Cache-Aware Dynamic Skewed Tree for Fast Memory Authentication

Saru Vig
Siew-Kei Lam
Nanyang Technological University, Singapore

Rohan Juneja
Qualcomm, India

ABSTRACT

Memory integrity trees are widely-used to protect external memories in embedded systems against bus attacks. However, existing methods often result in high performance overheads incurred during memory authentication. To reduce memory accesses during authentication, the tree nodes are cached on-chip. In this paper, we propose a cache-aware technique to dynamically skew the integrity tree based on the application workloads in order to reduce the performance overhead. The tree is initialized using Van-Emde Boas (vEB) organization to take advantage of locality of reference. At run time, the nodes of the integrity tree are dynamically positioned based on their memory access patterns. In particular, frequently accessed nodes are placed closer to the root to reduce the memory access overheads. The proposed technique is compared with existing methods on Multi2Sim using benchmarks from SPEC-CPU2006, SPLASH-2 and PARSEC to demonstrate its performance benefits.

CCS CONCEPTS

• **Security and privacy** → *Hardware-based security protocols.*

KEYWORDS

Memory security, Integrity trees, Cache aware

ACM Reference Format:
Saru Vig, Siew-Kei Lam, and Rohan Juneja . 2021. Cache-Aware Dynamic Skewed Tree for Fast Memory Authentication. In *ASP-DAC 2021: Asia and South Pacific Design Automation Conference, June 18-21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Main memories, being an integral part of all embedded systems, become an obvious target for attackers whose motives are to exploit leakage or modification of information. Integrity trees are commonly used to provide security against replay, splicing, and spoofing bus attacks in external memories. However, the existing schemes that use memory integrity trees suffer from high computational overheads [5] [7] as they do not address the impact of the system's performance for traversing large trees. This poses an enormous security challenge as embedded systems generally have tight computational constraints with real-time requirements.

In an integrity tree, the memory contents are stored in the leaf nodes after encryption/hashing. A hierarchical tree structure is built on top of the leaf nodes by recursively applying a primitive authentication technique (e.g. MAC, hash). Authentication of a leaf node would involve verification of each node against its parent node, from the leaf node up till the root of the tree. A copy of the root, stored on-chip, is assumed to be safe from attack. After the final step of successfully verifying the root node with its copy on-chip, data is considered safe and forwarded to the data cache. The whole process which entails verification at each level of the tree, increases the execution time substantially. One of the earlier versions of integrity tree is the Bonsai Merkle tree [8]. A number of variations have been proposed to the original Merkle tree in order to enhance the performance with size, cost, and complexity trade-offs [3]. One such approach is using a dedicated TreeCache for caching the integrity tree [2]. However, such approaches may lead to overall performance degradation due to cache contention. Using a large TreeCache to reduce the cache misses of the tree nodes becomes prohibitive for low-cost embedded systems.

In this paper, we propose a cache-aware memory integrity tree organization to fully take advantage of a dedicated TreeCache to reduce the memory bandwidth for verification. In order to further reduce the memory authentication overhead, we propose a method called Cache-aware Dynamic Skewed Tree (CADST) that leverages the proposed memory layout to move the frequently accessed nodes closer to the root node of the integrity tree at run time. This led to further reduction in the overall number of verification levels.

2 SECURITY MODEL

In Fig. 1, we present the security design model. The memory controller (MC) is responsible for the authentication process. It consists of of the following sub-parts:

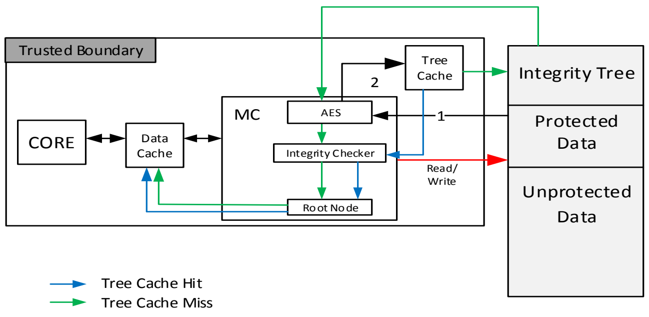


Figure 1: Security design model

(1) Encryption/Decryption Unit: We employ AES-128 as the encryption standard. This unit can be implemented as a hardware custom block to accelerate encryption/decryption.
(2) Integrity Checker (IC): Each tree node is matched to its parent node. In case of a mismatch, an alarm is raised.
(3) Root Node: A copy of the root node of the tree is stored on-chip and is assumed to be safe. The final requirement for data to be considered safe is that the root node of the tree must match with the root value stored on-chip.

