This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Framework for Fast Memory Authentication Using Dynamically Skewed Integrity Tree

Saru Vig, *Member, IEEE*, Rohan Juneja, *Student Member, IEEE*, Guiyuan Jiang, *Member, IEEE*, Siew-Kei Lam, *Member, IEEE*, and Changhai Ou, *Member, IEEE*

***Abstract*— Integrity trees are widely used in computer systems to prevent replay, splicing, and spoofing attacks on memories. Such mechanisms incur excessive performance and energy overhead. We propose a memory authentication framework that combines architecture-specific optimizations of the integrity tree with mechanisms that enable it to restructure at runtime based on memory access patterns. The integrity tree structure is customized based on the cache configuration in order to minimize the performance and energy overhead through speculative authentication. At runtime, the tree nodes that are accessed more frequently will be dynamically shifted closer to the root such that fewer levels of the tree are accessed during authentication. The framework is simulated with Multi2Sim and compared with other existing mechanisms [i.e., tamper-evident counter (TEC) tree and ASSURE] to demonstrate its performance and energy benefits. Experimental results using benchmarks from SPEC-CPU2006, SPLASH-2, and PARSEC show that the proposed dynamic integrity tree leads to an average reduction in instruction per cycle of 13% and 10% over TEC tree and ASSURE, respectively. The corresponding average reduction in authentication time is 30% and 20%, respectively. We show that the proposed framework facilitates the selection of a processor with a smaller cache size such that the energy consumption is reduced without sacrificing performance.**

***Index Terms*— Access patterns, cache, dynamic tree, memory integrity, Multi2Sim.**

I. Introduction

EMBEDDED systems have become a pervasive part of our lives and are the driving force behind technological advancements in many commercial sectors, such as health, automotive, and instrumental control. Due to our high dependability on such systems, it is crucial that they are secure and cannot be tampered with. This poses an enormous challenge as such systems are generally energy-constrained, and the existing security schemes contribute to excessive energy consumption. Thus, we need techniques that are capable of providing security without excessive energy and performance overhead.

Memory is an integral part of all embedded systems that become an obvious target for attackers whose motives are to exploit leakage or modification of information. In addition, securing sensitive data on computers is rapidly becoming a matter of high urgency. There are numerous schemes that have been reported for mitigating memory attacks [1], [2]. These methods inevitably make use of some form of encryption and a memory integrity tree. Memory contents are stored in the leaf nodes of an integrity tree and are authenticated whenever any access is made to them. The authentication process compares each node against its parent node (using a primitive authentication function) recursively up till the root. All such models are built with the assumption that the on-chip memory is safe and the root node is stored on-chip, where it cannot be tampered with. An authentication step for each level of the tree necessitates multiple memory accesses of nodes. This is repeated up till the root where the final authentication step is made with the root node that is stored on-chip. This results in a performance overhead for memory accesses that are directly proportional to the height of the tree.[1]

The main contribution of this paper is a framework that combines architecture-specific customization of the integrity tree with runtime mechanisms for reducing the performance and energy overhead of memory authentication. The former tailors the leaf node structure of the tree based on the cache configuration of the target processor in order to take advantage of speculative authentication and also to maintain cache line granularity of memory accesses. At runtime, the tree is dynamically skewed such that frequently accessed data are moved closer to the root. This minimizes the authentication steps of the frequently accessed data, resulting in an overall reduction in instruction per cycle (IPC) and energy overhead compared with the existing methods. We performed the experiments on benchmark applications with varying degree of workloads and memory access patterns to demonstrate the performance/energy gains of our proposed method.

Given an application that needs to be protected against memory attacks, the proposed framework that combines cache-aware integrity tree structure design and a mechanism to dynamically skew the tree is able to select a suitable cache configuration to minimize energy consumption. We show that this can lead to over 40% energy savings without sacrificing

Manuscript received December 19, 2018; revised April 8, 2019 and June 1, 2019; accepted June 8, 2019. This work was supported in part by the National Research Foundation Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme with the Technical University of Munich at TUMCREATE. *(Corresponding author: Saru Vig.)*

S. Vig, G. Jiang, S.-K. Lam, and C. Ou are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798 (e-mail: saru001@e.ntu.edu.sg; gyjiang@ntu.edu.sg; assklam@ntu.edu.sg; chou@ntu.edu.sg).

R. Juneja is with Qualcomm, Bangalore 560048, India (e-mail: rohan14156@iiitd.ac.in).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2019.2923004

[1]The maximum height of a tree is $\log_k n$, where n is the number of leaf nodes to be protected and k is the arity or number of children for a node.

