Offline Integrity Checking of Untrusted Storage

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

MIT Laboratory for Computer Science
{declarke, gassend, suh, marten, devadas}@mit.edu

Abstract

We extend the offline memory correctness checking scheme presented by Blum et. al [BEG+91] to develop an offline checker that can detect attacks by active adversaries. We introduce the concept of incremental *multiset hashes*, and detail one example: `MSet-XOR MAC`, which uses a secret key, and is efficient as updating the hash costs a few hash and XOR operations. Using multiset hashes as our underlying cryptographic tool, we introduce a primitive, *bag integrity checking*, to explain offline integrity checking; we demonstrate how this primitive can be used to build cryptographically secure integrity checking schemes for random access memories and disks.

Recent papers describe processors, file systems, and databases in which hash trees are used to verify the integrity of data in untrusted storage. Checkers using hash trees are referred to as online checkers, as the trees are used to check, after *each* operation, whether the storage behaved correctly. The offline checker we describe is designed for checking *sequences* of operations on an untrusted storage, and, for some applications, performs better and uses less space than a checker using a hash tree.

In this paper, we also introduce a hybrid checker, which can capture the best of both the online and offline schemes. The hybrid checker can operate mainly as an online checker when integrity checks need to be performed frequently, and as an offline checker when checks can be performed less frequently. The performance of the checker is expected to be close to the better scheme for every checking period.

1 Introduction

Recent papers, [GSC+03], [MS01], [MVS00], describe systems in which a trusted program, running in a trusted computing base (TCB), maintains data stored on untrusted storage. The untrusted storage is typically some arbitrarily large, easily accessible, bulk store in which the program regularly stores and loads data which does not fit in a cache in the TCB. In [GSC+03], the program runs on a trusted processor; the untrusted storage is the external random access memory (RAM). [MS01] and [MVS00] describe a file system and database respectively, in which the program runs on a protected client and the data is maintained on an untrusted server.

Each of the systems provides strong data integrity guarantees without trusting the storage. The systems use Merkle (hash) trees [Mer79] to verify the integrity of data stored on untrusted storage. A tree of hashes is maintained over the data, and the root of the tree is kept in the TCB. On each program store, the path from the data leaf to the root is updated. On each program load, 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. The scheme, thus, is a pessimistic approach to integrity checking; more importantly, on each program load or store, there are an additional number of accesses to the untrusted storage, which grows logarithmically with the size of the data set. As the operations are checked after each operation, the approach is referred to as *online* integrity checking [BEG+91].

This paper investigates an alternative approach to verifying the integrity of untrusted storage. We extend the work presented by Blum et al. [BEG+91] on *offline* memory correctness checking. The offline approach is used to check whether the untrusted storage performed correctly after a *sequence* of operations is performed. The benefit of the approach is that there is a constant overhead on the number of storage accesses on each program load or store. Blum's correctness checker was implemented using ϵ-biased hash functions [NN90]; these hash functions can be used to detect random errors, but are not cryptographically secure. We extend Blum's work to produce an offline integrity checker, secure against attacks by an active adversary.

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, as a hash tree does not have to be built, 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.

1.1 Model

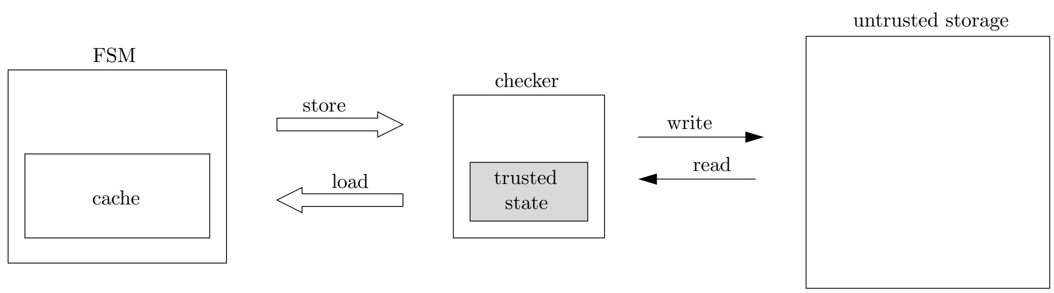


Figure 1: Model

Figure 1 illustrates the model we use. 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 FSM's operations on the untrusted storage. The TCB consists of the checker and its trusted state.

The FSM may also maintain a fixed-sized *cache*. The cache is initially empty, and can store data frequently accessed by the FSM. Data that is loaded into the cache is checked by the checker and the FSM can trust the data once it is loaded into the cache.

We consider a few examples. In the case of a processor accessing data in untrusted off-chip RAM (main memory), the FSM is the unmodified processor running a user program. The processor may have an on-chip cache. The checker is special hardware that is added to the processor to check the processor's loads and stores from the RAM. In the case of a network file system, or a database, where a client computer accesses data stored on a disk at an untrusted server, the FSM is the client running a program. The client can contain a cache. The checker is software that is added to the client to verify the client's operations on the server.

The problem this paper addresses is that of checking that the untrusted storage behaves like valid storage i.e., that the data the FSM loads from the storage is the data it is supposed to get based on what it had previously stored on the untrusted storage. For example, for random access storage, such as RAM or disks, the storage behaves like valid random access storage if the data value that the FSM loads from a particular address is the same data value that the FSM most recently stored to that address.

In our model, the untrusted storage is assumed to be actively controlled by an adversary. The adversary can perform any software or hardware-based attack on the storage. The untrusted storage may not behave like valid storage if the storage has malfunctioned because of errors, or the data stored has been somehow altered by the adversary. We are interested in *detecting* whether the storage is behaving correctly (like valid storage). Recovery after detection of a malfunction or an attack by an active adversary is not considered in this paper.

The checker must perform a check of the FSM's operations before exporting FSM results which depend on those operations to other programs or users. Thus, the FSM implicitly determines when it is necessary to perform checks based on the functions it performs. If the FSM runs to completion to produce a result, it is not necessary to check each load operation. The checker can perform a single check of all the FSM's operations on the storage when the FSM has completed. If the FSM produces intermediate results, the operations used to produce those results must be checked before the results are exported to other programs or users.

1.2 A Simple, Faulty, Approach

To help explain the complexity of the problem, we present an approach for random access storage that is simple, but faulty. The approach is for the checker to compute a message authentication code (MAC) of the data value and the address at which it is stored, and store the (data value, MAC) pair at the address on the untrusted storage. The address needs to be included in the computation of the MAC so that an adversary cannot copy data at one address to another address. When the FSM stores a data value to an address, the checker updates the MAC at the address. When the FSM loads a data value from the address, the checker recomputes the MAC and checks that it is the same as the MAC stored at the address. The data value is considered valid if the MAC check passes, and is considered invalid otherwise.

The simple approach does not work: it does not prevent replay attacks. By our definition, a storage behaves like valid random access storage if the data value that the FSM loads from a particular address is the same data value that the FSM most recently stored to that address. In the simple approach, an adversary can replace the (data value, MAC) pair stored at an address with pairs stored at that address on earlier stores. Even worse, the adversary could execute the first store, and then block all of the stores thereafter, and the checker would not detect the attack.

In the approaches described in this paper, the checker uses its small fixed-sized trusted state to ensure that the value the FSM loads is the most recent value stored on the storage. The state must be updated on each FSM store to maintain a snapshot of the current state of the storage.

1.3 Organization

Section 2 describes related work. Section 3 gives an overview of hash-tree based integrity (online) checking, the scheme with which we compare our work. In Section 4, we describe multiset hashes, and present two examples: `MSetMuHash` and `MSet-XOR MAC`; `MSet-XOR MAC` is described in detail. Our bag integrity checking primitive is introduced in Section 5. In Section 6, we show how to use this primitive to build offline integrity checking schemes for random access storage. A summary of an analysis of offline integrity checking is given in Section 7, and experimental results are presented in Section 8. Section 9 introduces hybrid online-offline integrity checking. Section 10 concludes our work. Appendix A gives a proof for the set-collision resistance of `MSet-XOR MACs`. Appendix B provides proofs of the equivalence of the bag definitions that we use in Section 5. Appendix C provides detailed analysis of the offline scheme.

2 Related Work

Our work is inspired by the work on memory correctness checkers by Blum et al. [BEG+91]. They proposed using hash (Merkle) trees to detect attacks by active adversaries on RAM. They also proposed offline schemes to check the correctness of RAM. Their implementation of offline checkers uses ϵ-biased hash functions [NN90]; these hash functions can be used to detect random errors, but are not cryptographically secure. We introduce a bag integrity checking primitive, and introduce multiset hash functions as a cryptographic tool to implement it. We demonstrate how the bag primitive can be used as a basis to build offline RAM and disk integrity checking checkers secure against active adversaries. Blum et. al also discuss offline checkers for stacks and queues, and we note that these checkers can also be easily explained using our bag integrity checking primitive. We demonstrate how caches can be used to make offline integrity checking substantially more efficient and present theoretical and practical evaluations of the scheme. We also introduce a hybrid online-offline integrity checker, which can be particularly useful in checking untrusted storages in large systems, like storages in file systems, databases, or on the Internet.

As described previously, [GSC+03], [MS01], [MVS00] describe systems which use hash trees to protect memory, file systems, and databases respectively. For particular applications, like certified execution, the storage could be checked more efficiently using offline integrity checking schemes. [GSC+03] and [MVS00] do not propose schemes to recover from an adversary that corrupts data in the untrusted storage. In [MS01], backups and client caches are used to repair damage from adversaries after the damage has been detected.

The eXecute Only Memory (XOM) architecture [LTM+00] is designed to run security requiring applications in secure compartments that can only communicate with the rest of the world on an explicit request from the application. XOM protects data stored in memory by appending the data blocks with a MAC of itself. To prevent an adversary from copying blocks from one memory address to another, the block's address is included in the MAC. The Protected File System (PFS) [SHS01] and the Transparent Cryptographic File System (TCFS) [CCSP01] use similar integrity protection mechanisms. As described in Section 1.2, this approach is vulnerable to replay attacks. For example, XOM will not notice if stores to memory are never performed (except when memory is first initialized).

The Byzantine-fault-tolerant file system [CL99, CL00] is implemented as a state machine that is replicated across different nodes in a distributed system. Each replica maintains the service state and implements the service operations. The system provides recovery from tampering, though it relies on the assumption that at least two-thirds of the replicas will be honest. The expectation is that the replicas are weakly protected but not hostile, so the difficulty of an adversary taking over k hosts increases significantly with k.