The process of authentication begins with a read/write request sent to an address in the Protected Data region of the memory.

(1) The leaf node corresponding to that specific address is forwarded to the AES unit for decryption followed by verification.
(2) For verification, a request to access the parent node in the Integrity Tree region is initiated by the MC for the TreeCache.

The latter can lead to the following outcomes:

- Request sent to TreeCache is met with a cache hit: TreeCache stores the decrypted data. Thus the requested node value can be directly sent to the Integrity Checker for matching.
- Request sent to TreeCache is met with a cache miss: The node is then forwarded from the Integrity Tree region of the external memory to the AES unit for decryption. Once decrypted, it is then sent to the Integrity Checker to be matched with its child node.

The above mentioned steps are performed recursively till we reach the root node of the tree. The root node is compared against the root value stored on-chip and if they match, data is passed to the data cache for processing. In case of a write request, MC will then initiate a command to re-structure the tree.

3 PROPOSED CADST FRAMEWORK

3.1 Baseline

In this work, we have chosen Tamper Evident Counter (TEC) tree as our baseline. TEC tree provides security using Block AREA (Added Redundancy Explicit Authentication) technique. Tree nodes can be divided into two categories, Data chunk and Counter chunk, which store either data or count values respectively. A nonce, which is unique to each node, is added to each tree node before encryption. Nonce comprises of a count, and address of each node. Count value, representing the number of write requests performed on each node, is added to detect replay attacks. For an authentication to be successful, the count values of any given node must match with the value stored in its immediate parent node. An important advantage of the TEC tree is it relies on block encryption for authentication which provides for confidentiality at no additional cost. Encrypted data is resilient towards stolen memory attacks as well. Although we have chosen TEC tree as our baseline due to the above-mentioned advantages, the proposed technique can be adapted to different versions of integrity trees (e.g. Hash Trees, Merkle Tree), and is complementary to prior works based on caching tree-nodes and increasing arity. The original TEC tree does not cache tree nodes. We have implemented TEC tree with caching of tree nodes on a dedicated TreeCache.

To design integrity trees that can take advantage of a dedicated TreeCache, the most straightforward approach is to enforce the size of the tree nodes to be the same as a single cache line (CL). For example in SGX, the granularity of the Memory Encryption Engine (MEE) is fixed to 512 bits (same as CL size) and the MEE tree data structure is designed based on this constraint. In integrity trees employing AES block encryption, continuous memory addresses are segregated into blocks and a fixed number of blocks are combined to meet the CL size. These blocks are encrypted/decrypted together.

3.2 Proposed Tree Node Structure

For the proposed tree structure, the Protected Data are divided into equal sized blocks and each block is used to create a single Data Chunk (DC) as can be seen in Fig. 2. A DC comprises of data concatenated with a nonce value as shown in Fig. 2. Nonce is created using a count value, c_i. This value equals the number of write requests made to the DC. c_i concatenated with the node ID, n_i, making the nonce unique to each node. Other attributes of the DC are as shown in Fig. 2. p_i, s_i, LR_i denote the parent, sibling, side of n_i respectively. p_i stores the node ID of the parent and s_i stores the node ID of the sibling of n_i. LR_i denotes whether the node is a left child or a right child to its parent. These attributes help in skewing the tree as discussed later in Section 3.4. Each node has a separate counter. This makes overflow of counters a rare scenario. All the DC*s* are encrypted using block AES-128 encryption and are stored in the Protected Data region of memory.

The counter, c_i, are stored in the off-chip memory in a hierarchical tree structure. In the rest of the paper, we refer to Counter Chunk (CC) as a block where the counters are stored. Each CC also comprises of its own nonce and other attributes, similar to that of a DC. An entire CC is encrypted as a single block before being stored off-chip in the Integrity Tree region of external memory. This scheme is recursively applied to subsequent levels of the tree to form CC*s* till we obtain a single CC, called the root node of the tree. The count of the root node is stored securely on-chip. Thus, the tree structure helps to reduce the on-chip memory overhead to a single root node, while providing full memory integrity.

