $d = 1$, $\mathcal{B}'$ outputs s as the discrete log of y in $\mathbb{Z}_p^*$). $y = g^x$ which means that $y^d = g^{dx} = g^s$, that is $dx = s \mod \rho$. Because d divides ρ, d must also divide s. Thus, the equation $dx = s \mod \rho$ has exactly d solutions between 0 and $\rho - 1$, all of which are congruent mod $\frac{\rho}{d}$. Because $d \leq |w_j| \leq m^e$, there are a polynomial number of solutions. $\mathcal{B}'$ computes $\rho' = \frac{\rho}{d}$ and $s' = \frac{s}{d}$. For each k, $0 \leq k \leq d-1$, $\mathcal{B}'$ computes $x_k = s' + k\rho' \mod \rho$. (The solutions repeat themselves when $k > d - 1$; for example, when $k = d$, $(s' + d\rho') \mod \rho = s' \mod \rho$.) The discrete log, x, is one of the solutions. Thus, for each k, $0 \leq k \leq d - 1$, $\mathcal{B}'$ computes g^{x_k} and checks to see if it is equal to y. When $\mathcal{B}'$ finds which x_k is such that $g^{x_k} = y$, it outputs the x_k as the discrete log of y in $\mathbb{Z}_p^*$.

This concludes the proof of Theorem 5.7.1. □

Recall that p is an m-bit prime, and that $\rho = |\mathbb{Z}_p^*| = p - 1$. Again, let g be a generator of $\mathbb{Z}_p^*$. The discrete log assumption states that, for all ppt, in m, algorithms $\mathcal{D}(.)$, for any number a, for p large enough,

$$Prob\left\{y \leftarrow \mathbb{Z}_p^*,\ x \in \mathbb{Z}_\rho \leftarrow \mathcal{D}(y, p, g) \ : \ y = g^x \mod p\right\} < m^{-a}.$$

We are now ready to prove Theorem 5.4.2.

Proof. We reduce the difficulty of finding collisions in `MSet-Mu-Hash` to the difficulty of finding the discrete log of a random element in $\mathbb{Z}_p^*$. We note that, because $\mathcal{H}$ does not use a secret key, and because, for $v \in \mathcal{V}, \mathcal{H}(v) = H(v)$, oracle access to $(\mathcal{H}, \equiv_{\mathcal{H}}, +_{\mathcal{H}})$ also means

oracle access to H. By $\mathcal{A}(H)$ we denote a probabilistic polynomial time, in $n = m$, algorithm with oracle access to H.

Suppose that $(\mathcal{H}, \equiv_{\mathcal{H}}, +_{\mathcal{H}})$ is not multiset-collision resistant. This means that there exists a ppt algorithm $\mathcal{A}(H)$, and there exists a number e, and there exists a sufficiently-large $n = m$, such that the probability that $\mathcal{A}(H)$ finds a (M, M'), $M \neq M'$, such that $\mathcal{H}(M) \equiv_{\mathcal{H}} \mathcal{H}(M')$ is $\geq m^{-c}$. Since, for our trace-hash application, for a collision to be useful to the adversary, the multisets in the collision must each be of polynomial size in $n = m$, $|M|$ and $|M'| \leq m^e$, for some number e.

$$\mathcal{H}(M) = \prod_{v \in \mathcal{V}} H(v)^{M_v} = \prod_{v \in \mathcal{V}} H(v)^{M'_v} = \mathcal{H}(M') \ .$$

This means that

$$1 = \prod_{v \in \mathcal{V}} H(v)^{M_v - M'_v} \ ,$$

there exists a $v \in \mathcal{V}$ such that $(M_v - M'_v) \neq 0$, and, for all $v \in \mathcal{V}$, $|(M_v - M'_v)| \leq m^e$.

Because we model H as a random function from $\{0,1\}^l$ to $\mathbb{Z}_p^*$, for all $v \in \mathcal{V}$, $H(v)$ cannot be distinguished from a random element in $\mathbb{Z}_p^*$. Because $\mathcal{A}(H)$ runs in probabilistic polynomial time, the number of distinct values $\in \mathcal{V}$ on which $\mathcal{A}(H)$ can compute the function H is $u \leq m^e$. Thus $\mathcal{A}(H)$ is an algorithm satisfying (5.6), with the inputs being the replies to $\mathcal{A}$'s oracle queries to H.

By Theorem 5.7.1, if $\mathcal{A}(H)$ exists, then there exists a ppt, in m, algorithm, $\mathcal{D}$, and there exists a number, a, such that $\mathcal{D}$ can find the discrete log of a random element in $\mathbb{Z}_p^*$ with probability at least $m^{-a} = m^{-(c+e)}$. For a large enough p, this contradicts the discrete log assumption. Thus, for p large enough, $\mathcal{A}(H)$ does not exist, which proves multiset-collision resistance.

□

Recall that p is an m-bit prime. Given a random element in $\mathbb{Z}_p^*$, let ϵ be the probability of finding the discrete log of the element in $\mathbb{Z}_p^*$. For any number a, for p large enough, $\epsilon < m^{-a}$. Because $\mathcal{A}(H)$ runs in polynomial time, it runs in time m^e, for some number e. From Theorem 5.7.1, the probability that $\mathcal{A}(H)$ finds a collision (M, M'), $M \neq M'$, such

that $\mathcal{H}(M) \equiv_{\mathcal{H}} \mathcal{H}(M')$, is at most $m^e\epsilon$.

Remark. Supposing that H is a random function (oracle) is a strong assumption. Compared to the `MSet-XOR-Hash` and `MSet-Add-Hash` we do not need a secret key at all. We refer to [21, 22] for a discussion into what extent the random oracle model can be met in practice.

Remark. In [4], it is proven, under the strong RSA assumption [24], that the RSA group $\mathbb{Z}_N^*$ is pseudo-free [27] when N is the product of two safe primes, i.e., no polynomial time algorithm can find, with non-negligible probability, an unsatisfiable equation over the free abelian group generated by the elements $g_1, g_2, \ldots, g_n$, together with a solution to the equation in $\mathbb{Z}_N^*$, when $g_1, g_2, \ldots, g_n$ are instantiated to randomly-chosen quadratic residues in $\mathbb{Z}_N^*$. Consider the function I that maps elements of $\mathcal{M}$ to randomly-chosen quadratic residues in $\mathbb{Z}_N^*$. Consider the multisets M and M'. We note that the multiset hash function defined by $\Big(\mathcal{H}(M) = \prod_{v \in \mathcal{V}} I(v)^{M_v} \bmod N;\ (h) \equiv_{\mathcal{H}} (h') = (h = h');\ (h) +_{\mathcal{H}} (h') = (h \times h' \bmod N) \Big)$ is multiset-collision resistant because it is infeasible to find a relation of the form $1 = \prod_{v \in \mathcal{V}} I(v)^{M_v - M'_v}$ when $M \neq M'$.

Chapter 6

Bag Integrity Checking

We introduce bag integrity checking as an abstraction for explaining and proving various lemmas on trace-hash integrity checking on dynamically-changing, sparsely-populated, address spaces. The scenario consists of a bag checker, a bag, and an adversary. The bag checker performs two operations on the bag: the checker can put items into the bag and the checker can take items out of the bag. The bag is untrusted storage and under the control of the adversary, who can put items into the bag, take items from the bag, or alter any of the items in the bag.

The bag checker is interested in whether the bag behaves correctly. A bag behaves correctly if it behaves as a valid bag. *Intuitively, a bag is a valid bag if it is a bag which only the checker has manipulated with put and take operations.*

6.1 Definitions

Definition 6.1.1. A bag is a triple $(\mathcal{B}, \textsc{Put}, \textsc{Take})$ consisting of a multiset $\mathcal{B}$ and two operations
$\textsc{Put}, \textsc{Take} \in \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$ defined by

$$\textsc{Put}(\mathcal{B}, M) = \mathcal{B} \cup_{\mathcal{M}} M \tag{6.1}$$

and

$$\mathcal{B} = \textsc{Take}(\mathcal{B}, M) \cup_{\mathcal{M}} M. \tag{6.2}$$

Intuitively, $\textsc{Put}(\mathcal{B}, M)$ puts the multiset M into bag $\mathcal{B}$. $\textsc{Take}(\mathcal{B}, M)$ takes the multiset M from bag $\mathcal{B}$.

Definition 6.1.2. A history is a finite sequence of put/take operations $O_1(., M_1)$, $O_2(., M_2)$, $\ldots$, $O_n(., M_n)$.

Definition 6.1.3. A bag's history is valid if and only if the bag's multiset, $\mathcal{B}_n$, follows from a sequence of well defined put/take operations starting from the empty set. That is, $\mathcal{B}_n$ results from the recurrence relation $\mathcal{B}_i = O_i(\mathcal{B}_{i-1}, M_i)$ with $\mathcal{B}_0 = \emptyset$ where the put/take operations O_i satisfy the previous definitions (6.1) and (6.2).

Definition 6.1.4. A bag is a valid bag if and only if the bag's history is valid.

Intuitively, $\mathcal{B}_i$ is the state of the bag at moment i. Also intuitively, a bag has a valid history if the checker can take an item from the bag only if the item has previously been put in the bag by the checker and has not yet been taken from the bag by the checker. This leads to the following equivalent definition of a valid history.

Given a history of put/take operations $O_1(., M_1)$, $O_2(., M_2)$, $\ldots$, $O_n(., M_n)$, we define the put multisets P_i and the take multisets T_i by the recurrence relations

$$P_i = \begin{cases} \emptyset & \text{if } i = 0, \\ P_{i-1} \cup_{\mathcal{M}} M_i & \text{if } i > 0 \text{ and } O_i = \textsc{Put}, \\ P_{i-1} & \text{if } i > 0 \text{ and } O_i = \textsc{Take}, \end{cases} \tag{6.3}$$

and

$$T_i = \begin{cases} \emptyset & \text{if } i = 0, \\ T_{i-1} \cup_{\mathcal{M}} M_i & \text{if } i > 0 \text{ and } O_i = \textsc{Take}, \\ T_{i-1} & \text{if } i > 0 \text{ and } O_i = \textsc{Put}. \end{cases} \tag{6.4}$$

Definition 6.1.5. The history is valid if and only if for all $0 \leq i \leq n$, $T_i \subseteq_{\mathcal{M}} P_i$.

We prove the equivalence of Definition 6.1.3 and Definition 6.1.5 in Section 6.3. It turns out that the difference in P_i and T_i is the bag $\mathcal{B}_i$, that is for all $0 \leq i \leq n$

$$T_i \cup_{\mathcal{M}} \mathcal{B}_i = P_i. \tag{6.5}$$

6.2 Bag Checker

Our solution to checking the integrity of an untrusted bag is described in Figure 6-1. The bag checker maintains multiset hashes and a counter. The checker performs two operations on the bag:

- `cbag-put`: the checker puts an item into the bag
- `cbag-take`: the checker takes an item out of the bag.

The essence of this bag checker is that a "trace" of the sequence of its operations on the bag is maintained in its fixed-sized trusted state. (The checker cannot maintain an actual physical trace because it only has a small fixed-sized trusted state.)

In the bag, each item is accompanied by a time stamp. Each time the checker performs a `cbag-put` operation, it appends the current value of the counter (a time stamp) to each item, and puts the (item, time stamp) pairs into the bag. When the checker performs a `cbag-take` operation, it takes pairs from the bag, and, if necessary, updates the counter so that it is strictly greater than each of the time stamps in the pairs that were taken from the bag. The multiset hashes are updated ($+_{\mathcal{H}}$) with the (item, time stamp) pairs that were put or taken from the bag.

The FSM uses the bag checker as an interface to the bag. The FSM calls the bag checker's `cbag-put` and `cbag-take` interface, and the checker performs the put and take operations for the FSM as described in Figure 6-1. The checker records its operations on the bag, i.e., the bag's history, in PutHash and TakeHash. When the FSM wants to check the integrity of the bag, it tells the checker to take all of the items out of the bag, by calling `cbag-take`(true). At this point, the FSM calls the checker's `cbag-check` operation. The `cbag-check` operation returns true if the checker's PutHash is equal to its TakeHash. If

The checker's fixed-sized state is:

- 2 multiset hashes: PUTHASH and TAKEHASH. Initially both multiset hashes are $\mathcal{H}(\emptyset)$.
- 1 counter: TIMER. Initially TIMER is 0.

`cbag-put`($\mathcal{I}_{ms}$) puts a multiset of items, $\mathcal{I}_{ms}$, into the untrusted bag:

1. Let t be the current value of TIMER. For each item $\in \mathcal{I}_{ms}$, put the pair, (item, t), into the untrusted bag.
2. For each item $\in \mathcal{I}_{ms}$, update PUTHASH: PUTHASH $+_{\mathcal{H}}$ hash(item, t).

`cbag-take`($\mathcal{P}$) takes the multiset of items that match the predicate $\mathcal{P}$ from the untrusted bag:

1. Take all pairs whose items match $\mathcal{P}$ from the untrusted bag. Denote this multiset of pairs as $\mathcal{Q}_{ms}$. ($\mathcal{Q}_{ms} = \{$(item, t): $\mathcal{P}$(item) = `true`$\}$).
2. For each pair (item, t) $\in \mathcal{Q}_{ms}$, update TAKEHASH: TAKEHASH $+_{\mathcal{H}}$ hash(item, t).
3. For each pair (item, t) $\in \mathcal{Q}_{ms}$, update TIMER: TIMER = max(TIMER, $t + 1$).
4. Return $\{$item: (item, t) $\in \mathcal{Q}_{ms}\}$.

`cbag-check`() returns `true` if, (i) the untrusted bag has behaved correctly (as a valid bag) and, (ii) the untrusted bag is empty, according to the untrusted bag's history of accesses:

1. If PUTHASH $\equiv_{\mathcal{H}}$ TAKEHASH is `false`, then return `false`.
2. Reset TIMER to zero (this is an optimization).
3. Return `true`.

Figure 6-1: Bag checker

the check operation returns true, the FSM concludes that the bag is empty (according to the checker's operations on the bag, i.e., according to the bag's history) and that the bag has behaved correctly (i.e., the bag's history is valid). In other words, if the check operation returns true, the FSM concludes that the bag is empty and that the bag has behaved as if only the checker has manipulated it with put and take operations.

The PutHash records information on the items that, according to the checker, should be in the bag at any given point in time. The TakeHash records information on the items that the checker takes from the bag.

Because the checker checks that PutHash is equal to TakeHash, insertion/substitution (the checker takes an item from the bag that the checker has never put into it) and replay (the checker takes a stale item from the bag; a stale item is one which the checker has put into the bag, and has already taken from the bag) attacks on the bag are prevented.

The purpose of the time stamps is to prevent reordering attacks in which the checker takes an item from the bag that the checker has not yet put into the bag, but will put in the bag at a later time. Suppose we consider the `cbag-put` and `cbag-take` operations that occur as occurring on a timeline. Line 3 in the `cbag-take` operation ensures that, with regard to each `cbag-put` operation, each item that the checker puts into the bag has a time stamp that is strictly greater than all of the time stamps of all of the pairs that the checker had previously taken from the bag. Therefore, the first time an adversary tampers with a particular (item, time stamp) pair that the checker takes from the bag, there will not be an entry in the PutHash matching the adversary's entry in the TakeHash, and that entry will not be added to the PutHash at a later time. Thus, PutHash will not be equal to TakeHash when a `cbag-check` operation is performed.

The Timer is not solely under the control of the checker, and is a function of what is taken from the bag, which is untrusted. Therefore, the PutHash cannot be guaranteed to be over a set. For example, for a sequence of `cbag-put` and `cbag-take` operations, an adversary can decrease the time stamp in a pair that is taken from the bag and have pairs be added to the PutHash multiple times. The TakeHash can also not be guaranteed to be over a set because the adversary controls the pairs that are taken from the bag. Thus, neither set-collision resistance nor set-multiset-collision is not sufficient, and we require multiset-collision

resistant hash functions.

In Section 6.4, we prove the following theorem to show that our solution to bag integrity checking is secure:

Theorem 6.2.1. *Let P_n be the multiset of pairs that the checker has put into the bag. Let T_n be the multiset of pairs that the checker has taken from the bag. That is, P_n hashes to* PutHash *and T_n hashes to* TakeHash*. Then, $P_n = T_n$ if and only if the bag is empty and the bag's history O_1, O_2, ..., O_n is valid.*

The following corollary shows the hardness of breaking our bag integrity checking scheme.

Corollary 6.2.2. *Tampering with the bag to make its history invalid without the bag checker detecting the tampering is as hard as finding a collision $P_n \neq T_n$ for the multiset hash function.*

If the bag's history is valid, the FSM knows that whenever it took an item from the bag, it was from among the FSM's items currently in the bag. Note that if an adversary takes items out of the bag such that `cbag-take`($\mathcal{P}$) only returns some of the FSM's items that match predicate $\mathcal{P}$, the bag's history is still valid. (The adversary would have to put the items back into the bag at a later time for `cbag-take`(true) to empty the bag for `cbag-check` to pass.) This is not a problem in our bag checker abstraction because this tampering will be detected by our trace-hash integrity checker (cf. Chapter 7). In the trace-hash checker, the items are (address, value) pairs. The trace-hash checker uses a predicate, P_a, that returns true on a pair (a', v) for which $a = a'$. The trace-hash checker ensures that each address is present in the bag exactly once, and checks that `cbag-take`(P_a) always returns a singleton set.

6.3 Proof of the Equivalence of the Bag Definitions

In this section, we prove the equivalence of Definition 6.1.3 and Definition 6.1.5 (cf. Section 6.1).

Define $\mathcal{B}_i$ as:

$$\mathcal{B}_i = O_i(\mathcal{B}_{i-1}, M_i) \tag{6.6}$$

Define $\mathcal{B}'_i$ as:

$$T_i \cup_{\mathcal{M}} \mathcal{B}'_i = P_i \tag{6.7}$$

We prove, by induction, that, for all $0 \leq i \leq n$, $Q(i)$ is true where

$$Q(i) ::= \ \mathcal{B}_i = \mathcal{B}'_i.$$

That is, (6.6) defines a unique bag, $\mathcal{B}_i$. (6.7) defines a unique bag, $\mathcal{B}'_i$. We prove that, for all $0 \leq i \leq n$, $\mathcal{B}_i \ = \ \mathcal{B}'_i$; i.e., for all $0 \leq i \leq n$, the two bags are the same.