Schneier and Kelsey describe a technique for securing audit logs on untrusted machines [SK98]. Each log entry contains an element in a linear hash chain that serves to check the integrity of the values of all previous log entries. It is this element that is actually kept in trusted storage, which makes it possible to verify all previous log entries by trusting a single hash value. The technique is suitable for securing append-only data that is read sequentially by a verifying trusted computer. However, it is not suitable for a system that requires frequent and random read and write access to individual blocks of data.

3 Hash Trees

The scheme with which we compare our work is integrity checking using hash trees (Merkle trees) [Mer79]. Figure 2 illustrates a hash tree. The data values are located at the leaves of a tree. Each internal node contains a collision-resistant hash of the data that is in each one of the nodes below it. The root of the tree is stored in the checker where it cannot be tampered with.

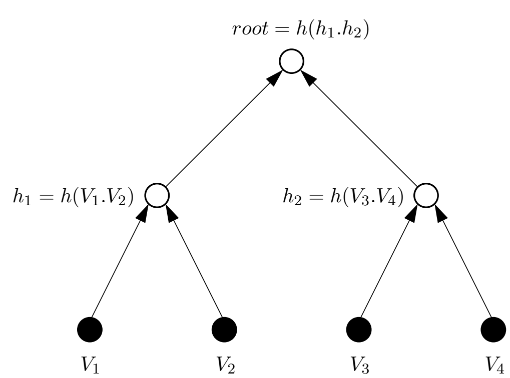


Figure 2: A binary ($m = 2$) hash tree. Each internal node is a hash of the concatenation of the data in the node's children.

To check the integrity of a node in the tree, the 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.

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, as 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 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 [GSC+03], 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 [Riv92] is used for the hashes). The experiments use 4 GBytes of RAM and about one quarter of the memory is used to store the hashes of the tree.

4 Multiset Hashes

The principal cryptographic tool we use for offline integrity checking is a set-collision resistant multiset hash function. This section defines the tool, and shows some example implementations.

In what follows, we will be working with a countable set of values $\mathcal{V}$. We refer to a *multiset* as an unordered group of elements where an element can occur as a member more than once [Ros99]. All sets are multisets, but a multiset is not a set if an element appears more than once. Multiset union $\cup_{\mathcal{MS}}$ combines two sets into a multiset in which elements appear with a multiplicity that is the sum of their multiplicities in the initial multisets. We shall call $\mathcal{MS}$ the set of multisets of elements of $\mathcal{V}$, and $\mathcal{S}$ the set of subsets of $\mathcal{V}$.

Definition 4.1. Let $(\mathcal{H}, +_{\mathcal{H}}, \equiv_{\mathcal{H}})$ be a triple of probabilistic polynomial time algorithms. That triple is a *multiset hash function* if it satisfies:

compression: $\mathcal{H}$ maps elements of $\mathcal{MS}$ into elements of $\{0,1\}^k$, where k is some integer. Compression guarantees that we can store hashes in a small bounded amount of memory.

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

incrementality: We would like to be able to efficiently compute $\mathcal{H}(A \cup_{\mathcal{MS}} B)$ knowing $\mathcal{H}(A)$ and $\mathcal{H}(B)$. The $+_{\mathcal{H}}$ operator makes that possible: $\forall A, B \in \mathcal{MS} : \mathcal{H}(A \cup_{\mathcal{MS}} B) \equiv_{\mathcal{H}} \mathcal{H}(A) +_{\mathcal{H}} \mathcal{H}(B)$

In particular, knowing only $\mathcal{H}(A)$ and e we can easily compute $\mathcal{H}(A \cup_{\mathcal{MS}} \{e\}) = \mathcal{H}(A) +_{\mathcal{H}} \mathcal{H}(\{e\})$

As it is, this definition is not very useful, as $\mathcal{H}$ could be any constant function. We need to add some kind of collision resistance to have a useful hash function.

Definition 4.2. A multiset hash function is *set-collision resistant* if it is computationally infeasible to find a set S and a multiset M such that $|S|$ and $|M|$ are of polynomial size in k, $\mathcal{H}(S) \equiv_{\mathcal{H}} \mathcal{H}(M)$ and $S \neq M$. [1]

4.1 Incremental cryptography

We use incremental cryptography, introduced by Bellare, Goldreich and Goldwasser [BGG94], as the basis of our design of $\mathcal{H}$. In particular, we present two instantiations of $\mathcal{H}$, a `MSetMuHash` instantiation ($\mathcal{H}_{\mathcal{M}}$) and a `MSet-XOR MAC` instantiation ($\mathcal{H}_{\mathcal{X}}$). $\mathcal{H}_{\mathcal{M}}$ is simple, but costly as a multiplication ($\times$) modulo a large prime is used to update the hash. $\mathcal{H}_{\mathcal{X}}$ is less simple as it requires a secret key, but it is more efficient, as adding elements to the hash costs a few hash and XOR ($\oplus$) operations only. We present $\mathcal{H}_{\mathcal{M}}$ in Section 4.1.1, and $\mathcal{H}_{\mathcal{X}}$ is detailed in Section 4.1.2.

4.1.1 Multiset Multiplication Hashes

Definition 4.3. Let H be a pseudorandom function over $\mathcal{V}$ (in practice one would use MD5 or SHA1). Let p be a k-bit prime. Then we can define `MSetMuHash` by:[2]

$$\boxed{\begin{aligned} \mathcal{H}_{\mathcal{M}}(M) &= \prod_{e \in_{\mathcal{MS}} M} H(e) \bmod p \\ A +_{\mathcal{M}} B &= A \times B \bmod p \\ A \equiv_{\mathcal{M}} B &= (A = B) \end{aligned}}$$

Theorem 4.1. `MSetMuHash` *is a set-collision resistant multiset hash function.*

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

Compression: As the operations are performed modulo a large prime, a `MSetMuHash` will be $k = \lceil \log_2(p) \rceil$ bits large.

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

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

Set-collision resistance: We model our proof of `MSetMuHash` on Bellare and Micciancio's proof in [BM97] that their `MuHash` is collision resistant.

`MuHash` is computed as follows. Let message, M, be viewed as an ordered sequence of n blocks: $M = M[1] \dots M[n]$, where $M[i] \in \{0,1\}^b$. Let $\langle i \rangle$ denote the binary representation of block index i. Let p be a large prime. Then

$$\mathcal{H}_{\mathrm{MuHash(M)}} = \prod_{i=1}^{n} H(\langle i \rangle \cdot M[i]) \bmod p$$

Observe that if the block indices $\langle i \rangle$ were omitted, permuting message blocks would leave the hash unchanged. As it is, the function is collision resistant under the assumption that the discrete log problem is hard in Z_p^*, i.e., it is computationally infeasible to find two distinct messages which produce the same `MuHash`.

Our proof of `MSetMuHash` is similar to the proof of `MuHash` in [BM97] (the details of the proof can be found in one of our technical reports).

[1] To make this notion formal, we need to consider a family of hash functions indexed by a seed s. This family satisfies collision resistance if for all probabilistic polynomial time algorithms $\mathcal{A}$, and any numbers c,
$\exists k_0 : \forall k \geq k_0, Prob\{s \leftarrow \{0,1\}^k, (S, M) \leftarrow A(s) : S \in \mathcal{S} \wedge M \in \mathcal{MS} \wedge M \neq S \wedge \mathcal{H}_s(M) \equiv_{\mathcal{H}} \mathcal{H}_s(S)\} < k^{-c}$
Note that because $\mathcal{A}$ is polynomial, we will consider that it can only output polynomial sized S and M (we are disallowing compact representations for multisets that would allow $\mathcal{A}$ to express larger multisets).

[2] Elements that appear multiple times in the multiset are iterated over multiple times in the product $\prod_{e \in_{\mathcal{MS}} M} H(e)$.

□

MSetMuHash is a simple and elegant example of a set-collision resistant multiset hash. Unfortunately it relies on modular arithmetic, which makes it too costly for some applications. We will therefore look at another candidate.

4.1.2 Multiset-XOR MACs

Definition 4.4. Let H_s be a pseudorandom function over $(\{1\} \times \mathcal{V}) \cup (\{0\} \times \{0,1\}^{\frac{k}{3}})$ keyed with a seed s (in practice, one would apply the HMAC method [KBC97] to MD5 or SHA1 to construct such a function). Let $|M|$ denote the number of elements in M, $R \leftarrow \{0,1\}^{\frac{k}{3}}$ denote uniform random selection of R from $\{0,1\}^{\frac{k}{3}}$, then define `MSet-XOR MAC` by:[3]

$$\mathcal{H}_{\mathcal{X}}(M) = \left(\left(H_s(0,R) \oplus \bigoplus_{e \in_{\mathcal{MS}} M} H_s(1,e)\right), |M| \bmod 2^{\frac{k}{3}}, R\right) \text{ where } R \leftarrow \{0,1\}^{\frac{k}{3}}$$

$$(A, C_A, R_A) +_{\mathcal{X}} (B, C_B, R_B) =$$

$$\Big(H_s(0,R) \oplus A \oplus H_s(0,R_A) \oplus B \oplus H_s(0,R_b)), C_A + C_B \bmod 2^{\frac{k}{3}}, R\Big) \text{ where } R \leftarrow \{0,1\}^{\frac{k}{3}}$$

$$(A, C_A, R_A) \equiv_{\mathcal{X}} (B, C_B, R_B) = \Big(A \oplus H_s(0,R_A) = B \oplus H_s(0,R_B) \wedge C_A = C_B\Big)$$

Theorem 4.2. `MSet-XOR MAC` *is a set-collision resistant multiset hash function as long as the key s remains secret.*

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

Compression: The output of $\mathcal{H}_{\mathcal{X}}$ is k-bits long by construction.

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

Set-collision resistance: We reduce the set collision resistance of `MSet-XOR MAC` to the collision resistance of Bellare, Guérin and Rogaway's incremental XOR message authentication scheme [BGR95].

In their scheme, a message authentication code is computed as follows. Let message, M, be viewed as an ordered sequence of n blocks: $M = M[1] \dots M[n]$, where $M[i] \in \{0,1\}^b$. Let $\langle i \rangle$ denote the binary representation of block index i, and F_s be a pseudorandom function keyed by s. Then the `XOR-MAC` of M is:

$$\mathcal{F}_{\text{XOR}-\text{MAC}}(M) = F_s(0,R) \oplus \bigoplus_{i=1}^{n} F_s(1, \langle i \rangle, M[i])$$

Observe that if the block indices $\langle i \rangle$ were omitted, permuting message blocks would leave the hash unchanged. The nonce R prevents an adversary from forming new MACs via linear combinations of old ones. Bellare et. al prove that `XOR-MAC` is collision resistant if the key s used to create the MAC is not known.