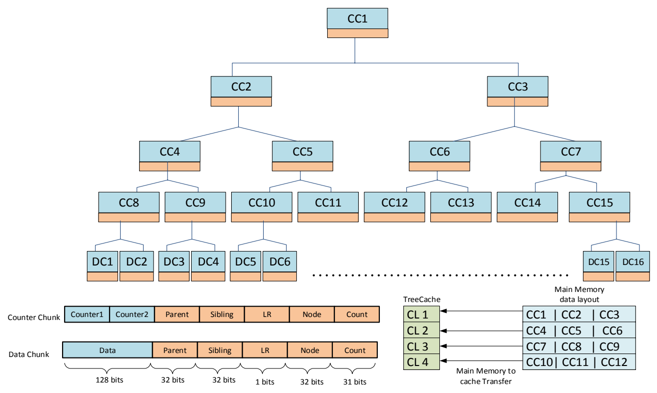


Figure 2: Standard memory integrity tree structure

3.3 Initializing Tree Structure with Cache aware Organization

We start with a binary tree of height h, which is equal to $\log N$, for a tree with N elements. Conceptually, we split the edges at the middle level of the tree, between nodes of height h/2 and h/2 + 1. This partitions the tree into a top recursive sub-tree A of height h/2, and several bottom recursive sub-trees $B_1, \ldots, B_k$ of height h/2. If all non-leaf nodes have the same number of children, then there will be $\sqrt{N}$ recursive sub-trees at the bottom, each with roughly $\sqrt{N}$ nodes. After splitting, we group all the upper sub-tree elements followed by lower sub-tree elements. The vEB layout is recursively applied to each sub-tree. At each step of recursion, size of the sub-tree being

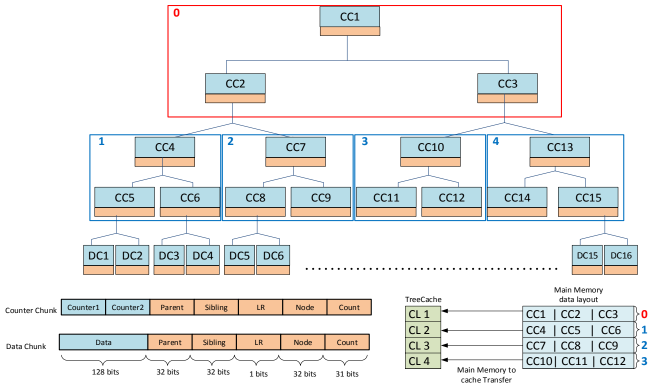


Figure 3: Proposed vEb memory integrity tree structure

grouped is the square-root of the size of the sub-trees in the previous step. Consequently, at some point we will be grouping sub-trees that can be retrieved in a single memory transfer, and this final partition creates a tree with sub-trees of size equivalent to a single CL.

The vEB version of the standard memory integrity tree in Fig 2 is shown in Fig. 3. The node ID of the CC*s* differ in both the trees. We use the vEB layout as the starting point of our tree. Based on this layout, the cache lines will be filled as shown in Fig. 3. It can be observed that for the proposed vEB memory integrity tree structure, both child and parent node will be cached in the same line. This reduces number of cache misses during a memory access because during verification, the parent of a child node is always accessed. Note that the proposed layout of the memory integrity tree is cache aware with the exception of the assumption that each sub-tree is aligned with the cache block boundaries.

3.4 Proposed Dynamic Skewing of Integrity Tree

Skewing operation is performed after write requests. We check if the c_i value of a given chunk is similar to its neighbourhood. If a chunk has higher c_i value than its immediate neighbours, then it should be shifted one level up, closer to the root node. In this manner, we are able to reduce the height difference between the node and the root. This reduces the number of verifications to be performed. The *ShiftUp* operation is performed before nodes are cached so that upon eviction, they are placed at their most recent position. In order to take advantage of the proposed memory layout, *ShiftUp* must take into consideration that the nodes forming a single sub-tree need to be kept intact in order for the cache misses to remain minimal. To avoid this, the re-structuring operation should be performed on sub-trees and not singular tree nodes. The re-balancing process and verification process happen concurrently. The decrypted counter chunks are checked for re-balancing during the verification process. All changes are made prior to re-encrypting the nodes at the end of the verification process.