Proof. **Base case:** Prove $Q(0)$ is true. $\mathcal{B}_0 = \emptyset$. Also, $P_0 = T_0 = \emptyset$; thus, $\mathcal{B}'_0 = \emptyset$. Therefore, when $i = 0$, $\mathcal{B}_0 = \mathcal{B}'_0$.

Inductive Step: Assume $Q(i)$ is true, prove $Q(i+1)$ is true. That is, assume $\mathcal{B}_i = \mathcal{B}'_i$, prove $\mathcal{B}_{i+1} = \mathcal{B}'_{i+1}$.

Case 1: PUT operation:

$$\begin{aligned}
& \mathcal{B}_{i+1} = \textsc{Put}(\mathcal{B}_i, M_{i+1}) \\
\xleftarrow{\texttt{by (6.1)}} \quad & \mathcal{B}_{i+1} = \mathcal{B}_i \cup_{\mathcal{M}} M_{i+1} \\
\xleftarrow{\texttt{by inductive hypothesis}} \quad & \mathcal{B}_{i+1} = \mathcal{B}'_i \cup_{\mathcal{M}} M_{i+1} \\
\xleftarrow{\texttt{by (6.7)}} \quad & \mathcal{B}_{i+1} \cup_{\mathcal{M}} T_i = P_i \cup_{\mathcal{M}} M_{i+1} \\
\xleftarrow{\texttt{by (6.3) and (6.4)}} \quad & \mathcal{B}_{i+1} \cup_{\mathcal{M}} T_{i+1} = P_{i+1} \\
\xleftarrow{\texttt{by (6.7)}} \quad & \mathcal{B}_{i+1} = \mathcal{B}'_{i+1}.
\end{aligned}$$

Case 2: TAKE operation:

$$\begin{aligned}
& \mathcal{B}_{i+1} = \textsc{Take}(\mathcal{B}_i, M_{i+1}) \\
\xleftarrow{\texttt{by (6.2)}} \quad & \mathcal{B}_{i+1} \cup_{\mathcal{M}} M_{i+1} = \mathcal{B}_i \\
\xleftarrow{\texttt{by inductive hypothesis}} \quad & \mathcal{B}_{i+1} \cup_{\mathcal{M}} M_{i+1} = \mathcal{B}'_i \\
\xleftarrow{\texttt{by (6.7)}} \quad & \mathcal{B}_{i+1} \cup_{\mathcal{M}} M_{i+1} \cup_{\mathcal{M}} T_i = P_i \\
\xleftarrow{\texttt{by (6.3) and (6.4)}} \quad & \mathcal{B}_{i+1} \cup_{\mathcal{M}} T_{i+1} = P_{i+1} \\
\xleftarrow{\texttt{by (6.7)}} \quad & \mathcal{B}_{i+1} = \mathcal{B}'_{i+1}.
\end{aligned}$$

□

Also, as, for all $0 \leq i \leq n,\ \mathcal{B}_i = O_i(\mathcal{B}_{i-1}, M_i) \iff T_i \cup_{\mathcal{M}} \mathcal{B}_i = P_i$, then,
for all $0 \leq i \leq n,\ \mathcal{B}_i = O_i(\mathcal{B}_{i-1}, M_i) \iff T_i \cup_{\mathcal{M}} \mathcal{B}_i = P_i \iff T_i \subseteq_{\mathcal{MS}} P_i$.

6.4 Proof of Security of Bag Checker

In this section, we prove Theorem 6.2.1 (cf. Section 6.1).

Consider the `cbag-put` and `cbag-take` operations that occur on an address as occurring on a time line. To avoid confusion with the values of TIMER, we express this time line in terms of processor cycles.

Let t be the current value of TIMER. t is the value of the TIMER when pairs are put into the bag. When the checker `cbag-puts` a multiset of items $\mathcal{I}_{ms}$ into the bag on cycle i, $O_i = \textsc{Put}$ and

$$M_i = \{(v, t) : v \in \mathcal{I}_{ms}\}. \tag{6.8}$$

When the checker `cbag-takes` a multiset, M_j, from the bag on cycle j, $O_j = \textsc{Take}$ and

$$\mathcal{I}_{ms} = \{v' : (v', t') \in M_j\}. \tag{6.9}$$

If necessary, the TIMER is increased at the end of a `cbag-take` operation.

Proof. We show that, if $P_n = T_n$, then the history O_1, O_2, ..., O_n is valid. We use Defini-

tion 6.1.5 of a valid history, and give a direct proof. Thus, we show that, if $P_n = T_n$, then, for all $0 \leq i \leq n$, $T_i \subseteq_{\mathcal{MS}} P_i$.

Let $(v', t') \in T_i$. By (6.4), $(v', t') \in M_j$ with $O_j = \textsc{Take}$ for some $j \leq i$. Because $T_n = P_n$, $(v', t') \in P_n$. By (6.3), $(v', t') \in M_k$ with $O_k = \textsc{Put}$ for some $k \leq n$. Because of line 3 in the `cbag-take` operation, all items that are `cbag-put` into the bag after cycle j have a time stamp strictly greater than t'. Therefore, the checker cannot put (v', t') into the bag after cycle j. Therefore, we have that $k \leq j$. Therefore we have $k \leq j \leq i$. We conclude that $(v', t') \in M_k \subseteq P_k \subseteq P_i$. This proves $T_i \subseteq P_i$ as required.

We show that, if $P_n = T_n$, then the bag is empty. Suppose $P_n = T_n$; then, by (6.5), the bag $\mathcal{B}_n$ is empty.

Finally, by (6.5), if the bag $\mathcal{B}_n$ is empty and the history O_1, O_2, ..., O_n is valid, then $T_n = P_n$.

□

Chapter 7

Trace-Hash Integrity Checking

Our trace-hash[1] checker [8, 11], intuitively, maintains a "write trace" and a "read trace" of its write and read operations to the external memory. At runtime, the checker updates its traces with a small constant-sized bandwidth overhead. When the checker needs to check a sequence of its operations, it performs a separate integrity-check operation using the traces. When sequences of operations are checked, the amortized bandwidth cost of the integrity-check operation is very small and the checker's principal bandwidth overhead is the constant-sized runtime bandwidth overhead.

7.1 Trace-Hash Checker

In Chapter 6, we developed a *bag checker*. Its interface is made of three functions: `cbag-put`($\mathcal{I}_{ms}$), which places all the elements in the multiset $\mathcal{I}_{ms}$ into the untrusted bag; `cbag-take`($\mathcal{P}$), which takes all the elements that match predicate $\mathcal{P}$ from the untrusted bag; `cbag-check`() which returns true if and only if the untrusted bag has behaved correctly and is empty according to its history of accesses (i.e., if PutHash $\equiv_{\mathcal{H}}$ TakeHash). We now show how the bag checker can be used to create the trace-hash checker.

Figure 7-1 shows how to produce the trace-hash checker for random access memory, using the bag checker. The trace-hash checker provides the following interface: `trace-hash-add`(S) adds S, a set of (address, data value) pairs, to the address space; `trace-hash-remove`(a)

[1] In [8], the trace-hash checker was referred to as an offline checker [20]. In [11], the trace-hash checker was referred to as the log-hash checker.

removes address a from the address space; `trace-hash-store`(a, v) stores data value v at address a; `trace-hash-load`(a) returns the data value that is stored at address a; if `trace-hash-check`() returns true, then each `trace-hash-load` from an address returned the data value that was most recently placed at that address by `trace-hash-store` (or `trace-hash-add`).

Essentially, RAM is simulated by placing (address, data value) pairs into the bag. `trace-hash-add`(S) calls `cbag-put`(S) to put, S, a set of (address, data value) pairs, into the bag; `trace-hash-remove`(a) calls `cbag-take`(P_a) on the bag. To perform a `trace-hash-store`, the pair previously in the bag for that address is taken from the bag, and replaced by the new pair. To perform a `trace-hash-load`, the pair for the desired address is taken from the bag, the value in the pair is returned to the caller, and the pair is put back into the bag. We denote an untrusted bag with its bag checker as its interface as a checked bag. To check the bag, `trace-hash-check` empties it into a fresh checked bag, and once the current bag is empty, it `cbag-check`s it before throwing it out. The operation resets the TIMER. Because the fresh checked bag has a zero TIMER, when `trace-hash-check` replaces the current checked bag with the fresh checked bag, it replaces the high TIMER in the current checked bag with the zero TIMER in the fresh checked bag.

If, during the FSM's execution, the FSM wishes to increase its address space, the FSM calls `trace-hash-add` on the set of new addresses. The FSM can then store and load from the larger address space. If, during the FSM's execution, the FSM wants to decrease its address space, the FSM calls `trace-hash-remove` on each of the addresses that will no longer be used. The FSM then stores and loads from the smaller address space. Only the addresses in the FSM's current address space are traversed during a `trace-hash-check` operation.

Because the FSM can `trace-hash-add` and `trace-hash-remove` arbitrary addresses from its address space, there are two new issues that must be addressed. Section 7.2 describes the issues and how we use the bag abstraction we developed in Chapter 6 to resolve them.

To create the trace-hash checker, we use a bag checker, into whose bag we will place (address, data value) pairs. P_a is a predicate that returns true on a pair (a', v) for which $a = a'$.

Intuitively, a valid RAM is a valid bag in which each address in the bag is present exactly once. To check the RAM, we must both check that the bag is behaving like a valid bag and that the *RAM invariant* (cf. Section 7.2.2) is maintained. To help check the RAM invariant, we set an ERROR flag if `cbag-take`(P_a) does not return a singleton set; if the ERROR flag is set, `check` returns false.

`trace-hash-add`(S) adds S, a set of (address, data value) pairs, to the address space.

1. `cbag-put`(S) into the bag.

`trace-hash-remove`(a) removes address a from the address space.

1. `cbag-take`(P_a) from the bag.

`trace-hash-store`(a, v) stores v at address a.

1. `cbag-take`(P_a) from the bag.
2. `cbag-put`($\{(a, v)\}$) into the bag.

`trace-hash-load`(a) loads the data value at address a.

1. ($\{(a, v)\}$) =`cbag-take`(P_a) from the bag.
2. `cbag-put`($\{(a, v)\}$) into the bag.
3. Return v to the caller.

`trace-hash-check`() returns true if and only if the memory has behaved correctly.

1. Create a temporary new checked bag T. We refer to the current checked bag as B.
2. M =`cbag-take`(true) from B.
3. If the ERROR flag is set, return false.
4. If `cbag-check`() on B is false, then return false (the check failed).
5. If `cbag-check` returned true, it means a set, as opposed to a multiset, of (a, v) pairs was read in step 2. Thus, we refer to this set as S. `trace-hash-add`(S) into T.
6. $B = T$.
7. Return `true`.

Note that steps 2 and 5 involve a set S that is huge. In an actual implementation, we would merge both steps, putting items into T as soon as they were removed from B.

Figure 7-1: Trace-hash checker

7.2 Checking a dynamically-changing, sparsely-populated address space

7.2.1 Maintaining the set of addresses that the FSM uses

In real life, the untrusted bag that the bag checker uses is actually implemented with some untrusted random access memory. The bag checker's `cbag-puts` and `cbag-takes` of (address, data value) pairs result in puts and takes of (address, data value, time stamp) triples to the untrusted bag. The untrusted bag is implemented using untrusted RAM by storing (address, data value, time stamp) triples as *(data value, time stamp) pairs* at the address from the triple in the RAM.

The addresses that the FSM uses may be any arbitrary subset of the addresses in the memory. The number of addresses that the FSM uses can also grow as the FSM uses the RAM. When an untrusted bag is implemented using RAM, where (address, data value, time stamp) triples are stored as (data value, time stamp) pairs, there is a problem of determining which addresses to read to empty the bag in step 2 of a `trace-hash-check` function call. As the checker only has a fixed-sized trusted state, it is not feasible to maintain a data structure that records which addresses the FSM uses in the checker. This data structure must then be maintained in the RAM, where it can potentially be tampered with by an adversary. As an example, this data structure can be a bitmap to keep track of the addresses that the FSM uses. The issue is that, if an adversary tampers with this bitmap, the wrong addresses will be read in step 2 of `trace-hash-check`.

To address this issue, we argue that, implicitly, it is the untrusted bag's job to keep track of the addresses the FSM uses. *Thus the bitmap does not need to be protected when it is stored in untrusted RAM: regardless of whether it is protected or not, if an adversary tampers with it, the tampering will be detected and* ***`cbag-check`*** *will return false.* In particular, the `cbag-take`(true) call in step 2 of `trace-hash-check` tells the bag checker to take all of the items out of the bag. If an adversary tampers with the bitmap so that the wrong addresses are read, the bag will not be empty after `cbag-take`(true) is called. The bag checker will detect the tampering because `cbag-check`() will return false if the untrusted bag is not empty according to the checker's history of accesses.

Though we have used a bitmap as an example for the data structure the untrusted bag can use to maintain the addresses the FSM has used, we note that, since this data structure does not have to be protected, the untrusted bag can use any data structure that will allow it to most efficiently determine the addresses that the FSM has used.

7.2.2 Satisfying the RAM Invariant

The second issue is that, with the `trace-hash-add` and `trace-hash-remove` operations, it is possible for the FSM to use the RAM in a way that is not well-defined. In particular,

`(a)` if `trace-hash-add` is performed on an address that is present in the bag, the address will be in the bag twice. An adversary can then choose to return either of the values of the address when the FSM next performs a `trace-hash-load` operation. The bag checker we have been using so far may return true because the bag may have behaved like a valid bag; however, the RAM may still not behave like valid RAM because the adversary may replay the value the FSM loads from an address with an older value because there are two pairs for the address in the bag.

`(b)` if `cbag-take` in `trace-hash-load` is performed on an address that is not present in the bag, a valid bag will return the empty set, which does not present a value that the FSM can interpret. (A similar problem exists for the `cbag-take` in `trace-hash-remove` and `trace-hash-store`).

Therefore, to determine whether the RAM is behaving like valid RAM, we must both check that the bag is behaving like a valid bag and that the *RAM invariant* is maintained. The RAM invariant is stated as: according to the bag checker's operations on the bag, no `cbag-put` operation is performed on an address that is already present in the bag, and no `cbag-take` operation is performed on an address that is not present in the bag. *Intuitively, a valid RAM is a valid bag in which each address in the bag is present exactly once.*

Part `(b)` of the issue is easily addressed: an ERROR flag is set if `cbag-take`(P_a) does not return a singleton set. If the ERROR flag is set, `check` returns false[2].

[2]In an actual implementation, if the FSM performs an operation that causes `cbag-take` to be performed on an address that is not in the bag, an entry that is not currently in the bag is read from the RAM.

Part (a) is the more difficult issue. It is resolved by putting a restriction on the addresses that are added to the bag or a restriction on the addresses that are removed from the bag. The following two cases demonstrate how either of these restrictions is sufficient to help check that the RAM is behaving like valid RAM. The cases demonstrate how to check the integrity of RAM such that both the integrity of the bag and the RAM invariant are checked.

Case 1: Restricting the `trace-hash-adds`

In this case, the FSM ensures that, for a particular run, by the time `cbag-check` is called on the bag in step 4 of the `trace-hash-check` operation, no address has had `trace-hash-add` called on it more than once. In other words, for a particular run, the addresses which `trace-hash-add` has been called on for a particular bag form a set (a simple example is, when the new bag is created, the FSM calls `trace-hash-add` on a set of addresses in the set of (address, data value) pairs, and does not call it again until `trace-hash-check` is called). All of the operations in Figure 7-1 are present in the interface.

Theorem 7.2.1. *Denote the addresses on which* ***`trace-hash-add`*** *has been called on as the multiset, $M_{\text{th-add}}$. Assuming $M_{\text{th-add}}$ is a set (*`trace-hash-add`* has been called exactly once on each address), the RAM that is being checked has behaved like a valid bag that has satisfied the RAM invariant (i.e., like valid RAM) and the* ***`trace-hash-check`*** *operation has read exactly the addresses in $M_{\text{th-add}}$ if and only if the* ***`trace-hash-check`*** *operation returns true.*

Proof. The validity condition is that, assuming $M_{\text{th-add}}$ is a set, if the RAM that is being checked has behaved like valid RAM and the `trace-hash-check` operation has read exactly the addresses in $M_{\text{th-add}}$, then the `trace-hash-check` operation returns true. The validity condition is easy to verify.

We present an argument for the safety condition: assuming $M_{\text{th-add}}$ is a set, if the `trace-hash-check` operation returns true, then the RAM that is being checked has behaved like valid RAM and the `trace-hash-check` operation has read exactly the addresses in $M_{\text{th-add}}$. If the `trace-hash-check` operation returns true, then, by Theorem 6.2.1, the bag

Therefore `cbag-check` will return false, and thus, `check` will return false. Also, in an actual implementation, `cbag-take`(P_a) will not return a set/multiset with two or more elements.

has behaved like a valid bag and the bag is empty i.e., the `trace-hash-check` operation has read exactly the addresses in $M_{\text{th-add}}$. We argue that the RAM invariant has been satisfied by contradiction. Assume that $M_{\text{th-add}}$ is a set and that the `trace-hash-check` operation returns true, and that the RAM invariant has not been satisfied. This means that either:

1. a `cbag-put` operation (in `trace-hash-add`) was performed on an address already present in the bag, so the address is in the bag multiple times. This contradicts the fact that the addresses which `trace-hash-add` has been called on form a set.

2. a `cbag-take` operation (in `trace-hash-load`, `trace-hash-store`, or `trace-hash-remove`) was performed on an address not present in the bag. Since the bag behaved like a valid bag, that `cbag-take` operation returned the empty set, so the ERROR flag was set. This contradicts the fact that `trace-hash-check` returned `true`.