1063-8210 © 2019 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.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

performance. In some cases, energy savings and performance gain can be concomitantly achieved. We perform the experiments with a dedicated TreeCache on-chip to store the contents of the tree. In such scenarios, verification is terminated whenever a node is found in the cache, as cache contents are considered safe. The proposed method still manages to achieve timing gains compared to a balanced tree.

This paper is organized as follows. Section II discusses the related work. Section III describes the threat model, and Section IV introduces the proposed memory integrity tree structure and framework. Section V introduces the tree restructuring algorithm, and Section VI discusses the security aspects. The experimental results and discussions are presented in Section VII. We conclude this paper in Section VIII.

II. Related Work

Securing memories often makes use of an integrity tree. One of the earlier versions of the integrity tree is the Bonsai Merkle tree [3]. Merkle trees are one-way trees based on a cryptographic message authentication code (MAC) function. Each data node is stored along with its MAC value and is verified whenever it is read from the memory. The Bonsai Merkle tree uses address independent seed encryption and reduces the memory storage requirements by creating the tree over counters rather than MAC's data. A number of variations have been proposed to the original Merkle tree in order to enhance the performance with size, cost, and complexity [1] tradeoffs. Intel′s SGX incorporates a memory encryption engine (MEE) [4] that uses a hash tree with a tweaked version of the Advanced Encryption Standard (AES)-128 for authentication. The design has been incorporated in the sixth-generation Intel Core processor. The memory engine uses a balanced Merkle Tree for data integrity alongside proven cryptographic bounds for the confidentiality of data. For performance enhancement, they have a dedicated cache for storing tree data. The tamper-evident counter (TEC) tree proposed in [5] employs the block-level added redundancy explicit authentication (AREA) as their protection mechanism. The block-level AREA performs block encryption on the tree nodes, both data and counters. Hence, data confidentiality is provided at no extra cost. In a TEC tree, the nodes are composed of D||N (||-concatenation operation), where D is the data/counter of k-bits and N is an n-bit nonce (number only used once). A nonce consists of the address and a counter, which depicts the number of times a write operation has been performed on that address. After encrypting this node, it is near impossible to distinguish the D and N in the ciphertext. If the content of a tree node is flipped by one bit, there is a very high probability that the last n-bits of the nonce are different from the original nonce. This probability depends on the nonce size, n.

Although the above-mentioned methods provide protection to the memory, they incur high-performance overhead as traversing the integrity trees during authentication requires numerous memory access. These methods do not address the impact of the system's performance for traversing large trees. The works in [6] and [7] suggest the means to reduce the depth of the integrity tree by using a different counter configuration and increasing arity of the tree. Saileshwar *et al.* [6] designed a tree with 128-arity, and hence, they are able to considerably reduce the depth of integrity trees for securing large memories. For protecting a 16-GB memory, they were able to reduce the tree size to four levels. However, the use of larger arity in order to obtain shallow integrity trees introduces additional problems, such as frequent counter overflowing, which increases the overhead of the scheme. In addition, the success of methods that rely on increasing the tree arity is dependent on the counter nodes being cached efficiently, as this contributes to reducing the number of counter writes and overflows. However, caching counter nodes is not always possible, especially for system with tight constraints.

In addition, approaches that rely on caching the tree nodes have reported the overall performance degradation due to cache contention [8], as storing the metadata of the tree nodes in caches for large trees consumes substantial memory space. In particular, [8] reported that unless the cache is reasonably big, the tree nodes will occupy 25%–50% of the cache, leading to a large number of cache misses. Lee *et al.* [9] suggested using a type-aware dynamic cache insertion technique, i.e., Dynamic Insertion Policy [10] and Bimodal Insertion Policy [11], to improve the caching efficiency. However, it is difficult to find the best technique for any given application without a considerable static analysis. Moreover, a caching strategy in itself does not tackle the problem of tree size. These methods should be used orthogonally, in addition to other techniques to avail maximum benefits. None of the methods discussed earlier use memory access patterns to improve the performance.

ASSURE [12] employs smart MACs and dual root trees to provide for memory security. The tree reduces the authentication steps by creating a smaller subtree within a standard binary tree. The smaller subtree has its roots closer to the leaf node. Thus, authenticating the leaf nodes of the smaller subtree requires fewer verification steps. Conceptually, the memory is divided into groups called memory block groups (MBGs) of continuous memory locations. One of the MBGs is denoted as HOT and the remaining MBGs are denoted as COLD. The HOT MBG consists of memory locations that are frequently accessed. A subtree is built for the HOT MBG with its root stored on-chip. The remaining memory locations form the COLD MBGs with their root being the main tree root. Thus, at any given time, two roots are stored securely on-chip. The HOT MBG is dynamically adjusted based on the memory access patterns. The number of MBGs for a given memory, n, is pre-determined. On the one hand, a large value of n will limit the size of each MBG, which may result in frequent accesses outside of the HOT MBG. On the other hand, smaller values of n result in larger subtrees, which increases the authentication steps of the HOT MBGs. In ASSURE, a mechanism is used to predict the next HOT MBG after a fixed number of accesses based on the *counter* value that is associated with each MBG. Another factor that is pre-determined is how frequently a new HOT MBG is predicted. Predicting the HOT MBG over large intervals may not guarantee the most frequently accessed locations in HOT MBG at all times, while predicting too frequently will only marginally affect the performance, especially for workloads with poor