ShiftUpSubTree: Let T be the sub-tree to be checked for shifting. And let Q be its parent sub-tree. We should note here that the count of a sub-tree is equal to the count of its root node. The process of shifting is as shown in Fig. 4.

(1) Check if, c_i, of T is greater than its corresponding sibling sub-tree count, c_j, by 1, and is also greater that its uncle sub-tree count, c_k.
(2) Exchange node T with its uncle sub-tree.
(3) Exchange the new children sub-trees of Q.
(4) Update the counter value of all the nodes of Q.
(5) Recursively perform steps 2-4 for all sub-trees on the path from T to the final sub-tree with the root node.

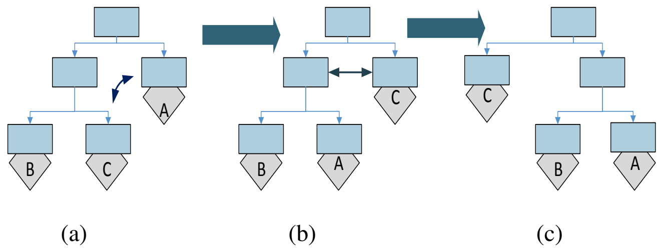


Figure 4: Shifting up sub-tree C: a) Exchanging C with uncle A, (b) rotation, (c) final state

3.5 Authentication

To avoid cache contention, we use a separate TreeCache on-chip to store the tree meta-data. Algorithm 1 describes the steps for verification on the integrity tree.

Algorithm 1: Dynamic Skewed Tree

```
begin
    Initialize tree with all data elements as data chunk on leaf
      nodes
    if (memory access request(addr)) then
        if read_request(addr) then
            ReadNCheck(addr)
        end
        if write_request(addr) then
            ReadNCheck(addr)
            WriteNUpdate(addr)
            Rebalance_flag ← rebalance_check(addr);
        end
        if Rebalance_flag then
            ShiftUpSubTree(addr);
        end
    end
end
```

ReadNCheck: This function is called when there is a read request sent to the Protected Data region that has not been cached and thus requires verification.

(1) Call the requested DC'*s* parent CC. If it is cached in the TreeCache, return the cached data for matching. The process completes, otherwise
(2) Fetch CC from external memory.
(3) Forward decrypted CC to IC for verification. If verified, place the contents of the child chunk into the TreeCache and proceed to verify the parent chunk (note that this step only applies when a CC is being verified).
(4) Repeat Step 1 until a cache hit is encountered or the root node is reached.
(5) If root node verifies correctly, return the requested data.

WriteNUpdate: This function is called when there is a write request sent to the Protected Data region that is not cached and thus requires verification.

(1) Call the requested DC'*s* parent CC. If it is cached, return the cached data for matching. Also, increment the c_i value. The process completes, otherwise
(2) Fetch CC from external memory, increment c_i.
(3) Check for *ShiftUpSubTree*.
(4) Place modified decrypted CC in TreeCache and forward to IC for verification.
(5) Repeat Step 1 until a cache hit is encountered or the root node is reached.
(6) If root node verifies correctly, return the requested data.

WriteBack: This function is performed upon eviction of a dirty CC from the TreeCache.

(1) Update the CC in the external memory.
(2) Recursively update the parent CC*s*, if not cached, up till the root with the new c_i values.

4 EXPERIMENTAL RESULTS

For our evaluations, we simulated applications from SPEC-CPU2006, SPLASH-2, and PARSEC benchmark suites on Multi2Sim [9]. We ran the benchmarks for four different tree architectures with a dedicated TreeCache: 1) Conventional Balanced TEC Tree (BTEC) [4], 2) Cache Aware Balanced TEC Tree (CABTEC) 3) Dynamic Skewed Tree (DST) [10, 11], 4) Proposed Cache Aware Dynamic Skewed Tree (CADST). BTEC [4] is a balanced integrity tree employing block-level AREA encryption as described in Section 3.1. CABTEC is BTEC tree implemented with the vEB layout in memory. DST [10] skews the tree nodes depending on frequency of access with the most frequently access closer to the root. The DST implementation does not employ the vEB memory layout. CADST is the proposed method where we initialize the DST with the vEB layout and use *ShiftUpSubTree* to skew the tree during run time. We have presented results for four different TreeCache configurations i.e. 16Kb and 8KB size with 128, 64 byte cache lines.