□

Case 2: Omitting `trace-hash-remove` from the interface

In this case, there is no `trace-hash-remove`(a) operation in the trace-hash checker interface. For this case, the `trace-hash-check` procedure must ensure that each address is taken from the bag at most once in step 2 of the procedure in order to ensure that the RAM invariant is met.

Theorem 7.2.2. *Denote the addresses on which* ***`trace-hash-add`*** *has been called on as the multiset, $M_{th\text{-}add}$. Assuming there is no* ***`trace-hash-remove`****(a) operation, the RAM that is being checked has behaved like a valid bag that has satisfied the RAM invariant (i.e., like valid RAM) and the* ***`trace-hash-check`*** *operation has read exactly the addresses in $M_{th\text{-}add}$ if and only if the* ***`trace-hash-check`*** *operation returns true and has taken a set (as opposed to a multiset) of addresses from the bag.*

Proof. The validity condition is that, assuming there is no `trace-hash-remove` operation, if the RAM that is being checked has behaved like valid RAM and the `trace-hash-check` operation has read exactly the addresses in $M_{\text{th-add}}$, then `trace-hash-check` operation returns true and has taken a set of addresses from the bag. The validity condition is easy to verify.

Again, we present an argument for the safety condition: assuming there is no `trace-hash-remove` operation, if the `trace-hash-check` operation returns true and has read a set of addresses, then the RAM that is being checked has behaved like valid RAM and the `trace-hash-check` operation has read exactly the addresses in $M_{\text{th-add}}$. If the `trace-hash-check` operation returns true, then, by Theorem 6.2.1, the bag has behaved like a valid bag and the bag is empty i.e., the `trace-hash-check` operation has read exactly the addresses in $M_{\text{th-add}}$. We argue that the RAM invariant has been satisfied by contradiction. Assume that there is no `trace-hash-remove` operation and that the `trace-hash-check` operation returns true and has taken a set of addresses from the bag, and that the RAM invariant has not been satisfied. This means that either:

1. a `cbag-put` operation (in `trace-hash-add`) was performed on an address already present in the bag, so the address is in the bag multiple times. Because there is no `trace-hash-remove` operation, it will remain that way until `trace-hash-check` is performed. If `trace-hash-check` ensures that it reads each address at most once, then the bag is not empty after `cbag-take`(true) is called. But, `trace-hash-check` returns true, so `cbag-check` returned true. Thus, we have a contradiction of Theorem 6.2.1.

2. a `cbag-take` operation (in `trace-hash-load` or `trace-hash-store`) was performed on an address not present in the bag. Since the bag behaved like a valid bag, that `cbag-take` operation returned the empty set, so the ERROR flag was set. This contradicts the fact that `trace-hash-check` returned `true`.

□

We note that case 1, where the addresses that `trace-hash-add` has been called on form a set, is important for proving the correctness of the tree-trace checker in Chapter 8. Case 2, where there is no `trace-hash-remove` operation and `trace-hash-check` takes a set of addresses from the bag, corresponds to our implementation of the trace-hash checker in Section 7.5; we used this case for the trace-hash checker implementation in Section 7.5 because of its simplicity. In the implementation, there is no `trace-hash-remove` operation. New addresses are added on demand while the FSM is running. The page table is used to keep track of which addresses the FSM has used during its execution. When there is a new page

allocated, the RAM checker calls `trace-hash-add` on each address in the page. When the RAM checker performs a `trace-hash-check` operation, it walks through the page table in an incremental way and reads all of the addresses in a valid page, also in an incremental way. This ensures that, during a check, each address is read at most once (i.e., a set of addresses is read). The page table does not need to be protected because, if an adversary changes the page table so that the checker calls `trace-hash-add` on the same address multiple times or skips some addresses during the `trace-hash-check` operation, the `trace-hash-check` operation will fail, because the underlying bag will not be empty.

7.3 Caching

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

7.4 Analysis

In this section, we present a detailed comparison of the trace-hash and hash tree schemes when caches are used (cf. Section 7.3 and Section 4.1.1). In Section 7.4.1, we analyze the space overhead and bandwidth overhead of trace-hash integrity checking. In Section 7.4.2, we analytically compare the trace-hash scheme with the hash tree scheme.

In the analysis, to compare the performance of the trace-hash scheme with that of the hash tree scheme, we compare the bandwidth overheads of the schemes. In hardware, on each FSM access to the untrusted RAM, the FSM speculatively continues execution as soon as the data value arrives from RAM and the checker performs the hash computations in the background. Because the checker performs the hash computations in the background, hash

	Trace-Hash	Hash Tree
per-bit space overhead	$\frac{b_t}{b_b}$	$\Theta(\frac{b_h}{b_b - b_h})$
bandwidth overhead	$t(1-h_v)(2b_t) + (n_{th} - C)(b_b + 2b_t)$	$t((1-\delta h_v)zb_b - (1-h_v)b_b)$

Table 7.1: Comparison of the trace-hash and hash tree integrity checking schemes

computation latency does not generally affect the performance of the FSM. When critical operations occur, hash computation latency affects the FSM's performance because the FSM must wait for the checker to complete the hash computations before the FSM can perform the critical operation. However, we are assuming that critical operations are, generally, not very frequent. With respect to bandwidth, the checker is using the same bus for fetching time stamps/hashes from the untrusted RAM as the FSM is using to fetch data values from the untrusted RAM. Thus, the time stamps/hashes are competing for the same memory bus bandwidth as the FSM's data values. Thus, the less the bandwidth that is consumed by the time stamps/hashes, the faster the FSM will retrieve data values from the RAM and the faster the FSM will perform.

Table 7.1 summarizes the space overheads and bandwidth overheads of the trace-hash and hash tree schemes. In the table, for the trace-hash scheme, b_t is the number of bits in a time stamp and b_b is the number of bits in a data value/hash cache block. t is the number of FSM stores and loads that the FSM performs before the checker performs a `trace-hash-check` operation. We refer to t, the period between intermediate `trace-hash-check` operations, as a *check period*. n_{th} is the number of data value blocks that the FSM uses, i.e., the number of blocks to which a time stamp is appended when the block is stored in the untrusted RAM. C is the number of blocks that can be stored in the cache, and h_v is the cache hit rate - the fraction of FSM load/store operations that find their data in the cache.

In addition, for the hash tree scheme, m is the number of children of each node in the tree. b_h is the number of output bits of the hash function used for the hash tree. We assume that, in the hash tree scheme, one hash covers one cache block implying $b_b = mb_h$. δh_v, where $0 \leq \delta \leq 1$, is the data value cache hit rate in the hash tree scheme (for simplicity, we assume that this is less than the data value cache hit rate of the trace-hash scheme because both hash blocks and data value blocks must be stored in the cache for the hash tree scheme

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 principal points of the analysis are that:

- for a large untrusted RAM, if the size of a time stamp is less than the size of a hash, the trace-hash scheme will have a smaller space overhead than the hash tree scheme. In the experiments in Section 7.5, 4 GBytes of memory are used, with $\frac{1}{16}$ of the memory being consumed by 32-bit time stamps. We contrast this with the experiments in [3], where $\frac{1}{4}$ of memory is used for 128-bit MD5 hashes for a 4-ary hash tree in a similar experimental setup.

- if t is greater than $\frac{(n_{th}-C)(b_b+2b_t)}{((1-\delta h_v)zb_b-(1-h_v)b_b)\ -\ (1-h_v)(2b_t)}$, the overhead of fetching and caching hashes in the hash tree scheme will exceed the cost of reading addresses to perform a `trace-hash-check` operation in the trace-hash scheme, and the trace-hash scheme will consume less bandwidth and perform better than the hash tree scheme. Importantly, we can observe that, *the smaller n_{th} is, the smaller t will be before the trace-hash scheme will perform better than the hash tree scheme.* The tree-trace scheme in Chapter 8 uses this observation to optimize the trace-hash scheme. In the experiments in Section 7.5, we are concerned with the number of FSM accesses to the untrusted RAM, $T = (1-h_v)t$, in the trace-hash scheme. If we assume $\delta = 1$ for large caches, the trace-hash scheme is better than the hash tree scheme if $T \geq \frac{(n_{th}-C)(b_b+2b_t)}{(z-1)b_b-2b_t}$.

7.4.1 Trace-Hash Checker

Space Overhead

Let b_t be the number of bits in a time stamp and b_b is the number of bits in a data value cache block. The per-bit space overhead of the time stamps is $\frac{b_t}{b_b}$ bits.

Bandwidth Overhead

Let n_{th} be the number of data value blocks protected by the trace-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 `trace-hash-check` operation. C is the number of blocks that can be stored in the cache, and h_v is the 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 trace-hash scheme. During a `trace-hash-check` operation, the blocks that are not in the cache are read, with their associated time stamps; if the `trace-hash-check` operation is an intermediate check, time stamps are also written back to memory to reset them. Thus, the bandwidth consumption of the `trace-hash-check` operation is $(n_{th}-C)(b_b+2b_t)$. Therefore, the total bandwidth overhead of the trace-hash scheme is $t(1-h_v)(2b_t)+(n_{th}-C)(b_b+2b_t)$.

7.4.2 Comparison with Hash Tree Checker

Space Overhead

We provide some comparison of the overhead of storing time stamps in the trace-hash scheme to the overhead of storing hashes in the hash tree scheme. The per-bit space overhead in the trace-hash scheme is $\frac{b_t}{b_b}$ bits. If the size of a data value block is the same as the size of a hash block, an m-ary hash tree has a per-bit space overhead of $\Theta(\frac{1}{m-1})$ bits. The parameter, m, is typically small, like 2, 4, or 8.

Suppose that, in the trace-hash scheme, one time stamp protects one data value block, and, in the hash tree scheme, one hash covers one data value block. Let b_h be the size of a hash. Because one hash covers m leaves, $b_b = mb_h$. The space overhead of the hash tree is thus, $\Theta(\frac{b_h}{b_b-b_h})$. If $b_t < b_h$, then $\frac{b_t}{b_b} < \frac{b_t}{b_b-b_t} < \frac{b_h}{b_b-b_h}$. This means that, for a large untrusted RAM, if the size of a time stamp is less than the size of a hash, the trace-hash scheme will have a smaller space overhead than the hash tree scheme.

Bandwidth Overhead

For the hash tree scheme, let δh_v, where $0 \leq \delta \leq 1$, be the cache hit rate of data values in the hash tree scheme (for simplicity, we assume that this is less than the data value cache hit rate of the trace-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 $t > \frac{(n_{th}-C)(b_b+2b_t)}{((1-\delta h_v)zb_b-(1-h_v)b_b) \ - \ (1-h_v)(2b_t)}$, the overhead of fetching and caching hashes in the hash tree scheme will exceed the cost of reading addresses to perform a `trace-hash-check` operation in the trace-hash scheme, and the trace-hash scheme will consume less bandwidth and perform better than the hash tree scheme. Importantly, we can observe that, the smaller n_{th} is, the smaller t will be before the trace-hash scheme will perform better than the hash tree scheme. The tree-trace scheme in Chapter 8 uses this observation to optimize the trace-hash scheme. In the experiments in Section 7.5, we are concerned with the number of FSM accesses to the untrusted RAM, $T = (1 - h_v)t$, in the trace-hash scheme. If we assume $\delta = 1$ for large caches, the trace-hash scheme is better than the hash tree scheme if $T \geq \frac{(n_{th}-C)(b_b+2b_t)}{(z-1)b_b-2b_t}$.

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 `trace-hash-check` is very small and the principal bandwidth overhead is the constant-sized runtime overhead of reading and writing time stamps, which are also very small (32 bits). The trace-hash scheme thus performs very well and its bandwidth overhead is very small. It is interesting to note that, if the number of FSM load and store operations is large, trace-hash integrity checking can consume less bandwidth and, thus, perform better than a simple, faulty, scheme, in which a 128-bit MAC is appended to a block that is stored in RAM (cf. Chapter 3).

When `trace-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 trace-

hash scheme between the checks. In this case, the amortized bandwidth consumption of reading the entire set of addresses protected by the trace-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 trace-hash scheme. (When check periods are small, the trace-hash scheme is not optimal because the `trace-hash-check` operation has to read all of the addresses that are protected by the trace-hash scheme to perform the check, instead of just the addresses that are used by the FSM during that check period.)

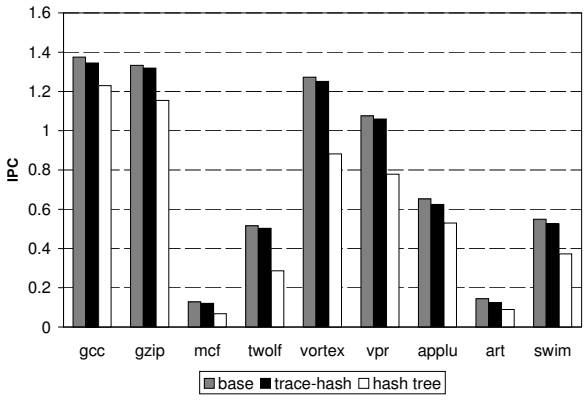
7.5 Experiments

This section evaluates the trace-hash integrity checking scheme compared to the hash tree scheme for computer processors through detailed simulations. We first investigate the impact of the trace-hash scheme on processor performance ignoring the overhead of reading memory in the `trace-hash-check` operation. These experiments provide the cost of trace-hash integrity checking when memory integrity needs to be checked only at the end of program execution, or very infrequently in comparison to the total execution time of the program. Then, we study the performance of the trace-hash scheme when more frequent integrity checks are required.

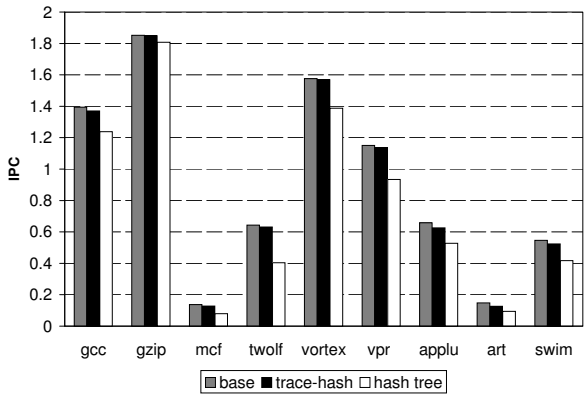
Our simulation framework is based on the SimpleScalar tool set [6]. The simulator models speculative out-of-order processors, which are standard modern microprocessors such as the Intel Pentium. For all of the experiments in this section, nine SPEC2000 CPU benchmarks [16] are used as representative applications: `gcc`, `gzip`, `mcf`, `twolf`, `vortex`, `vpr`, `applu`, `art`, and `swim`. The SPEC benchmark suite is a standard set of programs that are generally used in the computer architecture community to evaluate processor performance. These benchmarks show varied characteristics and represent various types of applications.

7.5.1 Asymptotic Performance

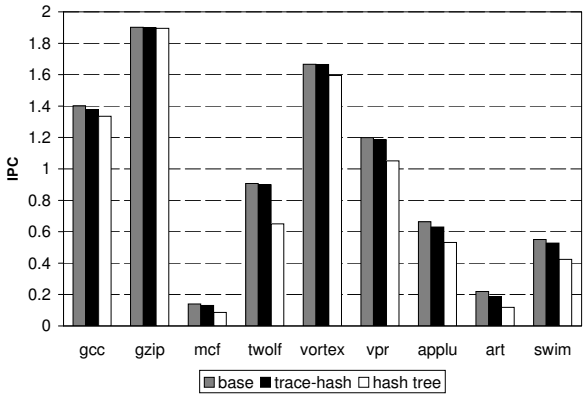
Applications that sign results at the end of execution, or do so relatively infrequently, are first considered. For these applications, the overhead of reading the untrusted RAM to perform the `trace-hash-check` operation in the trace-hash scheme is negligible. If a typical



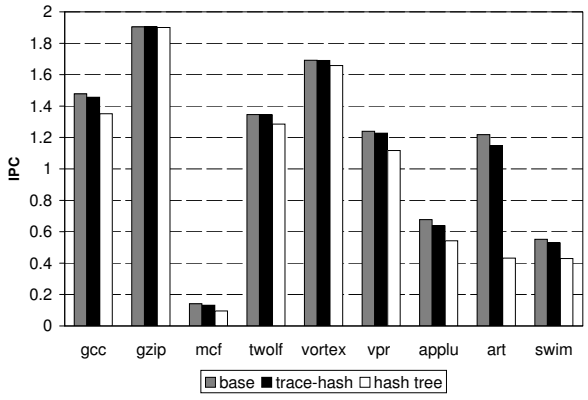
(a) 256KB



(b) 512KB



(c) 1MB



(d) 2MB

Figure 7-2: Comparison of the asymptotic performance of the trace-hash scheme and the hash tree scheme

program executes for a minute, this corresponds to roughly 100 billion instructions on a state-of-the-art 2- or 4-way superscalar 1 GHz processor. The `trace-hash-check` operation typically takes less than a billion cycles, and if this is performed once, at the end of the execution, the overhead is very small.

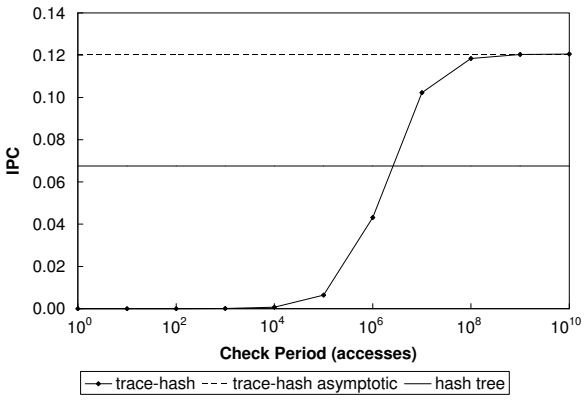
In this section, we compare the trace-hash scheme with the hash tree scheme and quantify their overheads relative to a standard processor without integrity checking. We will ignore the overhead of the `trace-hash-check` operation. This comparison will demonstrate the advantage of the trace-hash scheme over the hash tree scheme when integrity checking is infrequent.