Our reduction of `MSet-XOR MAC` to `XOR-MAC` can be found in Appendix A.

□

A few interesting variants of `MSet-XOR MAC` exist. First, it is possible to replace the random nonce R by a counter that gets incremented on each use of $\mathcal{H}_{\mathcal{X}}$ and $+_{\mathcal{X}}$, or by any other value that never repeats itself in polynomial time. This removes the need for a random number generator from the scheme. Also, if hashes are never revealed to the adversary (he knows what is being hashed, but not the value of the hash) then we can remove $H_s(0,R)$ from the scheme altogether.

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

5 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.

Definition 5.1. A bag is a triple $(\mathcal{B}, \textsc{Put}, \textsc{Take})$ consisting of a multiset $\mathcal{B}$ and two operations $\textsc{Put},\ \textsc{Take}\ \in\ \mathcal{MS} \times \mathcal{MS}\ \rightarrow\ \mathcal{MS}$ defined by

$$\textsc{Put}(\mathcal{B}, M) = \mathcal{B} \cup_{\mathcal{MS}} M \tag{1}$$

and

$$\mathcal{B} = \textsc{Take}(\mathcal{B}, M) \cup_{\mathcal{MS}} M. \tag{2}$$

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).

Definition 5.2. A history is a finite sequence of put/take operations $O_1(., M_1)$, $O_2(., M_2)$, $\ldots$, $O_n(., M_n)$. A bag with a valid history is a bag, $\mathcal{B}_n$, that follows from a sequence of well defined put/take operations starting from the empty set. That is, $\mathcal{B}_n$ results from the recurrence relation $\mathcal{B}_i = O_i(\mathcal{B}_{i-1}, M_i)$ with $\mathcal{B}_0 = \emptyset$ where the put/take operations O_i satisfy the previous definitions (1) and (2).

Intuitively, $\mathcal{B}_i$ is the state of the bag at moment i. Also intuitively, a bag has a valid history if the checker can take an item from the bag only if the item has previously been put in the bag by the checker and has not yet been taken from the bag by the checker. This leads to the following equivalent definition of a valid history. Given a history of put/take operations $O_1(., M_1)$, $O_2(., M_2)$, $\ldots$, $O_n(., M_n)$ we define the put multisets P_i and the take multisets T_i by the recurrence relations

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

and

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

The history is valid if and only if for all $0 \le i \le n$

$$T_i \subseteq_{\mathcal{MS}} P_i \tag{3}$$

as may easily be verified by the reader[4]. It turns out that the difference in P_i and T_i is the bag $\mathcal{B}_i$, that is for all $0 \le i \le n$

$$T_i \cup_{\mathcal{MS}} \mathcal{B}_i = P_i. \tag{4}$$

5.1 Checker with a Log

It is instructive to analyze how the checker can verify the authenticity of the bag if the checker maintains a log. The solution is intuitive: when the checker puts an item into the bag, it appends a record saying that it put the item into the bag to its log; when the checker takes an item out of the bag, it appends a record saying that it took the item out of the bag to its log. To check some arbitrary sequence of put and take operations, the checker iterates through the log and ensures that each item was put into the bag before it was taken from the bag. That is, as it iterates through the log, it checks that (3) is always true.

Of course, the problem with a log is that the state the checker maintains grows at least linearly with the number of items in the bag. In our model, the state the checker maintains must be small and of a fixed size.

[4] To prove (3) is equivalent to Definition 5.2, we define bags $\mathcal{B}_i$ by (4) and show, by using induction in i, that they satisfy Definition 5.2. (3) immediately follows from (4).

5.2 Checker with Multiset Hashes

To deal with the problem of a growing trusted space, we equip the checker with multiset hashes, instead of a log. We describe a simple approach (which does not work). In this approach, the checker maintains two multiset hashes, PUTHASH and TAKEHASH. As described in Section 4, the multiset hash is incremental in that when a new element is added to the set, the multiset hash is updated by 'adding' ($+_{\mathcal{H}}$) a hash of the element to the multiset hash. The multiset hash is of a small, fixed size, which is maintained when it is updated. The multiset hash function is also set-collision resistant, as described in Section 4.

Initially, the bag is empty and PUTHASH and TAKEHASH are 0. The checker performs some arbitrary sequence of put and take operations. To put an item into the bag, the checker:

1. puts the item into the bag
2. updates PUTHASH: PUTHASH $+_{\mathcal{H}}=$ hash(item).

To take an item out of the bag, the checker:

1. takes an item from the bag
2. updates TAKEHASH: TAKEHASH $+_{\mathcal{H}}=$ hash(item).

Thus, after each put or take operation, O_i, by the checker, (PUTHASH, TAKEHASH) is a hash of the items currently in P_i and T_i respectively (i.e., it is a hash of the items currently in the bag). When the checker wants to check the integrity of the bag after a sequence of operations, it takes all of the items currently in the bag out of the bag, and, for each item, updates TAKEHASH to be (TAKEHASH $+_{\mathcal{H}}$ hash(item)). If, after all of the items have been removed, PUTHASH is equal to TAKEHASH, the bag is considered to have a valid history.

This scheme does not work. The problem is that the checker can take an item, $item_a$, say, out of the bag that it did not put in. The attack follows. Suppose there is currently no $item_a$ in the bag, but an adversary knows that sometime in the future, the checker will put $item_a$ into the bag. The adversary can put $item_a$ in the bag now. The checker can take it out of the bag, i.e., take something out of the bag that the checker did not put in. When the checker takes $item_a$ from the bag, it updates TAKEHASH to be (TAKEHASH $+_{\mathcal{H}}$ hash($item_a$)). The checker later puts $item_a$ in the bag and updates PUTHASH to be (PUTHASH $+_{\mathcal{H}}$ hash($item_a$)). When the checker puts $item_a$ in the bag, the adversary then takes $item_a$ back out of the bag. When the checker performs an integrity check on the bag, the two multiset hashes will be equal, but the bag will not be a valid bag: the checker took $item_a$ out of the bag when it did not put one into the bag. At the point in time that the checker takes $item_a$ from the bag, T_i is not a submultiset of P_i. In other words, even though the two multiset hashes match when the integrity of the bag is checked, the history O_1, O_2, etc., is not valid for the final bag.

To prevent the attack, the checker is augmented with a counter and time stamps are appended to items to prevent reordering. We present this solution in the next subsection.

5.3 Checker with Multiset Hashes and using Time stamps

This solution is described in Figure 3. The checker maintains multiset hashes, as in the previous scheme, and also has a counter. The checker increments the counter each time it puts a set of items[5] into the bag, and appends the new value of the counter (a time stamp) to each item 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.

More formally, the mulitsets, M_i, which the checker puts into the bag are sets which contain pairs (v, t) where v represents a data value and t represents a time stamp. If the checker puts a set of items $\mathcal{I}_s$ into the bag at time t then $O_t =$ PUT and

$$M_t = \{(v, t) : v \in \mathcal{I}_s\}. \tag{5}$$

If the checker takes a multiset of items $\mathcal{I}_{ms}$ from the bag at time t then $O_t =$ TAKE and for all $v \in \mathcal{I}_{ms}$ there exists a time stamp t' such that $(v, t') \in M_t$, and the checker *checks* whether $t' \leq t$.

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

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 3: Bag Offline Integrity Checking

Theorem 5.1. *Suppose all the time stamp checks pass. Then,* PutHash = TakeHash *if and only if the bag is empty and the history O_1, O_2, ..., is valid.*

Proof. To prove that the scheme is secure, that is the sequence of operations O_i, $0 \leq i \leq n$, is a valid history, assume that the time stamp checks pass and that in addition PutHash = TakeHash. From (5) it follows that pairs in different sets M_t with $O_t =$ Put have different time stamps t, so sets M_t with $O_t =$ Put are disjoint. Therefore, their multiset union P_n is a *set*. Since the multiset hash function is set-collision resistant, $T_n = P_n$, thus T_n is a set as well and by (4) the bag $\mathcal{B}_n$ is empty. To prove that the sequence of operations O_i is a valid history we verify (3). Let $(v, t) \in T_i$. Clearly, $(v, t) \in M_j$ with $O_j =$ Take for some $j \leq i$. Since all time stamp checks passed, $t \leq j$ proving $t \leq i$. The tuple $(v, t) \in T_n = P_n$, hence, for some k, $(v, t) \in M_k$ with $O_k =$ Put. By (5) (i.e. as the checker controls the put pairs and stamps each put pair with the current time), $t = k$. Together with $t \leq i$, we conclude $(v, t) \in M_t \subseteq P_t \subseteq P_i$. This proves $T_i \subseteq P_i$ as required.

If the bag $\mathcal{B}_n$ is empty and the history O_1, O_2, ..., O_n is valid, then $T_n = P_n$ by (3,4) implying that PutHash = TakeHash.

□

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 3. 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. By Theorem 5.1, 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 6 demonstrates how this can be done for random access storages, such as RAM and disks.

6 Integrity Checking of Random Access Storage

In Section 5, 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 6.1. *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 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 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 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 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 **??**, we optimize this requirement with a hybrid online-offline approach).

In the following subsections, we discuss the implementation issues involved in checking dynamically-changing, sparsely-populated address spaces, and show how the offline scheme can be easily extended to incorporate caches.

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 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: Offline integrity checking of random access storage using a checked bag

6.1 Dynamically-changing, sparsely-populated, address space

6.1.1 General case

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; if the Error flag is set, `checkRAS` returns false[6]. The augmented interface is shown in Figure 5.

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 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

[6]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, the checker sets an Error flag; if the Error flag is set, `checkRAS` returns false.

Figure 5: Offline integrity checking of random access storage on a dynamically-changing, sparsely-populated, address space

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 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[7] 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.

Lemma 6.1. *Assuming the FSM meets the FSM requirement, the RAS behaves like valid random access storage (i.e. like a valid bag, which is empty after* `checkRAS` *has been performed, and the RAS invariant has been satisfied) if and only if the* `checkRAS` *operation returns true.*

Proof. The validity condition, that, assuming the FSM meets the FSM requirement, if the RAS behaves

[7] for example, the pointer to the top of the heap

like valid RAS, then the `checkRAS` will return true, is easy to verify. We present an argument for the safety condition: assuming the FSM meets the FSM requirement, if the RAS does not behave like valid random access storage, then the `checkRAS` operation will return false.

