Algorithms for Verifying the Integrity of Untrusted Storage

by

Ajay Sudan

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

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2004

Author .....
Department of Electrical Engineering and Computer Science
January 29, 2004

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

Accepted by .....
Arthur C. Smith
Professor of Electrical Engineering and Computer Science
Chairman, Department Committee on Graduate Students

Algorithms for Verifying the Integrity of Untrusted Storage

by

Ajay Sudan

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2004, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This work addresses the problem of verifying that untrusted storage behaves like valid storage. The problem is important in a system such as a network file system or database where a client accesses data stored remotely on an untrusted server. Past systems have used a hash tree-based checker to check the integrity of data stored on untrusted storage. This method has high overhead as the tree must be traversed on each load or store operation. In the offline approach, developed by Clarke et al. in [6], multiset hashes are used to verify a sequence of load and store operations. The overhead of this scheme is very low if checks are infrequent, but can be quite high if checks are performed frequently. The hybrid scheme combines the advantages of the two schemes and is efficient in most real world situations.

The various schemes were implemented on top of Berkeley DB, an embedded database. Real world performance measurements were taken using OpenLDAP, a lightweight directory service, which relies heavily on Berkeley DB. All read and writes to the database were replaced with secure read and secure write operations. Using the DirectoryMark LDAP test suite, the online scheme had an overhead of 113% when compared to the an unmodified server, while the offline scheme with infrequent checks (T=50000) resulted in 39% fewer DOPS. The offline scheme, however, outperformed the online scheme by 31%, while the hybrid scheme outperformed the online scheme by only 19%. In the worst case, when checks were frequent (T=500), the hybrid scheme was 185% slower (65% fewer DOPS) than the online scheme. With frequent checks, the offline scheme was 101% slower (50% fewer DOPS) than the online scheme.

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

Acknowledgments

This work was funded by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership, and by DARPA through the Office of Naval Research under contract number N66001-99-2-891702.

I would like to thank my thesis advisor, Srinivas Devadas, for his guidance, advice, and support. His energy and enthusiasm were a catalyst for the many great ideas which came out of our weekly "meetings," from which I was graciously excused after the first 3.5 hours.

I am indebted to Dwaine Clarke and Edward Suh, my office mates, without whom this thesis would not have been possible. Thank you both for being patient with me, explaining your ideas clearly, answering my countless questions, and helping me overcome any problems that I had.

I also greatly appreciate the support that I have received from all my friends and family. Thank you to all those who kept checking in to make sure things were going well. I am grateful to my roommate, Stuart Blitz, for driving me to and from my office every morning at 9 AM during the subzero temperatures in January.

And finally, I would like to thank my parents and Emily Fuchs, my biggest fan, for all their love and support and without whom I would not be where I am today.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

In today's networked world, users are increasingly trying to protect against attacks from malicious adversaries. Large, remote, untrusted storage devices may be used to house vast quantities of information. Ensuring that the data retrieved has been stored correctly and has not been tampered with is a real-world problem. Previous schemes of protecting the integrity of untrusted storage have relied on computationally intensive algorithms relying on collision-resistant cryptographically secure hash trees. These schemes are expensive and can have a debilitating effect on performance. More efficient algorithms are needed if widespread adoption of such techniques is expected.

Corporations frequently outsource their data storage needs to third-party vendors without any guarantee that the data is in fact being stored securely. The remote storage servers may be placed in locations such that they are vulnerable to physical attacks, hardware malfunctions, or software attacks by a malicious third party. Databases containing corporate information, social security and credit card numbers, medical records, and credit histories may all be stored remotely. It is imperative that the integrity of such data be verified prior to its use.

Integrity verification schemes are not only applicable to databases, but also to file systems and even a computer's memory. A computer's RAM may be untrusted and

susceptible to attack. In applications such as certified execution, it is necessary to ensure that the requested computation was executed correctly on the processor and that the RAM behaved correctly. Tampering with memory can cause incorrect results to be produced and may allow an adversary to circumvent protection mechanisms, such as in DRM applications.

The goal of this thesis is to implement three different integrity verification schemes to measure the amount of overhead incurred and gauge the relative performance advantages of each scheme in a real-world application with real workloads.

1.2 Related Work

A number of systems have addressed the problem of verifying data stored on untrusted storage. The Byzantine fault-tolerant system, BFS [4], relies on replication in order to ensure the integrity of a network file system. BFS allows up to 1/3 of the servers to be compromised, while still ensuring that any data read from the file system is legitimate.

Other systems, such as SUNDR, do not require any replication, placing trust requirements on the client side [14]. SUNDR and BFS also differ in their freshness guarantees. SUNDR relies on large client-side caches and utilizes hash trees, which are used to protect disk blocks at the file system level.

The OceanStore file system [1] assigns a unique "GUID" handle to each file. For immutable files, the GUID is a collision-resistant cryptographic hash of the file's contents. A client can verify the contents of the file using the GUID. For mutable files, the GUID is the hash of a public key and username. The file's contents are digitally signed with a private key corresponding to the GUID, which are mapped to the file's name using SDSI. Unlike SUNDR which signs entire file systems, OceanStore signs individual files and therefore may be susceptible to replay attacks.

SFSRO [8] provides secure read-only access to file systems without the use of public key infrastructure. SFSRO replicates data on a number of untrusted servers, and guarantees data integrity even if an adversary compromises any one of the read-

only servers.

Maheshwari's TDB uses hash trees and a limited amount of trusted storage to create a trusted database on untrusted storage [13]. TDB employs hash trees and validates the database against a collision-resistant root hash kept in trusted storage.

1.3 Organization

This thesis is structured as follows. Chapter 2 gives an overview of three different algorithms for verifying the integrity of untrusted storage. Chapters 3-5 examine the three algorithms in greater detail. First, online and offline schemes are explored in Chapters 3 and 4, followed by an examination of a hybrid scheme in Chapter 5.

Once we have familiarized ourselves with the three schemes, in Chapter 6, we take a look at the implementation details and issues that arise when putting theory into practice.

Chapter 7 analyzes the three schemes using real-world performance benchmarks. Finally, we wrap things up in Chapter 8 with some concluding remarks and suggestions for future research.

Chapter 2

Integrity Verification Schemes

This thesis addresses the problem of verifying that untrusted storage behaves like valid storage. Storage behaves like valid storage if the data value that a program loads from a particular address is the same data that the program most recently stored to that address. The problem is important in a system, such as a network file system [14] or a database [13], where a client computer accesses data stored remotely on an untrusted server.

Typically, systems use a hash tree-based checker to check the integrity of data stored on the untrusted storage. A tree of hashes is maintained over the data, and the root of the tree is kept in the client's protected store. On each store operation, the path from the data leaf to the root is updated. On each load operation, the path from the data leaf to the root is verified before the data is treated as valid. The hash tree is used to check, after *each* load operation, whether the untrusted storage performed correctly. Because of this, a checker which uses a hash tree is referred to as an *online* checker.

In the offline approach, developed by Clarke et al. [6, 7, 18], the client uses *multiset hash functions* [6] to efficiently log the minimum information necessary to check its operations on the untrusted storage. Since only the minimum information is collected at program runtime, the performance overhead of the scheme can be very small and the offline scheme can be very efficient when integrity checks are performed

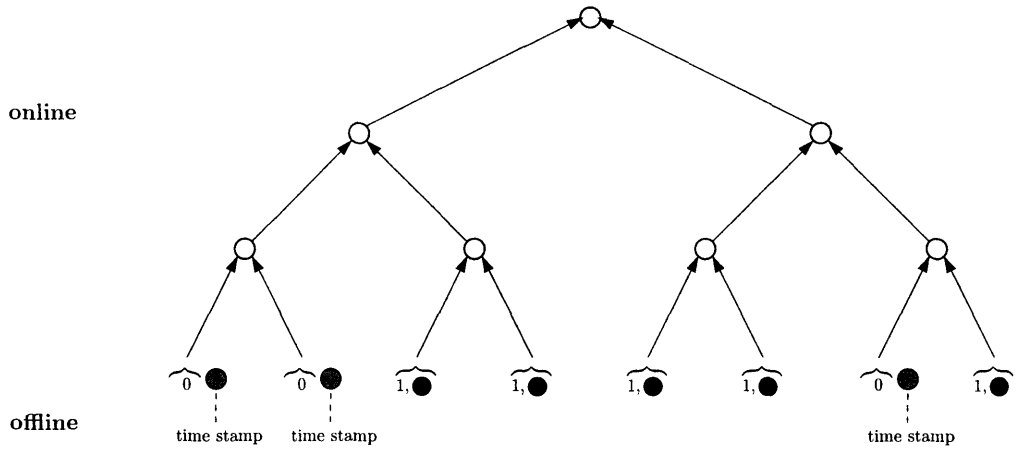

Figure 2-1: A hybrid checker. There is an extra bit, called a STATUSBIT, associated with each leaf node. STATUSBITs are always protected by the tree. If a leaf node's STATUSBIT is 1, the data value is protected by the online scheme; if a leaf node's STATUSBIT is 0, the data value is protected by the offline scheme.

infrequently. However, because the checker needs to read all the pages it used to perform the check, the scheme can perform poorly if checks are frequent.

The online checker can be combined with the offline checker to create a hybrid online-offline checker. This approach is illustrated in Figure 2-1.

In the hybrid approach, the checker maintains both a hash tree and the offline multiset hashes and timer. Data is initially stored in the hash tree. When the client wants to perform a particular computation on a subset of the data in the storage, it can either work on the data in an online fashion using the tree, or it can take the data out of the tree and work on it in an offline fashion. When the client performs an intermediate offline check, the client only needs to read addresses that were used since the last intermediate check, rather than having to read all of the addresses it used since the beginning of execution. If the intermediate offline integrity check is successful, data can then be returned to the tree. Per-address STATUSBITS are used to identify whether a particular data value is protected by the online or offline scheme.

When the client works on data in an offline fashion, the hash tree acts as a repository for the data values, while the offline scheme gives the client some work space for it to perform its computation. To work on a subset of the data in the

storage, the client checks the data out of the repository and operates on it in the work space. When the client has completed the computation, it checks the computation's operations, and then exports the result of the computation. The client can then deposit the data back into the repository.

The client can dynamically employ several strategies during its execution that would maximize its performance. For example, the client may protect data it uses regularly with the offline scheme and protect data it uses rarely using the hash tree, thereby reducing the number of addresses that are read to perform an offline integrity check. Alternatively, if the client will regularly be exporting results during some part of its execution, the client can protect the data using the online scheme, which has less overhead when integrity checks are frequent. If the client performs a computation for which it will not export a result for some time, the data that is being used can be moved to the offline scheme, which performs better when integrity checks are less frequent.