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

spatial locality. Although Rakshit and Mohanram [12] have provided the recommended values for n and the prediction rate, these cannot be generalized across different applications and workloads. The predetermined value of n and the prediction rate will also not be applicable to workloads with highly random-access patterns.

In [13], we introduced a static-skewed integrity tree where the tree is constructed offline based on the frequency of memory accesses derived from static application profiling. Tree nodes that are accessed frequently are placed closer to the root based on the static application profiles. However, the tree structure remains unchanged at runtime. Although this paper demonstrated the performance advantages of skewing a tree based on the application workload (about 18% lower performance overhead over the TEC tree), the method is only applicable to scenarios where the memory access patterns of the applications are known beforehand, which is highly unlikely in practical applications.

In [14], we presented our initial approach for dynamically skewing the integrity tree at runtime based on the memory access patterns. This is achieved by placing data elements with the same memory access frequency together in a single set, with each set being treated as a single unit for encryption. Each set is associated with a frequency, and data elements migrate among the sets as their access frequency changes. Although this method achieved higher performance gains compared to [13], it suffered from several drawbacks, which restricts the method from practical realization. First, the number of data elements that can be grouped in a set is limited by the size of the encryption blocks. This could lead to scenarios where multiple sets are associated with the same frequency and the tree nodes cannot be optimally placed in the tree structure to reduce the authentication steps. This leads to additional complexity during tree traversal and results in fewer performance gains. Second, the method in [14] requires an additional on-chip memory overhead in the form of a lookup table (LUT) to retrieve the tree locations in memory. This storage overhead is considerable. If the number of nodes of the tree is N, the memory overhead of the LUT is $(3 * \log_2 N)$ bits. Finally, the designs of our tree structures in [13] and [14] are agnostic to the cache, and the experiments were undertaken with the assumption that the target system does not have a cache. Moreover, only the runtime gains were measured and the power and energy were not evaluated.

The following highlights the novelties of this paper compared to our prior works in [13] and [14].

1) *Memory Authentication Framework:* Our proposed framework in this paper overcomes the limitations of our prior work with an entirely different tree structure design for dynamically skewing the tree at runtime compared to [14]. We introduce a new node structure for the memory integrity tree that contains all the necessary information to perform dynamic restructuring of the tree based on the memory access patterns. Unlike [14], the new node structure and the grouping of data elements in continuous memory address into a single leave node, remove the need for a large LUT, and enable the integrity tree to be restructured dynamically with low-performance overhead. Unlike [13], we do not require any offline static analysis to be performed on the applications. In addition, unlike the work in [12], our integrity tree does not conform to any rigid parameters and, hence, it is able to adapt more effectively to the dynamic memory access patterns. This is achieved by maintaining a separate counter for each leaf node that is incremented on each write request.
2) *Cache-Aware Tree Structure:* The designs of our tree structures in [13] and [14] are agnostic to the cache, and the experiments were undertaken with the assumption that the target system does not have a cache. The proposed framework in this paper customizes the integrity tree node structure based on the cache configuration of the target processor to take advantage of speculative authentication. In addition, the number of data elements in a leaf node is determined based on the cache line size. This scheme exploits the data locality of reference, where during a cache line replacement, only a single authentication step is made for multiple data elements that are likely to be accessed in the near future. An example is shown in Fig. 2, where data from four successive memory locations have been combined to form one leaf node. Such a design is suitable for a cache configuration with a 512-bit cache line. In Section VII-B, we show that the proposed framework provides the advantage for choosing the systems with smaller caches to minimize energy consumption without compromising performance.
3) *Experiments*: Finally, in Section VII, we perform the extensive simulations of our framework on systems with varying cache configurations using applications from the SPEC-CPU2006, SPLASH-2, and PARSEC benchmarks. Our prior works in [13] and [14] were not evaluated with such intensive workloads. Moreover, only the runtime gains were reported and the IPC and power/energy were not evaluated.