Authentication Time. Fig. 6 compares the normalized performance of all four models. Our performance metric is based on the run time required to perform the memory authentication. It can be observed that incorporating a cache-aware memory layout for the balanced integrity tree (CABTEC) can reduce the authentication time by approximately 14% compared to BTEC. By introducing the dynamic skewed tree mechanism which leverages on the cache-aware memory layout, the proposed CADST achieves an average speedup of approximately 25% and 17% over BTEC and CABTEC respectively. When compared to the DST method without cache-aware memory layout, the proposed CADST method achieves a performance overhead reduction of approximately 15%.

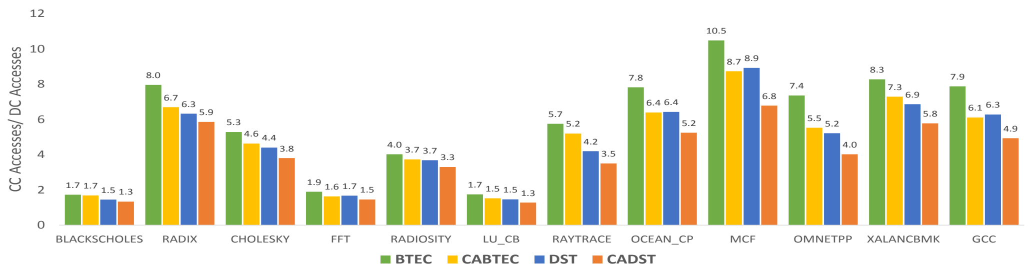
Table 1: Architectural parameters used in simulations

Architectural Parameters	Specifications
Clock Frequency	1 GHz
Data Cache	256 KB
TreeCache	16KB/8KB
Cache Latency	4
Replacement Policy	LRU
Write Policy	Write Back
External Memory Latency	100
AES Latency	40

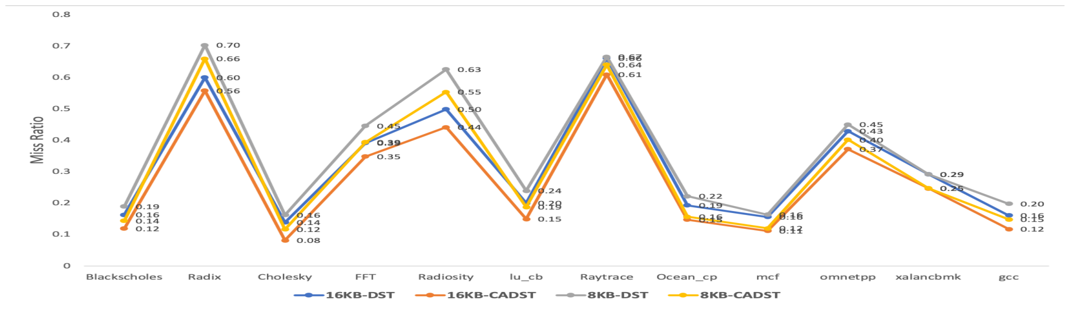
It is evident that the benefits of the proposed method depends on the application and its workloads. The height of the balanced integrity trees range from 11 to 26 levels. Large trees obviously results in a lot of storage and run time overhead, and thus such application always have a higher potential for improvement. Also, for memory intensive applications such as Lu_cb, Mcf, Omnetpp, Xalancbmk, and Gcc ($\geq$1 memory access per 1000 instructions), the benefit from skewing are more pronounced. Fig. 6 shows the effect of varying the TreeCache size and the block size on the performance. In general, a larger cache reduces the number of off-chip accesses and overall execution time of applications. Having a larger cache line also reduces the overhead of memory verification due to the reduction in the levels of the integrity tree. We should also consider the number of times the average number of CC*s* are accessed per DC. As such, we analyzed the memory traffic bottlenecks due the CC accesses, which is main cause of overhead. Fig. 5(a) shows the memory traffic bloat (counter accesses per data access) for all the models. CADST being more cache friendly, reduces the burden of the authentication and makes fewer access from the external memory for the CC*s* in all the applications. The improved performance due to the vEB layout is also highlighted in the difference between BTEC and CABTEC models.

TreeCache Miss Ratio. Fig. 5(b) compares the TreeCache miss ratio between the proposed model and DST with a 16KB size and 64B cache line size. We can observe that the 16KB cache performs better than an 8KB cache for both the models, as expected. But on average, the miss ratio improves further by 20% for the model with vEB structure which can directly be attributed to the customized memory organisation.