The security of the offline and hybrid approaches have been proven by Clarke et al. in [6]. A client using the hybrid checker is guaranteed to detect the attack if the data value it loads from an address is not the most recent value it stored to that address.

In the following chapters, we take a closer look at each of the three schemes.

Chapter 3

Online Scheme

The *online* scheme is referred to as such since the integrity of the untrusted storage is verified after *each* load or store operation. The underlying data structure for the online scheme is a hash tree, or Merkle tree [15]. In a hash tree, data is located at the leaves of a tree. Each node contains a collision resistant hash of the data in the child nodes. The root hash is stored in secure, tamper-resistant memory on the client and is the only item which must be protected.

To check that a node or leaf in a hash tree has not been tampered with, on each load operation, the path from the data leaf to the root is verified before the data is treated as valid. On each store operation, the path from the data leaf to the root is updated. Hash trees have been used for integrity checking in many systems such as Duchamp, BFS, SFSRO, and TDB.

Clarke et al. describe the online algorithm in [6] as follows:

To check the integrity of a node in the tree, the checker:

1. reads the node and its siblings
2. concatenates their data together ($\alpha . \beta$ is the concatenation of strings α and β.)
3. hashes the concatenated data
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 in the tree, the checker:

1. authenticates the node's siblings 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. This step must be performed in a manner which makes changes to the node and its parent visible simultaneously (using the cache in the checker, say).

Again, the steps are repeated until the root is updated.

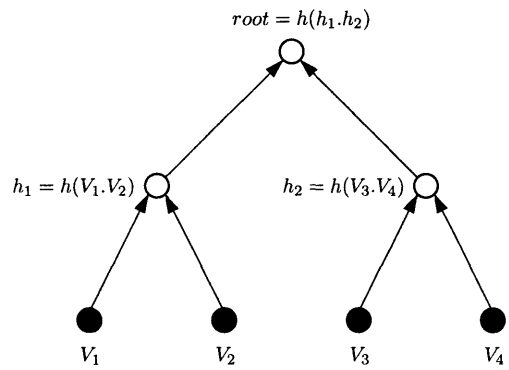

Figure 3-1: A binary ($m = 2$) hash tree. Each internal node is a hash of the concatenation of the data in the node's children.

Hash trees allow individual data values in an arbitrarily large amount of data to be checked and updated securely using a small, fixed-sized, trusted hash in the checker. On each FSM load, the checker checks the path from the leaf, corresponding to the address containing the data, to the trusted root. On each FSM store, the checker updates the path from the leaf to the trusted root. The number of accesses to the untrusted storage on each FSM load or store is logarithmic in the size of the storage. If the trusted hash were calculated directly over the data set, the overhead on each FSM load would be linear, because the entire storage would have to be read on each load.

With a balanced m-ary tree, the number of nodes to check on a FSM load is $log_m(N)$, where N is the number of leaves of data to be protected. If the checker frequently adds and deletes data values from the tree, steps may have to be taken to ensure that the tree remains balanced to maintain this logarithmic overhead.

If the size of a leaf is the size of a hash, an m-ary hash tree allows integrity verification with a per-bit space overhead of about $\frac{1}{m-1}$ bits. For example, in [9], a 4-ary hash tree is used, with the lowest level hashes computed over 64-byte data value blocks. Each hash is 128 bits (MD5 [17] is used for the hashes).

Chapter 4

Offline Scheme

The offline scheme was developed by Clarke et al. in [6] as an alternative to the online tree-based scheme described above. This scheme is an extension of Blum's offline memory correctness checking scheme [2] and is able to detect attacks by active adversaries. Clarke et al. introduce the incremental *multiset hash* as the underlying cryptographic tool for all offline operations. Building on the notion of *bag integrity checking*, Clarke et al. use this primitive to build cryptographically secure integrity checking schemes for random access memories and disks. In addition, Clarke et al. also introduce a hybrid checker which combines the advantages of both the online and offline schemes. The hybrid checker will be discussed further in Chapter 5.

The remainder of this chapter and the next contain an abbreviated explanation of the offline and hybrid schemes, excerpted from [6, 7] by Clarke et al. A more detailed treatment, including proofs and additional information on multiset hashes, can be found in [6, 7].

4.1 Introduction

The offline approach is used to check whether untrusted storage performed correctly after a *sequence* of operations is performed. The benefit of this approach is that there is a constant overhead on the number of storage accesses on each program load or

store.

Offline integrity checking can be particularly efficient in an application like certified execution. In certified execution, a program is run on a processor, and the processor produces a certificate proving that the computation was carried out in an authentic manner on the processor and that the program produced a particular set of results. In the application, the processor needs to know, at the end of the program's execution, whether the RAM performed correctly. However, in the case the check fails, it is not necessary to know which particular operation malfunctioned, or which stored value was tampered with. Certifying executions can be important in applications like distributed computation, where an expensive computation is carried out by several networked computers in a highly distributed manner. The person requesting the computation must have some assurances that, when he receives results, the results are of authentic program executions.

Offline integrity checking can also be useful in memory-constrained devices, since a hash tree is not required and the memory checking code and its resources (stack and heap) can be smaller. There is some space overhead to store time stamps, but it is typically smaller than the overhead of storing a tree of hashes.

4.2 Model

In our model, there is a checker that keeps and maintains some small, fixed-sized, trusted state. The untrusted storage 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 storage. The checker uses its trusted state to verify the integrity of the untrusted storage. The trusted computing base (TCB) consists of the FSM, and the checker with its trusted state. For example, the FSM could be a processor. The checker would be special hardware that is added to the processor to detect tampering in the external memory.

The checker checks if the untrusted storage behaves correctly, i.e., like valid storage. *Storage behaves like valid storage if the data value that the checker reads from a*

particular address is the same data value that the checker had most recently written to that address. In our model, the untrusted storage is assumed to be actively controlled by an adversary. The untrusted storage may not behave like valid storage if the storage has malfunctioned because of errors, or if it has been somehow altered by the adversary.

For this problem, a simple solution 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. The essence of an offline checker is that a "log" of the sequence of FSM operations is maintained in fixed-sized trusted state in the checker.

4.3 Bag Integrity Checking

We introduce bag integrity checking as our primitive for thinking about offline integrity checking. The scenario consists of the checker and a bag. The bag is the untrusted storage and the checker performs two operations on the bag:

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

The checker is interested in whether the bag behaves correctly. A bag behaves correctly if it behaves as a valid bag, a bag in which the sequence of puts and takes performed by the checker is a valid history for the bag (i.e., only the checker has manipulated the bag with put and take operations).

The checker's fixed-sized state is:

- 2 multiset hashes: PUTHASH and TAKEHASH. Initially both multiset hashes are 0:
- 1 counter: TIMER. Initially TIMER is 0.
- 1 flag: ERROR. Initially ERROR is `false`.

`put`($\mathcal{I}_s$) puts a set of items $\mathcal{I}_s$ into the bag:

1. Increment TIMER.
2. For each item $\in \mathcal{I}_s$, put the pair, (item, TIMER), into the bag.
3. For each item $\in \mathcal{I}_s$, update PUTHASH: PUTHASH $+_{\mathcal{H}}=$ hash((item, TIMER)).

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

1. Take all pairs that match $\mathcal{P}$ from the bag.
2. If, for any pair, (item, T), we have T>TIMER, then set ERROR to `true`.
3. update TAKEHASH: TAKEHASH $+_{\mathcal{H}}=$ hash((item, T)).

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

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

Figure 4-1: Bag Offline Integrity Checking

4.3.1 Checker with Multiset Hashes and Time stamps

This solution is described in Figure 4-1[1]. The checker maintains multiset hashes and has a counter. The checker increments the counter each time it puts a set of items[2] into the bag, and appends the new value of the counter (a time stamp) to each item

[1]We assume that `take`($\mathcal{P}$) is a deterministic function that removes all of the items in the bag that match predicate $\mathcal{P}$ and returns them to the caller; `take`(true) removes all of the items currently in the bag and returns them to the caller.

[2]Recall that, in a set, each element is distinct.

as it puts it into the bag. When the checker takes a multiset of items from the bag, for each item, it checks that the time stamp on the item is less than or equal to the current value of the counter.

The time stamp is included with the item when the multiset hashes are updated. The checker uses time stamps to help check that items it takes from the bag have been put into the bag by the checker at an earlier time.

Note that the put pairs that are added to PUTHASH are added by the checker and thus, we are guaranteed that they will form a set. The adversary can control the take pairs, and thus, take pairs can be duplicated. The set-collision resistance property implies that it is computationally infeasible to find a multiset of take pairs different from the set of put pairs that will result in PUTHASH being equal to TAKEHASH at the end of an integrity check.

The FSM uses the checker as an interface to the bag. The checker performs the put and take operations for the FSM as described in Figure 4-1. 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 (`take`(true)). At this point, the FSM performs a checker `check` operation. The `check` operation returns true if all of the time stamp checks have passed and PUTHASH is equal to TAKEHASH. If the check operation returns true, the FSM knows that the bag has behaved correctly and is empty (according to the bag's history of accesses). So, by asking the checker to compare the put hash and the take hash, the FSM determines whether the bag's history is valid.

With the checker's put and take operational primitives, we can build more complex operations for more complex data structures. Section 4.4 demonstrates how this can be done for random access storages, such as RAM and disks.

4.4 Integrity Checking of Random Access Storage

In Section 4.3, we developed a *bag checker*. It makes a checkable bag from an untrusted bag. Its interface is made of three functions: `put`(S), which places all the elements in the set S into the untrusted bag; `take`(P), which takes all the elements that match

predicate P from the untrusted bag; `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 and all the time stamp checks passed).

We now show how a checked bag can be used to create checked random access storage (RAS). We call this algorithm the offline algorithm because checks are performed after a sequence of storage accesses, rather than on each storage access.

Definition 1. Checked Random Access Storage *is a primitive that provides the following interface:* `store`(a, v) *stores data value* v *at address* a. `load`(a) *returns the data value that is stored at address* a. `checkRAS`() *returns true if and only if for each address* a *and each* `load` *from* a*, the* `load` *returned the data value that was most recently placed by* `store` *at address* a.

Figure 4-2 shows how to produce checked random access storage from a checked bag. Essentially, the random access storage is simulated by placing (address, data value) pairs in a checked bag. Therefore we must maintain the invariant, which we term the *RAS invariant*, that there is always exactly one pair in the bag for each address, according to the bag checker's operations on the bag. During initialization, a pair is placed in the bag for each address. To perform a `load`, the pair for the desired address is taken from the bag, inspected, and then put back into the bag (to maintain the RAS invariant). To perform a `store`, the pair previously in the bag for that address is taken from the bag, and replaced by the new pair. To check the bag, we empty it into a fresh bag, and once the old bag is empty, we `check` it before throwing it out.