III. Security Model and Objectives

This paper is concerned with bus attacks that aim to perform unauthorized reading or tampering of data stored in an external (untrusted) memory. As such, countermeasure must be in place to preserve data confidentiality by encrypting the data with a secret key and storing the key in a secure location, e.g., on-chip memory. In addition, data authentication methods are necessary to detect the following widely known data tampering attacks.

1) *Spoofing:* Attacker exchanges a memory block with a tampered one.
2) *Splicing:* Attacker replaces the memory block at address A with a memory block at address B, where $A \neq B$.
3) *Replay Attacks:* Attacker records data at an address and inserts it at the same address at a later point in time. Thus, the present value of the data is replaced by an older value.

While side-channel attacks are beyond the scope of this paper, we wish to briefly discuss its security impact on our approach. In side-channel attacks, the attacker measures

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

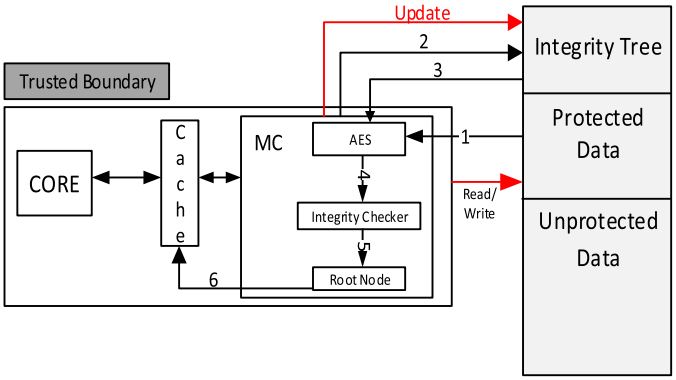


Fig. 1. Security design model.

side-channel information leakage, such as power or timing to decipher encryption keys being used for authentication. Unterluggauer *et al.* [15] demonstrated the feasibility of cache side-channel attacks on systems employing a standard memory integrity tree. A cache-based side-channel analysis falls into one of two main categories: trace-driven attacks [16] where individual cache hits and misses yield information, and time driven attacks [17] where the aggregate number of misses and, hence, the execution time of the algorithm give similar information. We wish to highlight that the security of cache attacks on systems employing standard memory integrity tree (see [15]) and the proposed dynamic-skewed integrity tree in this paper is the same due to the following reasons. First, the cache characteristics (hits and misses) will not differ in both methods as long as the grouping of data elements in leaf nodes is the same. This is because only the data elements are cached and the contents of the integrity tree are not. The proposed skewed integrity tree only reduces the time taken to traverse the integrity tree for authentication. It is still stored in the external main memory. Second, any side-channel analysis of the tree cannot give information about the keys as the keys are not stored in the tree. Only by studying the cryptographic algorithm used, attackers can deduce such information. The success of a side-channel attack is independent of the structural format of the tree. We have used the standard cryptographic algorithm of AES, and its execution and the processing times do not change due to the proposed changes in the organization of the tree. Thus, there is no additional leakage of information to aid an attacker performing side-channel analysis. Unterluggauer *et al.* [15] also proposed the prevention methods against side-channel attacks on systems with memory integrity tree. This entails some masking procedure for the keys or employing more rigorous mechanisms, such as updating keys in short intervals. Such mechanisms can also be employed in our proposed dynamic-skewed integrity tree structure as countermeasures against side-channel attacks.

A. Overview of Proposed Security Model

In Fig. 1, we show our security design model. The model consists of a memory controller (MC) that is responsible for data authentication. The entire memory can be split into three distinct regions: a section consisting of sensitive data that need protection (i.e., Protected DATA), a section of unprotected data, and a region for storing the integrity tree. The protected data and the integrity tree region will be encrypted with AES.

An integrity tree consists of equal-sized blocks of data known as nodes that are organized in a tree structure. The root of the tree (which holds the root value) is stored in the first level and also on-chip (i.e., root node in MC). The nodes on the last level are known as leaf nodes that are stored in the protected data region. The remaining nodes comprising of the counter nodes are stored in the integrity tree region. The tree operations after a read/write request is sent to the protected data region can be described as follows with the help of Fig. 1.

1) The leaf node corresponding to the data address requested is sent to AES for decryption
2) MC sends a request to the integrity tree to obtain the parent of the node that was just decrypted.
3) The parent node is sent to AES for decryption.
4) Once both the child and parent nodes are decrypted, they are verified by the integrity checker. Steps 2–4 are repeated until the root of the tree is reached. If verification fails in any one of these steps, an alarm is triggered.
5) The root of the tree is matched with the root node stored on-chip. In case of mismatch, an alarm is raised.
6) If verification is successful, the data requested root in Step 1 is sent to the cache. Any further accesses to this address will be made from the cache itself and thus will not need to be verified again.
7) *Update*: If the request in step 1 is a write request, an update command will be sent from the MC to the Integrity Tree region to restructure the tree. The details of this procedure will be discussed in Section V.