In the first case, we assume that, even though the adversary tampered with (data value, time stamp) pairs in the untrusted storage, the RAS invariant is still satisfied. In this case, either the bag did not behave like a valid bag, or the bag is not empty after `checkRAS` was performed. In either case, the bag `check` operation will return false, by Theorem 5.1. Thus, `checkRAS` will return false.

In the second case, we assume that the RAS invariant was not met. If we consider the first time the RAS invariant was violated, the bag had to have previously misbehaved, as the FSM meets the FSM requirement. That misbehavior will be caught by the bag checker, by Theorem 5.1. Thus, again, `checkRAS` will return false. □

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 6, 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[8]. We note that the bitmap does not have to be protected as 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.

6.1.2 First Monotonic case

It is interesting to examine the case where there is no `remove`(a) operation in the checked random access storage interface, and we allow the FSM to access some untrusted, and unchecked, storage directly (i.e. data structures that are used to maintain the FSM requirement can be stored on untrusted, and unchecked, storage). All of the other operations are present in the interface, and are as described in Figure 5. In this case, the `checkRAS` procedure must ensure that each address is taken from the bag at most once in step 2 of the procedure.

Lemma 6.2. *The RAS that is being checked has behaved like valid random access storage (i.e. like a valid bag, which is empty after `checkRAS` has been performed, and the RAS invariant has been satisfied) if and only if the `checkRAS` operation returns true and has taken a set (as opposed to a multiset) of addresses from the bag.*

Proof. The validity condition, that if the RAS has behaved like valid random access storage, then `checkRAS` operation returns true and has taken a set of addresses from the bag is easy to verify. Again, we present an argument for the safety condition: if `checkRAS` operation returns true and has read a set of addresses, then the RAS has behaved like valid random access storage. The fact that the bag has behaved like a valid bag, and is empty, follows directly from the `checkRAS` operation returning true, according to Theorem 5.1.

We argue that the RAS invariant has been satisfied by contradiction. Assume that the `checkRAS` operation returns true and has taken a set of addresses from the bag, and that the RAS invariant has not been satisfied. This means that either:

1. a `put` operation (in `add`) was performed on an address already present in the bag, so the address is in the bag multiple times. As there is no `remove` operation, it will remain that way until `checkRAS` is performed. If `checkRAS` ensures that it reads each address at most once, then the bag is not empty after `take`(true) is called. But, `checkRAS` returns true, so `check` returned true. Thus, we have a contradiction of Theorem 5.1.

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

□

[8] 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.

6.1.3 Second Monotonic case

It is also interesting to examine the case where, for a particular bag, by the time `check` is called on the bag in step 3 of the `checkRAS` operation, no address has had `add` called on it more than once (either by using the add operation, or in step 4 of `checkRAS`). In other words, the addresses which `add` has been called on for a particular bag form a set (a simple example is, when the new bag is created, the FSM calls `add` on a set of addresses in the set of (address, data value) pairs, and does not call it again until `checkRAS` is called).

Again, in this case, we can allow the FSM to access some untrusted, and unchecked, storage directly (i.e. data structures that are used to maintain the FSM requirement can be stored on untrusted, and unchecked, storage). All of the operations, including the `remove` operation, are present in the interface, and are as described in Figure 5.

Lemma 6.3. *Assuming the addresses which* `add` *has been called on form a set (as opposed to a multiset), the RAS that is being checked has behaved like valid random access storage (i.e. like a valid bag, which is empty after* `checkRAS` *has been performed, and the RAS invariant has been satisfied) if and only if the* `checkRAS` *operation returns true.*

Proof. The validity condition, that assuming the addresses which `add` has been called on form a set, if the RAS that is being checked has behaved like valid random access storage then the `checkRAS` operation returns true, is easy to verify. We present an argument for the safety condition: assuming the addresses which `add` has been called on form a set, if the `checkRAS` operation returns true, then the RAS that is being checked has behaved like valid random access storage. Again, the fact that the bag has behaved like a valid bag, and is empty, follows directly from the `checkRAS` operation returning true, according to Theorem 5.1. Again, we argue that the RAS invariant has been satisfied by contradiction. Assume that the addresses which `add` has been called on form a set and the `checkRAS` operation returns true, and that the RAS invariant has not been satisfied. This means that either:

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

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

□

6.2 Caches

Integrating a cache into the FSM can produce a drastic reduction in bandwidth usage to the untrusted storage. Up to now we have based the offline RAS checker on the checked bag abstraction. In this section we break the abstraction and add a write allocate cache[9] to optimize the scheme. We discuss this optimization in the context of a processor working with untrusted RAM.

In a cache, data is organized into blocks, with each block containing multiple data values. As the cache is trusted, and RAM (main memory) is untrusted, the cache contains just value blocks, while the RAM contains (value block, time stamp) pairs. We detail the caching algorithm:

- when the cache evicts a block of data,
  - if the block is dirty, the checker increments Timer, writes the (value block, current time) pair to the appropriate RAM address, and updates PutHash with the (RAM address, value block, current time) triple.
  - if the block is clean, the checker increments Timer, writes only the current time to the appropriate RAM address, and updates PutHash with the (RAM address, value block, current time) triple.
- when the cache brings a block of data in from RAM, the checker reads the time stamp and checks that it is less than or equal to the current value of Timer. The checker then updates TakeHash with the (RAM address, value block, time) triple.

[9] In a write allocate cache, the whole data block is brought into the cache when one of its words is read or written.

In essence, when the cache evicts a block, the (address, value block) pair is put into RAM. When the cache brings in a block from RAM, the (address, value block) pair is taken from RAM. The `checkRAS` operation iterates through the addresses. For each address, it checks to see if the block at the address is in the cache; if it is in the cache, no action is taken for the address (TakeHash has already been updated when the block at the address was brought into the cache). If the address's block is not in the cache, it takes the (address, value block) from RAM. If PutHash is equal to TakeHash at the end, the integrity check is successful.

There are a few important points to note when evaluating this caching algorithm. The first is that *the entire cache is used to store data values, so the cache hit rate - the fraction of FSM storage operations that find their data in the cache - is the same as that in a base FSM without storage integrity checking* (in Section 7, we contrast this hit rate with the data value hit rate of an FSM using a checker which uses hash trees, in which both hashes and data values must be stored in the cache for the checker to perform well.)

The algorithm has an overhead of reading the FSM's addresses in RAM whose blocks are not in the cache when performing `checkRAS`. The second point we would like to note is that, *besides this overhead, the only other overhead of the caching algorithm is the bandwidth consumed by reading and writing time stamps.* The cache hit rate remains the same, and when blocks are brought into or ejected from the cache, time stamps are also read from or written to RAM. If the number of loads and stores performed by the FSM is large, the amortized cost of the overhead of reading the RAM to perform `checkRAS` is very small, and the principal overhead comes from reading and writing time stamps. If a cache block is 64 bytes, say, and a time stamp is 32 bits, say, the overhead of the time stamps is small. In other words, with this caching algorithm, if the number of FSM storage operations before `checkRAS` is performed is large, we can expect the performance of offline integrity checking to be very close to the base performance of the standard FSM without any storage integrity checking. In Section 7 and Appendix C, we formalize this analysis. It is interesting to note that, if the number of FSM storage operations is large, offline integrity checking can perform better than the simple, faulty, scheme described in Section 1.2, in which a 128-bit MAC (assuming MD5 is used for the MACs), is appended to a block which is stored in RAM.

7 Analysis

In this section, we present a summary of our analysis of the online and offline integrity checkers. Our analysis is detailed in Appendix C. Table 1 summarizes the space and bandwidth overhead of the offline and online schemes.

	Offline	Online
per-bit space overhead	$\frac{b_t}{b_v}$	$\frac{b_h}{b_v - b_h}$
b/w consumption of initialization	$N(b_v + b_t)$	$\frac{b_v N}{m-1}$
b/w overhead at run time, with a cache	$2b_t t(1-h_v) + (N-C)(b_v+b_t)$	$t(1-\delta h_v)b_h z$

Table 1: Comparison of Offline and Online Integrity Checking

In the table, b_t is the number of bits in a time stamp, b_v is the number of bits in cache block and N is the number of data value blocks the FSM uses, i.e. the number of blocks to which a time stamp is appended when the block is stored in the untrusted storage. t is the number of FSM loads and stores the FSM can perform before a check is required. C is the number of blocks that can be stored in the cache, and h_v is the cache hit rate - the fraction of FSM storage operations that find their data in the cache. We assume that, in the online scheme, one hash covers one cache block implying $b_v = mb_h$.

As in Section 3, m is the number of children of each node in a hash tree. b_h is the number of bits in hash. δh_v, where $0 \leq \delta \leq 1$, is the cache hit rate of data values in an online checker (this is less than the cache hit rate of an offline checker as both hashes and data values must be stored in the cache for an online checker to perform well). z is the average number of hashes the online checker fetches from the storage on each FSM storage operation.

The principal points of the analysis are that:

- if the size of a time stamp is less than the size of a hash, the offline scheme will have a smaller space overhead than the online scheme. As described in Appendix C, 4 GBytes of memory are used in the experiments in Section 8, with $\frac{1}{16}$ of the memory being consumed by 32-bit time stamps. We contrast this with the experiments in [GSC+03], where $\frac{1}{4}$ of memory is used for 128-bit MD5 hashes for a 4-ary hash tree in a similar experimental set up.

- if the period between offline integrity checks, t, is greater than $\frac{(N-C)(b_v+b_t)}{(1-\delta h_v)b_h z - 2b_t(1-h_v)}$, the cost of fetching hashes on each FSM storage operation in an online checker will exceed the cost of reading the addresses the FSM used to perform a `checkRAS` operation in an offline checker, and the offline checker will consume less bandwidth than an online checker. In the experiments in Section 8, we are concerned with the number of FSM storage accesses $T = (1 - h_v)t$ for the offline case: these are the number of FSM accesses to the untrusted storage. If we assume $\delta = 1$ for large caches, the offline scheme is better than the online scheme if $T \geq \frac{(N-C)(b_v+b_t)}{b_h z - 2b_t}$.

8 Experiments

This section evaluates the offline integrity checking scheme compared to the online hash trees for computer processors through detailed simulations. We first investigate the impact of the offline scheme on processor performance ignoring the overhead of reading memory in the `checkRAS` operation (cf. Figure ??). This analysis provides the cost of offline integrity checking when memory integrity needs to be checked only at the end of program execution, or very infrequently in comparison to the total execution time of the program. Then, we study the performance of the offline scheme when more frequent integrity checks are required.