In real life, the untrusted bag that the checked bag is based on is actually implemented with some untrusted random access storage (RAM or block storage for example). This untrusted bag is being accessed by the bag checker when we access the checked bag. Takes and puts of (address, data value) pairs to the checked bag result in takes and puts of (address, data value, time stamp) triples to the untrusted bag. If the untrusted bag behaves correctly, the invariant that there is exactly one (address, data value) pair per address carries over to (address, data value, time stamp) triples in the untrusted bag. Therefore, it is possible to implement the untrusted bag

Let P_a be a predicate that returns true on a pair (a', v) for which $a = a'$. To produce checked random access storage, we use a checked bag, in which we will place (address, data value) pairs. We will arrange to always have exactly one pair per address in the bag, according to the bag checker's operations on the bag. Therefore, we will assume that `take`(P_a) always returns exactly one pair (we can add an explicit check for this and set an ERROR flag if this does not happen).

Initialization the bag must be filled at startup.

1. `put`(S) into the checked bag, where S is a set of (address, data value) pairs that represents the initial state of the checked random access storage.

`store`(a, v) stores v at address a in the checked random access storage.

1. `take`(P_a) on the checked bag.
2. `put`(a, v) into the checked bag.

`load`(a) loads the data value at address a from the checked random access storage.

1. $(\cdot, v)$ =`take`(P_a) on the checked bag.
2. `put`(a, v) into the checked bag.
3. Return v to the caller.

`checkRAS`() returns true if and only if the storage has behaved correctly up until now.

1. Create a temporary checked bag T (call the current checked bag B).
2. S =`take`(true) from B.
3. If `check`() on B is false, then return false (the check failed).
4. `put`(S) into T.
5. $B = T$.
6. Return `true`.

Note that steps 2 and 4 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 4-2: Offline integrity checking of random access storage using a checked bag

using untrusted random access storage by storing (address, data value, time stamp) triples as (data value, time stamp) pairs stored at an address that is proportional to the address from the triple.

Unfortunately, when we implement the untrusted bag using untrusted random access storage, we implicitly limit the range that the time stamp can take. Consequently, it will be necessary to call `checkRAS` on the checked random access storage each time the time stamp reaches its maximum value. When `checkRAS` replaces the checked bag that is in use by a fresh one, we replace the high time stamp from the old bag by a low one in the new bag.

The advantage of this offline RAS checker over an integrity checker using a hash tree is that there is a constant overhead per FSM load and store, as compared with the logarithmic overhead of using a hash tree. However, the offline approach does require that all of the addresses that the FSM used be read whenever an integrity check is desired. (In Section 5, we optimize this requirement with a hybrid online-offline approach.)

In the following subsections, we explore the implementation issues involved in checking dynamically-changing, sparsely-populated address spaces.

4.4.1 Dynamically-changing address space

Thus far, we have looked at the problem of checking the integrity of fixed-sized RAS. However, in practice, it is often desirable to check a RAS with a dynamically-changing, sparsely-populated, address space.

To enable a dynamically-changing address space, we augment the RAS interface with two methods: `add`(S), and `remove`(a). `add`(S) calls `put`(S) to put, S, a set of (address, data value) pairs, into the bag; `remove`(a) calls `take`(P_a) on the bag. Moreover, we arrange to set an ERROR flag if `take`(P_a) does not return a singleton set, as was assumed in Figure 4-2; if the ERROR flag is set, `checkRAS` returns false[3]. The augmented interface is shown in Figure 4-3.

If, during the FSM's execution, the FSM wishes to increase its address space (for example, when the program increases its heap size), the FSM calls `add` on the new

[3]In an actual implementation, if the FSM performs an operation that causes `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 RAS. Therefore `check` will return false, and thus, `checkRAS` will return false. Also, in an actual implementation, `take`(P_a) will not return a set or multiset with two or more elements.

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

1. `put`(S) into the checked bag.

`remove`(a) removes address a from the address space.

1. `take`(P_a) on the checked bag.

`store`(a, v) stores v at address a in the checked random access storage.

1. `take`(P_a) on the checked bag.
2. `put`(a, v) into the checked bag.

`load`(a) loads the data value at address a from the checked random access storage.

1. $(\cdot, v)$ =`take`(P_a) on the checked bag.
2. `put`(a, v) into the checked bag.
3. Return v to the caller.

`checkRAS`() returns true if and only if the storage has behaved correctly up until now.

1. Create a temporary checked bag T (call the current checked bag B).
2. M =`take`(true) from B.
3. If `check`() on B is false, then return false (the check failed).
4. If `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. `add`(S) into T.
5. $B = T$.
6. Return `true`.

Note that steps 2 and 4 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.

Also, if `take`(P_a) does not return a singleton set, an ERROR flag is set; if the ERROR flag is set, `checkRAS` returns false.

Figure 4-3: Offline integrity checking of random access storage on a dynamically-changing, sparsely-populated, address space

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

`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 `checkRAS` operation. This approach for checking the integrity of a dynamically-changing address space is much simpler than checking the integrity of the space using a hash tree, which could require re-balancing the tree.

As we are now considering sparsely-populated address spaces, we re-define the RAS invariant to be that, according to the bag checker's operations on the bag, no `put` operation is performed on an address that is already present in the bag, and no `take` operation is performed on an address that is not present in the bag. It is possible for the FSM to use the checked RAS in a way that is not well-defined (by adding the same address twice, for example). Therefore, we will be particularly interested in FSMs whose implementations always maintain the RAS invariant on correctly-behaving RAS. We call this property the *FSM requirement.* The FSM maintains whatever data structures it needs to meet the FSM requirement[4] in either trusted or checked storage. We note that if the FSM requirement is met and the bag behaves like a valid bag, then the RAS invariant is maintained.

The addresses the FSM uses may be any arbitrary subset of the addresses in the storage. When an untrusted bag is implemented using RAS, as described in the beginning of Section 4.4, where (address, data value, time stamp) triples are stored as (data value, time stamp) pairs, there is the issue of determining which addresses to read in step 2 of a `checkRAS` function call. Implicitly, it is the untrusted bag's job to keep track of these addresses. As an example, the bag could use a bitmap (an extra bit per address) to keep track of the addresses the FSM uses[5]. We note that the bitmap does not have to be protected because it is an internal structure to the untrusted bag: `check`() returns true if and only if the untrusted bag has behaved correctly and is empty according to its history of accesses.

[4]For example, the pointer to the top of the heap.

[5]If the offline scheme is used to protect a process's virtual memory space, this bitmap could be the valid bits in a page table; in this case, addresses would be added or removed from the address space a page at a time.

4.5 Improved Offline Checker

We now take a look at a slightly modified version of the offline checker. In this improved checker, the TIMER is not incremented on every `load` and `store` operation. Rather, the TIMER is incremented only on each `put` operation, allowing time stamps to be smaller without increasing the frequency of checks.

Figure 4-4 shows the basic `put` and `get` operations that are used internally in the checker. Figure 4-5 shows the interface the FSM calls to use the offline checker to check the integrity of the memory.

In Figure 4-4, the checker maintains two multiset hashes and a counter. In memory, each data value is accompanied by a time stamp. Each time the checker performs a `put` operation, it appends the current value of the counter (a time stamp) to the data value, and writes the (data value, time stamp) pair to memory. When the checker performs a `get` operation, it reads the pair stored at an address, and, if necessary, updates the counter so that it is strictly greater than the time stamp that was read. The multiset hashes are updated ($+_{\mathcal{H}}$) with (a, v, t) triples corresponding to the pairs written or read from memory.

Figure 4-5 shows how the checker implements the `store-load` interface. To initialize the RAM, the checker `puts` an initial value to each address. When the FSM performs a `store` operation, the checker `gets` the original value at the address, then `puts` the new value to the address. When the FSM performs a `load` operation, the checker `gets` the original value at the address and returns this value to the FSM; it then `puts` the same value back to the address. To check the integrity of the RAM at the end of a sequence of FSM stores and loads, the checker `gets` the value at each address, then compares WRITEHASH and READHASH. If WRITEHASH is equal to READHASH, the checker concludes that the RAM has been behaving correctly.

Because the checker checks that WRITEHASH is equal to READHASH, substitution (the RAM returns a value that is never written to it) and replay (the RAM returns a stale value instead of the one that is most recently written) attacks on the RAM are prevented. The purpose of the time stamps is to prevent reordering attacks in which

The checker's fixed-sized state is:

- 2 multiset hashes: WRITEHASH and READHASH. Initially both hashes are 0.
- 1 counter: TIMER. Initially TIMER is 0.

put(a, v) writes a value v to address a in memory:

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

get(a) reads the value at address a in memory:

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

Figure 4-4: put and get operations

RAM returns a value that has not yet been written so that it can subsequently return stale data. Suppose we consider the put and get operations that occur on a particular address as occurring on a timeline. Line 3 in the get operation ensures that, for each store and load operation, each write has a time stamp that is strictly greater than all of the time stamps previously read from memory. Therefore, the first time an adversary tampers with a particular (data value, time stamp) pair that is read from memory, there will not be an entry in the WRITEHASH matching the adversary's entry in the READHASH, and that entry will not be added to the WRITEHASH at a later time.

The TIMER is not solely under the control of the checker, and is a function of what is read from memory, which is untrusted. Therefore, the WRITEHASH cannot be guaranteed to be over a set. For example, for a sequence of store and load operations occurring on the same address, an adversary can decrease the time stamp that is stored in memory and have triples be added to the WRITEHASH multiple times. The READHASH can also not be guaranteed to be over a set because the adversary controls the pairs that are read from memory. Thus, set-collision resistance is not sufficient,

`initialize()` initializes RAM.

1. put$(a, 0)$ for each address a.

`store`(a, v) stores v at address a.

1. get(a).
2. put(a, v).

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

1. $v =$ get(a). Return v to the caller.
2. put(a, v).

`check()` checks if the RAM has behaved correctly (at the end of operation).

1. get(a) for each address a.
2. If WRITEHASH is equal to READHASH, return true.

Figure 4-5: Offline integrity checking of random access memory

and we require multiset-collision resistant hash functions.

The original offline checker differs from the improved checker in that the TIMER is incremented on each put operation and is not a function of what is read from memory. The TIMER is solely under the control of the checker. This means that the pairs that are used to update WRITEHASH form a set. Therefore set-collision resistance is sufficient. Our offline checker improves on the original checker because TIMER is not incremented on every `load` and `store` operation. Thus, time stamps can be smaller without increasing the frequency of checks, which improves the performance of the checker.