Custom Instructions. Custom instructions (CI) enable the acceleration of time critical software algorithms by introducing custom hardware blocks that are tightly-coupled to the Arithmetic Logic Unit (ALU). To reduce the latency of encryption/decryption during memory integrity verification, we evaluate the performance benefits of implementing the 128-bit mix-column AES algorithm as a custom instruction [6]. For our future work, we plan to develop a



(a)



(b)

Figure 5: a) CC*s* per DC accessed, b) TreeCache miss ratio

programmable memory controller that consists of processors with custom ISAs that can be adapted to the memory workloads in order to reduce energy consumption [1]. As such, we have evaluated the benefits of integrating a custom hardware AES module that is tightly coupled with the processor in the memory controller. The advantages of this method is the flexibility of implementing different encryption/decryption techniques without the need for re-designing a new memory controller. As can be observed in Fig. 6, the utilization of CI's has reduced the authentication time of all the models.

5 CONCLUSION

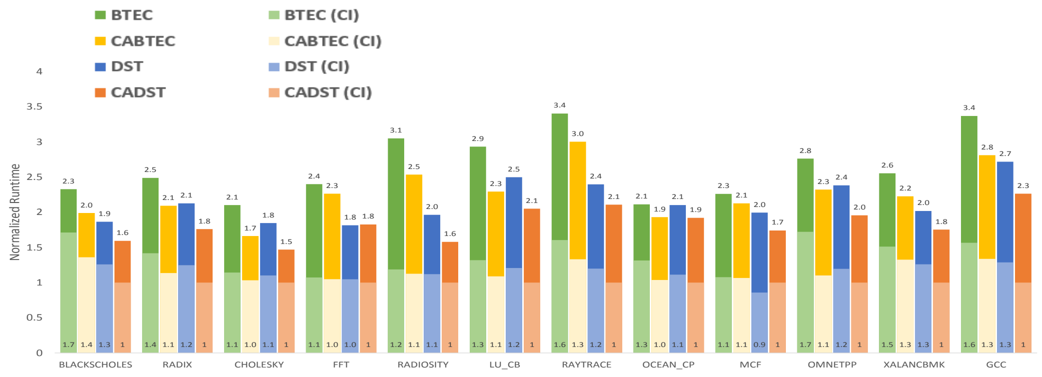
This paper proposes a dynamic skewed tree implemented with cache aware algorithms. The nodes in the tree are placed based on their frequency of access and are cached in a separate TreeCache on-chip. The tree layout is initialized with a vEB organization to reduce the memory-cache accesses and the skewing algorithm shifts memory blocks which form sub-trees so as to maintain minimal number of cache misses during tree traversal. Experimental results demonstrate that the proposed cache aware memory layout for a balanced integrity tree improved performance by approximately 14%. When a cache aware dynamic skewed tree model is utilized, a gain of about 25% can be achieved over the conventional balanced integrity tree model.

ACKNOWLEDGMENTS

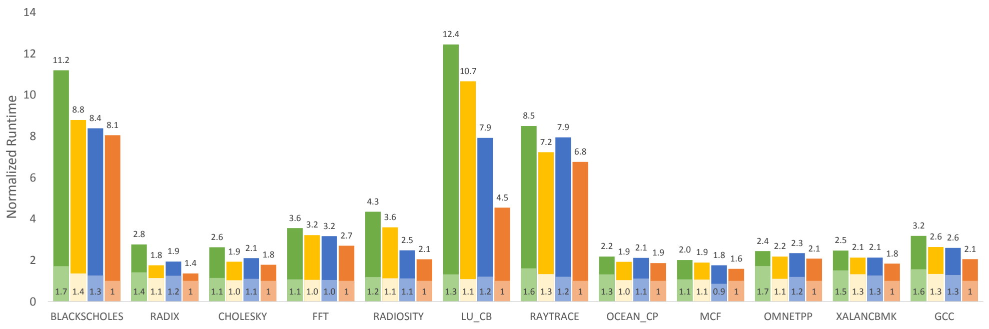
The research described in this paper has been supported by the National Research Foundation, Singapore, under grant number NRF2016NCR-NCR001-006.