Our simulation framework is based on the SimpleScalar tool set [BA97]. The simulator models speculative out-of-order processors, which are standard modern microprocessors such as the Intel Pentium. For all of the experiments in this section, nine SPEC2000 CPU benchmarks [Hen00] are used as representative applications: `gcc`, `gzip`, `mcf`, `twolf`, `vortex`, `vpr`, `applu`, `art`, and `swim`. The SPEC benchmark suite is a standard set of programs that are generally used in the computer architecture community to evaluate processor performance. These benchmarks show varied characteristics and represent various types of applications.

The offline integrity checking module is added next to the on-chip Level 2 (L2) cache [10]. The module computes the MSET-XOR MAC of (address, data value, time stamp) triples and updates appropriate set hashes as described in Section 6.2 on L2 cache misses and cache evictions. For each memory access, the processor speculatively continues computation as soon as the values arrive from memory, and the hash computations are computed in the background. Therefore, hash computations do not increase memory latency. When the processor checks memory integrity by reading the memory space that was used, i.e., the `checkRAS` operation in Figure ??, it waits for the check to be done before continuing computation.

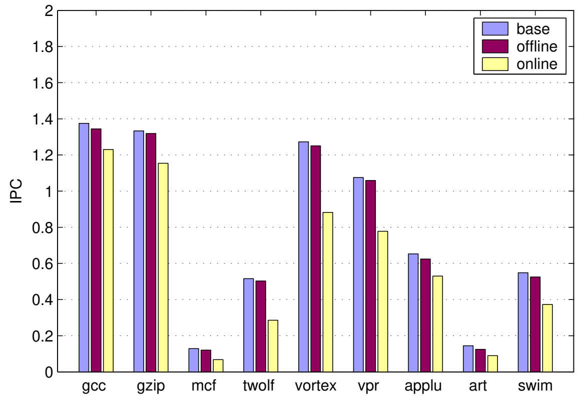
The online integrity checking is implemented in a similar way next to the L2 cache. The online module uses hash trees and computes the hashes in the background so that it does not increase the memory latency. Since memory integrity is checked on every access, there is no specific checking operation like `checkRAS`.

8.1 Steady-State Performance

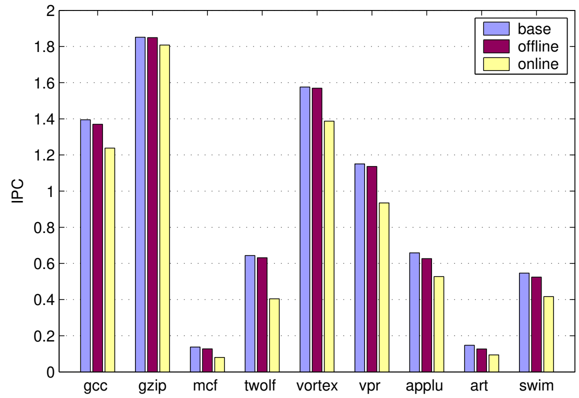
Applications that sign results at the end of execution, or do so relatively infrequently, are first considered. For these applications, the overhead of reading the memory space for the offline scheme during the `checkRAS` operation is negligible. If a typical program executes for a minute, this corresponds to roughly 100 billion instructions, on a state-of-the-art 2- or 4-way superscalar 1 GHz processor. The `checkRAS` operation typically takes less than a billion cycles, and if this is performed once, at the end of the execution, the overhead is very small.

In this subsection, we compare the offline scheme with the online scheme and quantify their overheads relative to a standard processor without integrity checking. We will ignore the overhead of the `checkRAS` operation. This comparison will demonstrate the advantage of the offline scheme over the online scheme when integrity checking is infrequent.

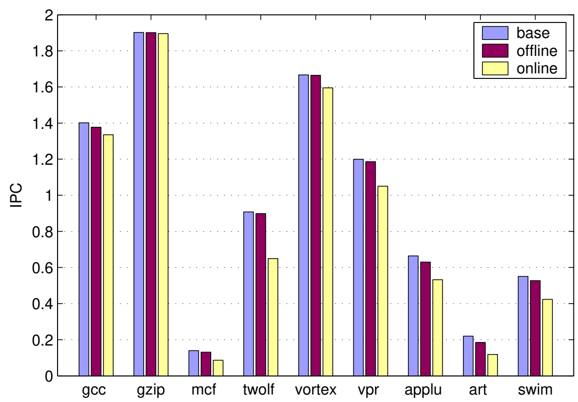
[10] A typical processor has two levels of on-chip caches. Main memory is accessed only if the requested data cannot be found in the caches (cache misses).



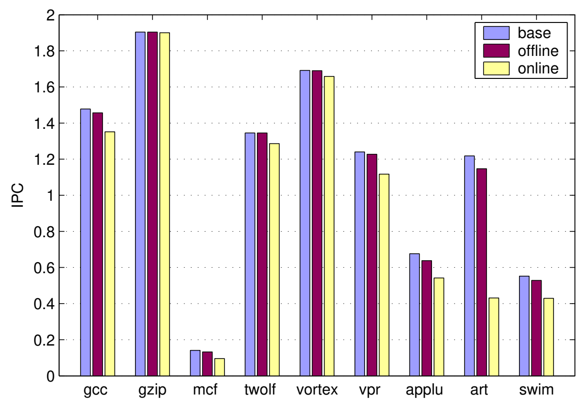
(a) 256KB



(b) 512KB



(c) 1MB



(d) 2MB

Figure 6: Steady-state performance overhead of memory integrity checking: standard processor without integrity checking (`base`), offline integrity checking (`offline`), and online integrity checking (`online`). Results are shown for four different cache sizes (256KB, 512KB, 1MB, 2MB) with cache block size of 64B. 32-bit time stamps for `offline` and 128-bit hashes for `online` are used.

Figure 6 illustrate the impact of integrity checking on the program performance. For four different L2 cache sizes, the IPCs (instructions per clock cycle) of three schemes are shown: a standard processor (`base`), offline integrity checking scheme (`offline`), and online integrity checking scheme (`online`). The IPC represents how fast a program executes. Therefore, higher IPC results indicate better program performance.

The experimental results clearly demonstrate the advantage of the offline scheme over the online scheme when we can ignore the `checkRAS` overhead. For all programs and configurations simulated, the offline scheme outperforms the online scheme. The overhead of the offline scheme compared to `base` is 15% in the worst case and often less than 5%, whereas the online scheme has over 50% overhead in the worst case and 10-20% in general.

Ignoring the checking overhead, the online scheme significantly reduces the performance overhead over the offline scheme in two ways: L2 cache pollution and memory bandwidth consumption. The online scheme stores hashes in the on-chip cache with program data, and degrades the cache performance for the program. In the figure, the effect of this cache pollution is evident for a small cache (256KB). The performance degradation of the offline scheme for cache-sensitive benchmarks such as `gcc`, `twolf`, `vortex`, and `vpr` in the 256-KB case is mainly due to the cache pollution. On the other hand, the offline scheme does not pollute the cache at all since it does not need to cache time stamps. As a result, the offline scheme only shows a marginal performance degradation even for the small cache.

The offline scheme can also consume less bandwidth than the online scheme. With the 64-B cache block ($b_v = 512$) and the 32-bit time stamp ($b_t = 32$), the offline scheme consumes an additional 64 bits (8 Bytes) of memory bandwidth for each access of 64-B cache block. The bandwidth overhead is 12.5%. On the other hand, the online scheme reads about 1.5 hashes (128-bit, $b_h = 128$) for each memory access on average ($z = 1.5$), which translates to 37.5% overhead in the memory bandwidth usage. Even though the online scheme improves significantly using caches (z reduces from 14 to 1.5), the offline scheme is considerably better than the online scheme. Due to less bandwidth consumption, the offline scheme is better than the online scheme even for large caches where cache pollution is not an issue (Figure 6 (d)).

8.2 Effects of Checking Periods

The last subsection clearly demonstrated that the offline integrity checking outperforms the online scheme when checking is infrequent. However, applications may need to check the memory integrity more often for various reasons such as exporting intermediate results to other programs. In these cases, we cannot ignore the overhead of frequent `checkRAS` operations. In this subsection, we compare the offline scheme and the online scheme including the overhead of periodic `checkRAS` operations.

We assume that the offline scheme checks memory integrity every T accesses to the untrusted RAM. A processor executes a program until it makes T main memory accesses, then checks the integrity of the T accesses by reading the memory it used before continuing. Obviously, the overhead of the offline checking heavily depends on the characteristics of the program and the check period T. We use two representative benchmarks `mcf` and `art` – the first has the most overhead for the `checkRAS` operation and the second has the least overhead. `mcf` uses 191MB of main memory and takes about 140 million cycles for each check. On the other hand, `art` uses only 2MB of memory and takes about 2.7 million cycles for a check.

Figure 7 compares the performance of the offline and the online integrity checking for varying check periods. In the figure, IPCs for the online scheme (`online`), the offline scheme ignoring the `checkRAS` operation (`offline-steady-state`), and the offline scheme including the `checkRAS` operation (`offline`) are shown.

Experimental results show that the performance of the offline scheme heavily depends on the checking period. The offline checking is infeasible when the application needs to assure the memory integrity after every memory access. In this case, the online integrity checking should be used. On the other hand, if checking is less frequent, the offline scheme can outperform the online scheme. Also, as the checking period increases, the performance of the offline scheme converges to the steady-state performance ignoring the `checkRAS` operation (around 100 million accesses for `mcf`, and ten million accesses for `art`).

