Offline Authentication of Untrusted Storage

Dwaine Clarke, Blaise Gassend, G. Edward Suh, Marten van Dijk[1], Srinivas Devadas

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

[1]Philips Research
marten@caa.lcs.mit.edu

August 23, 2002

Abstract

We extend the offline memory correctness checking scheme presented by Blum et. al [BEG+91], by using incremental cryptography, to detect attacks by an active adversary. We also introduce a hybrid offline-online checking scheme designed for untrusted storages in file systems and databases. Previous work [GSC+02] [FKM00] [MVS00] describe systems in which Merkle trees are used to verify the authenticity of data stored on untrusted storage. The Merkle trees [Mer79] are used to check, after *each* operation, whether the storage performed correctly. The offline and hybrid checkers are designed for checking *sequences* of operations on an untrusted storage and, in the common case, require only a constant overhead on the number of accesses to the storage, as compared to the logarithmic overhead incurred by online Merkle tree schemes.

1 Introduction

In [BEG+91], Blum et al. describe an offline checker using ϵ-based hash functions [NN90], that can be used to detect random errors after a *sequence* of operations has been performed on random access memory (RAM). The offline checker is contrasted with an online checker using Merkle trees [Mer79], which can be used to detect, after *each* operation, whether the untrusted storage malfunctioned or was tampered with by an active adversary. In this paper, we extend the work presented by Blum et. al [BEG+91] to implement an offline checker using incremental cryptography, thus enabling the detection of tampering by an active adversary of data stored on untrusted storage. We also introduce a hybrid offline-online checking scheme designed for untrusted storages in file systems and databases.

In the model we use, the *checker* keeps and maintains some small, trusted, state which is updated with each access to the *untrusted storage* by the *program* the checker is running. As an example, in the case of untrusted RAM, the checker is the processor and the program that is being run is some user program. At the end of the program's execution, or at periodic times during the program's execution, a check is performed using the state in the processor. If the RAM has malfunctioned, or has been tampered with by an adversary, the check fails.

Offline checking and hybrid checking can be particularly efficient in applications like certified execution [GCvDD02], in which it is necessary to know at the end of the program's execution, if the untrusted storage has performed correctly; however, in the case the check fails, it is not necessary to know which particular operation malfunctioned, or which value was tampered with.

2 Offline Checking using Incremental Cryptography

Our offline authentication scheme uses the basic strategy described in [BEG+91], but uses XOR MACs [BGR95] to update the state in checker. The authentic state consists of 2 data values, WriteValue and ReadValue, and a timer, Timer. WriteValue encodes all of the checker's writes to the untrusted storage, and ReadValue encodes all of the checker's reads from the untrusted storage. The data values are updated using XOR MACs, a subclass of keyed hash functions. There are 3 properties provided by XOR MACs that we require for our scheme:

- *compression*: XOR MACs map an input of arbitrary length to an output of fixed length. The output of the XOR MAC is also small enough to require only a small amount of trusted state in the checker.
- *incrementality*: updating MACs produced by XOR MACs can be performed quickly. That is, given the MAC of some data, updating the MAC after changing the data can be done quickly with a cost that is proportional to the amount of change of the data. Thus, if the data is changed, the MAC does not have to be re-computed over the entire data. Schemes have been developed to support incrementally updating the MACs after inserting, deleting, or replacing blocks in the original data [BGG95].
- *collision-resistance*: if the key used to create the MAC is not known, then it is computationally infeasible to find two distinct inputs which hash to the same output.

In particular, we will be using schemes to append a block to an XOR MAC and to replace a block of an XOR MAC with another block. We will also be using the counter-based XOR MAC scheme described in [BGR95]. Thus, the checker also keeps and maintains an XOR MAC counter, XOR-MAC-Counter; this counter is initialized to 0 at the beginning of the scheme. Let the XOR MAC counter-tag pair, (C, z), be an XOR MAC for an n block message M. Let F_a be the description of the XOR MAC function that the checker is using. To append block m to M as the $(n+1)^{th}$ block:

- increment XOR-MAC-Counter. Let C' be the new value of XOR-MAC-Counter.
- set $z' = z \oplus F_a(0.C) \oplus F_a(0.C') \oplus F_a(1.\texttt{<}n+1\texttt{>}.m)$. ($\alpha.\beta$ is the concatenation of strings α and β.)
- Then (C', z') is the XOR MAC for $M.m$.

