Distributed Memory Integrity Trees

Ofir Shwartz and Yitzhak Birk

Abstract—Ensuring the correct execution of a program running on untrusted computing platforms, wherein the OS, hypervisor, and all off-CPU-chip hardware, including memory, are untrusted, (also) requires protecting the integrity of the memory content against replay attacks. This requires dedicated tracking structures and in-chip state storage. For this purpose, integrity trees are used in various forms, varying in complexity, size, and performance; yet, existing integrity trees do not address distributed, shared-memory computations, for which one must also ensure the integrity of the coherence state of the memory. Observing that a block not residing at a given node merely needs to be known by that node as such, we present the novel Distributed Integrity Tree (DIT) method, and show that it can be used effectively to extend existing integrity trees to parallel and distributed environments. Using DIT, we constructed a Distributed Merkle Tree, a Distributed Bonsai Merkle Tree, and a distributed Intel SGX's Memory Encryption Engine integrity mechanism. All these extensions entail negligible overhead.

Index Terms—Distributed computing, computer security, shared memory, integrity tree

1 Introduction

Secure computing in untrusted environments like public clouds is an emerging requirement. A common (e.g., SGX [7], SeM [9]) assumption is that the secure CPU chip and the user's own code are the only trusted elements. Everything else, including the board and off-chip memory as well as the operating system (OS) and hypervisor, is untrusted. Encryption can protect the confidentiality of the data residing in untrusted memory, and message authenticating code (MAC) can protect against forged or misplaced data; however, replay attacks (whereby old data is maliciously restored) by a privileged attacker may harm the integrity (version) of the data, and this requires additional measures. For this, various secure CPU-managed integrity trees are used; however, these only protect a program running at a single compute node, whereas many relevant application programs are parallel or distributed.

Distributed applications commonly use message passing interface (MPI) [1] or distributed shared memory (DSM) [2]. In MPI, the memory space itself is not distributed, so the single-node integrity solutions suffice. DSM, wherein threads are spawned and access a shared address space for data sharing and synchronization, is natively easier to program and thus attractive. However, this sharing requires a distributed integrity-preserving mechanism. Also, since the DSM coherence state metadata is managed by the underlying (untrusted) OS, it is vulnerable to manipulations. (e.g., preventing the invalidation of a local memory block, causing the read of an old value; or granting write permission without remote invalidations, causing incoherent blocks.) Secure CPUs that protect at application granularity (e.g., [7], [9]) cannot protect this metadata against privileged attackers, and solutions like AMD-SEV [11] lack memory integrity protection.

In this work, we present the Integrity-Verified Local Coherence State (IVLCS) mechanism, which protects the coherence state of *shared memory* blocks against malicious manipulations. We then present *Distributed Integrity Tree* (DIT), a novel scheme that uses IVLCS, any existing inter-node secure coherent data transfer layer (a coherence preserving layer that transfers data securely, e.g., [3], [10]), and any single-node integrity tree to construct a corresponding distributed integrity tree. Finally, we use DIT to construct distributed versions of Merkle Tree [4], Bonsai Merkle Tree [5], and of Intel SGX's [7] Memory Encryption Engine (MEE) integrity mechanism [6]. Although we target protection at a process level, DIT may also protect at a virtual machine level.

- *Authors are with the Electrical Engineering Department, Technion, Haifa 3200003, Israel. E-mail: ofirshw@tx.technion.ac.il, birk@ee.technion.ac.il.*

Manuscript received 29 Sept. 2017; revised 1 Mar. 2018; accepted 27 Mar. 2018. Date of publication 2 Apr. 2018; date of current version 25 June 2018.
(Corresponding author: Ofir Shwartz).
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2018.2822705

2 Existing Single-Node Integrity Trees

Merkle Tree [4]. A secure hash (based on a secret key) is calculated from each memory block (data or instructions); the hashes are stored in the clear in 'hash blocks' in the untrusted memory. For each hash block, a secure hash is calculated, and this is repeated in a tree structure whose single "root" hash value protects the integrity of the entire tree. The root never leaves the CPU chip, so it cannot be forged. The use of a chip-resident secure root hash ensures that maliciously restored old data blocks with their old hashes will fail verification when read into the CPU chip.

By caching hash blocks on chip, a missing hash fetched from the unprotected memory only needs to be validated up to the first in-cache ancestor hash: the latter was verified when fetched (and possibly updated for subsequent sub-tree modifications), and cannot be modified by an attacker while cached, so it can be treated as a root hash and the hash validation is completed. A hash block is only updated upon eviction of its descendant block, and only if modified.

