CCIT Report #914 August 2017

# Distributed Bonsai Merkle Tree

Ofir Shwartz
Electrical Engineering Dept., Technion, Israel
ofirshw@tx.technion.ac.il

Yitzhak Birk
Electrical Engineering Dept., Technion, Israel
birk@ee.technion.ac.il

## Abstract

*Ensuring the correct execution of a program while running on untrusted computing services is challenging. Specifically, protecting the integrity of the memory content against replay attacks while running on a platform with an untrusted memory requires dedicated tracking structures and an in-chip state. For that, the Bonsai Merkle Tree (BMT)was suggested as an efficient integrity tree. We present the Distributed Bonsai Merkle Tree, a multi-node version of the BMT suitable for parallel and distributed environments.*

## 1. Introduction

Secure computing on untrusted environments is an emerging requirement. In these environments, it is commonly assumed that the CPU is the only trusted element, leaving everything else (including the board and the off-CPU memory) untrusted. The use of encryption enables protecting the confidentiality of the data while it resides in the untrusted memory, and the use of message authenticating code (MAC) enables protecting against forged or mis-located data; however, replay attacks, wherein old data is maliciously restored (e.g. by blocking the memory 'write' port) requires additional treatment.

The Bonsai Merkle Tree (BMT) [1] was suggested for protecting a program running on a single-node setting; however, most of the workloads that benefit from the use of untrusted environments (e.g., public clouds) are parallel and distributed in nature. Multi-node applications commonly use distributed shared memory (DSM) [2] or message passing interface (MPI) [3], where in the latter the memory space itself is not distributed, so the single-node integrity solutions are suitable. However, DSM is easier to program by simply spawning threads that access the shared address space, therefore an integrity pre-serving mechanism that supports DSM is required.

In this work we present the Distributed Bonsai Merkle Tree (DBMT). While extending the single node BMT's functionality, it does not require additional overhead on top of it.

## 2. Bonsai Merkle Tree

Bonsai Merkle Tree (BMT) [1] is an efficient integrity hash tree. BMT targets systems that protect their memory using counter mode encryption, wherein each memory block has its corresponding counter value. [1]'s observation is that instead of protecting the actual memory blocks using a secure hash tree, protecting the counters by a Merkle hash tree [4] (with an in-chip root hash) and using a per-block secure MAC will result in a smaller hash tree, the Bonsai Merkle Tree, which provides the same security guarantees as the original Merkle Tree. The smaller hash tree footprint results in better cache hit-rate, and therefore better performance than the Merkle Tree.

Each memory block has a small MAC alongside the data, so when it is fetched into the cache and decrypted by the counter mode technique (assuming a correct counter), forged data will result in a mismatch between the fetched MAC and the one computed over the fetched, decrypted block. Forging a counter (e.g. old counter with old data and MAC) will be detected on counter block fetch, since BMT directly protects the counter blocks (similarly to the way Merkle Tree protects the data blocks).

The BMT values are stored in the clear in the unprotected memory, and can be cached in the chip.

BMT cannot be simply used for protecting a distributed program, because its memory space spans multiple memories that are controlled by different CPUs.

## 3. Distributed Bonsai Merkle Tree

We assume the existence of a secure node-to-node data transfer method, such as SDSM [5].

In Distributed Bonsai Merkle Tree, we use private per-CPU encryption counters (not shared), so other CPUs cannot access them. We choose the counter corresponding to the data block to either contain the actual counter for an existing block, or NA for unmapped or a block currently resides in another CPU's local memory. Since counters are kept in blocks, counters of existing and non-existing blocks may reside in the same counter block. A per-CPU local-BMT is maintained normally, and each CPU maintains a local root hash of its local-BMT. See Fig. 1 for example.

When a memory block is needed, its counter is fetched first. If the counter value is NA, then it is considered not existing

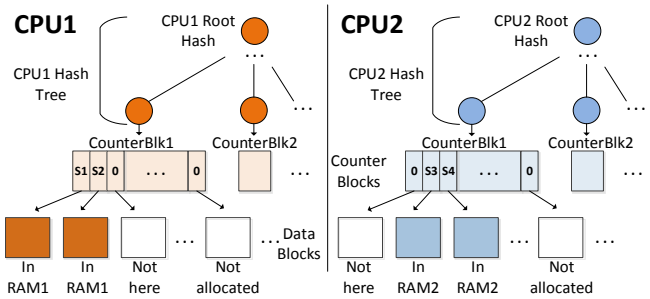


**Fig. 1. DBMT of blocks of multiple states: only in CPU1, shared between CPU1 and CPU2, and not allocated.**