To replace block i of message M with m:

- increment XOR-MAC-Counter. Let C' be the new value of XOR-MAC-Counter.
- set $z' = z \oplus F_a(0.C) \oplus F_a(0.C') \oplus F_a(1.\texttt{<}i\texttt{>}.M[i]) \oplus F_a(1.\texttt{<}i\texttt{>}.m)$. ($M[i]$ is the current i^{th} block of M.)

- Then (C', z') is the XOR MAC for M with the i^{th} block replaced with m.

The WRITEVALUE and READVALUE are updated on each program store and program load. Each time any data value is written to storage, the timer is incremented, and the current time is also stored with the data value. Keeping a time value with the data value is important as the checker uses it to verify, at the end, that the data value retrieved from reading a particular storage address at a particular time is the last data value previously written to that address.

Each write to storage is represented as a triple (data, address, time). WRITEVALUE is, in essence, a hash of an ordered array (log) of triples representing writes to storage. Similarly, each read from storage is represented as a triple (data, address, time) and READVALUE is, in essence, a hash of an ordered array (log) of triples representing storage reads. The triples in each array represented by WRITEVALUE and READVALUE are ordered by the time values in the triples[1].

A description of the offline authentication scheme follows. In the scheme, triples are appended to the WRITEVALUE, whereas stubs in the READVALUE are replaced with triples. For the stubs, we use a generic block, $F_a(1.0.0)$.

At the beginning of the scheme, for each address in the storage, the checker:

- increments TIMER.
- writes (0, currentTime) to the address. currentTime is the current value of TIMER.
- appends the triple (0, address, currentTime) to WRITEVALUE as the currentTimeth block. The checker also appends a stub block, $F_a(1.0.0)$, to READVALUE as the currentTimeth block. (TIMER is initialized to 1 at the beginning of the scheme, thus the stub does not conflict with other blocks).

On each program store of value, newValue, to address, address, the checker:

- reads the old data, (oldValue, oldTime) stored at address. oldTime is the time when oldValue was written to storage at address.
- checks that oldTime is less than or equal to the current time (the current value of TIMER).
- replaces the stub at the oldTimeth position in READVALUE with (oldValue, address, oldTime).
- increments TIMER.
- writes the new data and current time to address. Thus, the pair that is written to address in storage is (newValue, currentTime). Again, currentTime is the new current value of TIMER.
- appends the triple (newValue, address, currentTime) to WRITEVALUE as the currentTimeth block. The checker also appends a stub block, $F_a(1.0.0)$, to READVALUE as the currentTimeth block.

Program loads are similar to program stores. On each program load of the value at address, address, the checker:

- reads the old data, (value, oldTime) stored at address.

[1]For the scheme, the *set* of triples that is written is checked against the *set* of triples that is read. The ordering of the triples is only necessary as we are using MACs to create and update READVALUE and WRITEVALUE.

- checks that oldTime is less than or equal to the current time.
- replaces the stub at the oldTimeth position in ReadValue with (oldValue, address, oldTime).
- increments Timer.
- writes the same data and current time to address. Thus, the pair that is written to address in storage is (value, currentTime).
- appends the triple (value, address, currentTime) to WriteValue as the currentTimeth block. The checker also appends a stub block, $F_a(1.0.0)$, to ReadValue as the currentTimeth block.

After the program has finished executing, or when the checker wants to check the authenticity of the storage, the checker traverses the storage to read all of the storage addresses once again. For each address, the checker checks that the time read is less than or equal to the current time, and replaces the stub in the appropriate position in ReadValue with the triple corresponding to the data read at the address. If the storage malfunctions or has been tampered with, then WriteValue would not be equal to ReadValue after the storage has been read as the set of triples which were written would not be equal to the set of triples that were read.