Bonsai Merkle Tree (BMT) [5] targets systems that protect their memory using counter mode encryption, wherein each memory block has a corresponding counter value. BMT protects the *counters* using a Merkle tree, so forged counters (e.g., old counter with old data and MAC) are detected upon counter block fetch. Also, each data block has a small MAC alongside, so forged data (assuming a correct counter) will be detected upon fetch. The BMT values are stored in the clear in the unprotected memory, and can be cached in the chip. The Bonsai Merkle Tree is smaller than the Merkle Tree, thus increasing performance while providing the same security guarantees.

Intel Memory Encryption Engine (MEE) [6] integrity tree is used in SGX [7]. In MEE, each *data* block has a *version*, and a Tag that is calculated using a MAC algorithm over the *data* and the *version*. The versions blocks are protected similarly (using an upper level version and a Tag), forming a tree structure up to a root version that is kept on chip.

Unfortunately, a distributed program's memory space spans multiple memories, controlled by different CPUs, so none of the above **can** be used as is to protect it.

3 Distributed Integrity Trees

A single-node integrity tree is responsible for its CPU's fixed set of memory blocks. In a DSM, however, valid blocks move among nodes, so a collection of local integrity trees with fixed responsibilities does not work. Using instead a single root value (stored in one of the secure CPUs) to maintain the integrity of the entire distributed memory is impractical, because this value must change upon any write to memory, overwhelming both the network and the CPU holding the root value. A replicated global integrity tree is even more expensive to create and maintain.

Our key observation is that, by having a secure coherent inter-node data transfer layer (e.g., [3], [10]), each node need only 1) know with certainty which blocks are present in its local memory (*"locally resident"*), and 2) only maintain the integrity of those. Specifically, each node must maintain an *integrity-protected*

1556-6056 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: Technion Israel Institute of Technology. Downloaded on January 21,2021 at 12:33:54 UTC from IEEE Xplore. Restrictions apply.

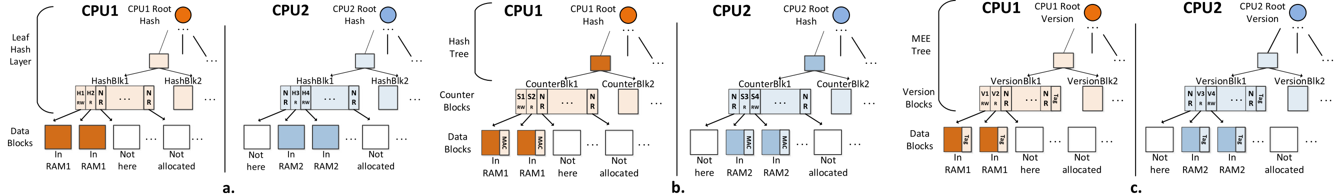

Fig. 1. DIT examples, a. DMT b. DBMT c. DMEE. Some blocks are writable, some shared, some only in one CPU, and some are not allocated.

coherence state ("locally resident + write permission"/"not resident") of all its shared memory blocks, be they locally resident or not. However, a node need not even be aware of changes being made to a non-resident block's content or location until it is fetched again. We next exploit this key insight.

Definition. The *local coherence state* of a block is *integrity verified* (IVLCS) iff the coherence state of the block is protected by a local integrity preserving mechanism: when the CPU checks if a block resides locally, the results are either correct or an error is declared.

Constructing the IVLCS is done via the following modifications to existing integrity verified structures:

1) Assign a special value 'NR' to mark a locally non-resident block (unallocated or currently present at a different node).
2) Add a per-block write-permission bit $(0 - \mathrm{R}, 1 - \mathrm{RW})$.

'NR' can be kept in the data block itself, in the leaf of the hash tree responsible for the integrity of this block, in place of the counter of this block (if counter mode is used), etc.; the write permission bit can be kept alongside. NR can either be marked using a dedicated value of existing bits, or using an additional bit (incurring memory overhead). In fact, one may use a partial integrity tree protecting only the allocated and locally existing part of the memory, in which case NR can also mark an unallocated or locally unavailable subtree (i.e., a block containing solely NR's is omitted, and its predecessor hash contains NR, recursively).

The Distributed Integrity Tree (DIT) scheme. DIT uses secure CPUs that employ a secure coherent inter-node data transfer layer (e.g., [3], [10]) to protect the data while in transit, along with integrity verified local coherence state (IVLCS) and single-node local memory integrity trees to protect the data while in a secure node.