Figure 7-2 illustrates the impact of integrity checking on program performance. For four different L2 cache sizes (256KB, 512KB, 1MB, 2MB), the IPCs (instructions per clock cycle) of three schemes are shown: a standard processor without integrity checking (`base`), the trace-hash integrity checking scheme (`trace-hash`), and the hash tree integrity checking scheme (`hash tree`). The IPC represents how fast a program executes. Therefore, higher IPC results indicate better program performance. The cache block size is 64B, and 32-bit time stamps for the trace-hash scheme and 128-bit hashes for the hash tree scheme are used.

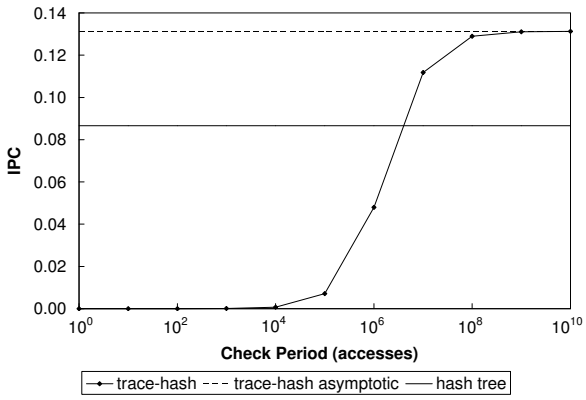
The experimental results clearly demonstrate the advantage of the trace-hash scheme over the hash tree scheme when we can ignore the cost of the `trace-hash-check` operation. For all programs and configurations simulated, the trace-hash scheme outperforms the hash tree scheme. The performance overhead of `trace-hash` compared to `base` is often less than 5% and less than 15% even for the worst case. On the other hand, `hash tree` has as much as 50% overhead in the worst case and 20-30% in general.

7.5.2 Effects of Checking Periods

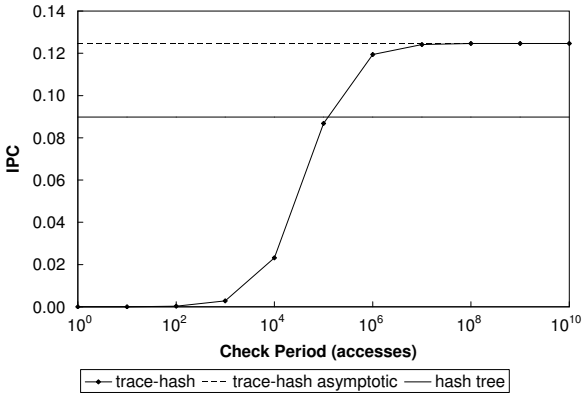
The experiments in the last section clearly demonstrated that the trace-hash scheme outperforms the hash tree scheme when checking is infrequent. However, applications may need to check the integrity of memory more often for various reasons such as exporting intermediate results to other programs. In these cases, we cannot ignore the cost of frequent `trace-hash-check` operations. In this section, we compare the trace-hash scheme and the hash tree scheme including the cost of periodic `trace-hash-check` operations.



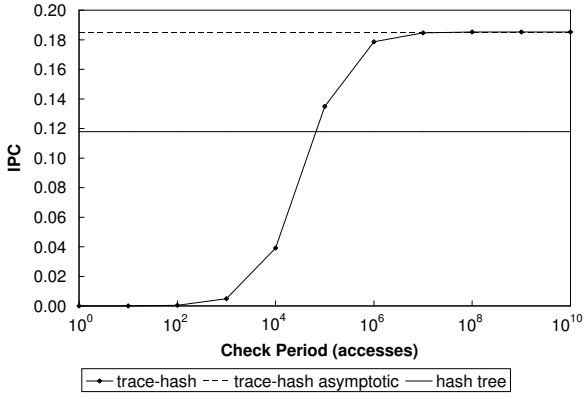
(a) `mcf, 256KB`



(b) `mcf, 1MB`



(c) `art, 256KB`



(d) `art, 1MB`

Figure 7-3: Performance comparison of the trace-hash scheme and the hash tree scheme for various trace-hash check periods

We assume that the trace-hash scheme checks memory integrity every T FSM accesses to the untrusted RAM. A processor executes a program until it makes T main memory accesses, then checks the integrity of the T accesses by reading the memory it used before continuing. Obviously, the overhead of the trace-hash checking heavily depends on the characteristics of the program and the period T. We use two representative benchmarks `mcf` and `art` – `mcf` is one of the benchmarks with the largest cost for the `trace-hash-check` operation and `art` is one of the benchmarks with the smallest cost for the `trace-hash-check` operation. `mcf` uses 195MB of main memory and takes about 140 million cycles for each check. On the other hand, `art` uses only 4MB of memory and takes about 2.7 million cycles for a check.

Figure 7-3 compares the performance of the trace-hash scheme and the hash tree scheme for various trace-hash check periods. In the figure, IPCs for the trace-hash scheme including the `trace-hash-check` operation (`trace-hash`), the trace-hash scheme ignoring the `trace-hash-check` operation (`trace-hash asymptotic`) and the hash tree scheme (`hash tree`) are shown. Results are shown for 256KB and 1MB L2 caches with 64B blocks. 32-bit time stamps and 128-bit hashes are used.

Experimental results show that the performance of the trace-hash scheme heavily depends on the checking period. The trace-hash checking is infeasible when the application needs to assure memory integrity frequently. In this case, the hash tree integrity checking should be used. On the other hand, if checking is less frequent, the trace-hash scheme can outperform the hash tree scheme. Also, as the checking period increases, the performance of the trace-hash scheme converges to the asymptotic performance ignoring the cost of the `trace-hash-check` operation (around 1 billion accesses for `mcf` and 10 million accesses for `art`).

The break-even point between the trace-hash scheme and the hash tree scheme depends on the characteristics of the program. For programs using a large amount of memory such as `mcf`, the checking period should be long so that the cost of `trace-hash-check` is amortized. However, the trace-hash scheme performs as well as the hash tree for much shorter checking periods for programs such as `art` with a small amount of memory usage. For example, with 256KB L2 caches, the break-even point for `mcf` is around $T = 4$ million accesses, which corresponds to about 160 million instructions (0.12 seconds with 4-way superscalar

1-GHz processors). The analysis in Section 7.4 predicts these break-even points reasonably accurately. For example, in the case of `mcf` with a 256KB cache, the analysis results in

$$T \geq \frac{(3194880 - 4096) \cdot (512 + 2 \cdot 32)}{0.5351 \cdot 512 - 2 \cdot 32} \approx 8.8 \times 10^6 \text{ accesses.}$$

The value of n_{th} was calculated based on 195MB (1MB = 16,384 cache blocks) memory usage, and $(z - 1) = 0.5351$. For `mcf` with a 1MB cache, the analysis results in $T \geq \frac{(3194880-16384)\cdot(512+2\cdot32)}{0.434\cdot512-2\cdot32} \approx 11.6 \times 10^6$ accesses. For `art` with a 256KB cache, the analysis results in $T \geq \frac{(65536-4096)\cdot(512+2\cdot32)}{0.5324\cdot512-2\cdot32} \approx 1.7 \times 10^5$ accesses. For `art` with a 1MB cache, the analysis results in $T \geq \frac{(65536-16384)\cdot(512+2\cdot32)}{0.5572\cdot512-2\cdot32} \approx 1.3 \times 10^5$ accesses.

As is also described in the analysis in Section 7.4, when check periods are large, the trace-hash scheme performs very well because the amortized bandwidth consumption of reading the memory to perform the `trace-hash-check` is very small and the principal bandwidth overhead is the constant-sized runtime overhead of reading and writing time stamps, which are also very small. When check periods are small, the program performs a small number of memory operations and uses a small subset of the addresses that are protected by the trace-hash scheme between the checks. In this case, the amortized bandwidth consumption of reading all of the memory protected by the trace-hash scheme, which is all of the memory that has been used by the program, is more costly and performance of the trace-hash scheme is not good. (When check periods are small, the trace-hash scheme is not optimal because the `trace-hash-check` operation has to read all of the addresses that are protected by the trace-hash scheme to perform the check, instead of just the addresses that are used by the program during that check period.)

In general, our experiments show that the trace-hash scheme is well-suited for certified execution applications [3], in, for example, commercial grid computing, secure mobile agents and trusted third party computation. In certified execution applications, the processor is equipped with a private key of a public-private key pair. The processor executes a program without allowing any interference from external sources. It executes the program to produce a result. The processor then creates a certificate that includes a hash of the program and its inputs, and the result of the program's execution. The certificate is signed with the processor's private key; anyone with the correct public key can verify the certificate and

trust the result in the certificate. In certified execution applications, the processor only needs to check the integrity of its memory operations at the end of the program's execution, when the processor signs the certificate. As our experiments show, the trace-hash scheme significantly outperforms the hash tree if critical security operations are performed after around one billion or more memory operations. Thus, the trace-hash scheme is the better scheme for certified execution applications.

Chapter 8

Tree-Trace Integrity Checking

Our tree-trace[1] checker [7] is a hybrid of the hash tree checker and the trace-hash checker. The tree-trace checker enables the trace-hash scheme to be optimized. The tree-trace checker can also use the hash tree scheme when critical operations are frequent and use the trace-hash scheme when sequences of FSM data operations can be checked.

8.1 Partial-Hash Tree Checker

The partial-hash tree abstraction is a simple abstraction that we use to help build the tree-trace checker in Section 8.2. A partial-hash tree is a hash tree (cf. Section 4.1) in which some addresses are protected by the tree and some addresses are not protected by the tree. The partial-hash tree checker interface consists of the hash tree checker's `hash-tree-load`(a) and `hash-tree-store`(a, v) operations, and a third operation: `hash-tree-updateParent`(a, v). 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 checker can call `hash-tree-updateParent`(a, `NULL`) to remove address a from the protection 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

[1] In [7], the tree-trace checker was referred to as the tree-log checker.

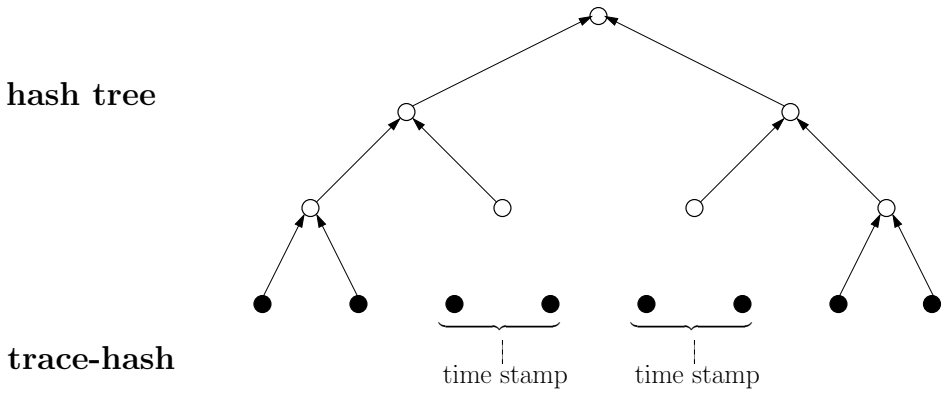


Figure 8-1: Illustration of tree-trace checker

for the address). If the checker performs a `hash-tree-load`(a) or a `hash-tree-store`(a, v) operation on an address, a, that is not protected by the tree, the hash tree integrity check will not pass (recall that `hash-tree-store` checks the integrity of the old value of the node and its siblings before updating the node).

The checker can call `hash-tree-updateParent`(a, v) to add address a back to the protection of the tree; the parent will be updated with a hash of value v. A `hash-tree-load`(a) or a `hash-tree-store`(a, v) operation on an address, a, that is protected by the tree operates as the standard hash tree load and store operations in Section 4.1.

8.2 Tree-Trace Checker

Figure 8-1 illustrates the tree-trace checker and Figure 8-2 shows the interface that the FSM calls to use the tree-trace checker to check the integrity of the untrusted RAM.

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

`tree-trace-moveToTraceHash` first calls `hash-tree-load`(a) to check the integrity of the value v at address a in the RAM. `hash-tree-updateParent`$(a,$ `NULL`$)$ is called to remove a from the tree. `trace-hash-add`(a, v) is then called to add a with value v to the trace-hash scheme[2].

[2]Because of the organization of the tree, whenever an address is moved to the trace-hash scheme, the

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

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

The tree-trace scheme allows for optimization of the trace-hash scheme. All of the addresses are initially in the tree. Recall that we call the period between intermediate `tree-trace-check` (`trace-hash-check`) operations a *check period* (cf. Section 7.4). During a check period, the checker can move an arbitrary set of addresses to the trace-hash scheme, where the FSM can perform store and load operations on them in the trace hash scheme. When a `tree-trace-check` operation is performed, all of the addresses in the trace-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 trace-hash scheme to be used by the FSM in the trace-hash scheme. The benefit is that, whenever a `tree-trace-check` operation is performed, *only the addresses of the data that have been moved to the trace-hash scheme since the last* ***`tree-trace-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-trace-check` operation needs to read addresses that are protected by the trace-hash scheme, but were not used during the check period, then the trace-hash scheme is not optimal. Thus, the ability of the tree-trace scheme to move the set of addresses that are accessed during a check period into the trace-hash scheme and move them back into the tree on a `tree-trace-check` operation so that a different set of addresses can be moved to the trace-hash scheme during a subsequent check period, helps to optimize the bandwidth overhead of the trace-hash scheme.

block consisting of the address's data value node and its siblings is `trace-hash-added` to the trace-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 trace-hash scheme.

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

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

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

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

`tree-trace-moveToTraceHash`(a): move address a from the tree to the trace-hash scheme:

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

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

1. `trace-hash-check`().
   Also, create a new TIMER′ and PUTHASH′. As the untrusted RAM is read to perform the `trace-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 `trace-hash-add`(a, v), using TIMER′ and PUTHASH′, to reset the time stamps in RAM and update PUTHASH′.

   Set TIMER and PUTHASH to TIMER′ and PUTHASH′; reset TAKEHASH to 0.

Figure 8-2: Tree-trace checker

The following theorem is proven in Section 8.3 to show that the tree-trace integrity checking scheme is secure:

Theorem 8.2.1. *The untrusted RAM has behaved like valid RAM if and only if the tree-trace integrity checks (using the hash tree and the* `tree-trace-check` *operation) return true.*

8.2.1 Caching

Caching is easily integrated into the tree-trace scheme using the approaches described in Sections 4.1.1 and 7.3. 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 4.1.1 is used. If the block's address is protected by the trace-hash scheme, the caching approach in Section 7.3 is used. `tree-trace-moveToTraceHash` 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 4.1.1. The parent is then updated in the cache. The `tree-trace-check` uses an approach similar to that in Section 7.3 when performing the `trace-hash-check` operation. If the block's address is in Y, the block's parent is brought into the cache as described in Section 4.1.1 and updated in the cache.

8.2.2 Bookkeeping

In Section 8.3, we prove that, with regard 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 trace-hash scheme, and to determine which addresses to read to perform a `tree-trace-check` operation, do not have to be protected. The necessary information is already implicitly encoded in the hash tree and trace-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 9.3, a range of addresses is moved to the trace-hash scheme when the trace-hash scheme is used. The checker maintains the highest address and the lowest address of the range in its fixed-sized trusted state. When the checker performs a `tree-trace-check` operation, it moves all of the addresses in the trace-hash scheme to the

tree, so that a separate range of addresses can be moved to the trace-hash scheme during a subsequent check period.

Maintaining a range of addresses in the trace-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.

8.3 Proof of Security of Tree-Trace Checker

In this section, we prove Theorem 8.2.1 (cf. Section 8.2).

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

We assume that the bookkeeping data structures (cf. Section 8.2.2) 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-trace-check` integrity checks return true. We will prove that an adversary is unable to affect the validity of the RAM.

Denote the addresses on which `trace-hash-add` has been called on as the multiset, $M_{\text{th-add}}$. First we show that $M_{\text{th-add}}$ is a set. Suppose `tree-trace-moveToTraceHash` is called on an address that has already been added to the trace-hash scheme. When the checker first called `tree-trace-moveToTraceHash` on the address in the tree to add it to the trace-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-trace-moveToTraceHash` 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 $M_{\text{th-add}}$ is a set and the results of Theorem 7.2.1 (cf. Section 7.2.2) 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-trace-check` operation would not read exactly the address in $M_{\text{th-add}}$, or a hash tree store or load operation would be performed on an address that is in the trace-hash scheme, or a trace-hash store or load operation would be performed on an address that is in the hash tree. Suppose that the `tree-trace-check` operation does not read exactly the addresses in $M_{\text{th-add}}$. This means that the `trace-hash-check` operation does not read exactly the addresses in $M_{\text{th-add}}$. By Theorem 7.2.1, the `tree-trace-check` operation will not pass. Suppose that a hash tree store or load operation is performed on an address that is in the trace-hash scheme. Because the `NULL` value was recorded in the address's parent in the tree when the address was first moved to the trace-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 trace-hash store or load operation is performed on an address that is in the hash tree. `trace-hash-store` or `trace-hash-load` is then called on the address before `trace-hash-add` is called to add the address to the trace-hash scheme. By Theorem 7.2.1, the `tree-trace-check` operation will not pass (because the RAM invariant is not satisfied). 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-trace-moveToTraceHash`, `tree-trace-store` and `tree-trace-load` each check the integrity of the data value read from the untrusted RAM. If the data value is tampered with, the hash 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 trace-hash scheme. By Theorem 7.2.1, the `tree-trace-check` operation will not pass (because the bag has not behaved like a valid bag). 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-trace-check` integrity checks return true, then the RAM has behaved like valid RAM. This concludes the proof of Theorem 8.2.1.■ □

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

Chapter 9

Adaptive Tree-Trace Integrity Checking

Our adaptive tree-trace[1] checker [7] adaptively chooses a tree-trace strategy for the FSM that indicates how the FSM should use the tree-trace scheme when the FSM is run. The checker enables FSMs to be run unmodified and still benefit from the tree-trace scheme's features.