The detailed description of the tree construction and verification will be discussed in Section IV-A.

IV. Proposed Dynamic Integrity Tree Structure

The proposed dynamic integrity tree adopts the principles of the TEC tree [5], which provides both data confidentiality and integrity. In this section, we describe the modifications made to the balanced TEC tree (BTEC) [5] to enable dynamic skewing. As discussed earlier, the TEC tree employs the AREA technique at the block level, where redundant data (a nonce) are added to each tree node before encryption and checked after decryption. Since TEC tree relies on block encryption to construct the tree, it provides data confidentiality without any additional latency by using the hardware already in place for integrity verification. This advantage makes it possible to employ the TEC tree even for non-volatile memories (NVMs) that face stolen memory attacks due to their data remanence properties. It is worth mentioning that the proposed method for dynamically skewing the integrity tree can also be applied to other existing forms of memory integrity trees. We will introduce the new tree node specifications and its design in Section IV-A and Fig. 2. This improved framework will have explained further with the help of an example in Section IV-C.

A. Integrity Tree Model

In the rest of this paper, we use the term data chunk (DC) to refer an atomic block for authentication. In order to reduce

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

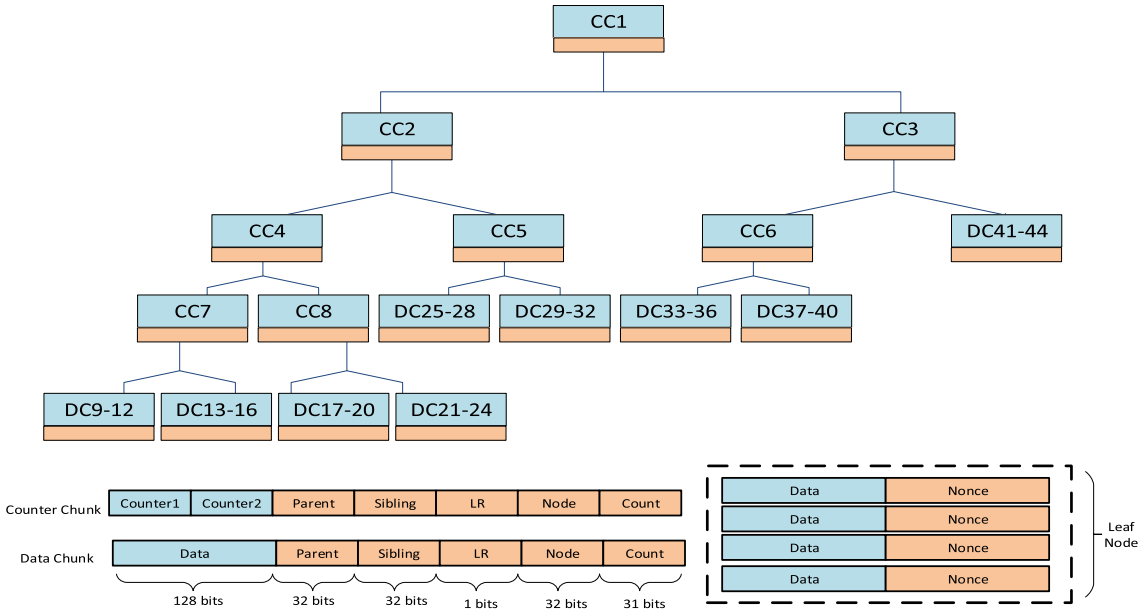


Fig. 2. Dynamic integrity tree structure.

the overhead of data verification, the processor should be able to easily obtain a parent's address from the node's address. Thus, the DCs have been modified to enable quick access to this information. The DC can be broadly split into two parts: 1) plaintext or data and 2) nonce. The attributes of a DC are shown in Fig. 2. Let n_i denote the node ID of the ith node on the tree. p_i, s_i, LR_i, and c_i denote the parent, sibling, side, and count of write access 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. c_i denotes the frequency of write access made to the data stored in n_i. c_i is a local counter dedicated to a specific chunk rather than a global counter. This is to minimize counter overflows. A counter overflow will require all the encryption keys to be re-generated, and hence, utilizing local counters can help extend the lifetime of the encryption key.