The data transfer layer, embedded in the secure CPU, is responsible for trustfully locating the current owner of a requested block (e.g., by communicating with a trusted directory), and providing encrypted (if desired) and integrity protected data transfer between the current owner and the requesting node. Any tamper attempt is detected, and an error is treated by the trusted compute node.

Each CPU has a single-node local memory integrity tree. Locally resident data blocks (along with their integrity-protecting metadata) are fetched into the CPU both for local use and for verification prior to being sent remotely. The local memory integrity mechanism detects modifications of both data and integrity blocks (as done in single-node integrity trees). This also holds for local permission checks, which are integrity protected (by IVLCS definition). Non-resident blocks are determined properly (by IVLCS), so only the missing ones are requested from their remote owner. These are requested by the secure CPU, their current owner is detected, it sends them, they are received correctly, and are invalidated (when required)—all this by the data transfer layer. Having arrived, their local coherence state is trustfully updated in the local integrity structure, and these become locally resident blocks. DIT can thus be used to preserve the integrity of the entire distributed shared memory. It thus provides all the truly required functions of a global integrity tree at a cost and performance that are very similar to those of independent local trees.

DIT was described with cache-block sharing granularity. For memory-page granularity DSM, an in-CPU buffer can hold the rest of the page while updating its integrity, not delaying the availability of the requested cache line. Other approaches, like bringing the entire page into the local cache and marking it 'dirty', remain for future research.

4 Distributed Tree Examples

We now apply DIT to the three cited single-node trees, yielding corresponding distributed integrity trees. For each, we choose the data structure that marks a non-resident block and write permission, and discuss its behavior.

Distributed Merkle Tree (DMT). We let the hash corresponding to a data block (leaf hash of the Merkle Tree) contain the actual hash for an residing block, and 'NR' for unmapped blocks or ones currently residing only in other CPUs' local memory. (An absent data block could itself be marked 'NR', but its hash is smaller so hash marking is more efficient.) A per-block write permission bit is kept alongside the leaf hash, and is checked before a write operation. Since hash values are also kept in blocks, hashes of residing and non- residing blocks ('NR') may reside in the same hash block. A per-CPU local Merkle Tree is maintained normally, and each CPU maintains a local root hash of its local Merkle Tree. (See Fig. 1a).

On a cache hit, a block is accessed directly; if a first write is requested, then write permission is requested from the secure coherent node-to-node data transfer layer. On a cache miss, the block's hash (and write-permission bit) is examined. If the stored hash value is not 'NR', the block is simply fetched from the local memory; if it is 'NR', a request is sent to the data transfer layer for validating and then bringing the block securely (by definition). Once the requested block arrives, it is securely stored in the cache, marked 'dirty', and its write-permission bit is updated based on the request type. Only when a 'dirty' block is locally evicted, its hash must be updated, since until then the block is simply accessed via the cache. Similarly, upper level hash blocks are updated as soon as their children are evicted. If a cached modified block is requested by others, its write permission is revoked; its local hash is either updated instantly or upon eviction. If a CPU invalidates a block, its hash becomes 'NR' and the local integrity structure is updated similarly.

Distributed Bonsai Merkle Tree (DBMT). We construct DBMT with private per-CPU encryption counters (not shared), so other CPUs cannot access them, thereby avoiding redundant management (inter-node communication for counter updates even for inactive blocks) and potential false sharing. The counter corresponding to the data block either contains the actual counter for a resident block, or 'NR' for unmapped blocks and ones residing only in other CPUs' local memory. A per-block write-permission bit is kept alongside the counter (also covered by the MAC calculation), and it is checked before a write operation. A per-CPU local BMT is maintained normally, and each CPU maintains a local root hash of its local BMT. See Fig. 1b.

Distributed Memory Encryption Engine (DMEE). Although Intel SGX currently does not support distributed execution, we show that its MEE integrity tree can also benefit from DIT. The tree structure is fairly similar to BMT, so we construct DMEE with private per-CPU versions. The version corresponding to the data block either contains the actual version for a resident block, or 'NR' otherwise, alongside a write permission bit. A per-CPU local-MEE is

Authorized licensed use limited to: Technion Israel Institute of Technology. Downloaded on January 21,2021 at 12:33:54 UTC from IEEE Xplore. Restrictions apply.