9.1 Interface Overview

The adaptive tree-trace interface consists of just three operations: `adaptive-tree-trace-store`(a, v), `adaptive-tree-trace-load`(a) and `adaptive-tree-trace-check`(); these operations call their respective tree-trace operations (cf. Chapter 8). During the FSM's execution, the FSM calls `adaptive-tree-trace-store` and `adaptive-tree-trace-load` to access the untrusted RAM. The FSM calls `adaptive-tree-trace-check` whenever it executes a critical operation (cf. Chapter 1 and Chapter 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

[1] In [7], the adaptive tree-trace checker was referred to as the adaptive tree-log checker.

all FSMs, the tree-trace 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 trace-hash scheme based on its bandwidth overhead. Whenever an `adaptive-tree-trace-check` operation occurs, the checker moves all of the addresses in the trace-hash scheme back to the tree to optimize the trace-hash scheme; the operation does not need any arguments from the FSM because the checker moves all of the addresses in the trace-hash scheme to the tree. The `adaptive-tree-trace-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.

9.2 Adaptive Tree-Trace Checker

Sections 9.2.1 and 9.2.2 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-trace strategy we adopt. Section 9.2.3 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 8.2.2). In Section 9.5, we extend the discussion to when the checker uses a more general data structure, such as a bitmap, for its bookkeeping.

9.2.1 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 trace-hash scheme each time it accesses an address that is in the tree. The naïve approach is a valid approach of using the tree-trace 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-trace-check` operation).

Thus, to provide the parameterizable worst-case bound, the checker needs to regulate the rate at which addresses are added to the trace-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-trace checker maintains two statistics: (1) its current total *potential*, Φ [29, Chapter 18], and (2) the number of data value blocks currently in the trace-hash scheme, n_{th}. $\Phi = (1+\omega)B_{ht} - B_{tt}$ where B_{tt} is the current total tree-trace 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, Φ is how many bits ahead the tree-trace 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. Φ and n_{th} are also maintained in the checker's fixed-sized trusted state. n_{th} is incremented whenever an address is moved from the tree to the trace-hash scheme, and reset to zero on a `tree-trace-check` operation after the operation has moved the addresses back to the tree. Φ is updated on each checker operation.

We itemize how Φ changes on each tree-trace operation:

- `tree-trace-store`/`tree-trace-load`: Φ increases with each operation.
- `tree-trace-moveToTraceHash`: Φ decreases with each operation. Let $C_{\text{tt-mv-to-th}}$ be the bandwidth consumed by the `tree-trace-moveToTraceHash` operation. Then, $\Delta\Phi = -C_{\text{tt-mv-to-th}}$.
- `tree-trace-check`: Φ decreases with each operation. Let $C_{\text{tt-chk}}(n_{th})$ be the bandwidth consumed by the `tree-trace-check` operation; $C_{\text{tt-chk}}(n_{th})$ increases with n_{th}. $\Delta\Phi = -C_{\text{tt-chk}}(n_{th})$.

Table 9.1 details the amounts by which Φ changes when a range is used for bookkeeping (cf. Section 8.2.2). The potential increases on each `tree-trace-store`/`tree-trace-load` operation. The essential idea of how we bound the worst-case tree-trace bandwidth overhead is to have the checker build up enough potential to cover the cost of the `tree-trace-moveToTraceHash` operation plus the increased cost of the `tree-trace-check` operation before the checker moves an address from the tree to the trace-hash scheme.

`tree-trace-store` `hash-tree-store`	$\Delta\Phi = \omega * (2hb_b - b_b)$
`tree-trace-load` `hash-tree-load`	$\Delta\Phi = \omega * (h-1)b_b$
`tree-trace-store` `trace-hash-store`	$\Delta\Phi = 2hb_b - (2(b_b + b_t)) + \omega * (2hb_b - b_b)$
`tree-trace-load` `trace-hash-load`	$\Delta\Phi = hb_b - (b_b + 2b_t) + \omega * (h-1)b_b$
`tree-trace-moveToTraceHash`	$\Delta\Phi = -(hb_b + (h-1)b_b + b_t)$
`tree-trace-check`	$\Delta\Phi = -n_{th}((b_b + b_t) \ + \ 2(h-1)b_b)$

Table 9.1: $\Delta\Phi$ if a range is used for bookkeeping (cf. Section 8.2.2). 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).

Whenever the checker wants to move an address to the trace-hash scheme, it performs a test to determine if it has enough potential to do so. For the test, the checker checks that the address is in the tree and that $\Phi > C_{\text{tt-mv-to-th}} + C_{\text{tt-chk}}(n_{th}+1)$. If these checks pass, the test returns true; otherwise, the test returns false. If the test returns true, the checker has enough potential to be able to move the address to the trace-hash scheme. Otherwise, the checker cannot move the address to the trace-hash scheme. Whenever an address is moved to the trace-hash scheme, n_{th} is incremented.

The mechanism described in this section is a safety mechanism for the adaptive tree-trace scheme: whenever an `adaptive-tree-trace-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 Φ, the larger ω is, the sooner the checker will be able to move addresses to the trace-hash scheme. Also, the larger ω is, the larger could be the potential loss in the case that the tree-trace scheme has to perform an `adaptive-tree-trace-check` soon after it has started moving addresses to the trace-hash scheme; however, in the case that the performance of the tree-trace scheme improves when the trace-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 trace-hash scheme and the greater can be the performance benefit of using the trace-hash scheme. Also from the expression for Φ, the smaller the tree-trace bandwidth overhead compared to the hash tree bandwidth overhead, the better the tree-trace scheme performs and the greater the rate at which addresses can be added to the trace-hash scheme. This helps the checker adapt to the performance of the scheme, while still guaranteeing the worst-case bound.

9.2.2 Without Caching: Tree-Trace Strategy

Section 9.2.1 describes the minimal requirements that are needed to guarantee the bound on the worst-case bandwidth overhead of the tree-trace checker. The approach described in Section 9.2.1 can be applied as a greedy algorithm in which addresses are moved to the trace-hash scheme whenever Φ 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 trace-hash scheme as long as Φ was large enough, for short check periods, the checker might move a lot of data into the trace-hash scheme and incur a costly penalty during that check period when the `adaptive-tree-trace-check` operation occurs. We do not want to risk gains from one check period in subsequent check periods. Thus, instead of using Φ, we use Φ_{cp}, the potential that the checker has gained during the current check period, to control the rate at which addresses are added to the trace-hash scheme when we adopt the greedy approach. By using Φ_{cp} instead of Φ, for short check periods, it is more likely that the checker will just keep addresses in the tree, instead of moving addresses to the trace-hash scheme. If we let $\Phi_{\text{cp-start}}$ be the value of Φ at the beginning of the check period, then $\Phi_{\text{cp}} = \Phi - \Phi_{\text{cp-start}}$. Φ_{cp} regulates the rate at which addresses are added to the trace-hash scheme during the current check period.

Figure 9-1 shows the interface that the FSM uses to call the adaptive tree-trace checker. The strategy we use is a simple strategy and more sophisticated strategies for moving addresses from the tree to the trace-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 9.2.1 to ensure that the strategy's bandwidth overhead is never worse than the parameterizable worst-case bound.

In the safety mechanism in Section 9.2.1, $(1+\omega)B_{ht} = B_{tt} + \Phi$. In the safety mechanism, at the beginning of the FSM's execution, $B_{ht} = B_{tt} = 0$, thus $\Phi = 0$; during the FSM's execution, because the checker ensures that it has enough potential to cover the cost of the `tree-trace-moveToTraceHash` operation plus the increased cost of the `tree-trace-check` operation before the checker moves an address from the tree to the trace-hash scheme, the potential is always greater than zero. Thus, as long as the checker's tree-trace strategy,

`adaptive-tree-trace-store`(a, v):

1. If a is in the tree, and $\Phi_{\text{cp}} > C_{\text{tt-mv-to-th}} + C_{\text{tt-chk}}(n_{th}+1)$, then `tree-trace-moveToTraceHash`$(a)$.
2. `tree-trace-store`(a, v).

`adaptive-tree-trace-load`(a):

1. If a is in the tree, and $\Phi_{\text{cp}} > C_{\text{tt-mv-to-th}} + C_{\text{tt-chk}}(n_{th}+1)$, then `tree-trace-moveToTraceHash`$(a)$.
2. `tree-trace-load`(a).

`adaptive-tree-trace-check`():

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

Figure 9-1: Adaptive tree-trace checker, without caching

whatever the strategy may be, is layered over this safety mechanism, the strategy will be $(1+\omega)$-competitive [2, Chapter 1] with the bandwidth overhead of the hash tree scheme (i.e., for any FSM, the strategy's bandwidth overhead will be guaranteed to be less than $(1+\omega)$ times the hash tree bandwidth overhead).

At this point, we precisely describe the three features of the adaptive tree-trace checker. Firstly, the checker *adaptively* chooses a tree-trace 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 never to be worse than the *parameterizable worst-case bound*. This feature allows the adaptive tree-trace 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 trace-hash integrity checks. This feature allows large classes of FSMs to take advantage of the constant runtime bandwidth overhead of the optimized trace-hash scheme to improve their integrity checking performance, because FSMs

typically perform many data operations before performing a critical operation.

Let |TIMER| be the bit-length of the trace-hash TIMER counter. In the third feature, we exclude intermediate trace-hash integrity checks because they become insignificant for sufficiently large |TIMER|. Whenever the TIMER reaches its maximum value during a check period, an intermediate trace-hash check is performed (cf. Section 8.2). However, by using a large enough |TIMER|, intermediate checks occur so infrequently that the amortized bandwidth cost of the check is very small, and the principal bandwidth overhead is the *constant runtime bandwidth overhead* of the time stamps.

9.2.3 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 9.2.1 and 9.2.2. Firstly, to accurately calculate the potentials, 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 potential can decrease on `tree-trace-store` and `tree-trace-load` operations. To handle this situation, the adaptive checker will need an additional tree-trace operation that allows it to *back off*, and will need to perform an additional test to determine whether it will need to back off. We describe the extensions.

Cache performance is very difficult to predict. Thus, to help determine B_{tt} 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-trace simulator that can be used to determine the cost of a particular tree-trace 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 [15]; the data values are not needed. The tag RAM is a small percentage of the cache [15]. 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 Chapter 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; in Chapter 11, we discuss how they can be dropped if the strictness of the bound is relaxed.

We expect `tree-trace-store` and `tree-trace-load` operations to generally perform better than the corresponding hash tree operations because the trace-hash scheme does not pollute the cache with hashes and because the runtime overhead of the trace-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-trace scheme, because the trace-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-trace and hash tree cache access patterns are different, and the tree-trace cache performance could be worse than the hash tree cache performance. Potential can sometimes decrease on `tree-trace-store` and `tree-trace-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 potential gets dangerously low.

The backoff procedure consists of performing a `tree-trace-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[2]. We refer to the backoff procedure as `tree-trace-bkoff`. Let C_{sync} be the cost of synchronizing the cache (it is independent of n_{th}). Then the bandwidth consumed by `tree-trace-bkoff` is $C_{\text{bkoff}}(n_{th}) = C_{\text{tt-chk}}(n_{th}) + C_{\text{sync}}$. Whenever the checker backs off, it continues execution just using the tree alone, until it has enough potential to try moving addresses to the trace-hash scheme again.

Again, we indicate how Φ changes with each tree-trace operation:

- `tree-trace-store`/`tree-trace-load`: With each operation, Φ usually increases; how-

[2]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 4.1.1).

ever it can decrease. Let $\Delta\Phi_{\text{tt-op}}$ be the change in Φ that occurs when the store/load operation is performed; $\Delta\Phi_{\text{tt-op}}$ can be positive or negative (and is different for each store/load operation). $\Delta\Phi = \Delta\Phi_{\text{tt-op}}$.

- `tree-trace-moveToTraceHash`: Φ decreases with each operation. $\Delta\Phi = -C_{\text{tt-mv-to-th}}$.
- `tree-trace-check`: Φ decreases with each operation. $\Delta\Phi \geq -C_{\text{tt-chk}}(n_{th})$.
- `tree-trace-bkoff`: Φ decreases with each operation. $\Delta\Phi \geq -C_{\text{bkoff}}(n_{th})$.

Figure 9-2 shows the `adaptive-tree-trace-store` operation when the FSM uses a cache. The `adaptive-tree-trace-load` operation is similarly modified.
`adaptive-tree-trace-check` is similar to the operation in Figure 9-1. The actual costs of $\Delta\Phi_{\text{tt-op}}$ and $C_{\text{tt-mv-to-th}}$ are obtained at runtime from the simulators. The worst-case costs of $C_{\text{tt-chk}}(n_{th})$ and $C_{\text{bkoff}}(n_{th})$ can be calculated; on the `tree-trace-check` and `tree-trace-bkoff` operations, Φ 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 Section 9.4. The worst-case cost of the `tree-trace-check` operation can be expressed as $C_{\text{tt-chk}}(n_{th}) = C_{\text{tt-chk}}(0) + C^{\text{marg}}_{\text{tt-chk}}(n_{th})$, where $C_{\text{tt-chk}}(0)$ is a fixed cost, a cost that is independent of n_{th}, and $C^{\text{marg}}_{\text{tt-chk}}(n_{th})$ is a marginal cost, a cost that is dependent on n_{th}. Recall that $C_{\text{bkoff}}(n_{th}) = C_{\text{tt-chk}}(n_{th}) + C_{\text{sync}}$.

In Figure 9-2, the first test in step 1 is similar to the test in Section 9.2.2. However, in this case, the potential that the checker uses to regulate the rate at which addresses are added to the trace-hash scheme is Φ'_{cp}, where Φ'_{cp} is the potential that the checker has gained after $\Phi > C_{\text{bkoff}}(0)$ during the current check period (or, if the checker has backed off, the potential that the checker has gained after $\Phi > C_{\text{bkoff}}(0)$ since backing off). Thus, $\Phi'_{\text{cp}} = \Phi - \max(C_{\text{bkoff}}(0),\ \Phi_{\text{cp-start}})$ (where $\Phi_{\text{cp-start}}$ is the value of Φ at the beginning of the check period, or, if the checker has backed off, the value of Φ after the checker has completed backing off). Φ'_{cp} only begins recording potential after Φ has become greater than $C_{\text{bkoff}}(0)$ because otherwise, the checker would not have enough potential to be able to back off if it needed to. The test also gives a small potential buffer per address in the trace-scheme, $C_{\text{buffer}}(n_{th})$, for the tree-trace scheme to start outperforming the hash tree.

`adaptive-tree-trace-store`(a, v):

1. If a is in the tree, and
   $\Phi'_{\text{cp}} > C_{\text{tt-mv-to-th}} + C^{\text{marg}}_{\text{tt-chk}}(n_{th}+1) + C_{\text{buffer}}(n_{th}+1)$,
   then `tree-trace-moveToTraceHash`(a).
2. If the tree-trace and hash tree caches are not synchronized,
   if $\Phi + \Delta\Phi_{\text{tt-op}} < C_{\text{bkoff}}(n_{th})$, then `tree-trace-bkoff`.
3. `tree-trace-store`(a, v).

Figure 9-2: `adaptive-tree-trace-store`, with caching

The test in step 2 determines whether the checker needs to back off. The tree-trace and hash tree caches are unsynchronized if the trace-hash scheme has been used (since the beginning of the FSM's execution or since the checker last backed off). It is only necessary to perform the test if the tree-trace and hash tree caches are unsynchronized. From the expression for Φ'_{cp}, $\Phi > \Phi'_{\text{cp}} + C_{\text{bkoff}}(0)$. From the first test, to successfully move an address to the trace-hash scheme, $\Phi'_{\text{cp}} > (C_{\text{tt-mv-to-th}} + C^{\text{marg}}_{\text{tt-chk}}(n_{th}+1) + C_{\text{buffer}}(n_{th}+1))$. Thus, $\Phi > (C_{\text{tt-mv-to-th}} + C^{\text{marg}}_{\text{tt-chk}}(n_{th}+1) + C_{\text{buffer}}(n_{th}+1) + C_{\text{bkoff}}(0))$. From the expression for $C_{\text{bkoff}}(n_{th})$, $\Phi > (C_{\text{tt-mv-to-th}} + C^{\text{marg}}_{\text{tt-chk}}(n_{th}+1) + C_{\text{buffer}}(n_{th}+1) + C_{\text{tt-chk}}(0) + C_{\text{sync}}) > (C_{\text{tt-mv-to-th}} + C_{\text{bkoff}}(n_{th}+1) + C_{\text{buffer}}(n_{th}+1))$. Thus, $\Phi > (C_{\text{tt-mv-to-th}} + C_{\text{bkoff}}(n_{th}+1) + C_{\text{buffer}}(n_{th}+1))$. This means that, whenever an address is successfully moved to the trace-hash scheme, $\Phi > (C_{\text{bkoff}}(n_{th}) + C_{\text{buffer}}(n_{th}))$, (recall that n_{th} is incremented when an address is successfully moved to the trace-hash scheme). If the trace-hash scheme has been used, the second test uses $\Delta\Phi_{\text{tt-op}}$, obtained from the simulators, to determine if performing the store operation would result in its potential dropping below $C_{\text{bkoff}}(n_{th})$. If it does, the checker backs off, then performs the operation. Otherwise, it just performs the operation in its current state.