Multiple encrypted DCs are combined to form a leaf node for the tree. The number of DCs that are grouped in a leaf node depends on the cache line size. Also, the data elements of the same leaf node are from consecutive memory addresses. During authentication, data verification is first performed on a leaf node. When there is a cache miss, the full leaf node that has a size equivalent to the cache line is transferred to the cache after authentication. This implies that all the DCs in the leaf node have to be verified even though an access is made to only one of them. Each DC in the leaf node is decrypted sequentially followed by authentication with the parent node. Note that each DC in a leaf node will have the same parent lineage since they are associated with consecutive memory addresses. As most applications exhibit the locality of reference, the other DCs in the same leaf node that has been authenticated and their decrypted data elements are in the cache, which will likely be accessed in the near future. As such, speculative verification of the data is performed to reduce the authentication overhead.

The verification starts by decrypting the DC to obtain the counter part of the nonce. The counter values are also stored in the off-chip memory in a tree structure. In the rest of this paper, we refer to counter chunk (CC) as a block where the counter part of the nonces is stored with its own nonce. The entire CC is encrypted as a single block before being stored off-chip. The layout of the CC and DC is shown in Fig 2. A block-level AREA scheme is recursively applied to the subsequent levels of the tree to create CCs with redundant data being added to them. Finally, we obtain a single CC called the root node of the tree. The root value is also stored securely on-chip.

The main objective of the recursive procedure described earlier, which creates the tree structure, is to reduce the on-chip memory overhead to a single root node while providing full memory integrity.

B. Authentication

1) Block Encryption: We implement the block-level AREA with AES encryption as our authentication primitive on b-bit size DC and CC, comprising l bits of data concatenated with n bit nonce, with an encryption key K. The probability of detecting a flipped bit in the DC depends on the size of the nonce. The probability that the decrypted chunk remains the same after tampering is $1/2^n$. Thus, having a reasonable size nonce is crucial to maintain the security of the system.

2) Reading and Writing to DCs: Whenever the data are read from or written to memory, the tree performs the functions of ReadNCheck and WriteNUpdate, as shown in Algorithm 1.

a) ReadNCheck: In case a read request to an address in the protected area is sent, the leaf node comprising the

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

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
            Shift_up(addr);
        end
    end
end
```

requested DC is decrypted along with its parent node. Once decrypted, a nonce is extracted from all the DCs on that leaf node. For authentication, p_i of the DCs and n_i of their parent CC along with their c_i values are matched. If the comparisons are successful, the process is recursively repeated for all the parent nodes until the root. Finally, c_i of the root node is verified against the value stored on-chip. If this last verification is successful, the data are considered to be safe and are forwarded to the cache.

b) WriteNUpdate: When a write request is sent to an address in the protected area, the leaf node comprising of that DC is loaded and decrypted. This is followed by a verification of the leaf node with a ReadNCheck function. Once verified, the data slot of the specific requested DC is updated and the c_i values of all the DCs are incremented by 1. The parent CCs also needs to be updated with the new counter values. This process of first verifying the nodes before updating is carried out for the entire branch from the leaf node to the root node. Verification is performed to prevent adversaries from inserting fake data onto a node before the write, with the motive of corrupting parts of the node which are not affected by the write. Once updating is complete, a check is performed to determine whether rebalancing is required based on the c_i values of nodes. The procedure for rebalancing is discussed in Section V.

C. Example

As an example, we consider the tree shown in Fig. 2. We implement AES-256 encryption. The DC and the CC are 256-bit blocks. In the case of DC, 128-bit data are concatenated with a 128-bit nonce. The CC has two 64-bit counters concatenated with a 128-bit nonce. We consider a cache line size of 512 bits. Thus, as shown in Fig. 2, we combine four DCs to form one leaf node (128 data bits $\times$ $4 = 512$) so that 512 bits of verified data can be passed after one complete round of leaf to root authentication. In the event of a cache miss, four DCs are verified and the data extracted from these nodes is transferred to the cache. As these four DCs are always transferred and evicted together to/from the cache, they share the same count and have a common parent storing

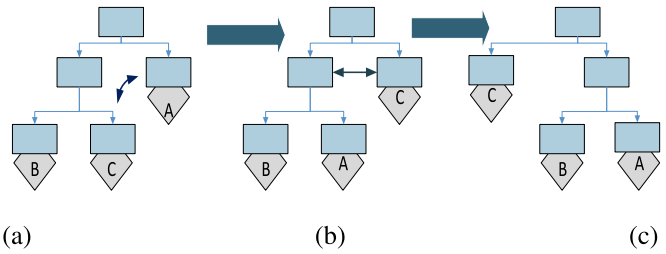


Fig. 3. Shifting up tree node C. (a) Exchanging C with uncle A. (b) Rotation. (c) Final state.

their c_i *values*. This format has been adopted to keep the address calculations simple, keeping in mind the constraints due to the required nonce size, AES block size, and cache line size.

V. Dynamic Tree Restructuring

Initially, when the protected data have not been accessed, the proposed dynamic tree structure is similar to the standard two-ary binary tree since all the leaf nodes have the same probability of occurrence. At runtime, the tree is progressively restructured to place leaf nodes that are more frequently accessed closer to the root to reduce the overall authentication time.

Algorithm 1 describes the steps for verification on the integrity tree. During verification, when memory requests for the protected region arrive at the MC, the integrity checker performs verification/updating depending on whether it is a read or write request. The verification is performed by matching the counter values of nodes against their parent values recursively until the root node. The change in the parent nodes will also not affect the data stored off-chip. This is due to the fact that the protected data and the integrity tree data are stored in different memory regions, as shown in Fig. 1.

On a write request, after increasing the count value c_i, it is checked whether rebalancing should be performed or not. Two important factors to be taken into consideration are when and how should the tree be re-balanced. We adopt the following criterion used in [18]:

$$\textbf{if } (c_i > (c_{s_i} + 1) \wedge (\ c_i > c_{s_{p_i}})\ ) \textbf{ then } \text{rebalance.}$$

The above-mentioned criterion states that if the count of a node is greater than the weight of its sibling node by at least 2 and is greater than the weight of its uncle node, then it should be relocated. The algorithm used to perform rebalancing has been described in Algorithm 2. The rebalancing process and the verification process happen concurrently. When the CCs are decrypted during the verification process, they are checked for rebalancing. The changes made are on the counter nodes before they are reencrypted at the end of the verification process. The idea is based on the principle that leaf nodes with a higher probability of occurrence (based on the c_i values) are shifted up closer to the root node. This is achieved in two steps, as shown in Fig. 3: 1) exchanging subtree with its uncle and 2) rotation.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 2 ShiftUp {*T* : *Pointer to Tree node*}