The break-even point between the offline scheme and the online scheme depends on the characteristics of the program. For programs using large amount of memory such as `mcf`, the checking period should be long so that the cost of `checkRAS` is amortized. However, the offline scheme performs as well as the online for much shorter checking periods for programs with small amount of memory usage. For example, with 256-KB L2 caches, the break point, for `mcf` is around $T = 5$ million accesses, which corresponds to about 200 million instructions (0.15 seconds with 4-way superscalar 1-GHz processors), and for `art` is 100 thousand accesses, which is about 800 thousand instructions (6ms). The analysis in Section 7 predicts these break-even points quite accurately. For example, in the case of `mcf` with a 2-MB cache, the analysis results in,

$$T \geq \frac{(3129344 - 32768) \cdot (512 + 32)}{128 \cdot 1.9 - 2 \cdot 32} \approx 9.4 \times 10^6.$$

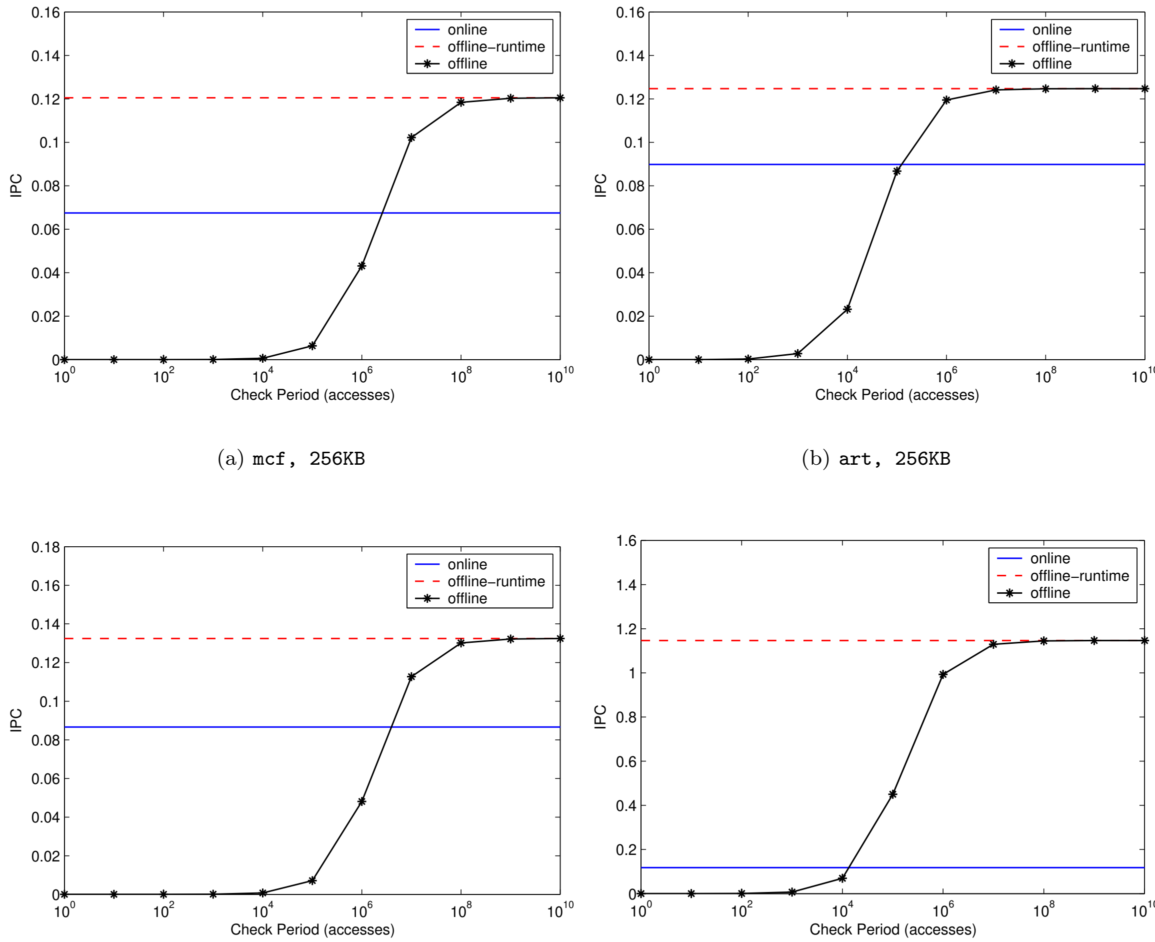
The value of N was calculated based on 191MB (1MB = 16,384 cache blocks) memory usage, and $z = 1.9$.

9 Hybrid Integrity Checker

Figure 8 illustrates the various interfaces discussed in this paper. In Section 5, we constructed a *checked bag*, with the interface $\texttt{put}(\mathcal{I}_s)$, $\texttt{take}(\mathcal{P})$, and $\texttt{check}()$[11], from an underlying *untrusted bag*, whose interface was standard bag $\texttt{put}(\mathcal{I}_m)$ and $\texttt{take}(\mathcal{P})$ operations[12]. In Section 6, we used the checked bag to construct a *checked random access storage (RAS)*, with the interface $\texttt{add}(S)$, $\texttt{remove}(a)$, $\texttt{store}(a, v)$, $\texttt{load}(a)$, and

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

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



(a) `mcf, 256KB`

(b) `art, 256KB`

(c) `mcf, 2MB`

(d) `art, 2MB`

Figure 7: Performance comparision between the offline scheme and the online scheme for various offline checking periods. Results are shown for 256-KB and 2-MB L2 caches with 64-B blocks. 32-bit time stamps and 128-bit hashes are used.

`checkRAS`()[13]. 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 9 illustrates a hybrid checker. To construct the hybrid checker, we use the checked RAS, and a variant of the hash tree described in Section 3, which we call a *partial-hash tree*. Partial-hash trees are described in Section 9.2.

9.1 Motivation for the hybrid checker

One of the disadvantages of the offline checker in Section 6 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

[13]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`$()$.

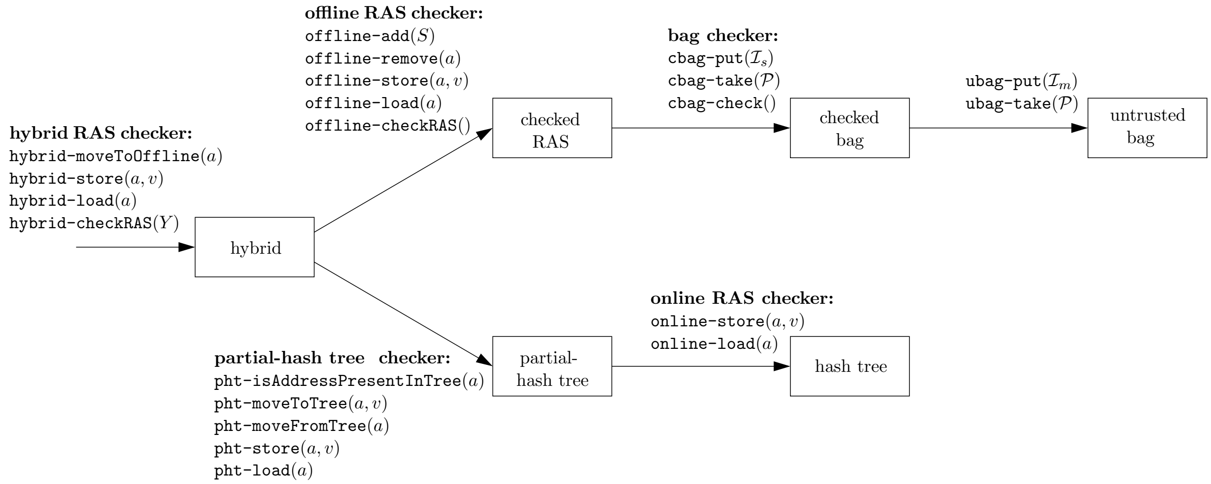


Figure 8: Interfaces

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.

9.2 Partial-Hash Tree

Besides the checked RAS developed in Section 6, 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

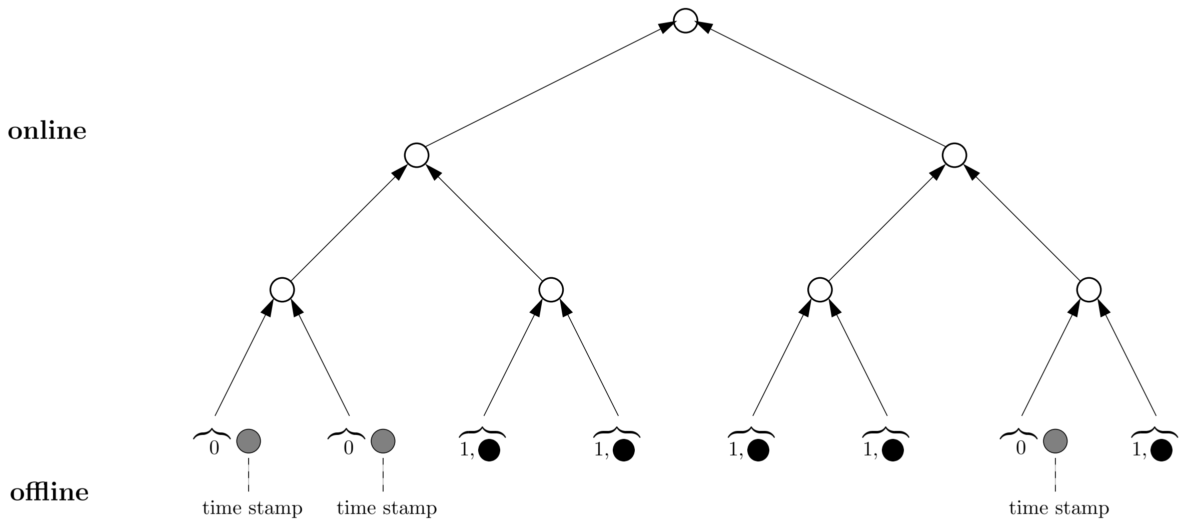


Figure 9: A hybrid checker. There is an extra bit, called a statusBit, associated with each leaf node. statusBits are always protected by the tree. If a leaf node's statusBit is 1, the data value is protected by the online scheme; if a leaf node's statusBit is 0, the data value is protected by the offline scheme.

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 10. 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 Section 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.

9.3 Hybrid Checker

The interface for the hybrid checker is shown in Figure 11. 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 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 6, 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. As 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, as implied by Lemma 9.1, which follows, this attack, and other attacks on the storage, will be detected by the hybrid checker.

Lemma 9.1. *The RAS (that is protected by the offline scheme) has behaved like valid random access storage* `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 Section 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 Section 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 Section 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 Section 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 Section 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 10: Partial-hash tree

if and only if the `hybrid-checkRAS` operation returns true.

Proof. `hybrid-moveToOffline` calls `offline-add`. An adversary cannot have `offline-add` be called on an address accessed in an offline manner *zero* times, by changing its StatusBit from 1 to 0. If the adversary did change the bit from 1 to 0:

- if `hybrid-moveToOffline` is called on the address, `pht-isAddressPresentInTree` would check the StatusBit and detect the tampering.
- if either `hybrid-store` or `hybrid-load` is called on the address, the address would be operated on in an offline manner. In this case, the RAS would not behave like valid RAS as the RAS invariant was not maintained: a `cbag-take` operation would be performed on an address that is not present in the bag. The `offline-checkRAS` would return false, and the tampering would be detected.

An adversary cannot have `offline-add` be called on an address accessed in an offline manner *two or more* times, by changing its StatusBit from 0 to 1. If the adversary did change the bit from 0 to 1:

- `hybrid-moveToOffline` calls `pht-isAddressPresentInTree` which would check the StatusBit and detect the tampering.