Chapter 5

Hybrid Scheme

In Section 4.3, we constructed a *checked bag*, with the interface put($\mathcal{I}_s$), take($\mathcal{P}$), and check()[1], from an underlying *untrusted bag*, whose interface was standard bag put($\mathcal{I}_m$) and take($\mathcal{P}$) operations[2]. In Section 4.4, we used the checked bag to construct a *checked random access storage (RAS)*, with the interface add(S), remove(a), store(a, v), load(a), and checkRAS()[3]. We refer to the interface to the checked bag as a *bag checker* and the interface to the checked RAS as an *offline RAS checker (offline checker)*.

In this section, we will introduce a *hybrid RAS checker (hybrid checker)*, with the operations:
hybrid-moveToOffline(a), hybrid-store(a, v), hybrid-load(a), and *hybrid-checkRAS*(Y).
Figure 2-1 illustrates a hybrid checker. To construct the hybrid checker, we use the checked RAS, and a variant of the hash tree described in Chapter 3, which we call a *partial-hash tree*. Partial-hash trees are described in Section 5.2.

[1]In this section, we will refer to these operations as cbag-put($\mathcal{I}_s$), cbag-take($\mathcal{P}$) and cbag-check().

[2]In this section, we will refer to these operations as ubag-put($\mathcal{I}_m$) and ubag-take($\mathcal{P}$); I_m denotes a multiset of items.

[3]In this section, we will refer to these operations as offline-add(S), offline-remove(a), offline-store(a, v),
offline-load(a), and offline-checkRAS().

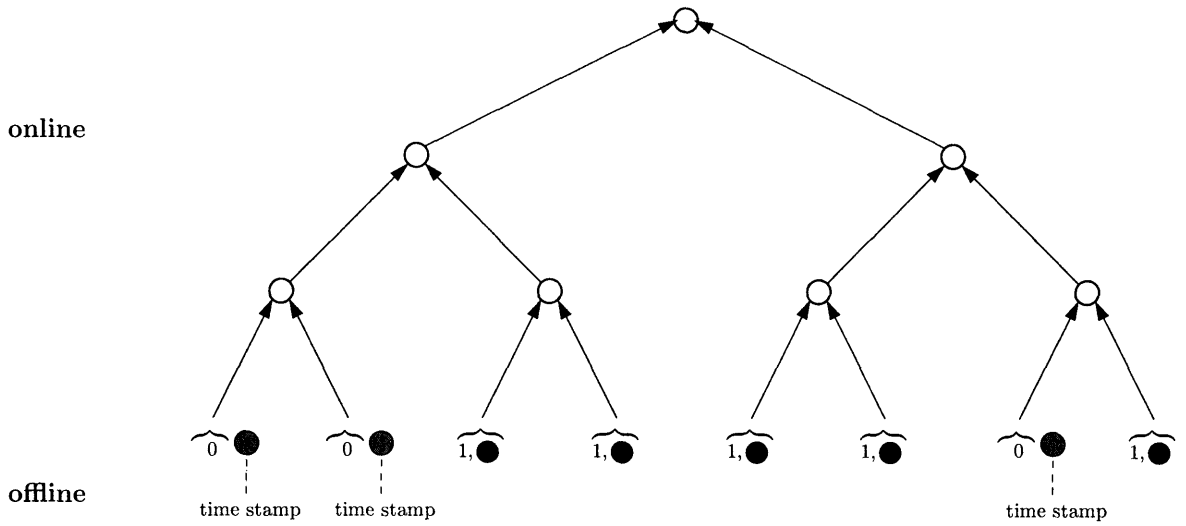

Figure 5-1: A hybrid checker. There is an extra bit, called a STATUSBIT, associated with each leaf node. STATUSBITs are always protected by the tree. If a leaf node's STATUSBIT is 1, the data value is protected by the online scheme; if a leaf node's STATUSBIT is 0, the data value is protected by the offline scheme.

5.1 Motivation for the hybrid checker

One of the disadvantages of the offline checker in Section 4.4 is that all of the addresses that the FSM used since the beginning of its execution must be read during each `offline-checkRAS` operation; otherwise, even if the RAS were behaving like valid RAS, the underlying untrusted bag would not be empty and the `cbag-check` operation would fail. This approach is feasible for RAM or small disks, but impractical for large-scale storage in file systems or databases. For large scale storage, it is desirable to use a scheme in which the FSM would only need to read addresses it used since the last integrity check when it is performing its current integrity check.

The second disadvantage of the offline checker is that, if integrity checks are frequent, the offline checker can perform worse than the online checker, because of the overhead it incurs reading the addresses it used since the beginning of its execution to perform the integrity check. It is desirable to construct a checker which could take advantage of the benefits of both the online and offline checkers; the FSM could use this checker to maximize its performance during its execution.

We propose using a hybrid RAS checker to address these issues. In the hybrid approach, the checker maintains both a hash tree and the offline multiset hashes and timer. Data is initially stored in the tree. The idea is that, when the FSM wants to perform a particular computation on a subset of the data in the storage, it can either work on the data in an online fashion using the tree, or it can take the data out of the tree, and work on it in an offline fashion. When the FSM performs an intermediate offline integrity check, the FSM only needs to read addresses that were used since the last intermediate check, instead of having to read all of the addresses it used since the beginning of its execution. If the intermediate offline integrity check is successful, data can then be returned to the tree.

With respect to when the FSM works on data in an offline fashion, the hash tree can be seen as acting as a repository for the data values, and the offline scheme gives the FSM some work space for it to perform some computation. To work on a subset of the data in the storage, the FSM checks the data out of the repository and operates on it in the work space. When the FSM has completed the computation, it checks the computation's operations, and then exports the result of the computation. The FSM can then deposit the data back into the repository.

The FSM can dynamically employ several strategies during its execution that would maximize its performance. As an example of a strategy the FSM might employ, if there is data the FSM regularly uses and data it uses rarely, it can protect the data it uses regularly with the offline scheme, and protect the data it uses rarely using the hash tree; this can reduce the number of addresses that are read to perform an offline integrity check. As a second example of an FSM strategy, if, during some part of its execution, the FSM will be regularly exporting results, the FSM can protect the data using the online scheme, which has a smaller overhead when integrity checks are very frequent; if, during some other part of its execution, the FSM performs a computation for which it will not export a result for some time, the data that is being used can be moved to the offline scheme, which performs better when integrity checks are less frequent.

5.2 Partial-Hash Tree

Besides the checked RAS developed in Section 4.4, we also use a partial-hash tree to develop the hybrid checker. A partial-hash tree is similar to a hash tree, except that there is an extra bit associated with each leaf node (recall that the leaf nodes contain data in a hash tree). The extra bit, which we refer to as a STATUSBIT, is always protected by the tree.

If a leaf node's STATUSBIT is set (equal to '1'), the leaf node is protected by the tree, and said to be 'present' in the tree. If a leaf node's STATUSBIT is not set (equal to '0'), the leaf node is not protected by the tree, and is said to be 'not present' in the tree.

The partial-hash tree interface has the following operations: `pht-isAddressPresent`(a), `pht-moveToTree`(a, v), `pht-moveFromTree`(a), `pht-store`(a, v), and `pht-load`(a); the interface is described in Figure 5-2. The operations `pht-store`(a, v), and `pht-load`(a) simply call `online-store`(a, v), and `online-load`(a) respectively; `online-store`(a, v), and `online-load`(a) are the hash tree store and load operations described in Chapter 3.

Considering the layout of the tree, an address and its STATUSBIT can be protected by the same hash when the address is present in the tree. When the address is not present in the tree, the hash protects the STATUSBIT.

5.3 Hybrid Checker

The interface for the hybrid checker is shown in Figure 5-3. At first, we consider a fixed-sized RAS. The operations are: `hybrid-moveToOffline`(a), `hybrid-store`(a, v), `hybrid-load`(a), and `hybrid-checkRAS`(Y).

`hybrid-moveToOffline`(a) checks the integrity of the data value in address a in the tree, and moves the data from the protection of the tree to the protection of the offline scheme. `hybrid-store`(a, v) first reads (but does not check) a's STATUSBIT in the partial-hash tree. If it is 1, it checks the STATUSBIT and performs

`pht-isAddressPresentInTree`(a) returns true if and only if address a is present in the tree.

1. check the integrity of the STATUSBIT pertaining to address a in the manner described in Chapter 3.
2. if the STATUSBIT is 1, return `true`. If it is 0, return `false`.

`pht-moveToTree`(a, v) move address a to the partial-hash tree, and set its value to be v.

1. update the STATUSBIT pertaining to address a, in the manner described in Chapter 3, to be 1.
2. update the appropriate nodes so that the leaf node for address a is protected by the tree and has the value v. The nodes are updated in the manner described in Chapter 3. (This step can be performed at the same time as the previous step, so that the logarithmic cost of using the hash tree is incurred once.)

`pht-moveFromTree`(a) remove address a from the partial-hash tree.

1. update the STATUSBIT pertaining to address a, in the manner described in Chapter 3, to be 0.
2. update the appropriate nodes so that the internal nodes of the tree no longer protect the leaf node for address a. The nodes are updated in the manner described in Chapter 3. (This step can be performed at the same time as the previous step, so that the logarithmic cost of using the hash tree is incurred once.)

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

1. `online-store`(a, v)

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

1. v = `online-load`(a)
2. Return v to the caller.

Figure 5-2: Partial-hash tree

an online store; if it is 0, it performs an offline store. `hybrid-load` performs similarly. `hybrid-checkRAS`(Y) checks the data protected by the offline scheme. Addresses specified in Y are moved back to the protection of the online scheme. If the

`offline-checkRAS` call in `hybrid-checkRAS` returns `true`, `hybrid-checkRAS` returns true; otherwise, `hybrid-checkRAS` returns `false`.

Compared with the checked RAS described in Section 4.4, the new adversarial attack we must consider is that an adversary can change the STATUSBIT of an address from 1 to 0 and alter the data value corresponding to the address in the offline scheme. Because the STATUSBIT is not checked if it is 0, the FSM could do an offline load (or offline store) on this value, when it should have done an online load (or online store). However, it can be proven that this attack, and other attacks on the storage, will be detected by the hybrid checker.

As described in Section 4.4.1, there is the issue of determining which addresses to read in step 2 of the `offline-checkRAS` call in `hybrid-checkRAS`. Again, we argue that it is the underlying untrusted bag's job to keep track of these addresses, and thus, the data structures used to maintain this information do not have to be protected. One possibility is to maintain an extra bit per hash tree node. These bits are all initially one. When an address is moved to the offline scheme in `hybrid-moveToOffline`, the bits from the address's leaf node to the root are set to 0. In `hybrid-checkRAS`, the tree is traversed in either a depth-first or breadth-first manner to determine which addresses are in the offline scheme. The appropriate bits are reset to 1 when addresses are moved back into the online scheme.