The formal proof of the underlying scheme is presented in [BEG+91]. We present some intuition here. Suppose the storage was one address large. Then, on each program store and program load, if WriteValue is equal to ReadValue at the step at which Timer is incremented, then the (value, time) triple that was read from storage on the program operation is the same as that which was written to storage on the previous program operation. When an authentication check is desired, the storage has to be read again to update the ReadValue. If the adversary, tampers with the value or time stored at the address at any time, there will exist a triple that would be in WriteValue but not be in ReadValue.

The advantage of the offline scheme is that it requires only a small *constant* overhead per program store/load operation, making it more efficient for some applications. The tradeoffs are that the entire data storage needs to be read when an authenticity check is desired, and it may not be possible to determine which particular value was tampered with if a check fails.

2.1 Incremental Cryptography Alternatives

The offline scheme described in the previous section uses XOR-MACs. However, an incremental hashing scheme which does not use secret keys is sufficient as well. Also, as the scheme checks whether the *set* of triples that is written is the same as the *set* of triples that is read, it is possible to use incremental cryptographic schemes which produce the same output if two input sets are the same, and the order in which blocks were added to the inputs is different.

[BGG94] introduces incremental hashes and [BM97] studies several incremental hashing schemes with various degrees of efficiency and security. Incremental hashes described in these papers can be used for the offline checker, in place of XOR-MACs. We also believe that incremental cryptography schemes in the literature can be adapted to work for sets. Thus, henceforth, we will refer to operations like appending m to WriteValue or replacing a block in ReadValue with m as 'updating' WriteValue or ReadValue with m.

3 Hybrid Checking

One of the disadvantages of the offline checking scheme is that the entire storage needs to be read when a check is desired. This makes the scheme impractical for file systems and databases. To address this problem, we propose a hybrid offline-online scheme, which still has a constant overhead per program store/load operation in the common case, but does not require the entire storage to be read when a check is desired.

Essentially, a Merkle tree is kept over a copy of the data in the untrusted storage. To perform a sequence of operations on a subset of the data, the data is operated on as described in Section 2. When an authenticity check of these operations is desired, an offline check is performed on the data that was used during the operations. A linked list of pointers is used to keep track of which addresses were accessed since the operations were started. If the offline check fails, the checker reports that the storage has been tampered with. During the check, the checker copies the data back into the Merkle tree. If the check passes, the process can be repeated the next time a sequence of operations is performed. As the checker only maintains hash values/MAC values for the ReadValue and WriteValue, the Merkle tree is used to remember the values at the end of successful previous offline checks.

The checker still maintains a fixed set of data: Timer, HeadPointer, ReadValue, WriteValue, and MerkleTreeRoot. HeadPointer is a pointer to an address, and MerkleTreeRoot is the root of the Merkle tree.

Two bookkeeping issues to address are:

- determining when the checker accesses a particular address for the first time during the current sequence of operations. At this time, the checker should check the data at that address in the Merkle tree. Otherwise, the checker should just perform the offline store/load described in Section 2.
- determining the set of addresses that were accessed during the current sequence of operations. We address this problem using a linked list, the head of which is maintained in the checker. The pointers of the linked list are protected by the Merkle tree.

Each write and read to and from the untrusted storage is represented as a 5-tuple (data, address, time, FirstTimeBit, AddressPointer). FirstTimeBit is a single bit: a 0 indicates that this is the first time the checker is accessing the address; a 1 indicates that this is the second or more time the checker is accessing the address during this sequence of operations. AddressPointer is a pointer to the next address in a linked list of addresses.

For each address that is accessed during a sequence of operations, there are two datum, one that is protected by the Merkle tree, and one that will be protected by the offline checking. One way of handling this issue is to have a local space for the data protected by the Merkle tree (Merkle-Tree-Space) and another local space for the data protected by the offline checker (Offline-Space).

To describe the scheme, we present four cases: 1) program load when the FirstTimeBit is 0, 2) program store when FirstTimeBit is 0, 3) program load when the FirstTimeBit is 1, and 4) program store when the FirstTimeBit is 1. Changes that the checker makes to data that it reads are underlined.