Thus, the addresses which `offline-add` is called on form a set. By Lemma 6.3, if the addresses which `offline-add` has been called on form a set, the RAS that is being checked has behaved like valid random access storage if and only if the `offline-checkRAS` operation returns true. `hybrid-checkRAS` returns true if and only if `offline-checkRAS` return true.

□

As described in Section 6.1.1, 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 `hybrid-moveToOffline`(a) move address a to the offline scheme.

1. Call `pht-isAddressPresentInTree`(a).
   (a) If it returns false, a is already present in the offline scheme. Thus, simply return to the caller.
   (b) If it returns true, a is present in the online scheme. a must be moved to the offline scheme.
      i. Call $v =$ `pht-load`(a) to check the integrity of the data value stored at a in the tree.
      ii. Call $v =$ `pht-moveFromTree`(a). (Steps 1, 1(b)i, 1(b)ii can be performed at the same time, using a cache, so that the 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 the StatusBit pertaining to address a.
   (a) if the StatusBit is 1, a is in the online scheme.
      i. Call `pht-isAddressPresentInTree`(a). If it returns true, continue; otherwise, return an error to the caller.
      ii. `pht-store`(a, v). (This step can be performed at the same time as the previous step, using a cache, so that the logarithmic cost of using the hash tree is incurred once.)
   (b) if the StatusBit is 0, a is in the offline scheme.
      i. Call `offline-store`(a, v).

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

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

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 11: Hybrid Offline-Online RAS Checker (Hybrid Checker)

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.

9.3.1 Hybrid Add and Remove operations

The operations `hybrid-add`(a, v, f) and `hybrid-remove`(a) could be added to the interface in Figure 11. `hybrid-add` add's 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 with respect to one of the manners described in Section 6.1. 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 6.1. The appropriate nodes for the address and its statusBit are removed from the partial-hash tree.

9.3.2 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 objects 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 Section 3, the extra space overhead is the statusBits and time stamps. As described in Section 7, 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.

10 Conclusion

In this paper, we investigate offline integrity checking as an alternative method to using hash trees to check the integrity of untrusted storage. The advantages of offline integrity checking over hash tree-based (online) integrity checking include:

- Can perform better as untrusted memory accesses are constant time.
- Can have a smaller space overhead.
- Easier to implement.

In offline integrity checking, there is a constant overhead on the number of checker accesses to the untrusted storage on each program access, as opposed to the logarithmic overhead on the number of accesses incurred by using a tree. The cost is that the addresses that were used must be read to do an integrity check; schemes such as the hybrid online-offline scheme optimize this requirement and facilitate offline integrity checking on large storages.

Though the paper focuses on integrity checking, (data value, time stamp) pairs can be further encrypted when they are stored on the untrusted storage to prevent adversaries from reading them. As the time stamp is different each time a pair is written to the storage, the encrypted pairs will be different each time a pair is written to the storage. We expect offline integrity checking to be particularly useful in applications like certified execution on appliances, and the protection of memory in small devices. We also expect hybrid integrity checking will present an alternative to using hash trees to verify the integrity of very large storage, such as the storage in file systems and databases.

References

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[BEG+91] 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.

[BGG94] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *Extended abstract in Advances in Cryptology - Crypto 94 Proceedings, Lecture Notes in Computer Science Vol. 839, Y. Desmedt ed, Springer-Verlag*, 1994.

[BGR95] M. Bellare, R. Guerin, and P. Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In *CRYPTO '95*, volume 963 of *LNCS*. Springer-Verlag, 1995.

[BM97] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Extended abstract was in Advances in Cryptology- Eurocrypt 97 Proceedings, Lecture Notes in Computer Science Vol. 1233, W. Fumy ed, Springer-Verlag*, 1997.

[CCSP01] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for unix. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, Boston, MA, 2001.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation*, February 1999.

[CL00] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[GSC+03] 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.

[Hen00] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104: HMAC: Keyed-hashing for message authentication, February 1997. Status: INFORMATIONAL.

[LTM+00] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 169–177, November 2000.

[Mer79] R. Merkle. A certified digital signature. In *manuscript*, 1979.

[MS01] 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.

[MVS00] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of OSDI 2000*, 2000.

[NIS95] NIST. FIPS PUB 180-1: Secure Hash Standard, April 1995.

[NN90] J. Naor and M. Naor. Small-bias probability spaces: efficient constructions and applications. In *22nd ACM Symposium on Theory of Computing*, pages 213–223, 1990.

[Riv92] R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm, April 1992. Status: INFORMATIONAL.

[Ros99] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. WCB McGraw-Hill, 1999.

[SHS01] C. Stein, J. Howard, and M. Seltzer. Unifying file system protection. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, 2001.

[SK98] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *The Seventh USENIX Security Symposium Proceedings, USENIX Press*, pages 53–62, January 1998.

A Appendix: Reduction from `MSet-XOR MAC` to `XOR-MAC`

In this appendix, we reduce the set-collision resistance of `MSet-XOR MAC` to the collision resistance of `XOR-MAC`.

Definition A.1. Let $\equiv'_{\mathcal{X}}$ be the comparability operator which just compares the first elements of $\mathcal{H}_{\mathcal{X}}(S)$ triples. That is:

$$(A, C_A, R_A) \equiv'_{\mathcal{X}} (B, C_B, R_B) = \Big(A \oplus H_s(0, R_A) = B \oplus H_s(0, R_B)\Big)$$

Lemma A.1. *It is computationally infeasible to find two sets A and B of polynomial size such that $A \neq B$ and $\mathcal{H}_{\mathcal{X}}(A) \equiv'_{\mathcal{X}} \mathcal{H}_{\mathcal{X}}(B)$.*

Proof. We introduce a numbering v_i of the elements of $\mathcal{V}$. Let $n(S)$ be the index of the highest element in the set S. We define a set of encoding functions over sets, $\mathcal{E}_n : \mathcal{S} \to \{0,1\}^*$, by $|\mathcal{E}_n(S)| = n$ and for $i \leq n(S)$, $\mathcal{E}_n(S)[i] = 1$ if $v_i \in S$ or 0 otherwise.

Let $\mathcal{F}_{\mathrm{XOR-MAC}}$ be the `XOR-MAC` that we defined in section 4.1.2, and $\mathcal{H}^1_{\mathcal{X}}(S)$ denote the first element of the triple $\mathcal{H}_{\mathcal{X}}(S)$. We would now like the pseudorandom functions H_s and F_s to be such that

$$\mathcal{H}^1_{\mathcal{X}}(S) \oplus \bigoplus_{i=1}^{n} F_s(1 \cdot \langle i \rangle \cdot 0) = \mathcal{F}_{\mathrm{XOR-MAC}}(\mathcal{E}_n(S)) \quad \text{if } n \geq n(S) \tag{6}$$

when both algorithms pick the same random nonce R (effectively, we are XOR-ing out the nonce term on each side).

One can verify that a sufficient condition for (6) to hold is

$$H_s(0, x) = F_s(0, x) \tag{7}$$
$$H_s(1, v_i) \oplus F_s(1, \langle i \rangle, 0) = F_s(1, \langle i \rangle, 1) \tag{8}$$

If H_s is fixed, we use (7) to define $F_s(0, x)$. We then define $F_s(1, x, 1)$, to be some arbitrary pseudorandom function H'_s that is not correlated with H_s. In turn this defines $F_s(1, x, 0)$ via equation (8).

If some algorithm B, given oracle access to F_s, can distinguish it from a random function, then it can distinguish $(x, y) \to (H_s(x), H'_s(y))$ from a random function, which is impossible because H_s and H'_s are uncorrelated. So, as constructed, F_s is a pseudorandom function.

Now, let $\mathcal{A}$ be a polynomial time algorithm that outputs two sets A and B such that $A \neq B$ and $\mathcal{H}_{\mathcal{X}}(A) \equiv'_{\mathcal{X}} \mathcal{H}_{\mathcal{X}}(B)$ with high probability.

Let $n_0 = \max(n(A), n(B))$. Applying relation (6) twice gives $\mathcal{F}_{\mathrm{XOR-MAC}}(\mathcal{E}_{n_0}(A)) = \mathcal{F}_{\mathrm{XOR-MAC}}(\mathcal{E}_{n_0}(B))$, so we have found a collision for `XOR-MAC`. Since $\mathcal{E}_{n_0}(A)$ and $\mathcal{E}_{n_0}(B)$ are sparse, they can be used by a polynomial algorithm to find a polynomial sized change to make to a (possibly exponentially large) message to get a collision. This contradicts the collision resistance of `XOR-MAC`. ☐

Theorem A.2. `MSet-XOR MAC` *is set-collision resistant.*

Proof. Let $\mathcal{A}$ be a set-collision finder for $\mathcal{H}_{\mathcal{X}}$. We use $\mathcal{A}$ to produce a set S and a multiset M such that $\mathcal{H}_{\mathcal{X}}(S) \equiv_x \mathcal{H}_{\mathcal{X}}(M)$ and $S \neq M$.[14]

We construct the *set* $S' = \{e \in M : \text{mult}_M(e) = 1 \bmod 2\}$, where $\text{mult}_M(e)$ denotes the multiplicity of e in M. It follows that $\mathcal{H}_{\mathcal{X}}(S') \equiv'_{\mathcal{X}} \mathcal{H}_{\mathcal{X}}(M) \equiv'_{\mathcal{X}} \mathcal{H}_{\mathcal{X}}(S)$. Therefore, because of Lemma A.1, $S = S'$.

If M is a set, then $M = S' = S$, which contradicts the assumption that $\mathcal{A}$ produced a collision for $\mathcal{H}_{\mathcal{X}}$. So M is a multiset, and there exists e_0 in M such that $\text{mult}_M(e_0) > 1$. Moreover, by construction of S', $e_i \in S \implies \text{mult}_M(e_i) \geq 1$. So $|S'| = \sum_{e_i \in S} 1 < \sum_{e_i \in M} \text{mult}_M(e_i) = |M|$.

Summing up, we have $|S| = |S'| \neq |M|$. Because the cardinalities of S and M are polynomial in k, for large enough k the cardinalities will be less than $2^{\frac{k}{3}}$, from which we conclude that the inequality holds modulo $2^{\frac{k}{3}}$. This contradicts the hypothesis that $\mathcal{A}$ had found a collision. So `MSet-XOR MAC` is set-collision resistant. ☐

Note that this proof clearly shows the necessity of including the cardinality of the multiset M that is being hashed in $\mathcal{H}_{\mathcal{X}}(M)$, and shows why $\frac{k}{3}$ bits are sufficient to encode the cardinality.