5.3.1 Hybrid Add and Remove operations

The operations `hybrid-add`(a, v, f) and `hybrid-remove`(a) could be added to the interface in Figure 5-3. `hybrid-add` adds an address to the online scheme if f (flag) is 1, and to the offline scheme if f is 0. `hybrid-remove` checks to determine if the address to be removed is in the online or offline scheme; the address is then removed from the appropriate scheme.

- For `hybrid-add`,
  - If the address is being added to the protection of the online scheme, the appropriate nodes for the address and its STATUSBIT are added to the

partial-hash tree, with the STATUSBIT being set to 1. The address can be operated on with `hybrid-store` and `hybrid-load`. The address's value can be moved to the offline scheme using `hybrid-moveToOffline`.

- If the address is being added to the protection of the offline scheme, it is added as described in Section 4.4.1. The appropriate nodes for the STATUSBIT and leaf node for the address are added to the tree, with the STATUSBIT being protected by the tree and set to 0. The address can be operated on with `hybrid-store` and `hybrid-load`. `hybrid-checkRAS` can be used to move the address's value to the online scheme.

- For `hybrid-remove`,

  - If the address to be removed is currently being protected by the online scheme, the appropriate nodes for the address and its STATUSBIT are removed from the partial-hash tree.

  - If the address to be removed is currently being protected by the offline scheme, it is removed with respect to one of the manners described in Section 4.4.1. The appropriate nodes for the address and its STATUSBIT are removed from the partial-hash tree.

5.3.2 Space Considerations

We provide some discussion on the space layout of the hybrid scheme. To implement the hybrid scheme the layout of data values, STATUSBITS, hashes and time stamps should be determined. Data values should be stored at the addresses, as usual. Given an address, it should be easy for the checker to compute the location of its STATUSBIT. When the checker reads either the data value at an address or the address's STATUSBIT, it is likely to be more efficient if the checker reads both from the storage together.

Given an address it should be easy for the checker to compute the location of its time stamp. Also, given a node in the hash tree, it should be easy for the checker to

compute the location of its parent.

For memory, one possible space layout would be for the part of the storage that is addressable by the FSM (i.e., the part which would store data values) to be at the top of the storage. The non-FSM-addressable part of the storage would contain STATUSBITS, internal hash tree nodes, and time stamps. For storage in a file system or database, the STATUSBITS and time stamps could be part of a data object's meta data. The internal hash tree nodes could be in the non-FSM-addressable part of the storage.

Compared with the hash tree described in Chapter 3, the extra space overhead is the STATUSBITS and time stamps. As described in [6], the size of a time stamp can be small, relative to the size of a hash. Thus, the space overhead of the hybrid scheme should not be much larger than that of the online scheme.

`hybrid-moveToOffline`(a) move address a to the offline scheme.

1. Call `pht-isAddressPresentInTree`(a).
   (a) If false, a is already present in offline scheme. Thus, simply return to the caller.
   (b) If true, a is present in online scheme. a must be moved to the offline scheme.
      i. Call v = `pht-load`(a) to check integrity of the data value stored at a.
      ii. Call v = `pht-moveFromTree`(a). (Steps can be performed at the same time, using a cache, so logarithmic cost of using the hash tree is incurred once.)
      iii. Call `offline-add`(a, v).

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

1. Read STATUSBIT pertaining to address a. If STATUSBIT is 1, a is in the online scheme.
   (a) Call `pht-isAddressPresentInTree`(a). If it returns true, continue; otherwise, return an error to the caller.
   (b) `pht-store`(a, v). (Step can be performed at the same time as the previous step so logarithmic cost of using the hash tree is incurred once.)
2. if the STATUSBIT is 0, a is in the offline scheme.
   (a) Call `offline-store`(a, v).

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

1. Read STATUSBIT pertaining to address a. If STATUSBIT is 1, a is in the online scheme.
   (a) Call `pht-isAddressPresentInTree`(a). If it returns true, continue; otherwise, return an error to the caller.
   (b) v = `pht-load`(a). (Step can be performed at the same time as the previous step so logarithmic cost of using the hash tree is incurred once.)
   (c) Return v to the caller.
2. if the STATUSBIT is 0, a is in the offline scheme.
   (a) Call v = `offline-load`(a, v).
   (b) Return v to the caller.

`hybrid-checkRAS`(Y) returns true if and only if the storage (currently being used by the offline scheme) has behaved correctly; each of the addresses in Y is moved back into the tree.

1. Call `offline-checkRAS`(). In Step 4 of `offline-checkRAS`, if $a \in Y$, do not put it into bag T. Instead,
   (a) Call `pht-moveToTree`(a, v), where v is the data value at a.
2. If Step 1 returned `true`, return `true`; otherwise return `false`.

Figure 5-3: Hybrid Offline-Online RAS Checker (Hybrid Checker)

Chapter 6

Implementation

Now that we are familiar with the various schemes for integrity verification, we will describe an implementation of all three schemes and measure their relative performance advantages. Integrity verification is relevant in several different contexts including untrusted memory, network file systems, and databases. Previous experiments [10] have presented hardware mechanisms to verify the integrity of memory in a secure processor.

We will now examine the software architecture of SecureDB, an implementation of the three integrity verification schemes as they are applied to databases. SecureDB is a C++ library which can be quickly and easily incorporated into any application which currently makes use of the Berkeley DB embedded database. SecureDB provides secure get() and put() operations which will verify the integrity of all data stored in a database. It can be configured to use the online, offline, or hybrid schemes for integrity verification. SecureDB is compatible with the Berkeley DB API 4.1.25 and simply requires that all Berkeley DB get() and put() calls be replaced with calls to SecureDB's sget() and sput() calls, respectively. Like Berkeley DB, SecureDB supports the storage and retrieval of key/value pairs of arbitrary length.

Thousands of applications utilize Berkeley DB to provide fast, scalable, and reliable data management. Berkeley DB is integrated into such programs as Sendmail and MacOS X Server and is in use by companies such as Akamai, Alcatel, Ama-

zon.com, AOL, AT&T, Cisco, Google, HP, Motorola, and Sun.

6.1 Models and Assumptions

SecureDB was written with two models in mind, as shown in Figure 6-1. SecureDB assumes that Berkeley DB is running in client-server mode on untrusted storage. In this mode, the database server accepts requests via IPC/RPC, and issues calls to the Berkeley DB interfaces based on those requests. The database server is the only application linking the Berkeley DB library into its address space. The client-server model trades performance for protection. Since the size of the hash tree created by SecureDB is dependent on the number of items being protected, the hash tree is stored on a remote NFS server. Traversal of the tree requires communication between SecureDB and the NFS file server. The SecureDB server can handle requests from multiple clients, such as a PDA, via a simple get/put interface.

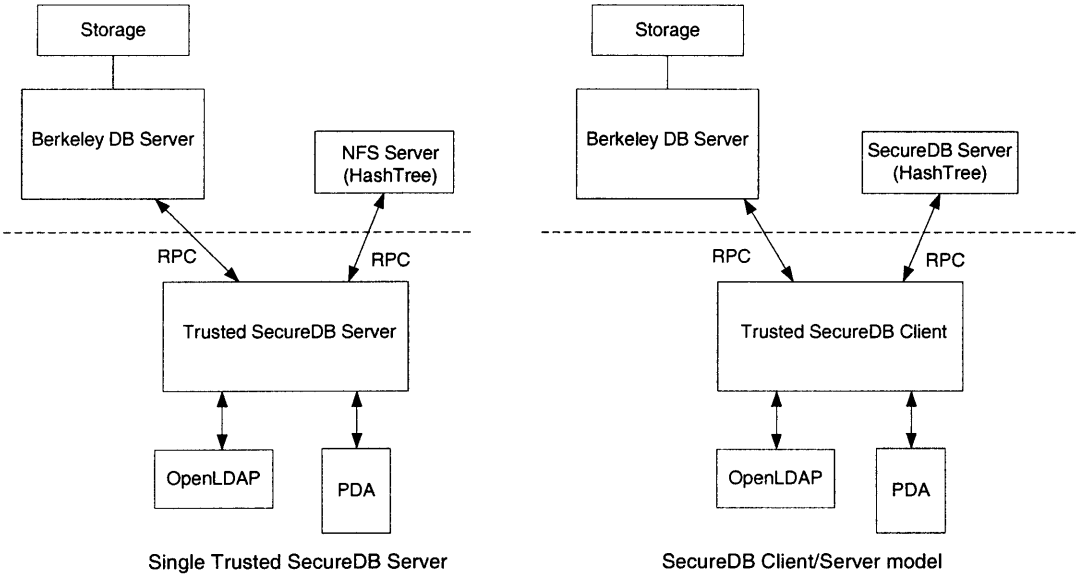

Figure 6-1: SecureDB Models

The two models differ in that, in the first, SecureDB runs on a single trusted server with a limited amount of memory. The trusted SecureDB server is responsible for maintaining the hash tree and the root hash. In the second model, SecureDB

is split into a client and server application. The trusted SecureDB client maintains the root hash, while the untrusted SecureDB server maintains the entire hash tree. The advantage of the former scheme is that we can guarantee that the hash tree has been constructed properly, eliminating the possibility of a denial of service attack (data element does not exist). In the latter scheme, more complicated algorithms are required to protect against denial of service since we are unable to guarantee that our code is run properly on the untrusted server.

6.2 Design Goals and Metrics

SecureDB was created with the following design goals in mind:

- *Strong security*: system should be secure and should detect all instances of data corruption or tampering of the untrusted store.
- *High reliability*: system should perform reliably and should reliably perform database operations on behalf of the user program.
- *Low client-side memory requirements*: we assume clients have only a limited amount of memory.
- *Low communication overhead*: communication between client and server should be kept at a minimum.

We can formalize the above goals as the algorithmic goal of minimizing the following:

- space used by the data structures maintained by client and server
- time required to get/put elements from the database
- amount of data required to get/put elements from the database

6.3 Data Structures

The online and hybrid schemes both require the use of a *hash tree.* A hash tree stores the elements of a set in the leaves of the tree. All internal nodes contain a collision-resistant cryptographic hash (typically MD5 or SHA-1) of the child nodes. In SecureDB, an open-source implementation of SHA-1 is used. The ordering of the leaves and all connectivity information in the tree must be known by the client so that the client can recompute and validate the hash value for the root.

As mentioned in Section 4.4.1, the offline scheme must maintain a data structure to track which keys are under its protection. This data structure can be unprotected and stored on untrusted storage. Tampering of the data structure will be detected by the `checkRAS()` function. SecureDB maintains a timestamp database which serves a dual purpose. First, as its name implies, the timestamp database maintains all the timestamps for every key being protected by the offline scheme. Second, the timestamp database is used to determine which keys are protected by the offline scheme.