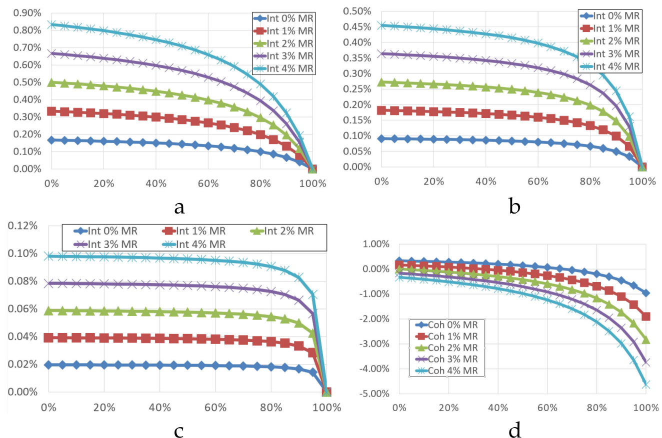

Fig. 2. DBMT average memory access time overhead for various node miss rates, with various integrity check miss rates and remote fetch times (a,b,c) and with various local coherence check miss rates (d).

maintained normally, and each CPU maintains a local root version of its local-MEE. See Fig. 1c.

Note that in DIT, migrating a thread to a new CPU does not require upfront data movement; rather, only an empty hash/version/counter structure, a new empty local-integrity tree, and a root hash. The thread's code and data will be subsequently fetched as shared data in the system.

5 Evaluation

We evaluate the additional operations performed by DIT relative to a baseline with secure coherent inter-node data transfer (SDSM) and single-node integrity trees, but without coherence integrity (and therefore no global memory integrity). We consider memory sharing at cache block granularity, though coarser granularity is possible. Therefore, the baseline's OS must verify the status of memory blocks upon access. Consider the following cases:

Locally resident block—cache hit. Both in the baseline and in DIT, a cached block is accessed directly for read. Before writing for the first time, write permission is first requested by the data transfer layer; however, an already *modified* block does not require a permission request. Therefore, performance is similar.

Locally resident block—cache miss. In the baseline, a missing cache block results in a local coherence lookup to check for the presence of the block. It is then fetched and verified with the local integrity system. In DIT, the value fetched for the local coherence lookup (hash / counter / version) also serves for verifying the integrity of the resident block, so no additional overhead is caused.

Locally non-resident block. In the baseline, a missing cache block results in a local coherence lookup to check for the existence of the block. If not resident, a data transfer request is sent, and the block arrives into the cache safely. The integrity structure is only updated when this block is evicted. With DIT, an integrity protected coherence lookup is performed, which is likely to be more costly than the local coherence lookup. Then, a remote request is performed similarly.

The average read time for the baseline is:

$$t_b = t_c + (1 - H) \cdot \left(t_{coh} + t_{fetch} + LE \cdot t_{int} + (1 - LE) \cdot t_{rem}\right),$$

where H is the cache hit rate, t_c is the cache hit time, t_{coh} is the average coherence lookup time, LE is the probability of finding a cache-missed block in the local memory, t_{int} is the mean integrity verification time, and t_{rem} is the mean time to fetch a block from another node.

The average read time for DIT is:

$$t_m = t_c + (1 - H) \cdot \left(t_{int} + t_{fetch} + (1 - LE) \cdot t_{rem}\right).$$

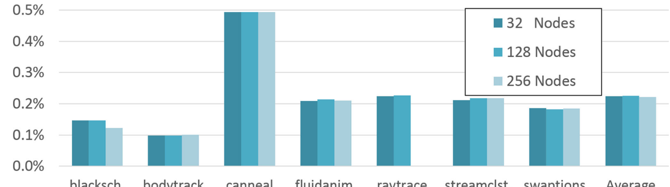

Fig. 3. DBMT performance overhead running PARSEC for various node count.

The difference between the two is:

$$t_m - t_b = (1 - H) \cdot (-t_{coh} + (1 - LE) \cdot t_{int}).$$

Performance differs only for cache misses, and the difference depends on t_{coh} versus $(1 - LE) \cdot t_{int}$.