With regard to the theoretical claims on the tree-trace algorithm in Section 9.2.2, 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-trace scheme will also perform well, because the tree-trace bandwidth overhead will be, at most, the parameterizable worst-case bound more than the hash tree bandwidth overhead.) With regard to the third feature, with a cache, the

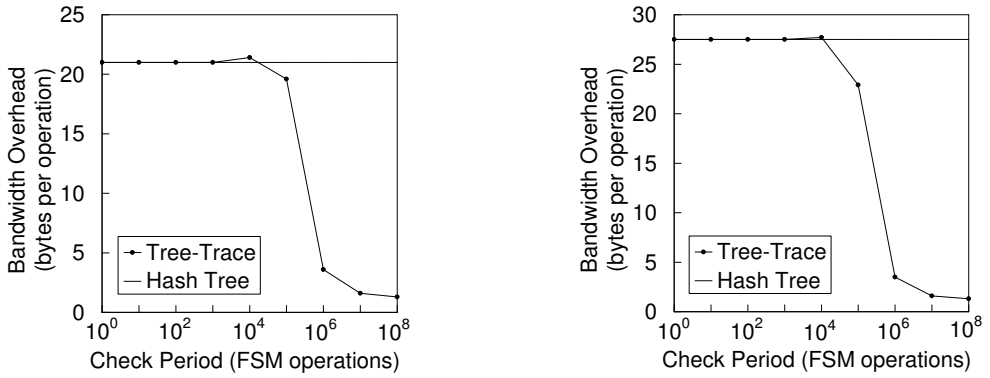
adaptive tree-trace checker will not improve over the hash tree for some FSMs. FSMs whose runtime performance improves when the trace-hash scheme is used experience the asymptotic constant bandwidth overhead behavior. From the expression for calculating potential (and from our experiences with experiments), we see that *the general trend is that the greater the hash tree bandwidth overhead, the less likely it is for the checker to back off and the greater the tree-trace 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-trace scheme will, when it has built up sufficient potential, automatically start using the trace-hash scheme to try to reduce the integrity checking bandwidth overhead.

Examples of cases in which the hash tree tends to be expensive, and in which one would expect to have a significant improvement over the hash tree when the trace-hash scheme is used are:

- an FSM that performs many stores. With many stores, there are many cache evictions of dirty cache blocks. Recall from the caching approach in Section 4.1.1 that, in the hash tree scheme, when the cache evicts a dirty block, the checker needs to check the integrity of the parent block and bring it into the cache, if it is not already in the cache (if the block were clean, the block is simply removed from the cache). There may not be a lot of spatial locality in the cache evictions and thus, checking the integrity of the parent block may require several hash blocks to be brought into the cache, making the hash tree expensive. The evicted block's parent block in the cache will also be dirty after the checker updates it, and the cost of its eviction will also contribute to the expense of the hash tree.

- an FSM with a large program stride. The program stride is the average number of addresses between consecutive program data operations. The general trend is that the larger the program stride, the smaller the spatial locality in the FSM's data operations, the larger the number of hashes that must be fetched on each FSM access to the untrusted RAM and the more expensive the hash tree.

- an FSM with a small cache. The general trend is that the smaller the cache, the smaller the number of hashes the cache contains, the larger the number of hashes that must be



(a) `b1`, cache size = 16 blocks

(b) `b1`, cache size = 12 blocks

Figure 9-3: Bandwidth overhead comparison of the tree-trace scheme and the hash tree scheme for various tree-trace check periods

fetched on each FSM access to the untrusted RAM and the more expensive the hash tree.

9.3 Experiments

We present some experimental evidence to support the theoretical claims on the adaptive tree-trace 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. ($C_{\text{buffer}}(n_{th})$ was about $4hb_bn_{th} = (4 * 10 * 64 * n_{th})$ bytes.) Figure 9-3(a) shows the bandwidth overhead for different check periods for a particular benchmark, `b1`. The cache size was 16 blocks. The tree-trace 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-trace: 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-trace scheme becomes significantly smaller. By check periods of 10^7 operations, the tree-trace 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 trace-

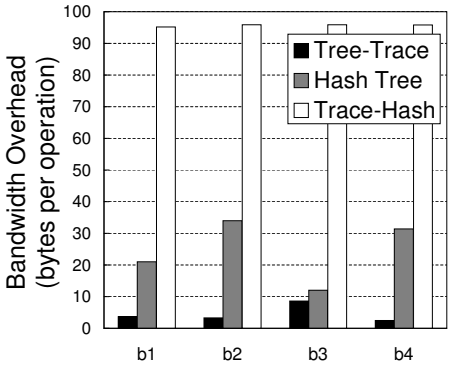


Figure 9-4: Bandwidth overhead comparison of the tree-trace scheme, the hash tree scheme and the trace-hash scheme for an example access pattern

hash scheme for this experiment because its bandwidth overhead is prohibitively large when check periods are small.) In Figure 9-3(b), the cache size is reduced to 12 blocks, making the hash tree more expensive. The figure shows a greater tree-trace scheme improvement over the hash tree bandwidth overhead when the tree-trace scheme improves on the hash tree. Figure 9-4 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-trace scheme outperforms both the hash tree and trace-hash schemes.

9.4 Worst-case Costs of `tree-trace-check` and `tree-trace-bkoff`, with caching

In this section, we give the worst-case costs of $C_{\text{tt-chk}}(n_{th})$ and $C_{\text{bkoff}}(n_{th})$ for the adaptive checker in the case that the FSM uses a cache (cf. Section 9.2.3). For this analysis, we assume a range is used for bookkeeping (cf. Section 8.2.2). Table 9.2 summarizes the costs.

The worst-case bandwidth consumption of the `tree-trace-check` operation is $2Chb_b + n_{th}((b_b + b_t) + 2(h-1)b_b)$, 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 $n_{th}((b_b + b_t) + 2(h-1)b_b)$ is the cost of reading the addresses in the trace-hash scheme and moving them to the tree.

The worst-case bandwidth consumption of the `tree-trace-bkoff` operation is

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

Table 9.2: Worst-case costs of `tree-trace-check` and `tree-trace-bkoff`, with caching, when a range is used for bookkeeping (cf. Section 8.2.2). 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.

$C_{\text{tt-chk}}(n_{th})+2Chb_b+Chb_b$, where $C_{\text{tt-chk}}(n_{th})$ is the worst-case cost of the `tree-trace-check` operation and $2Chb_b+Chb_b = 3Chb_b$ is the cost of synchronizing the cache (C_{sync}). $C_{\text{bkoff}}(n_{th})$ 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 trace-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-trace 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 9.2.3, $C_{\text{tt-chk}}(0) = 2Chb_b$ and $C^{\text{marg}}_{\text{tt-chk}}(n_{th}) = n_{th}((b_b+b_t) + 2(h-1)b_b)$. Also $C_{\text{bkoff}}(0) = 5Chb_b$.

9.5 Adaptive Tree-Trace Checker with a Bitmap

In this section, we describe the modifications that are made to the adaptive checker in Section 9.2 when the checker uses a bitmap for bookkeeping (cf. Section 8.2.2). Section 9.5.1 describes the modifications when the FSM does not use a cache. Section 9.5.2 describes the modifications when the FSM does use a cache.

9.5.1 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

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

Table 9.3: $\Delta\Phi$ if a bitmap is used for bookkeeping (cf. Section 8.2.2). 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 trace-hash scheme.

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 9.2.1 and Section 9.2.2. Firstly, the amounts by which Φ changes on the various tree-trace operations are slightly different. Secondly, because $C_{\text{tt-chk}}(0)$, the fixed-cost of the `tree-trace-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 trace-hash scheme needs to be slightly modified.

Table 9.3 details the new amounts by which Φ changes on the various tree-trace operations. For simplicity, we assume that the bitmap is on a per data value block granularity; it is simple to extend the analysis to a bitmap with a larger granularity. 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 trace-hash scheme and the checker then uses the bitmap. (A simple optimization is for

`adaptive-tree-trace-store`(a, v):

1. If a is in the tree, and $\Phi_{\text{cp}} > C_{\text{tt-mv-to-th}} + C^{\text{marg}}_{\text{tt-chk}}(n_{th}+1)$, then `tree-trace-moveToTraceHash`$(a)$.
2. `tree-trace-store`(a, v).

`adaptive-tree-trace-load`(a):

1. If a is in the tree, and $\Phi_{\text{cp}} > C_{\text{tt-mv-to-th}} + C^{\text{marg}}_{\text{tt-chk}}(n_{th}+1)$, then `tree-trace-moveToTraceHash`$(a)$.
2. `tree-trace-load`(a).

`adaptive-tree-trace-check`():

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

Figure 9-5: Adaptive tree-trace checker, without caching, when a bitmap is used for bookkeeping

the checker to maintain the highest and lowest addresses that have been moved to the trace-hash scheme during a check period in its fixed-sized trusted state. On `tree-trace-store` and `tree-trace-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-trace-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 potential, Φ, still increases on each `tree-trace-store`/`tree-trace-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 potential increases on each `tree-trace-store`/`tree-trace-load` operation.

The worst-case cost of the `tree-trace-check` operation can be expressed as $C_{\text{tt-chk}}(n_{th}) = C_{\text{tt-chk}}(0) + C^{\text{marg}}_{\text{tt-chk}}(n_{th})$, where $C_{\text{tt-chk}}(0)$ is a fixed cost, a cost that independent of n_{th}, and $C^{\text{marg}}_{\text{tt-chk}}(n_{th})$ is a marginal cost, a cost that is dependent on n_{th}. From Table 9.3, $C_{\text{tt-chk}}(0) = 2N_{\text{bmap}}$ and $C^{\text{marg}}_{\text{tt-chk}}(n_{th}) = n_{th}((b_b + b_t) + 2(h-1)b_b)$.

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

Table 9.4: Worst-case costs of `tree-trace-check` and `tree-trace-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.

Figure 9-5 shows the interface that the FSM uses to call the adaptive tree-trace checker. Compared with Figure 9-1 (c.f. Section 9.2.2), Φ_{cp} is the potential that the checker has gained after $\Phi > C_{\text{tt-chk}}(0)$ during the current check period. Thus, $\Phi_{\text{cp}} = \Phi - \max(C_{\text{tt-chk}}(0),\ \Phi_{\text{cp-start}})$ (where $\Phi_{\text{cp-start}}$ is the value of Φ at the beginning of the check period).

9.5.2 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 9.2.3 and Section 9.4. Firstly, an extra term is added to $C_{\text{tt-chk}}(n_{th})$ to account for using the bitmap. Secondly, bitmap segments may need to be read from RAM in the `adaptive-tree-trace-store` and `adaptive-tree-trace-load` operations. If a bitmap segment needs to be read, the checker needs to perform an additional test to ensure that it has enough potential to be able to do so. Thus, the adaptive checker's interface in Figure 9.2.3 also needs to be slightly modified.

Table 9.4 gives the worst-case costs of $C_{\text{tt-chk}}(n_{th})$ and $C_{\text{bkoff}}(n_{th})$. The tree-trace 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 9.2, the cost of $C_{\text{tt-chk}}(n_{th})$ increases by $3N_{\text{bmap}}$. As before, $C_{\text{bkoff}}(n_{th})$ costs an extra $3Chb_b$ more than the cost of $C_{\text{tt-chk}}(n_{th})$. $C_{\text{tt-chk}}(0) = 2Chb_b + 3N_{\text{bmap}}$ and $C^{\text{marg}}_{\text{tt-chk}}(n_{th}) = n_{th}((b_b + b_t) \ + \ 2(h-1)b_b)$. $C_{\text{bkoff}}(0) = 5Chb_b + 3N_{\text{bmap}}$.

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.

`adaptive-tree-trace-store`(a, v):

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

Figure 9-6: `adaptive-tree-trace-store`, with caching, when a bitmap is used for bookkeeping

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 trace-hash scheme. On `adaptive-tree-trace-store`/`adaptive-tree-trace-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-trace-store`/`tree-trace-load`. (Let b_{ht} be the average bandwidth overhead on a hash tree operation. When the trace-hash scheme is used, the data value cache hit rate tends to improve. Similarly to the case in Section 9.5.1, if $\omega * b_{ht} > C_{\text{read-bmap}}$, then, in the calculation of the potential, Φ, the extra cost of using the bitmap is covered by the overhead of the hash tree.)

Figure 9-6 shows the modified `adaptive-tree-trace-store`. The difference is in Step 1, where there is an extra test to determine if the checker has enough potential to read the bitmap segment if it needs to read the segment. `adaptive-tree-trace-load` is similarly modified. `adaptive-tree-trace-check` is similar to the check when a range is used for the bookkeeping (cf. Section 9.2.3 and Section 9.4).

On `adaptive-tree-trace-store`/`adaptive-tree-trace-load` operations, the tree-trace simulator is used to determine the costs of `tree-trace-moveToTraceHash` and `tree-trace-store`/`tree-trace-load`. These simulations occur after the checker has de-

termined 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-trace-store`/`adaptive-tree-trace-load` operations do not make any accesses to the RAM. (In the simulations, if the `tree-trace-moveToTraceHash` operation in `adaptive-tree-trace-store`/`adaptive-tree-trace-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-trace-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-trace checker, with caching, when a bitmap is used for bookkeeping. When check periods are large, the amortized cost of the `tree-trace-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.

9.6 Disk Storage Model

Thus far, we have been considering hash trees with a small branching factor, such as binary and 4-ary trees. In such trees, the bandwidth overhead on each FSM store or load operation is clearly logarithmic in the number of data values that are being protected.

Some storage devices, such as magnetic disks, use B-trees [29, Chapter 19] to help manage data. A B-tree is a balanced search tree that can have a large branching factor, often between 50 and 2000 nodes. A large branching factor greatly reduces the height of the tree. In disks, besides the cost of the bandwidth consumed by the bytes being transferred from the disk, there is also an additional fixed-cost for each disk access because of the time to position the disk's read/write head and wait for the disk to rotate to the correct position. Thus, the disk latency is an important performance consideration. If a single disk access fetches a node

and its siblings, using a B-tree greatly reduces the number of extra disk accesses that are required to fetch any leaf of the tree and thus reduces the latency incurred when fetching a leaf in the tree.

In this section, we consider hash trees with large branching factors. In Section 9.6.1, we study how the branching factor of the tree affects the overhead incurred on each FSM access to the untrusted storage. In Section 9.6.2, we describe how to modify the adaptive tree-trace checker for disks.

9.6.1 Branching Factor

Suppose that the storage consists of N data values and suppose that the hash tree's branching factor is m. Let us first consider bandwidth overhead alone. As the branching factor of a tree increases, the height of the tree decreases logarithmically, but the number of hashes that need to be read at each level increases linearly. The height of the tree is, thus, $\log_m N$ and the bandwidth overhead of the hashes on each FSM load/store is $\Theta(m \log_m N)$. Minimizing $m \log_m N$ for m gives $m = e$. For $m > e$, $m \log_m N$ is strictly increasing. Thus, if the goal is to construct the tree such as to minimize bandwidth overhead alone, then trees with a small branching factor, such as binary or 4-ary trees, are optimal.

Let us now consider latency alone. As the branching factor of a tree increases, the number of hashes decreases and, thus, the space overhead of the hashes decreases. The number of hashes of the tree is $\Theta(\frac{1}{m})$. If we fix the maximum amount of data value/hashes that can be retrieved on a single disk access, as m increases, the maximum number of disk accesses required to retrieve and verify any leaf in the tree decreases. Thus maximizing m minimizes the maximum latency that is incurred when retrieving and verifying a leaf in the tree.

Let us consider bandwidth overhead and latency together. Suppose the cost to access a node and its siblings is given by the cost function: $c(m) = F + m$, where F is the fixed cost incurred when accessing a node and its siblings. For memory, because silicon memory chips are entirely electronic, F is small. For disk, because of the large latency to position the disk's read/write head and position the disk, F is large. The cost to retrieve and verify a leaf in the tree is $c(m) * \log_m N$. Minimizing for m gives $m \ln(\frac{m}{e}) = F$. For $m > e$, as F increases, m increases. Thus, for $m > e$, if F is large, then m should be large; if F is small,

then m should be small.

9.6.2 Adaptive Tree-Trace Checker for Disks

For disks, because disk latency is such an important performance consideration, we compare and bound the overhead on the number of disk accesses instead of the bandwidth overhead. On each `hash-tree-store`/`hash-tree-load` there are extra disk accesses for reading/writing the hashes. On each `trace-hash-store`/`trace-hash-load`, there are extra disk accesses for reading/writing the time stamps. `tree-trace-moveToTraceHash` consumes disk accesses to move an address (and its siblings) from the tree to the trace-hash scheme. `tree-trace-check` consumes disk accesses to read the addresses currently protected by the trace-hash scheme and to move them back to the protection of the tree. The equation to calculate potential is $\Phi = (1 + \omega)D_{ht} - D_{tt}$, where D_{tt} is the current total tree-trace disk accesses overhead and D_{ht} is the current total hash tree disk accesses overhead, if the hash tree had been used to check the disk.

Chapter 10

Initially-Promising Ideas

In this chapter, we explore ideas that showed initial promise, but, upon careful examination, are vulnerable to attacks. The chapter helps to demonstrate the difficulty of the problem of checking the integrity of untrusted data when using only a small fixed-sized trusted state.

Section 10.1 studies replacing standard hash functions with incremental multiset hash functions in the hash tree. The potential advantage is that the siblings of a node will not have to be read to update the node. However, as we shall see, care must be taken when implementing hash trees using incremental hash functions instead of standard hash functions. If just incremental hash functions are used, the approach is vulnerable to replay attacks. To prevent the replay attack, we augment the approach with time stamps. In our resulting scheme, when compared with the typical hash tree, though stores can consume less bandwidth, loads will consume more bandwidth because of the time stamps. The checker will also need to have an extra operation to reset time stamps periodically, which will consume bandwidth that would not have been consumed if the typical hash tree had been used.

Section 10.2 studies an approach in which the `trace-hash-check` operation is modified such that checker requests that the untrusted storage read the addresses that the FSM used, instead of the checker reading the addresses itself. The untrusted storage computes a single multiset hash and sends this to the checker to add to its ReadHash for the check. The potential advantage is that the `trace-hash-check` operation would become much cheaper even for frequent checks. However, as we will see, the approach is vulnerable to attacks.