Hash trees have previously been created using a variety of data structures including binary trees, 2-3 trees, and skip lists [12, 16]. Other variations on hash trees have been proposed in [3, 11]. The SecureDB implementation makes use of a 2-3 tree as its primary data structure.

6.4 2-3 Trees

In our implementation, we use 2-3 trees, despite warnings by Naor and Nissim that dynamic 2-3 trees are non-trivial to program correctly [16]. Our 2-3 tree implementation supports dynamic addition but not dynamic deletion of elements. As described above, each leaf stores an element while each internal node contains a hash of its children's values. In a 2-3 tree, each node can have exactly two or three children, which are classified as being a left child, a middle child, or a right child.

2-3 search trees are part of a larger classification of data structures called dictio-

naries, which support insertion, deletion, and searching of key/value pairs. 2-3 search trees were first introduced in 1970 as an improvement to balanced binary trees and were later generalized to B-trees. They were later simplified to form red-black trees [5].

2-3 trees maintain the following invariants:

- All data appears at the leaves
- Data elements are ordered from left to right
- Every path through the tree is the same length
- Interior nodes have two or three subtrees

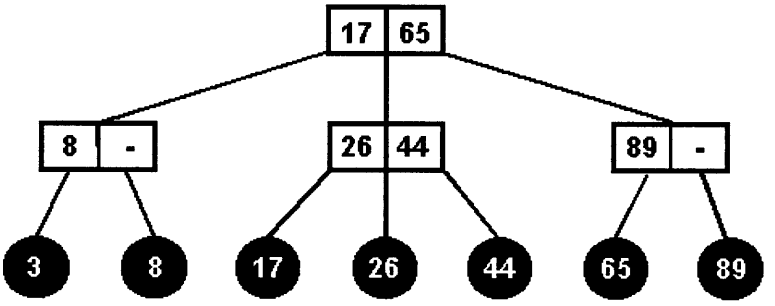

Figure 6-2: A 2-3 search tree. Data elements appear in the leaves while meta-data is stored in interior nodes [5].

6.4.1 Searching

A tree with N data items will always have a height between $log_3 N$ and $log_2 N$. Searching requires a single traversal along a path from the root to a leaf, and is thus bounded by the height of the tree. Interior nodes do not contain data but rather contain meta-data about the keys stored in the nodes' subtrees. Each interior node contains an *mlow* and *rlow* field, indicating the smallest key in the middle and right subtrees, respectively. These are used to efficiently traverse the tree and locate a desired key. Searching requires only traversing those subtrees that contain the desired element. The complete searching algorithm is given in [5].

6.4.2 Insertion

Insertion into a 2-3 tree can be a complex process and involves two different cases. Insertion involves finding the correct location for insertion, splitting nodes if necessary, and finally updating meta-data along a path from the newly inserted leaf to the root.

Every element can be inserted into one and only one correct location since elements must remain ordered from left to right. Finding this location involves using the meta-data to traverse the tree from the root until the appropriate leaf node is found. If the parent has just two children, then insertion is relatively painless. We can simply determine if the new element should become the left, middle, or right child, and adjust the meta-data along a path from leaf to root accordingly.

If, however, the parent of the leaf already has three children, then additional work is required. The three children and the new element to be inserted must be split into two interior parent nodes, each having two children. The exact location of the split depends on the key being inserted and must maintain the property that elements should remain ordered from left to right. After all interior node values have been updated, the newly created parent node with its two children can be attached.

If the preceding level in the tree also already has 3 children, then this process of splitting nodes must continue until a node with just two children is found. If the insertion propagates to the root of the tree, it is possible that a new root must be created to accommodate the new node.

6.5 `HashTree` class

The 2-3 tree described above was implemented as a C++ class. The `HashTree` class consists of a root node and supports the dictionary operations `insert()` and `find()`. Each node element in the tree contains a key, the corresponding data value, the hash of the node's children, a status bit, and pointers to the left, middle, and right children.

`HashTree` provides the following public API:

- `get();`
- `getRoothash();`
- `getStatusBit();`
- `insert();`
- `moveToOffline();`
- `getNextQupule();`
- `printout();`

6.5.1 Qupules

Qupules are the primary structure used to communicate information from the HashTree on the server to the client. As client-server communication is typically via an RPC mechanism, it is imperative that pointers in the server's memory space not be returned to the client. Instead, actual data values must be returned to the client. This is achieved through use of the Qupule. A Qupule sent from the server to the client contains the data values for the left, middle, and right child, the hash of the three children (parent hash), and positioning information indicating where the Qupule is attached to the parent. The client can traverse the tree upwards, toward the root, by calling `getNextQupule()`. The client must initiate a traversal by first calling `initiateInsert()` and passing the key of a node from which traversal should begin.

6.5.2 Client-Server Synchronization

Insertion of a new key/value pair by a user program requires two things to occur. First, the client must simulate the insertion and calculate a new root hash, making sure to verify that all values it uses in the calculation have not been tampered with. The client does this through the repeated use of Qupules from the point of insertion to the root. The verification and insertion are done in parallel to minimize the cost

of traversing the tree. The client maintains an oldHash and newHash during the traversal, using the old hashes for verification. After the client has calculated the new root hash, the server must also perform the insertion and update its hash tree to reflect the addition of the new element. After an insertion operation has completed, the client and server root hashes should match.

Communication between the client and server is currently simulated. Every time the client must communicate with the server, an NFS read is performed. The amount of data read is intended to accurately approximate the data that must be transferred from the server to the client. Network latency and available bandwidth can significantly affect the performance of the three integrity verification schemes, whether or not simulated RPC calls are used.

6.5.3 Hybrid Strategy

The client can dynamically employ one of several strategies for determining which data should be moved to and from the offline scheme and when. A client with knowledge of its data access patterns would typically protect frequently used information with the offline scheme and less frequently used data by the online scheme. If the client is performing a sequence of operations before returning a result, then the data required for the computation can be protected by the offline scheme. When the data access pattern is not known, a reasonable solution is to move data from the online to the offline scheme each time it is used. This strategy works well where there is a high degree of temporal locality. SecureDB leaves the choice of strategy up to the client. Data can be moved to the offline scheme via calls to `hybridMoveToOffline()`. Forcing an integrity check by calling `checkRAS()` results in all data being moved back to the protection of the online scheme.

6.6 Implementation Notes

SecureDB consists of five source files and their corresponding header files: securedb.cpp, online.cpp, offline.cpp, hybrid.cpp, tree.cpp, and util.cpp. The SecureDB source files were integrated into OpenLDAP 2.1.23 in the `/servers/slapd/back-bdb` directory and compiled with `gcc 2.96`. Several modifications were required to the OpenLDAP source code. Specifically, the file `init.c` contains the following line which initializes SecureDB and specifies the integrity verification scheme to be used:
`init_secure_db(ONLINE);`

All instances of `db->put` and `db->get` were replaced in `id2entry.c` with calls to `sget` and `sput`, passing all of the original parameters. Prior to inserting or retrieving elements from the database, the `check` procedure is called (in the offline and hybrid scheme only) to determine whether an integrity check is required.

OpenLDAP can be compiled and installed using the accompanying Makefiles. After the slapd daemon has been installed, it can be configured by modifying the slapd.conf file located at `/usr/local/etc/openldap/`.

A series of scripts to start, stop, populate, clean, and dump data from the OpenLDAP server are available.

Chapter 7

Performance

In an attempt to gauge the relative performance of the three integrity verification schemes, a series of carefully controlled tests were performed. The goal of the experiments was to determine the performance of each scheme under real workloads. As discussed in Chapter 6, each scheme was implemented and then tightly integrated into the Berkeley DB embedded database.

OpenLDAP 2.1.23 was selected to measure the performance of the three schemes and was modified minimally to make use of SecureDB. OpenLDAP is a commercial-grade, full featured, open source implementation of the Lightweight Directory Access Protocol (LDAP). OpenLDAP was determined to be a suitable candidate for our tests as it is widely used, its source code is freely available, and its performance is heavily affected by the performance of Berkeley DB.

7.1 What is LDAP?

LDAP was developed as an efficient way for PCs to access complex directories based on the X.500 global directory standards. The X.500 standard was overhead intensive and led to the development of LDAP to greatly simplify access to global directories. After the Internet Engineering Task Force's adoption of LDAP, it quickly became the preferred solution for all types of directory services applications running over IP.

The complete LDAP specification for LDAP v1, v2, and v3 can be found in RFC 1487, 1777, and 2251, respectively. LDAP v3 adds increased security (SASL and SSL), allows for multiple LDAP servers to handle requests, and adds support for international characters.

LDAP directories are frequently used for global corporate directories, e-mail contact lists, telephone directories, and user authentication, but its use is not restricted to these applications.

7.1.1 How LDAP directories work

Directories are typically read more often than they are written. LDAP directories are hierarchical, beginning at the root with a global name, working down to increasing detail such as employee names, ID numbers, divisions, contact information, etc. LDAP fields are called *attributes* and begin with a Distinguished Name (DN), an Organizational Unit (OU), a Common Name (CN), and finally an attribute type and attribute value.

Directory information may be accessed from a PC or other client device and may be used to search email directories, network administration directories, customer directories, product catalogs, etc. The LDAP API is widely used and supported and can be accessed from C, C++, Java, JavaScript, Perl, and many other platforms. Microsoft Active Directory is also LDAP compliant.

7.1.2 OpenLDAP and BerkeleyDB

OpenLDAP makes extensive use of BerkeleyDB in order to maintain its databases of directory information. OpenLDAP maintains one database (`dn2id.c`) of key/value pairs mapping DNs to IDs and a second database (`id2entry.c`) mapping IDs to directory entries. In order to make use of the SecureDB API, only a few changes to the OpenLDAP source were required. Upon database initialization, a call to `init_secure_db()` was required to initialize SecureDB and specify the integrity verification scheme to be used. The calls to `db->get()` and `db->put()` were replaced

with calls to `sget()` and `sput()`, respectively, passing the same parameters as before. In order to get the SecureDB C++ library to compile with OpenLDAP's C source, minor tweaking was necessary and `gcc 2.96` was required.

7.2 DirectoryMark Test Framework

7.2.1 Platform

The tests were run on a Dell Intel system with:

- 2 x 733 MHz Intel Pentium processors (256K L2 cache)
- 256 MB RAM
- 18 GB Maxtor ATA IDE disk
- RedHat Linux 2.4.20-8smp kernel

7.2.2 Test Method

The primary testing was conducted using Mindcraft's DirectoryMark 1.3 test suite. DirectoryMark provides a Solaris client and supports multiple multithreaded client processes, allowing a single system to simulate many clients. For our tests, only a single client with a single thread was used. The client executed a prepared script of 1000 LDAP operations, generated by the DirectoryMark test suite. Queries were executed continuously without any delay between operations, until the entire script had completed once.