Case 1: Program load when the FirstTimeBit is 0

On each program load of value, value, to address, address, the checker:

1. reads the data, (value, oldTime, firstTimeBit, addressPointer), stored at address in the OFFLINE-SPACE.
2. if firstTimeBit is 0
   (a) checks (value, oldTime, 0, addressPointer) in Merkle tree. If Merkle tree check fails, exit with failure signal.
   (b) updates the entry in Merkle tree to be (value, oldTime, 1, HEADPOINTER).
   (c) increments TIMER.
   (d) writes (value, currentTime, 1, HEADPOINTER) to address in the OFFLINE-SPACE.
   (e) updates WRITEVALUE with (value, address, currentTime, 1, HEADPOINTER).
   (f) updates HEADPOINTER to be address.

Case 2: Program store when the FIRSTTIMEBIT is 0

On each program store of value, newValue, to address, address, the checker:

1. reads the data, (oldValue, oldTime, firstTimeBit, addressPointer), stored at address in the OFFLINE-SPACE.
2. if firstTimeBit is 0
   (a) checks (oldValue, oldTime, 0, addressPointer) in Merkle tree. If Merkle tree check fails, exit with failure signal.
   (b) updates entry in Merkle tree to be (oldValue, oldTime, 1, HEADPOINTER).
   (c) increments TIMER.
   (d) writes (newValue, currentTime, 1, HEADPOINTER) to address in the OFFLINE-SPACE.
   (e) updates WRITEVALUE with (newValue, address, currentTime, 1, HEADPOINTER).
   (f) updates HEADPOINTER to be address.

Case 3 and 4 are basically the same as the program load and store in Section 2. We restate them again for clarity.

Case 3: Program load when the FIRSTTIMEBIT is 1

On each program load of value, value, to address, address, the checker:

1. reads the data, (value, oldTime, firstTimeBit, addressPointer), stored at address in the OFFLINE-SPACE.
2. if firstTimeBit is 1
   (a) checks that oldTime is less than or equal to the current time.
   (b) updates READVALUE with (value, address, oldTime, 1, addressPointer)
   (c) increments TIMER.
   (d) writes (value, currentTime, 1, addressPointer) to address in the OFFLINE-SPACE.

(e) updates WriteValue with (value, address, currentTime, 1, addressPointer).

Case 4: Program store when the FirstTimeBit is 1

On each program store of value, newValue, to address, address, the checker:

1. reads the data, (oldValue, oldTime, firstTimeBit, addressPointer), stored at address in the Offline-Space.
2. if firstTimeBit is 1
   (a) checks that oldTime is less than or equal to the current time.
   (b) updates ReadValue with (oldValue, address, oldTime, 1, addressPointer)
   (c) increments Timer.
   (d) writes (newValue, currentTime, 1, addressPointer) to address in the Offline-Space.
   (e) updates WriteValue with (newValue, address, currentTime, 1, addressPointer).

At the end of the sequence of operations, the checker iterates through the linked list, starting with HeadPointer. The checker reads the data stored at each address and:

- checks the next pointer in the tuple in the Merkle-Tree-Space using the Merkle tree.
- adds the tuple in the Offline-Space, (value, address, time, firstTimeBit, addressPointer), to ReadValue.
- updates the tuple in the Merkle-Tree-Space with the tuple (value, time, 0, NULL), corresponding to the tuple that was read from the Offline-Space.
- writes (value, time, 0, NULL) to the address in the Offline-Space.

After all of the addresses that were accessed during the sequence of operations have been read again, the offline test of checking that WriteValue is equal to ReadValue is performed. If WriteValue is equal to ReadValue, the offline check has succeeded, and HeadPointer is re-initialized to NULL (WriteValue, ReadValue, and Timer can also be re-initialized to 0).

One optimization is to keep a collision-free hash [Riv92] of the values in the tuples at the leaves of the tree in the Merkle-Tree-Space. Thus, each tuple in the Merkle tree is (hash(value), time, firstTimeBit, addressPointer). This reduces the space overhead of the Merkle tree when the values are large.

The implementation that we have described is chosen for clarity. We note that it is possible to optimize the implementation by leaving out some of the entries in the tuples in the Merkle-Tree-Space and the Offline-Space.