We evaluated DBMT's overhead "synthetically" on top of a system with single-node BMTs and SDSM. We chose $t_c = 1$, $t_{fetch} = 100$ <clock cycles>. First, using $t_{coh} = 0$ (no overheads for the baseline's coherence check) and $t_{rem} = 5 \cdot t_{fetch}$, we evaluated the average memory access time (AMAT) overheads for the entire range of node miss rates (0-100 percent), where the high miss rates simulate intensively shared blocks; since [5] measured 1 percent cache miss for the BMT verification, we use 0-4 percent the for the DIT verification (Fig. 2a). We then repeated for $t_{rem} = 10 \cdot t_{fetch}$, $50 \cdot t_{fetch}$ (Figs. 2b, 2c). *Results:* not only is the AMAT overhead always less than 1 percent, it moreover drops to zero as the node miss rate rises. Next, we fixed the DIT cache miss to 2 percent (double than [5]'s, reflecting DIT's slightly larger tree) and $t_{rem} = 5 \cdot t_{fetch}$, and evaluated t_{coh} as a single memory access with miss rates of 0-4 percent (baseline's coherence check) and (0-100 percent) node miss rates (Fig 2d). We see that, relative to a baseline with non-zero overhead for coherence check ($t_{coh} > 0$), DIT may even shorten the AMAT.

To evaluate with real workloads, we used DBMT and ran the PARSEC benchmark suit [8] with 32, 128, and 256 compute nodes. Fig. 3 shows that the performance overhead is <0.5 percent relative to the baseline, with an average of ~ 0.2 percent. The node miss rates that were measured for PARSEC are up to 7 percent (*canneal*, *fluidanimate*), with memory/compute intensiveness of up to 45 percent.

Memory overhead. A per-block bit in the local integrity tree's leafs is required to mark NE, and in some implementations this can be obviated (e.g., a DBMT counter / DMEE version may be set to its max value ('0xffff..') for 'NR'). As mentioned before, DIT supports partial trees by using a high level NE to compress an entire unused subtree, similarly to the integrity tree of the baseline. DIT's write-permission bit is merely moved to an integrity protected structure, since any DSM system must maintain that anyhow.

6 Conclusion

We presented the integrity verified local coherence state (IVLCS) mechanism, and used it with a prior-art secure inter-node coherent data transfer layer and existing single-node integrity trees to construct the Distributed Integrity Tree scheme (DIT). This enabled us to extend various single-node integrity trees (e.g., Merkle Tree, Bonsai Merkle Tree, and Intel SGX's Memory Encryption Engine) into corresponding distributed versions (DMT, DBMT, DMEE), adding the capability to preserve memory integrity of a distributed shared memory system. While providing the extra security guarantees, DIT exhibits only a slight performance reduction across a wide range of runs, and only minor additional complexity. Furthermore, the local nature of IVLCS allows DIT-supporting systems to scale well. DIT is thus a major step towards high performance, scalable secure computing.

Acknowledgments

This work was supported in part by the Hasso Plattner Institute.

Authorized licensed use limited to: Technion Israel Institute of Technology. Downloaded on January 21,2021 at 12:33:54 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, 6, pp. 789–828, 1996.

[2] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: concepts and systems," *IEEE Parallel Distrib. Technol.*, vol. 4, no. 2, pp. 63–77, Apr.-Jun. 1996.

[3] O. Shwartz and Y. Birk, "SDSM: Fast and scalable security support for directory-based distributed shared memory," in *Proc. IEEE Int. Symp. Hardware Oriented Security Trust*, 2016, pp. 114–119.

[4] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2003, vol. 12, pp. 295–306.

[5] B. Rogers, S. Chhabra, Y. Solihin, and M. Prvulovic, "Using address independent seed encryption and bonsai merkle trees to make secure processors OS-and performance-friendly," *Proc. Ann. Int. Symp. Microarch.*, 2007, pp. 183–194.

[6] S. Gueron, "A memory encryption engine suitable for general purpose processors," in *Proc. IACR Cryptol. ePrint Arch.*, 2016, Art. no. 204.

[7] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. Hardware Architectural Support Secur. Privacy (HASP)*, ACM, 2013.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Tech.*, 2008, Art. no. 72.

[9] O. Shwartz and Y. Birk, "SeM: A CPU architecture extension for secure remote computing," in *Proc. Hardware Architectural Support Secur. Privacy (HASP)*, ACM, 2017.

[10] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2008, pp. 161–172.

[11] Advanced Micro Devices, "Secure encrypted virtualization API version 0.16," (2017). [Online]. Available: http://support.amd.com/TechDocs/55766_SEV-KM%20API_Specification.pdf

Authorized licensed use limited to: Technion Israel Institute of Technology. Downloaded on January 21,2021 at 12:33:54 UTC from IEEE Xplore. Restrictions apply.