The DirectoryMark Perl scripts were modified so that:

- The inetOrgPerson schema is correctly used
- The correct attribute type is used for "unique identifier" matches
- The seeAlso field is inhibited as its syntax was incorrect (does not have DN syntax)

7.2.3 Test Scenarios

A test script containing 1000 LDAP operations was used for all tests. The test script contained a mixture of operations intending to simulate an e-mail / messaging server using a directory server, individual clients looking up names in an address book or expanding a group for e-mail, and an administrator making modifications to the directory. Searches consisted of exact UID matches, CN wildcard searches, and non-existent entries. The test scenario consisted of additions (17.5%), modifications (15.6%), comparisons (11.7%), and searches (55.2%). Binds were performed after every 5 operations.

Tests were performed with databases containing 1000, 5000, and 10,000 entries.

Prior to each test, the following steps were performed:

- The directory was started.
- The directory was populated "over protocol" using `ldapadd`.
- Each entry was read once to "warm" the entry cache.
- The specified test was run.

7.2.4 DirectoryMark configuration

The DirectoryMark test suite with detailed instructions is available from http://www.mindcraft.com/directorymark/.

OpenLDAP must be populated with properly formatted entries from an LDIF file. An LDIF file containing an arbitrary number of entries can be created using the `dbgen.pl` script. A 5000 entry LDIF file can be created by executing `perl dbgen.pl -v -x o=example.com -o ../Ldif/data5000 5000`. The LDIF file must be concatenated with the appropriate LDIF header as specified in the DirectoryMark instructions.

Once the LDIF file has been created, the test script must be generated. It is imperative that a new test script be created for each LDIF file, as the script is

dependent on the contents of the directory. Test scripts can be produced using the `scriptgen.pl` tool:

```
Usage: scriptgen2.pl [options] outputname ldif-file num-transactions
```

The following command produces a script similar to the ones that were used for testing: `perl scriptgen2.pl -b -v -i -a 35 -m 30 -c 25 -w 25 -n 25 -u 20 -g 25 -s 7 -X 8 -B 5 ../Scripts/data5000 ../Ldif/data5000.ldif 1000`

Once the appropriate server name and desired test script are specified in the DirectoryMark config file, the test suite can be run. The general procedure followed between tests was:

- Kill existing slapd daemon using killLDAP.sh.
- Clean residual files left by previous instance of slapd using cleanLDAP.sh.
- Start new instance of slapd daemon using startLDAP.sh.
- Populate LDAP server using populateLDAP.sh.
- Dump contents of directory using dumpLDAP.sh.
- Run DirectoryMark test suite from Solaris client.

7.3 DirectoryMark Results

The DirectoryMark test suite was run against an OpenLDAP server employing one of the three integrity verification schemes. An unmodified OpenLDAP server was also tested. Tests were conducted to determine the effects on performance of the number of entries in the database, and for the offline and hybrid schemes, the checking frequency. The period between integrity checks is indicated by the value of T, the number of get/put operations prior to a check being forced. A longer T value indicates less frequent checks. Performance was measured in terms of absolute time in seconds and in DirectoryMark Operations Per Second (DOPS). The average time for each search,

add, modify, and compare operation in milliseconds was also recorded. All data can be found in Appendix A.

Table 7.1 illustrates how the various schemes performed in decreasing number of DOPS. The number of entries was held constant at 1000, while the scheme and checking frequency were varied. As expected, the unmodified LDAP server performed the best while the offline scheme with frequent checks performed the worst. Unexpectedly, the hybrid scheme performed poorly when compared to the offline scheme. Possible reasons for the poor performance of the hybrid scheme will be discussed below. When checks were infrequent (T=50000), the offline scheme resulted in 39% fewer DOPS when compared to an unmodified LDAP server. The offline scheme, however, outperformed the online scheme by 31%, while the hybrid scheme outperformed the online scheme by 19%. In the worst case, when checks were frequent (T=500), the hybrid scheme was 185% slower (65% fewer DOPS) than the online scheme. With frequent checks, the offline scheme was 101% slower (50% fewer DOPS).

Scheme	T	RPC (k)	Time (s)	**DOPS**
Unmodified	-	-	30	33.3
Offline	50000	100	49	20.4
Offline	10000	150	52	19.2
Hybrid	50000	400	54	18.5
Online	-	1500	64	15.6
Hybrid	10000	900	69	14.5
Offline	1000	500	82	12.2
Offline	500	900	129	7.8
Hybrid	1000	5700	178	5.6
Hybrid	500	5700	183	5.5
Hybrid	100	5700	257	3.9
Offline	100	4400	817	1.2

Table 7.1: DOPS for various checking periods (T) 1000 entries

7.3.1 Checking frequency vs. performance

The performance of the offline scheme is bound solely by the overhead required to perform a check operation. The check operation requires updating the TAKEHASH

and the PUTHASH for all items in the database. This operation is computationally and communications intensive as the entire database must be read. Therefore, the frequency of checks can have a significant impact on the performance of the offline scheme as illustrated in Figure 7-1.

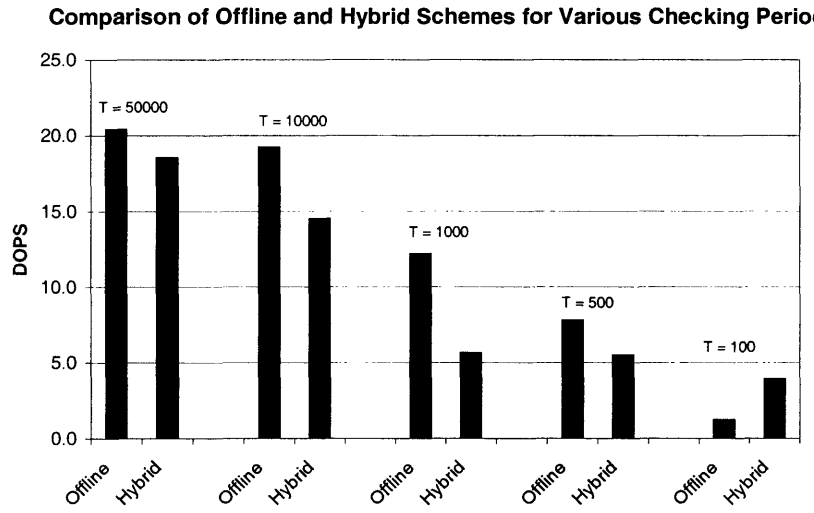

Figure 7-1: Comparison of Offline and Hybrid Schemes for Various Checking Periods (T = 50000, 10000, 1000, 500, 100) 1000 entries. *(longer bars are better)*

Performance of the offline scheme degrades rapidly as the checking period is decreased, suggesting the use of long checking periods to maximize performance. Long check periods, however, result in delayed tamper detection, which may or may not be an issue depending on the situation.

Similarly, the performance of the hybrid scheme is also affected by the checking period. The offline scheme outperforms the hybrid scheme which must traverse the hash tree to locate the correct STATUSBIT and must also perform the same check operation as the offline scheme. The hybrid scheme shines when an optimal strategy for moving data to the protection of the offline scheme is selected. When database accesses occur at random, however, no such optimal strategy is possible. The hybrid scheme outperforms the offline scheme when temporal locality is high. Our tests exhibited little to no temporal locality, resulting in poor performance of the hybrid scheme.

7.3.2 Effects of large data sets on performance

In order to determine the scalability of each of the three integrity verification schemes, each scheme was tested with an increasing number of database entries. Tests were performed for 1000, 5000, and 10,000 entries. More than 10,000 entries could not be tested due to the length of time involved to execute each test. A test with 100,000 entries would require 5 hours to initially populate the database over protocol. One million entries would require a population time in excess of 80 hours, even on an unmodified OpenLDAP server.

Scaling from 1000 entries to 10,000 entries resulted in an 8.3x slowdown for the unmodified server. The online scheme scaled the best, with a 10.4x slowdown, as its running time is logarithmic. The offline scheme with T=10,000 exhibited a 16x slowdown, while the hybrid scheme resulted in a 36x slowdown.

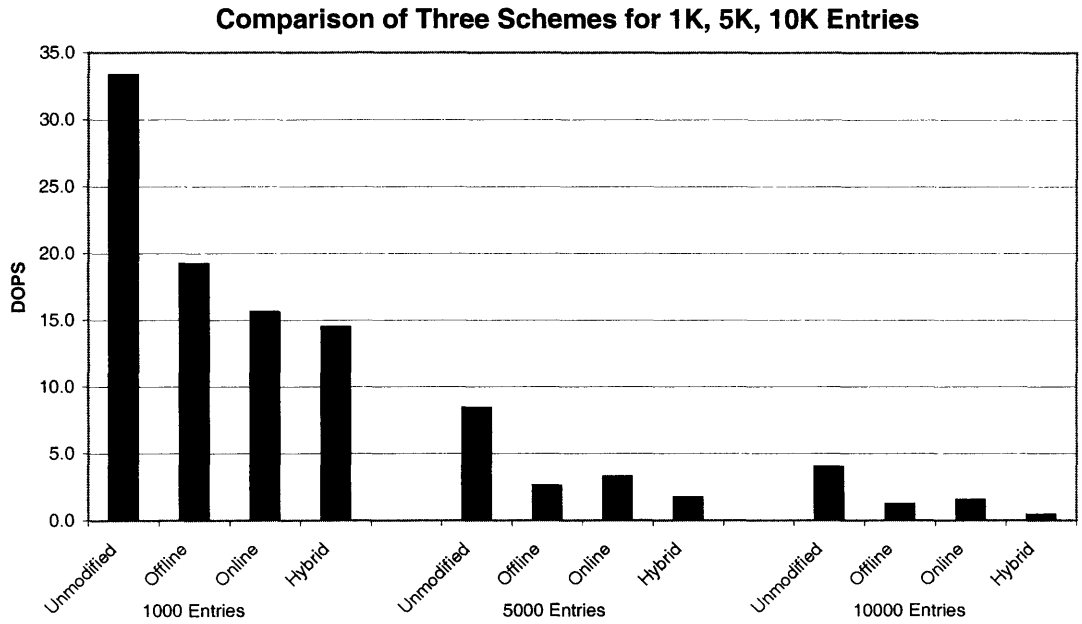

Figure 7-2: Comparison of Offline, Online, and Hybrid Schemes for 1000, 5000, and 1000 Entries (T=10000). *(longer bars are better)*

7.3.3 Search, add, modify, and compare operations

The DirectoryMark test suite consisted of four types of operations: Additions (17.5%), Modifications (15.6%), Comparisons (11.7%), and Searches (55.2%). Searches required the most amount of time as they typically required accessing large portions of the database. Figure 7-3 shows the relative time spent executing comparisons, modifications, and additions. Search time is shown in Figure 7-4.