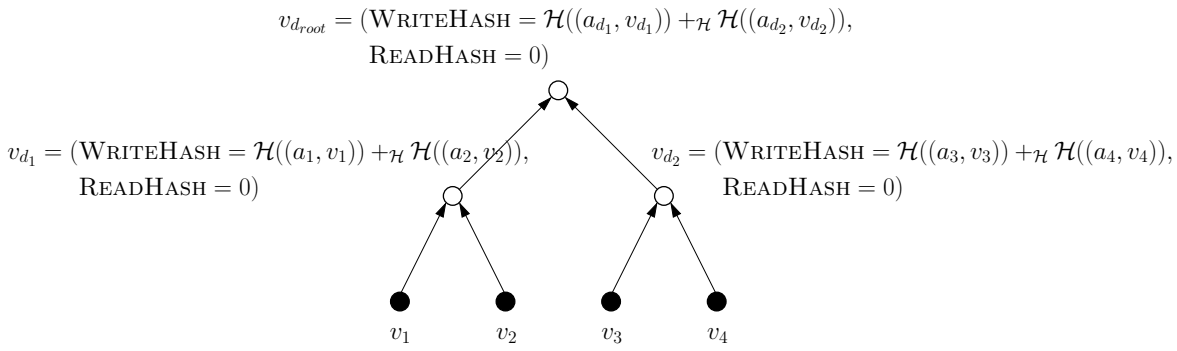


Figure 10-1: Initial state of broken version of an incremental hash tree

10.1 Incremental Hash Trees

10.1.1 Broken version

Consider a binary hash tree. A typical hash tree uses a standard hash function [1, Chapter 9] to create the collision resistant hash of the concatenation of the data that is in each of the children; the operations to check a node's integrity and to update a node are as described in Section 4.1. Consider replacing the standard hash function with an incremental multiset hash function. Figure 10-1 illustrates the approach. Each internal node contains a WriteHash and ReadHash. Consider that `Encrypted-Set-XOR-Hash` (cf. Section 5.5.3) is used. Initially, each internal node's WriteHash is $P_{k'}(H_k(a_1.v_1) \oplus H_k(a_2.v_2))$, where a_1 and a_2 are the addresses of the internal node's first and second child nodes respectively, and v_1 and v_2 are the data values at a_1 and a_2 respectively. Initially, each internal node's ReadHash is 0.

The operation to check the integrity of a node is similar to that in the typical hash tree. To check the integrity of a node, the checker:

1. reads the node and its siblings,

2. for each of the node and its siblings, creates an (address, data value) pair, and computes an incremental multiset hash of the (address, data value) pairs; denote this hash as childHash.

3. using the parent's WriteHash and ReadHash, checks that:

$\text{WriteHash} \equiv_{\mathcal{H}_k} \text{ReadHash} +_{\mathcal{H}_k} \text{childHash}$.

The steps are repeated on the parent node, and on its parent node, all the way to the root of the tree.

The operation to update a node is different than that in the typical hash tree, with the advantage that the node's siblings are not read when a node is updated. To update a node, the checker:

1. reads the node. *The node's siblings are not read* (note that since the checker does not read the node's siblings, it cannot check the integrity of the node's data value),

2. creates an (address, data value) pair using the node's address and the node's current value, and computes $\mathcal{H}_k$ on the pair; denote this hash as childHash,

3. creates an (address, data value) pair using the node's address and the node's new value, and computes H_k on the pair; denote this hash as $\text{childHash}_{\texttt{new}}$,

4. changes the parent's data value: using the parent's WriteHash and ReadHash,
   updates ReadHash: $\text{ReadHash} +_{\mathcal{H}} = \text{childHash}$,
   and updates WriteHash: $\text{WriteHash} +_{\mathcal{H}} = \text{childHash}_{\texttt{new}}$.

5. writes the node's new data value to the node in the untrusted RAM.

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 update's the path from a's data value leaf to the trusted root.

As it is, the scheme that is presented does not work because the checker does not check the integrity of a node when it is updating the node [3, Section 5.5]. Suppose that, at address a_1, the checker:

1. updates the address's current data value, $v_{1_{\texttt{old}}}$, with a new data value, $v_{1_{\texttt{new}}}$, then,

2. subsequently checks the integrity of the data value at address a_1.

Let $\mathtt{w}(.)$ denote a value that the checker writes to RAM and $\mathtt{r}(.)$ denote a value that the checker reads from RAM. Then, the check that the checker performs in Step 3 of the node integrity check operation when it checks the integrity of the data value at address a_1 is, essentially, whether:

$$H_k(a_1.\mathtt{w}(v_{1_{\mathtt{old}}})) \oplus H_k(a_1.\mathtt{w}(v_{1_{\mathtt{new}}})) = H_k(a_1.\mathtt{r}(v_{1_{\mathtt{old}}})) \oplus H_k(a_1.\mathtt{r}(v_{1_{\mathtt{new}}})) \ .$$

If the RAM behaves like valid RAM, then $H_k(a_1.\mathtt{w}(v_{1_{\mathtt{old}}}))$ matches $H_k(a_1.\mathtt{r}(v_{1_{\mathtt{old}}}))$, and $H_k(a_1.\mathtt{w}(v_{1_{\mathtt{new}}}))$ matches $H_k(a_1.\mathtt{r}(v_{1_{\mathtt{new}}}))$.

Attack 1: Unfortunately, there are other terms that can match. The checker's check also passes if $H_k(a_1.\mathtt{r}(v_{1_{\mathtt{old}}})) = H_k(a_1.\mathtt{w}(v_{1_{\mathtt{new}}}))$ and $H_k(a_1.\mathtt{w}(v_{1_{\mathtt{old}}})) = H_k(a_1.\mathtt{r}(v_{1_{\mathtt{new}}}))$. This means that if an adversary correctly predicts the new value for the address, $\mathtt{w}(v_{1_{\mathtt{new}}})$, the adversary can send this to the checker when the checker reads from the address during its update operation. The adversary can then replay the old value of the address, $\mathtt{w}(v_{1_{\mathtt{old}}})$, when the checker subsequently reads the node and checks its integrity during a load operation. The checker's check will pass, even though the adversary has replayed the old value of the address.

Attack 2: Because the $\oplus$ operation is used to update WriteHash and ReadHash, the checker's check also passes if $H_k(a_1.\mathtt{w}(v_{1_{\mathtt{old}}})) = H_k(a_1.\mathtt{w}(v_{1_{\mathtt{new}}}))$ and $H_k(a_1.\mathtt{r}(v_{1_{\mathtt{old}}})) = H_k(a_1.\mathtt{r}(v_{1_{\mathtt{new}}}))$. This means that, if the checker updates the node with the same value, the adversary can choose any value that it wants to give the checker to read during a load operation. The checker's check will pass, even though the adversary has tampered with the value of the address.

Section 10.1.2 shows how to fix the scheme by associating a time stamp with each node in the untrusted RAM. A node's time stamp is kept in its parent node's data value (cf. Figure 10-2). Thus the parent node's data value consists of a WriteHash, ReadHash and a time stamp for each of the children. The parent node's data value's WriteHash and ReadHash are computed over the (address, data value, time stamp) triples corresponding to the children.

Each time a node is updated, the checker increments the node's time stamp. Considering

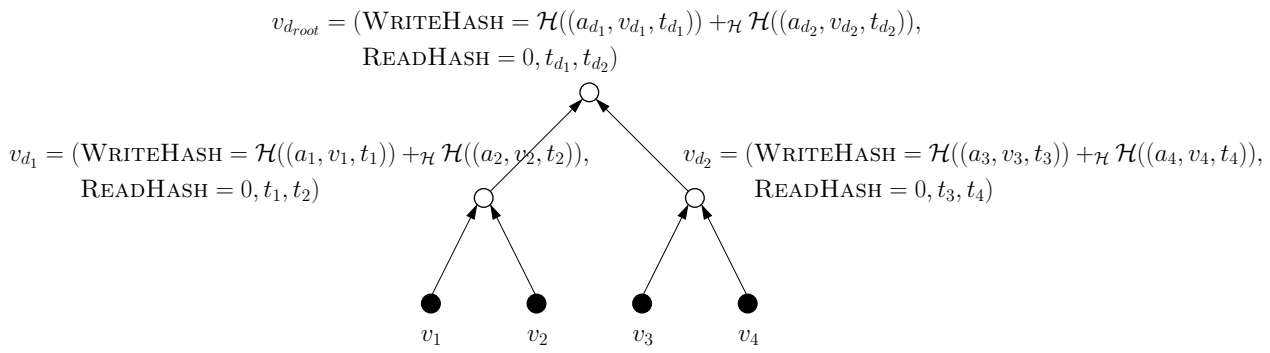


Figure 10-2: Initial state of `iHashTree`

the attacks described earlier, with the time stamps, the check that the checker performs in Step 3 of the node integrity check operation when it checks the integrity of the data value at address a_1 is whether:

$$H_k(a_1.\mathtt{w}(v_{1_{\mathtt{old}}}).t_{1_{\mathtt{old}}}) \oplus H_k(a_1.\mathtt{w}(v_{1_{\mathtt{new}}}).t_{1_{\mathtt{new}}}) = H_k(a_1.\mathtt{r}(v_{1_{\mathtt{old}}}).t_{1_{\mathtt{old}}}) \oplus H_k(a_1.\mathtt{r}(v_{1_{\mathtt{new}}}).t_{1_{\mathtt{new}}}) \,.$$

The purpose of the time stamp is to prevent the $v_{1_{\mathtt{old}}}$ terms from ever being identical to the $v_{1_{\mathtt{new}}}$ terms. Thus, $v_{1_{\mathtt{old}}}$ terms can only match $v_{1_{\mathtt{old}}}$ terms, and $v_{1_{\mathtt{new}}}$ terms can only match $v_{1_{\mathtt{new}}}$ terms.

Because the time stamp is incremented when the node is updated, we know that $t_{1_{\mathtt{new}}} \neq t_{1_{\mathtt{old}}}$. Consider the first of the attacks described earlier. If the adversary tries the attack, the attack will fail because, when the hashes are checked in the load operation, $H_k(a_1.\mathtt{r}(v_{1_{\mathtt{old}}}).t_{1_{\mathtt{old}}}) \neq H_k(a_1.\mathtt{w}(v_{1_{\mathtt{new}}}).t_{1_{\mathtt{new}}})$. Consider the second attack. If the adversary tries the attack, the attack will fail because, when the hashes are checked in the load operation, $H_k(a_1.\mathtt{w}(v_{1_{\mathtt{old}}}).t_{1_{\mathtt{old}}}) \neq H_k(a_1.\mathtt{w}(v_{1_{\mathtt{new}}}).t_{1_{\mathtt{new}}})$. Section 10.1.2 details the approach.

10.1.2 `iHashTree` Checker

Figure 10-2 illustrates an example tree in the `iHashTree` scheme. Each non-leaf node in the tree consists of a data value consisting of a WRITEHASH, a READHASH, and a time stamp for each of the node's children.

Notation

Each node d consists of a data value v_d.

We denote the parent node of d as $\mathcal{P}(d)$. We denote $\mathcal{P}(d)$'s data value as $v_{\mathcal{P}(d)}$.

We denote d and its sibling nodes as the set $\mathcal{S}(d)$. We denote the set of $\mathcal{S}(d)$'s data values as $v_{\mathcal{S}(d)}$.

Each node d is also associated with a time stamp t_d. Time stamp t_d is kept in $v_{\mathcal{P}(d)}$.

Consider the case that d is a leaf. In this case, v_d is simply the data value that the FSM stored at the address.

Consider the case that d is a parent. We denote the set of children nodes of d as $\mathcal{C}(d)$. In this case,

$$v_d = (\text{WriteHash}, \text{ReadHash}, \forall_{i \in \mathcal{C}(d)} t_i).$$

As before, let `w`(.) denote a value that the checker writes to RAM and `r`(.) denote a value that the checker reads from RAM.

Interface

Each node d's WriteHash is initially $\sum_{\mathcal{H}\ i \in \mathcal{C}(d)} \mathcal{H}((a_i, v_i, t_i))$ and its ReadHash is initially zero. The root is stored in the fixed-sized trusted state in the checker. Figure 10-3 shows the basic `iHashTree-nodeIntegrityCheck`, `iHashTree-nodeUpdate` and `iHashTree-nodeUpdate` operations.

On each FSM load from address a, the checker calls `iHashTree-nodeIntegrityCheck` on each node along the path from a's leaf node to the trusted root. On each FSM store of value v to address a, the checker calls `iHashTree-nodeUpdate` on each node along the path from a's leaf node to the trusted root. We refer to these load and store operations as `iHash-tree-load`(a) and `iHash-tree-store`(a).

When a leaf node's time stamp has reached its maximum value, the checker must reset it. To reset the time stamp of the leaf node at address a, the checker first calls `iHash-tree-load`(a) to check the integrity of data value at address a. The checker then calls `iHashTree-nodeTimeStampReset` with this data value to update the node's parent's data value with a zero value for the node's time stamp. The checker repeats the

Let a_d be the address of node d in the untrusted RAM.

`iHashTree-nodeIntegrityCheck`(a_d): checks the integrity of d:

1. read $v_{\mathcal{S}(d)}$ and $v_{\mathcal{P}(d)}$ from the untrusted RAM,
2. for each $i \in \mathcal{S}(d)$, create a (a_i, v_i, t_i) triple using the data value in $v_{\mathcal{S}(d)}$ and the time stamp in $v_{\mathcal{P}(d)}$, and compute $\textsc{childHash} = \sum_{\mathcal{H}\ i \in \mathcal{S}(d)} \mathcal{H}((a_i, v_i, t_i))$.
3. using $v_{\mathcal{P}(d)}$'s WriteHash and ReadHash, check that:
   $\textsc{WriteHash} \equiv_{\mathcal{H}} \textsc{ReadHash} +_{\mathcal{H}} \textsc{childHash}$.

The steps are repeated on the parent node, and on its parent node, all the way to the root of the tree.

`iHashTree-nodeUpdate`($a_d, v_{d\texttt{new}}$): updates d to have data value $v_{d\texttt{new}}$:

1. read v_d and $v_{\mathcal{P}(d)}$ from the untrusted RAM. *d's siblings are not read* (note that since the checker does not read d's siblings, it cannot check the integrity of v_d),
2. create a (a_d, v_d, t_d) triple using t_d in $v_{\mathcal{P}(d)}$ and v_d, and compute $\textsc{childHash} = \mathcal{H}((a_d, v_d, t_d))$,
3. let $t_{d\texttt{new}} = (t_d + 1)$. Using $v_{d\texttt{new}}$ and $t_{d\texttt{new}}$, create a $(a_d, v_{d\texttt{new}}, t_{d\texttt{new}})$ triple and compute $\textsc{childHash}_{\texttt{new}} = \mathcal{H}((a_d, v_{d\texttt{new}}, t_{d\texttt{new}}))$,
4. change $v_{\mathcal{P}(d)}$: using $v_{\mathcal{P}(d)}$'s WriteHash, ReadHash and t_d
   update ReadHash: $\textsc{ReadHash} +_{\mathcal{H}}= \textsc{childHash}$,
   update WriteHash: $\textsc{WriteHash} +_{\mathcal{H}}= \textsc{childHash}_{\texttt{new}}$,
   update t_d: $t_d \mathrel{+}= 1$.
5. write $v_{d\texttt{new}}$ to the untrusted RAM.

The steps are repeated on the parent node, and on its parent node, all the way to the root of the tree.

`iHashTree-nodeTimeStampReset`($a_d, v_{d\texttt{new}}$): resets t_d:

1. `iHashTree-nodeIntegrityCheck`(a_d) (this checks the integrity of d's siblings).
2. compute $\textsc{childHash} = \sum_{\mathcal{H}\ i \in \mathcal{S}(d)} \mathcal{H}((a_i, v_i, t_i))$ where $v_d = v_{d\texttt{new}}$ and $t_d = 0$.
3. change $v_{\mathcal{P}(d)}$: using $v_{\mathcal{P}(d)}$'s WriteHash, ReadHash and t_d
   update ReadHash: $\textsc{ReadHash} = 0$,
   update WriteHash: $\textsc{WriteHash} = \textsc{childHash}$,
   update t_d: $t_d = 0$.
4. write $v_{d\texttt{new}}$ to the untrusted RAM.

The steps are repeated on the parent node, and on its parent node, all the way to the root of the tree.

Figure 10-3: Basic `iHashTree` operations

`iHashTree-nodeTimeStampReset` call using the parent's new value to update the parent's parent's data value with a zero value for the parent's time stamp. The `iHashTree-nodeTimeStampReset` call is repeated to the root of the tree. We refer to the operation that resets a leaf node's time stamp as `iHash-tree-timeStampReset`(a).

With regard to the size of the time stamps, the maximum possible value of a parent's time stamp must be greater than the sum of the maximum possible values of its children's time stamps. Suppose the time stamp of a leaf node is $b_{t_{leaf}}$ bits large. For each leaf node, the FSM can perform $2^{b_{t_{leaf}}}$ store operations on the node before the checker must perform an extra `iHash-tree-timeStampReset` operation to reset the node's time stamp.

Proof of Security of `iHashTree` Checker

Theorem 10.1.1. *Assume that, for all of the nodes, the checker resets the node's time stamp when (or before) the node's time stamp reaches its maximum value. The RAM has behaved like valid RAM if and only if all of the checks pass.*

Proof. We assume that, for all of the nodes, the checker resets the node's time stamp when (or before) the node's time stamp reaches its maximum value. If the untrusted RAM has behaved like valid RAM, it is easy to verify that all of the checks pass. Suppose that the untrusted RAM has not behaved like valid RAM. We will prove that a check will fail.

Lemma 10.1.2. *Consider a node d. Assume that $v_{\mathcal{P}(d)}$ has not been tampered with. If v_d is tampered with, then a check will fail.*

Proof. Because we assume that $v_{\mathcal{P}(d)}$ has not been tampered with, we assume that, for each $i \in \mathcal{S}(d)$, t_i has not been tampered with.

Let R_d be the multiset of triples read from node d since the checker last reset t_d. Let W_d be the multiset of triples written to node d since the checker last reset t_d. Because R_d is only updated in the `iHashTree-nodeUpdate` operation (and reset in the `iHashTree-nodeTimeStampReset` operation), and because d's time stamp has not been tampered with and is incremented each time d is updated, each triple in R_d is unique and R_d forms a set. For a similar reason, each triple in W_d is unique and W_d forms a set. $\bigcup_{i \in \mathcal{S}(d)} R_i$ hashes to $v_{\mathcal{P}(d)}$'s ReadHash; $\bigcup_{i \in \mathcal{S}(d)} W_i$ hashes to $v_{\mathcal{P}(d)}$'s WriteHash. Because $\bigcup_{i \in \mathcal{S}(d)} R_i$

and $\bigcup_{i \in \mathcal{S}(d)} W_i$ are each sets, the multiset hash function used to update ReadHash and WriteHash simply needs to be set-collision resistant (cf. Section 5.5.3).