```
begin
    while T is not the root do
        T's weight = T's right child weight +T's left child weight
        if (T's weight > T's sibling weight +1) ∧ (T's weight >
          T's uncle weight) then
            Q ← parent of parent of T
            exchange T with T's uncle
            exchange Q's right and left children
            update T's ancient parent's weight
        end
        T ← T's parent
    end
end
```

A. Implementation Example

We consider an example shown in Fig. 4 to provide some insight into the internal working of a dynamic tree. We begin with the tree structure shown as TREE 1 with its leaf nodes placed in a skewed manner. The table below each figure describes the elements stored in that particular tree.

1) TREE 1: Tree nodes 1–3 and 6 are CCs and the remaining (4, 5, and 7–9) are leaf nodes comprising of multiple DCs. Nodes 4, 5, and 7 having a higher count value than nodes 8 and 9 are placed one level higher on the tree. We consider the following order: 9 → 5 → 5 for the subsequent write requests.

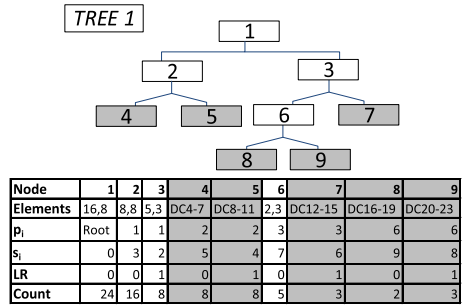
2) TREE 2: Based on the access sequence, the next node to be accessed is 9. Thus, the count value of node 9 along with its parent is incremented by 1. This makes it eligible to be considered for rebalancing. As the count of node 9 is greater than its sibling's count (node 8) by 2 and it is also greater than its uncle's count (node 7), it successfully meets the rebalancing criterion. After performing the rebalancing operation, the tree takes up the form of TREE 2 in Fig. 4. We can observe that node 9 has been shifted up by one level and node 7 has been pushed down by one level. The rebalancing criterion is checked for nodes 3 and 6 as well but is not met.

3) TREE 3: Next node to be accessed is node 5. This increases the count of nodes 5–9. Again, the rebalancing criterion is checked. As it is not yet eligible (its count is greater than its sibling by only 1), the rebalancing operation is not performed. Parent of node 2 is also checked against the criterion, but it is not eligible.

4) TREE 4: node 5 is then accessed again, increasing its count to 10. With this, the rebalancing criterion for node 5 is met. Thus, after the rebalancing operation, the tree takes up the form of TREE 4 in Fig. 4. Node 5 has been pushed one level up to level 2, and the remaining nodes are pushed one level below. Node 4 is still on level 3. The remaining nodes, i.e., 7–9, have been pushed one level down.

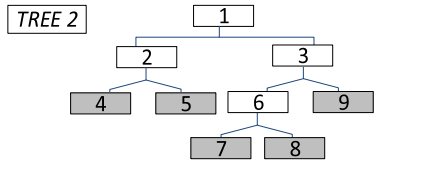
B. Analysis of Overhead

In this section, we analyze the storage and performance overhead of the proposed method in comparison to the TEC-tree implementation.



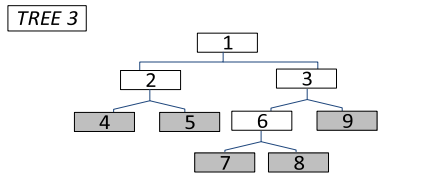
Node	1	2	3	4	5	6	7	8	9
Elements	16,8	8,8	5,3	DC4-7	DC8-11	2,3	DC12-15	DC16-19	DC20-23
p_i	Root	1	1	2	2	3	3	6	6
s_i	0	3	2	5	4	7	6	9	8
LR	0	0	1	0	1	0	1	0	1
Count	24	16	8	8	8	5	3	2	3

(a)



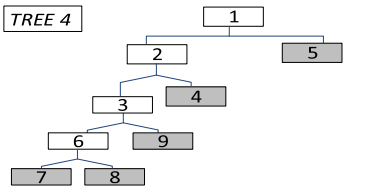
Node	1	2	3	4	5	6	7	8	9