REFERENCES

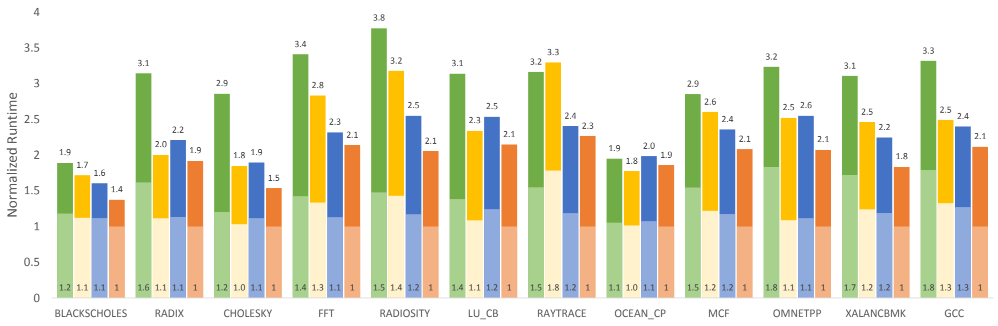
[1] Mahdi Nazm Bojnordi and Engin Ipek. 2012. PARDIS: A programmable memory controller for the DDRx interfacing standards. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 13–24.
[2] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.
[3] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B Lee, Nachiketh Potlapally, and Lionel Torres. 2009. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. In *Transactions on Computational Science IV*. Springer, 1–22.
[4] Reouven Elbaz, David Champagne, Ruby B Lee, Lionel Torres, Gilles Sassatelli, and Pierre Guillemin. 2007. Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 289–302.
[5] Shay Gueron. 2016. Memory Encryption for General-Purpose Processors. *IEEE Security & Privacy* 6 (2016), 54–62.
[6] Hua Li and Zachary Friggstad. 2005. An efficient architecture for the AES mix columns operation. In *2005 IEEE International Symposium on Circuits and Systems*. IEEE, 4637–4640.
[7] Joydeep Rakshit and Kartik Mohanram. 2017. ASSURE: Authentication Scheme for SecURE energy efficient non-volatile memories. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 1–6.
[8] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 183–196.
[9] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro Lopez. 2007. Multi2sim: A simulation framework to evaluate multicore-multithreaded processors. In *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*. IEEE, 62–68.
[10] Saru Vig, Guiyuan Jiang, and Siew-Kei Lam. 2018. Dynamic skewed tree for fast memory integrity verification. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 642–647.
[11] Saru Vig, Rohan Juneja, Guiyuan Jiang, Siew-Kei Lam, and Changhai Ou. 2019. Framework for Fast Memory Authentication Using Dynamically Skewed Integrity Tree. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 10 (2019), 2331–2343.



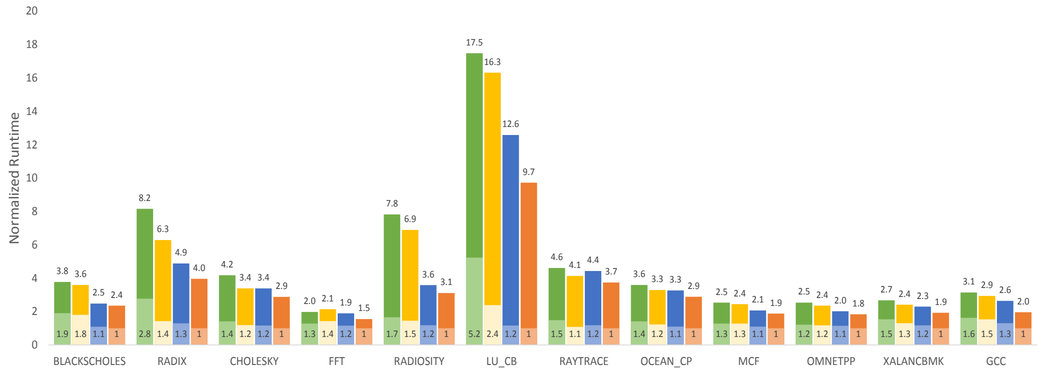
(a)



(b)



(c)



(d)

Figure 6: Run time for memory authentication with a) 16KB TreeCache, 64B cache line, b) 16KB TreeCache, 128B cache line c) 8KB TreeCache, 64B cache line d) 8KB TreeCache, 128B cache line