If the adversary tampers with v_d, when the checker performs the check in step 3 of the node integrity check operation during a load operation, there will be a triple in R_d that will not match its corresponding triple in W_d. $W_d \neq R_d$ implies that WriteHash is not equal to ReadHash $+_{\mathcal{H}}$ childHash, or that a collision has been found in the multiset hash function. Thus, the check in step 3 of the node integrity check operation will fail when d's integrity is checked during a load operation (or a collision will have been found in the multiset hash function). This concludes the proof of Lemma 10.1.2.

□

Theorem 10.1.1 follows by induction by noticing that $v_d =$ (WriteHash, ReadHash, $\forall_{i \in \mathcal{C}(d)} t_i$) for non-leaf nodes and v_d is equal to the FSM's data value for leaf nodes, and that the root's data value cannot be tampered with because it is stored in trusted state. □

Analysis

We compare the performance of `iHashTree` with that of a typical hash tree. We refer to the typical hash as `hashTree`.

We first consider a tree in the `iHashTree` scheme. Consider a complete m-ary tree, with $m = 2^c$. Let h be the height of the tree (the length of the path from the root to the leaf in the tree). Let $b_{t_{leaf}}$ be the number of bits of a leaf node's time stamp. For efficiency, each node can maintain the difference between WriteHash and ReadHash instead of two multiset hashes. For simplicity, let us assume that leaf nodes' data values and (WriteHash−ReadHash) are the same size; let b_d be the number of bits of a leaf node's data value.

The maximum possible value of a parent's time stamp must be greater than the sum of the maximum possible values of its children's time stamps. Let b_{t_0} be the number of bits of a time stamp of a node at depth h in the tree, b_{t_1} be the number of bits of a time stamp of a node at depth $h-1$ in the tree, etc. $b_{t_0} = b_{t_{leaf}}$ and $b_{t_{i+1}} = \lceil \log_2(m * 2^{b_{t_i}}) \rceil = (c + b_{t_i})$. Thus, by induction, $b_{t_i} = b_{t_{leaf}} + (i * c)$.

Let $\beta_{path_{\texttt{iHashTree}}}$ be the bandwidth consumed by reading a path in `iHashTree` from the leaf to the topmost node. $\beta_{path_{\texttt{iHashTree}}} = b_d + \sum_{i=0}^{h-2}(b_d + m(b_{t_{leaf}} + (i * c))) = hb_d + (h-1)mb_{t_{leaf}} + mc\sum_{i=0}^{h-2} i = hb_d + (h-1)mb_{t_{leaf}} + \frac{mc(h-1)(h-2)}{2}$. Let $\beta_{\texttt{iHashTreeStore}}$ be the bandwidth consumed by a `iHash-tree-store` operation. $\beta_{\texttt{iHashTreeStore}} = 2\beta_{path_{\texttt{iHashTree}}}$. Let $\beta_{\texttt{iHashTreeLoad}}$ be the bandwidth consumed by a `iHash-tree-load` operation. $\beta_{\texttt{iHashTreeLoad}} = m\beta_{path_{\texttt{iHashTree}}}$. Let $\beta_{\texttt{iHashTreeReset}}$ be the bandwidth consumed by a `iHash-tree-timeStampReset` operation. $\beta_{\texttt{iHashTreeReset}} \approx (m+1)\beta_{path_{\texttt{iHashTree}}}$.

Let f_s, $0 \leq f_s \leq 1$, be the average frequency of store operations per node (the frequency of store operations is defined as $\frac{\text{number of store operations}}{\text{number of store and load operations}}$). For each leaf node, the FSM can perform $2^{b_{t_{leaf}}}$ store operations on the node before the checker must perform an extra `iHash-tree-timeStampReset` operation to reset the node's time stamp. Thus, a `iHash-tree-timeStampReset` operation occurs every $\frac{1}{2^{b_{t_{leaf}}}}$ store operations on a particular node.

Let $\mathcal{B}_{\texttt{iHashTree}}$ be the average bandwidth per store or load operation consumed by `iHashTree`. $\mathcal{B}_{\texttt{iHashTree}} = f_s\beta_{\texttt{iHashTreeStore}} + (1-f_s)\beta_{\texttt{iHashTreeLoad}} + \frac{f_s}{2^{b_{t_{leaf}}}}\beta_{\texttt{iHashTreeReset}}$ $= \beta_{path_{\texttt{iHashTree}}}(\frac{2^{b_{t_{leaf}}}2f_s + 2^{b_{t_{leaf}}}m(1-f_s) + (m+1)f_s}{2^{b_{t_{leaf}}}})$.

We now consider a similarly-arranged tree in the `hashTree` scheme. Let $\beta_{path_{\texttt{hashTree}}}$ be the bandwidth consumed by reading a path in `hashTree` from the leaf to the topmost node. $\beta_{path_{\texttt{hashTree}}} = hb_d$. Let $\beta_{\texttt{hashTreeStore}}$ be the bandwidth consumed by a `hash-tree-store` operation. $\beta_{\texttt{hashTreeStore}} = 2m\beta_{path_{\texttt{hashTree}}}$. Let $\beta_{\texttt{hashTreeLoad}}$ be the bandwidth consumed by a `hash-tree-load` operation. $\beta_{\texttt{hashTreeLoad}} = m\beta_{path_{\texttt{hashTree}}}$. Let $\mathcal{B}_{\texttt{hashTree}}$ be the average bandwidth per store or load operation consumed by `hashTree`. $\mathcal{B}_{\texttt{hashTree}} = f_s\beta_{\texttt{hashTreeStore}} + (1-f_s)\beta_{\texttt{hashTreeLoad}} = \beta_{path_{\texttt{hashTree}}}(mf_s + m)$.

If $\frac{\mathcal{B}_{\texttt{hashTree}}}{\mathcal{B}_{\texttt{iHashTree}}} > 1$, then `iHashTree` performs better than `hashTree`; otherwise `hashTree` performs better than `iHashTree`. Thus, if

$$\frac{(mf_s + m)2^{b_{t_{leaf}}}}{2^{b_{t_{leaf}}}2f_s + 2^{b_{t_{leaf}}}m(1-f_s) + (m+1)f_s} > \frac{\beta_{path_{\texttt{iHashTree}}}}{\beta_{path_{\texttt{hashTree}}}},$$

then `iHashTree` performs better than `hashTree`. In this case, we see that, for $b_{t_{leaf}}$ sufficiently-large (so that `iHash-tree-timeStampResets` are not frequent), as f_s increases, `iHashTree`'s

gain over `hashTree` increases.

10.2 Intelligent Storage

In the trace-hash scheme in Chapter 7, the checker reads the addresses that the FSM used to perform the `trace-hash-check` operation. Suppose that `MSet-Mu-Hash` (cf. Section 5.4), which does not use a secret key, is used to update the checker's multiset hashes in the scheme. Consider a new `trace-hash-check` operation, `trace-hash-check′`, in which the checker requests that the untrusted storage read the addresses that the FSM used, instead of the checker reading the addresses itself. The untrusted storage computes the multiset hash of the (address, data value, time stamp) triples corresponding to the addresses that it reads, then sends this hash to the checker. The untrusted storage is able to compute the hash because `MSet-Mu-Hash` does not use a secret key. The checker performs the check by adding the hash to ReadHash and checking that WriteHash is equal to ReadHash. The advantage of `trace-hash-check′` is that the bandwidth consumption of the check is significantly reduced to primarily just the size of the hash. The `trace-hash-check′` operation is very cheap even for frequent checks.

Unfortunately, the scheme does not work. Let W be the multiset of triples that the checker writes to the untrusted storage. Let R be the multiset of triples that the checker reads from the untrusted storage. To successfully attack the scheme, an adversary tampers with a data value that the checker reads from the storage and maintains the multisets $(W - R)$ and $(R - W)$. Because the untrusted storage can calculate the hashes of multisets, the adversary can also calculate the hashes of multisets. When the checker performs `trace-hash-check′`, the adversary calculates $h_1 = \mathcal{H}(W - R)$ and $h_2 = \mathcal{H}(R - W)$, then $h_3 = (h_1 -_{\mathcal{H}_k} h_2)$. (In the case of `MSet-Mu-Hash`, $h -_{\mathcal{H}_k} h' = (h \times h'^{-1} \bmod p)$). The adversary sends h_3 to the checker for the checker to add to ReadHash. The checker's check will pass even though the adversary has tampered with a value read from the storage.

With the `trace-hash-check′` operation, the adversary is able to successfully attack the scheme because he is able to remove triples from R that are in R but not in W, as well as add triples to R that are in W but not in R, after he has tampered with a value that the

checker has read from the storage. In the original `trace-hash-check` operation, the checker reads the addresses in the untrusted storage and adds the triples to R itself. The adversary is unable to remove any triples from R and any attacks he tries will be detected.

Chapter 11

Conclusion

In this chapter, we present tradeoffs a system designer may consider making when using the adaptive tree-trace checker and the applicability of the adaptive tree-trace checker. We then conclude the thesis.

11.1 Tradeoffs

In this section, we discuss some of the tradeoffs a system designer may consider making when implementing the adaptive tree-trace checker in his system. We focus particularly on the data structures that are used for bookkeeping (cf. Section 8.2.2) and on the cache simulators (cf. Section 9.2.3).

The thesis has described implementing the adaptive tree-trace checker with two types of bookkeeping data structures: a range, where the checker moves a range of addresses to the trace-hash scheme and maintains the highest and lowest addresses of the range in its fixed-sized trusted state; and a bitmap maintained unprotected in the RAM, where the checker incurs the extra cost of reading a bitmap segment to determine which scheme an address is in, and incurs the extra cost of reading and updating the bitmap when performing a `tree-trace-check` operation. As described in Section 8.2.2, the hash tree and trace-hash schemes implicitly encode the necessary security information and a system designer is free to choose any data structures that allow the checker to most efficiently perform its bookkeeping. Using a range is very effective when there is spatial locality and a computer

architect may consider using ranges for the system's stack and heap. A bitmap is a more general bookkeeping data structure. If the architect uses a bitmap, the bitmap could be on a block granularity or on a larger granularity, such as on a page granularity. If there are unused bits in the page table, the architect may consider using page table bits for the bookkeeping. For software or hardware systems, the system designer may consider using a hierarchical bitmap which will make `tree-trace-moveToTraceHash` operations slightly more costly, but can significantly reduce the cost of finding the addresses in the trace-hash scheme during a `tree-trace-check` operation.

The cache simulators in Section 9.2.3 are being used to help guarantee the worst-case bound on the adaptive tree-trace checker when the FSM uses a cache. Though the simulators 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-trace operations. The tree-trace simulator could then 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. The hash tree simulator could then 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-trace strategies (cf. Section 9.2.2), that would work well in practice in different system implementations.

11.2 Applicability

In this section, we discuss the applicability of the adaptive tree-trace checker to other systems besides secure processors. Hash trees have also been used the check the integrity of data in software systems [14, 28, 30], where typically the client is trusted but the server and the data it contains are not trusted. A system designer may consider using the adaptive tree-trace

checker if the overhead of the hash tree checker is a performance bottleneck and applications can perform sequences of data operations before performing an operation critical to the security of the system. As the performance metric, the system designer may choose to use bandwidth overhead, to reduce network traffic, or may choose to use the number of RPC calls, to reduce the network latency experienced by applications.

In software systems, the storage access pattern may be more random than that in hardware systems. For each of the trace-hash checker and the adaptive tree-trace checker, the same general trends that are experienced with access patterns with spatial locality should be experienced with random access patterns, for the same reasons. For the trace-hash checker, if the number of store and load operations performed by the client is large, the amortized cost of reading the storage to perform the `trace-hash-check` operation is very small and the principal bandwidth overhead is the constant-sized runtime overhead of reading and writing time stamps, which are also very small. When check periods are large, the trace-hash checker should perform well. If the check period is small, the amortized cost of the `trace-hash-check` operation is more costly and the performance of the trace-hash checker will be poor. The adaptive tree-trace checker will be guaranteed to have a bandwidth overhead of no more than $(1+\omega)$ of the hash tree bandwidth overhead (if bandwidth overhead is the performance metric). Clients whose performance improves when the trace-hash scheme is used will experience the constant bandwidth overhead asymptotic performance. Even if a bitmap is used for bookkeeping and a large part of the bitmap needs to be read to perform a `tree-trace-check` operation, the amortized cost of reading and updating the bitmap during the `tree-trace-check` operation will be small when check periods are large, and the principal overhead will be the constant runtime bandwidth overhead of the time stamps and the bitmap segments. We observe that, with a random access pattern, the hash tree is likely to be more expensive than with an access pattern with spatial locality because the checker is likely to have to fetch more hashes from the storage before it finds a hash in the cache. Thus, there is the potential for the adaptive tree-trace checker to have a greater improvement with a random access pattern than with an access pattern with spatial locality when the checker experiences the constant bandwidth overhead asymptotic performance.

11.3 Summary

We have introduced a trace-hash scheme and an adaptive tree-trace scheme, which each have the feature that, for all programs, the scheme's bandwidth overhead approaches a constant bandwidth overhead as the average number of times the program accesses data between critical operations increases. The trace-hash scheme is well-suited for certified execution applications where a critical operation tends to occur after a billion or more memory operations. The adaptive tree-trace scheme is designed to be a general-purpose integrity checker. The adaptive tree-trace scheme 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. The general trend is that the greater the hash tree bandwidth overhead, the greater will be the adaptive tree-trace scheme's improvement when the scheme improves the checker's performance.

Bibliography

[1] Alfred J. Menezes and Paul C. van Oorschot and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press LLC, 1997.

[2] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[3] Blaise Gassend and G. Edward Suh and Dwaine Clarke and Marten van Dijk and Srinivas Devadas. Caches and Merkle Trees for Efficient Memory Integrity Verification. In *Proceedings of the* 9^{th} *International Symposium on High Performance Computer Architecture*, pages 295–306, February 2003.

[4] Daniele Micciancio. The RSA Group is Pseudo-Free. In *Advances in Cryptology - Eurocrypt 2005 Proceedings*, volume 3494 of *LNCS*, pages 387–403. Springer-Verlag, May 2005.

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

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

[7] Dwaine Clarke and G. Edward Suh and Blaise Gassend and Ajay Sudan and Marten van Dijk and Srinivas Devadas. Towards Constant Bandwidth Overhead Integrity Checking of Untrusted Data. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 139–153, May 2005.

[8] Dwaine Clarke and Srinivas Devadas and Marten van Dijk and Blaise Gassend and G. Edward 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.

[9] Eric Hall and Charanjit S. Jutla. Parallelizable Authentication Trees. In *Cryptology ePrint Archive*, December 2002.

[10] G. Edward Suh and Dwaine Clarke and Blaise Gassend and Marten van Dijk and Srinivas 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.

[11] G. Edward Suh and Dwaine Clarke and Blaise Gassend and Marten van Dijk and Srinivas 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.

[12] Hugo Krawczyk and Mihir Bellare and Ran Canetti. RFC 2104: HMAC: Keyed-Hashing for Message Authentication, February 1997. Status: INFORMATIONAL.

[13] Jan Camenisch and Anna Lysyanskaya. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In *Advances in Cryptology - Crypto 2002 Proceedings*, volume 2442 of *LNCS*, pages 61–76. Springer-Verlag, August 2002.

[14] Jinyuan Li and Maxwell Krohn and David Mazières and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the* 6^{th} *Symposium on Operating Systems Design and Implementation*, pages 91–106, December 2004.

[15] John L. Hennessy and David A. Patterson. *Computer Organization and Design*. Morgan Kaufmann Publishers, Inc., 1997.

[16] John L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, July 2000.

[17] Joseph Naor and Moni Naor. Small-Bias Probability Spaces: Efficient Constructions and Applications. In 22^{nd} *ACM Symposium on Theory of Computing*, pages 213–223, May 1990.

[18] Josh Benaloh and Michael de Mare. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In *Advances in Cryptology - Eurocrypt 1993 Proceedings*, volume 765 of *LNCS*, pages 274–285. Springer-Verlag, May 1993.

[19] Jun Yang and Youtao Zhang and Lan 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.

[20] Manuel Blum and Will Evans and Peter Gemmell and Sampath Kannan and Moni Naor. Checking the Correctness of Memories. In *Algorithmica*, volume 12, pages 225–244, August 1994.

[21] Mihir Bellare and Daniele Micciancio. A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost. In *Advances in Cryptology - Eurocrypt 1997 Proceedings*, volume 1233 of *LNCS*, pages 163–192. Springer-Verlag, May 1997.

[22] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the* 1^{st} *Annual Conference on Computer and Communications Security*, pages 62–73, November 1993.

[23] Mihir Bellare and Roch Guérin and Phillip Rogaway. XOR MACs: New Methods for Message Authentication using Finite Pseudorandom Functions. In *Advances in Cryptology - Crypto 1995 Proceedings*, volume 963 of *LNCS*, pages 15–28. Springer-Verlag, August 1995.

[24] Niko Barić and Birgit Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In *Advances in Cryptology - Eurocrypt 1997 Proceedings*, volume 1233 of *LNCS*, pages 480–494. Springer-Verlag, May 1997.

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

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

[27] Ronald L. Rivest. On the Notion of Pseudo-Free Groups. In *Theory of Cryptography Conference - Proceedings of TCC 2004*, volume 2951 of *LNCS*, pages 505–521. Springer-Verlag, February 2003.

[28] Sean Rhea and Patrick Eaton and Dennis Geels and Hakim Weatherspoon and Ben Zhao and John Kubiatowicz. Pond: the OceanStore Prototype. In *Proceedings of the* 2^{nd} *USENIX Conference on File and Storage Technologies (FAST '03)*, pages 1–14, March 2003.

[29] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 1994.

[30] Umesh Maheshwari and Radek Vingralek and William Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the* 4^{th} *Symposium on Operating Systems Design and Implementation*, pages 135–150, October 2000.