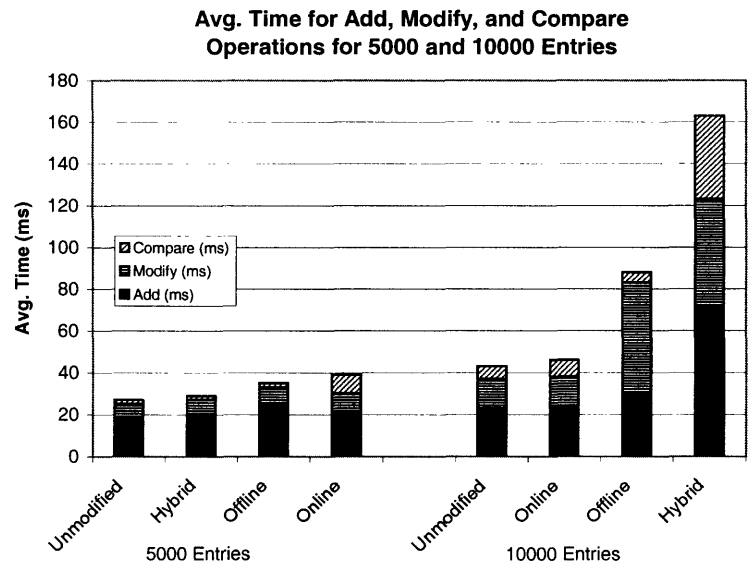

Figure 7-3: Avg. Time for Add, Modify, and Compare Operations for 5000 and 10000 Entries (T=10000).

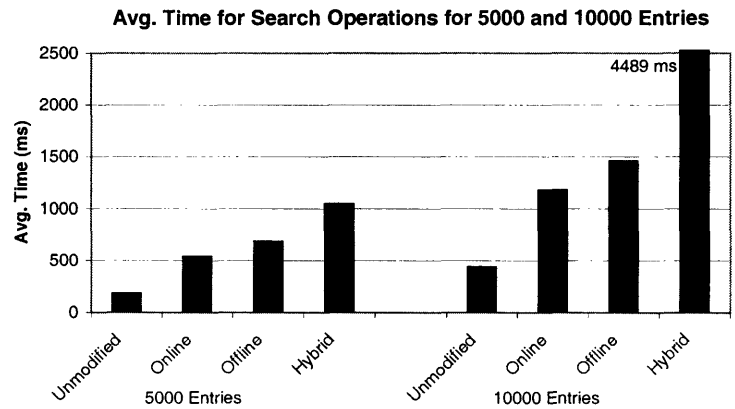

Figure 7-4: Avg. Time for Search Operations for 5000 and 10000 Entries (T=10000).

7.3.4 Loading over Protocol

Typically, direct tools are used for loading the OpenLDAP directories as this is significantly faster. Since SecureDB was not integrated into the direct tools (`slapadd`), the database had to be populated "over protocol" using the `ldapadd` tool. Loading over protocol requires that the OpenLDAP server be running, while direct loading makes modifications to the databases backing the directory directly. Direct population of a database is not always possible, so the performance of loading over protocol is a relevant and interesting measure.

Load measurements were taken for populating a database with 1000, 5000, and 10000 entries. Populating an unmodified server with 100,000 entries would have exceeded 5 hours and 1,000,000 entries would have exceeded 80 hours. As expected, the unmodified server performed the best, while the offline scheme once again proved to be the best of the three integrity verification schemes. The results for 10,000 entries is shown in Figure 7-5, and the complete results can be found in Appendix A.

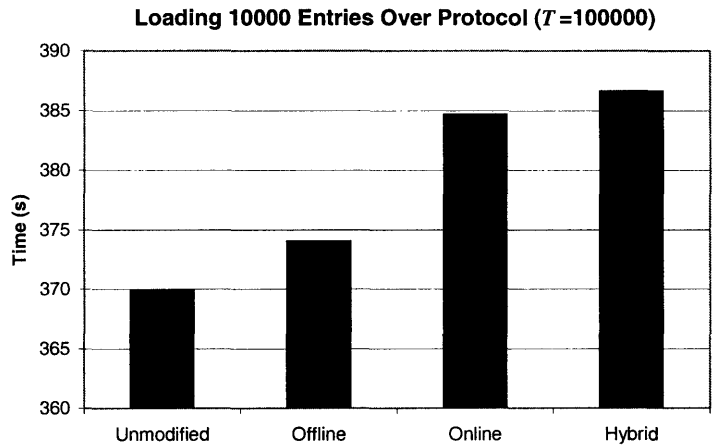

Figure 7-5: Time required to load 10000 entries over protocol (T=100000).

Chapter 8

Conclusion

We have presented three schemes for integrity verification of untrusted storage. The online scheme is in wide use today, while the offline and hybrid schemes are a new approach designed to outperform the use of hash trees alone. The SecureDB system was implemented in order to measure the performance of the three schemes when subjected to a real-world workload.

The offline scheme resulted in 39% fewer DOPS when compared to an unmodified LDAP server. The offline scheme, however, outperformed the online scheme by 31%, while the hybrid scheme was only 19% faster than the online scheme in the best case. In the worst case, when checks were frequent, the hybrid scheme was 400% slower (75% fewer DOPS).

Unfortunately, the hybrid scheme did not perform as well as expected due to the inability to accurately predict data access patterns generated by random LDAP queries. The overhead of the hybrid scheme can be reduced by selecting an optimal strategy for moving items to the protection of the offline scheme. When the data access pattern is known, selecting an optimal strategy is easy. However, when data accesses occur at random, finding the optimal strategy is significantly more difficult and can lead to poor performance of the hybrid scheme.

8.1 Future Work

The schemes presented focus on *tamper detection* while ignoring the issue of *recovery*. Additional work is required to integrate a system such as SecureDB with new or existing schemes for data recovery in the event of tampering or data corruption. In the future, we plan to investigate recovery techniques for data stored on untrusted storage in a database. One possible approach is to use backup disks and the client cache to restore data affected by an adversarial attack.

While integrity verification focuses on tamper detection, SecureDB could also be modified to provide encryption to prevent adversaries from reading data stored on untrusted storage. Encrypting data is essential for widespread adoption of SecureDB.

Offline integrity checking is useful for applications such as certified execution and memory protection on memory constrained devices. The hybrid scheme can be used as an alternative to hash trees for file systems and databases. For the hybrid scheme to truly be advantageous, an optimal strategy must be devised for moving data to the protection of the offline scheme in the presence of random accesses. Alternatively, the hybrid scheme may offer performance advantages if implemented at a higher level, say, the block or file level. This would give order to accesses which appear to be random at the key/value level.

The current implementation of SecureDB supports a single client and only simulates real client/server communication. For SecureDB to be truly useful it must provide mechanisms for data recovery and encryption and must be reworked to provide true client-server support.

Bibliography

[1] D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, and J. Kubiatowicz. Oceanstore: An extremely wide-area storage system, 2000.

[2] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.

[3] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable certificate management using undeniable attestations. In *ACM Conference on Computer and Communications Security*, pages 9–17, 2000.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[5] Arthur W. Chou. 2-3 trees as search trees. http://cs.clarku.edu/ achou/cs160/2-3Trees.htm.

[6] Dwaine Clarke, Blaise Gassend, G. Edward Suh, Marten van Dijk, and Srinivas Devadas. Offline integrity checking of untrusted storage. Technical report, MIT LCS TR-871, November 2002.

[7] Dwaine Clarke, Blaise Gassend, G. Edward Suh, Marten van Dijk, and Srinivas Devadas. Incremental multiset hash functions and their application to memory integrity checking. Technical report, MIT LCS TR-899, May 2003.

[8] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 181–196, San Diego, California, October 2000.

[9] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, February 2003.

[10] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, February 2003.

[11] Irene Gassko, Peter S. Gemmell, and Philip MacKenzie. Efficient and fresh certification. pages 342–353.

[12] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing, 2001.

[13] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of OSDI 2000*, 2000.

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

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

[16] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *Proceedings 7th USENIX Security Symposium (San Antonio, Texas)*, Jan 1998.

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

[18] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Hardware mechanisms for memory integrity checking. Technical report, MIT LCS TR-872, November 2002.

Appendix A

Appendix

Scheme	Entries	RAS	Populate (s)	Dump (s)	RPC (k)	checkRAS	Search (ms)	Add (ms)	Mod (ms)	Cmp (ms)	Total (s)	DOPS
Unmodified	1000	-	26.733	1.852	-	-	43	16	8	2	30	33.3
Unmodified	5000	-	151.668	2.1	-	-	186	18	7	2	119	8.4
Unmodified	10000	-	369.931	2.081	-	-	438	23	14	6	250	4.0
Offline	1000	50000	28.958	4.866	100	2	135	6	5	2	49	20.4
Hybrid	1000	50000	27.438	4.841	400	2	89	15	7	2	54	18.5
Online	1000	-	29.13	4.754	1500	-	105	16	9	3	64	15.6
Offline	1000	10000	29.567	4.716	150	16	85	16	7	2	52	19.2
Hybrid	1000	10000	28.914	4.637	900	14	116	17	7	2	69	14.5
Offline	1000	1000	30.493	4.966	500	162	139	15	7	2	82	12.2
Hybrid	1000	1000	28..835	5.015	5700	143	312	16	7	2	178	5.6
Offline	1000	500	27.225	4.874	900	371	222	16	6	2	129	7.8
Hybrid	1000	500	27.157	5.086	5700	287	321	19	10	4	183	5.5
Offline	1000	100	26.742	4.958	4400	2044	1439	16	117	1	817	1.2
Hybrid	1000	100	28.281	4.927	5700	1451	456	17	8	2	257	3.9
Online	10000	-	384.673	2.119	18600	-	1179	24	14	8	660	1.5
Offline	10000	100000	374.025	2.213	1600	14	845	36	26	12	479	2.1
Hybrid	10000	100000	336.02	2.149	9200	13	1227	25	13	5	686	1.5
Offline	10000	10000	360.919	2.023	4000	134	1459	30	53	5	821	1.2
Hybrid	10000	10000	338.311	1.93	64900	132	4489	72	51	40	2495	0.4
Offline	10000	1000	671.766	1.957	24300	1157	8026	26	11	7	4439	0.2
Online	5000	-	142.744	2.284	8700	-	541	21	9	9	307	3.3
Offline	5000	10000	166.365	1.959	2000	136	687	25	8	2	386	2.6
Hybrid	5000	10000	152.585	2.048	16300	66	1047	20	7	2	584	1.7
Offline	10000	Infinite	395.159	1.61			817	33	23	6	462	2.2