B Appendix: Proof of the equivalence of the bag definitions

Define $\mathcal{B}_i$ as:

$$\mathcal{B}_i = O_i(\mathcal{B}_{i-1}, M_i) \tag{9}$$

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

$$T_i \cup_{\mathcal{MS}} \mathcal{B}'_i = P_i \tag{10}$$

We prove, by induction, that, for all $0 \leq i \leq n$,

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

That is, (9) defines a unique bag, $\mathcal{B}_i$. (10) defines a unique bag, $\mathcal{B}'_i$. We prove that, for all $0 \leq i \leq n$, $\mathcal{B}_i = \mathcal{B}'_i$; i.e. for all $0 \leq i \leq n$, the two bags are the same.

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

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

Case 1: Put operation:

$$\begin{aligned}
& \mathcal{B}_{i+1} = \text{Put}(\mathcal{B}_i, M_{i+1}) \\
\xleftrightarrow{\texttt{by (1)}}\quad & \mathcal{B}_{i+1} = \mathcal{B}_i \cup_{\mathcal{MS}} M_{i+1} \\
\xleftrightarrow{\texttt{by inductive hypothesis}}\quad & \mathcal{B}_{i+1} = \mathcal{B}'_i \cup_{\mathcal{MS}} M_{i+1} \\
\xleftrightarrow{\texttt{by (10)}}\quad & \mathcal{B}_{i+1} \cup_{\mathcal{MS}} T_i = P_i \cup_{\mathcal{MS}} M_{i+1} \\
\xleftrightarrow{\texttt{by definition of } P_i \texttt{ and } T_i}\quad & \mathcal{B}_{i+1} \cup_{\mathcal{MS}} T_{i+1} = P_{i+1} \\
\xleftrightarrow{\texttt{by (10)}}\quad & \mathcal{B}_{i+1} = B_{i+1}.
\end{aligned}$$

[14] Note that because of the way $\mathcal{H}_{\mathcal{X}}$ and $\equiv_{\mathcal{X}}$ are defined, if S and M collide for some pair of random nonces, then they collide for all pairs of random nonces.

Case 2: TAKE operation:

$$
\begin{aligned}
& \mathcal{B}_{i+1} = \textsc{Take}(\mathcal{B}_i, M_{i+1}) \\
\xleftrightarrow{\texttt{by (2)}} \quad & \mathcal{B}_{i+1} \cup_{\mathcal{MS}} M_{i+1} = \mathcal{B}_i \\
\xleftrightarrow{\texttt{by inductive hypothesis}} \quad & \mathcal{B}_{i+1} \cup_{\mathcal{MS}} M_{i+1} = \mathcal{B}'_i \\
\xleftrightarrow{\texttt{by (10)}} \quad & \mathcal{B}_{i+1} \cup_{\mathcal{MS}} M_{i+1} \cup_{\mathcal{MS}} T_i = P_i \\
\xleftrightarrow{\texttt{by definition of } P_i \texttt{ and } T_i} \quad & \mathcal{B}_{i+1} \cup_{\mathcal{MS}} T_{i+1} = P_{i+1} \\
\xleftrightarrow{\texttt{by (10)}} \quad & \mathcal{B}_{i+1} = B_{i+1}.
\end{aligned}
$$

□

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

C Appendix: Detailed Analysis

In Section C.1, we analyze the space overhead and bandwidth overhead of offline integrity checking. In Section C.2, we analytically compare the offline scheme with the online scheme. Table 1 summarized the space overheads and bandwidth overheads of the offline and online schemes.

C.1 Offline Integrity Checking

C.1.1 Space Overhead

We analyze the per-bit space overhead of the offline scheme. Let t be the number of FSM load and store operations the FSM can perform before a check is required. Each time stamp has to be $log_2(t)$ bits large if there is no cache. If there is a cache, each time stamp needs to be $log_2(t(1-h_v))$ bits where h_v is the cache hit rate - the fraction of FSM storage operations that find their data in the cache.

Denote the size of a time stamp as b_t. Denote the number of data value bits in a block, to which a time stamp is appended, as b_v. The per-bit space overhead of the time stamps is $\frac{b_t}{b_v}$ bits.

C.1.2 Bandwidth Overhead

Bandwidth Consumption of Initialization

Let N be the number of data value blocks to which a time stamp will be appended. An offline checker makes N accesses to the untrusted storage to initialize it. The total number of bits the checker transmits is $N(b_v + b_t)$ bits.

Bandwidth Overhead at Run Time, with a trusted cache

The cache reduces the bandwidth of storage integrity checking. The offline caching algorithm is described in Section 6.2. In the algorithm, the entire cache is used for data values. If the cache hit rate is h_v, the number of FSM accesses to the untrusted storage is $(1-h_v)t$. On each of these accesses, the checker reads a time stamp and writes a time stamp as it brings in a block and evicts a block from the cache. If the evicted block is dirty, as a block is read from the storage and another block is written to the storage, the per-access overhead of time stamps is b_t. If the evicted block is clean (the more common case), as a block is read from the storage but no block is written to the storage, the per-access overhead of time stamps is $2b_t$. Thus, we express the bandwidth overhead of time stamps as $2b_t t(1-h_v)$. Let C be the number of blocks the cache can hold, each block being of size b_v. The bandwidth overhead of the `checkRAS` operation is $(N-C)(b_v+b_t)$ as the checker reads both data values and time stamps unless they are in the cache. Thus, the total bandwidth overhead of the offline scheme is $2b_t t(1-h_v) + (N-C)(b_v+b_t)$.

C.2 Comparison with Online Integrity Checking

C.2.1 Space Overhead

We provide some comparison of the overhead of storing time stamps to the space overhead of a hash tree. The space overhead in the offline scheme is $\frac{b_t}{b_v}$ bits. As described in Section 3, if the size of a leaf is the size of a hash, an m-ary hash tree has a space overhead of approximately $\frac{1}{m-1}$ bits. The parameter, m, is typically small, like 2, 4, or 8, say.

Suppose a cache block is b_v bits large. That is, suppose that, in the offline scheme, a time stamp covers one cache block, and, in the online scheme, a hash covers one cache block. Let b_h be the size of a hash. As one hash covers m leaves, $b_v = mb_h$. The space overhead of the hash tree is thus, $\frac{b_h}{b_v - b_h}$. If $b_t < b_h$, then $\frac{b_t}{b_v} < \frac{b_t}{b_v - b_t} < \frac{b_h}{b_v - b_h}$. This means that if the size of a time stamp is less than the size of a hash, the offline scheme will have a smaller space overhead than the online scheme. The size of a hash is usually 128 bits (MD5 [Riv92]) or 160 bits (SHA-1 [NIS95]). In Section 8, we use 32 bits for each time stamp on a 64 byte data value block (cache line). The experiments are run using 4 GBytes of memory, and $\frac{1}{16}$ of the memory is used to store the time stamps. We contrast this with the experiments in [GSC$^+$03], where over $\frac{1}{4}$ of the memory is used for the hashes of a 4-ary hash tree in a similar experimental set up (MD5 is used for the hashes).

C.2.2 Bandwidth Overhead

Bandwidth Consumption of Initialization

The total number of bits an offline checker transmits to initialize the storage is $N(b_v + b_t)$ bits. To compare, in the initialization approach described in [GSC$^+$03], the online checker uses about $\frac{mN}{m-1}$ accesses to initialize the storage. Thus, the online checker transmits $\frac{b_h mN}{m-1} = \frac{b_v N}{m-1}$ bits to initialize the storage. As $\frac{b_v}{m-1} < (b_v + b_t)$, offline initialization consumes more bandwidth than online initialization.

Bandwidth Consumption at Run Time, with a trusted cache

The cache can be used to improve the performance of the offline scheme described in Section 3. The cache can be used to store both recently-used data values and recently-used hashes. The checker trusts data values stored in the cache, and can perform FSM accesses directly on them without any hashing. If the checker fetches a data value from untrusted storage, it must check the integrity of the value before putting it into the cache. However, instead of checking the entire path from the value to the root of the tree, if the checker finds a hash along that path in the cache, this hash is trusted; the checker can treat the data value as valid when the hash's children are verified. The checker can also store the hashes used in the verification of the value in the cache and use them to check the integrity of other values. When a data value or hash is ejected from the cache, the checker brings its parent into the cache (if it is not already there), and updates the parent in the cache. Storing hashes and data in the cache has a significant performance advantage because each hash node in the tree stored in the cache acts as a root of a tree smaller than the original tree, which reduces the average lengths of the paths from leaves to trusted nodes that the checker must validate. The data value cache hit rate decreases as both data values and hashes are stored in the cache; however, the storage bandwidth consumption is significantly reduced as less hashes are fetched from the storage. A trusted cache, which stores both hashes and data values, is necessary to efficiently implement a hash tree integrity checking scheme [GSC$^+$03].

In the integrated hash tree/caching online algorithm, the cache hit rate of data values decreases. We denote this cache hit rate as δh_v, where $0 \leq \delta \leq 1$. We also denote the average number of hashes the checker fetches from the storage on each FSM storage operation as z. Thus, the checker's bandwidth overhead to read hashes from the untrusted storage is $t(1 - \delta h_v)b_h z$. The parameters δ and z depend on the access patterns of the FSM and the relative size of the cache to the size of untrusted storage.

For the comparison, we see that if $t \geq \frac{(N-C)(b_v+b_t)}{(1-\delta h_v)b_h z - 2b_t(1-h_v)}$, the cost of fetching hashes on each FSM storage operation in an online checker will exceed the cost of reading the addresses the FSM used to perform the `checkRAS` operation in an offline checker, and the offline checker will consume less bandwidth than an online checker. In the experiments in Section 8, we are concerned with the number of FSM storage accesses $T = (1 - h_v)t$ for the offline case: these are the number of FSM accesses (loads and stores) on the untrusted storage. If we assume $\delta = 1$ for large caches, the offline scheme is better than the online scheme if $T \geq \frac{(N-C)(b_v+b_t)}{b_h z - 2b_t}$.

Note that the smaller z is, the larger is the bound. In other words, the more the cache reduces the logarithmic overhead of using the hash tree, the larger the offline integrity checking period must be before the advantage of the constant overhead per access of the offline scheme takes effect. Also note that the larger δ is, the larger is the hit rate of data values in the cache, and the larger is the bound. This is also intuitive, as the larger the data value hit rate, the smaller the number of accesses the checker makes to the storage, and the less the likelihood the performance of the checker will be affected by the cost of using the tree.