With the hybrid checking scheme, the checker still maintains a fixed set of data. The overhead is mostly constant, assuming that most of the addresses are accessed multiple times during the sequence of operations. Also, only the addresses that were accessed during the sequence of operations need to read when a check is desired.

3.1 Analysis of the Hybrid Scheme

In this subsection, we argue that the hybrid scheme is correct, i.e., that if an adversary tampers with the data that is used for the sequence of operations, he will be detected. First we will use the FirstTimeBit bits to show that each address that is accessed is added once to the linked list, and is added to the list the first time the address is accessed; otherwise, the adversary will be detected. We will then argue that the adversary cannot tamper with any of the pointers in the linked list. This means that each address that was accessed during the sequence of operations will be read exactly once at the end when the check is desired. Thus, the scheme reduces to the offline scheme described in Section 2 and the adversary cannot tamper with either the values or times without being detected.

We show that the adversary cannot have an address that is accessed be added to the linked list *zero* times, by changing FirstTimeBit from 0 to 1. Suppose the adversary changes FirstTimeBit from 0 to 1. As the address has been accessed, there must be some access that happens for the first time. For this first access, as FirstTimeBit is 1, the checker adds the 5-tuple (value, address, time, 1, addressPointer) corresponding to the data it read to ReadValue. The checker checks the time on such reads so time must be less than the current time. The checker then increments the timer, and adds a tuple with the new time to WriteValue. The entry that was added to ReadValue will never match any entry in WriteValue for this first access, as entries for the address will be added to WriteValue with times strictly greater than time. Thus, the WriteValue and ReadValue will not be equal when the check is performed and the adversary will be detected.

We show that the adversary cannot have an address that is accessed be added to the linked list *two or more times*, by changing FirstTimeBit from 1 to 0 after the address has been added to the linked list. Suppose the adversary changes FirstTimeBit from 1 to 0. As FirstTimeBit is 0, the checker will check the tuple in the Merkle tree. Recall that after the first access to that address, the FirstTimeBit is updated to 1 in the Merkle tree, and it is not reset to 0 until during the final read at the end. When the checker checks the tuple in the Merkle tree, this check will fail, and the adversary will be detected.

Thus, an address can only be added to the linked list once. An adversary may try to have the address added sometime after the first time it is accessed, by changing the FirstTimeBit from 0 to 1 for the first storage access, then, sometime later, changing it from 1 to 0. This is a subset of the attack when the adversary tries to have the address added zero times. Thus, the adversary will still be detected. Thus, each address can only be added once to the linked list, at the first time it is accessed. When the address is added to the linked list, WriteValue is updated with the appropriate tuple.

The address pointers in the tuples are protected by the Merkle tree. They are securely updated the first time an address is accessed, and checked when the linked list is iterated at the end. Thus, the adversary cannot tamper with the address pointers, and try to delete addresses, say, from the linked list. HeadPointer is initialized to NULL before the sequence of operations are performed. When an entry is first added to the linked list, its address pointer is NULL, and this is protected by the Merkle tree. Thus, the adversary cannot add addresses to end of the linked list.

Thus, each address that was accessed during the sequence of operations will get read exactly once at the end when the check is desired. Therefore, the scheme reduces to the original offline checking scheme, and the adversary cannot tamper with the values or times either without being detected.

4 Conclusion

We use incremental cryptography to implement the offline checker described by Blum [BEG+91] and integrate Merkle trees into the offline scheme to develop a hybrid checker for file systems and databases. These checkers are useful for checking whether a sequence of operations on an untrusted storage are correct.

References

[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.

[BGG95] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography with application to virus protection. In *27th Annual Symposium on the Theory of Computing*, 1995.

[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.

[FKM00] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 181–196, San Diego, California, October 2000.

[GCvDD02] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Protocols and applications for controlled physical unknown functions. In *Laboratory for Computer Science Technical Report 845*, June 2002.

[GSC+02] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and merkle trees for efficient memory authentication. In *Laboratory for Computer Science Technical Report 857*, July 2002.

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

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

[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, 1992. Status: INFORMATIONAL.