Elements	16,9	8,8	5,4	DC4-7	DC8-11	3,2	DC12-15	DC16-19	DC20-23
p_i	Root	1	1	2	2	3	6	6	3
s_i	0	3	2	5	4	7	8	7	6
LR	0	0	1	0	1	0	0	1	1
Count	25	16	9	8	8	5	3	2	4

(b)



Node	1	2	3	4	5	6	7	8	9
Elements	17,9	8,9	5,4	DC4-7	DC8-11	3,2	DC12-15	DC16-19	DC20-23
p_i	Root	1	1	2	2	3	6	6	3
s_i	0	3	2	5	4	7	8	7	6
LR	0	0	1	0	1	0	0	1	1
Count	26	17	9	8	9	5	3	2	4

(c)



Node	1	2	3	4	5	6	7	8	9
Elements	17,10	9,8	5,4	DC4-7	DC8-11	3,2	DC12-15	DC16-19	DC20-23
p_i	Root	1	2	2	1	3	6	6	3
s_i	0	5	4	3	2	9	8	7	6
LR	0	0	0	1	1	0	0	1	1
Count	27	17	9	8	10	5	3	2	4

(d)

Fig. 4. Implementation example.

1) *Memory Overhead:* The memory overhead for the baseline TEC tree and the proposed method is as follows:

$$O_{\text{BTEC}} = \frac{l_p + n_{\text{BTEC}}}{l_p(A-1)} \tag{1}$$

$$O_{\text{Prop}} = \frac{l_p + n_{\text{Prop}}}{l_p(A-1)} \tag{2}$$

where l_p is the data bits in the DC, n are the nonce bits, and A is arity. Due to the additional attributes of

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

parent and sibling required for the proposed method as discussed in Section IV-A, the nonce size differs for both methods by 2*$\log_2$ N bits as follows:

$$n_{\text{BTEC}} = \text{count} + \log_2 N \quad (3)$$

$$n_{\text{Prop}} = \text{count} + 3 * (\log_2 N) \quad (4)$$

where N is the maximum number of tree nodes and count is the counter value size.

All integrity tree designs will typically incur additional memory overhead (about 1–2 times of the actual data protected). Although the proposed method incurs a slightly higher memory overhead than the TEC tree in order to facilitate fast tree traversal, the cost of this additional storage is not significant since the content of any integrity tree (i.e., MAC, hashes, and encryption counters) is always stored in the external memory.

2) *Update (Shift-Up) Procedure:* For every write request on a memory address protected by the tree, an update procedure had to be performed recursively on all nodes from the leaf node until the root. This recursive checking to update add to the overall time required for authentication. In spite of this additional overhead, we are able to achieve performance gain over the balanced tree. This can be attributed to the fact that verification in itself is a recursive process, where each node on the path from data to the root has to be verified. The verification process includes decrypting/encrypting the tree nodes, which in itself is computationally intensive. The relative overhead caused by the update procedure is small as it only involves changing pointers for the node attributes. As shown in our experiment results, we are able to achieve the reported timing performance gains despite this overhead.

VI. Security Analysis

Spoofing attacks are detected by making use of the block AREA scheme [19]. Under this scheme, we add redundant data (i.e., nonce) to our original data blocks before encryption. The nonce is checked during the verification step after decryption. The diffusion property of encryption engines ensures that any change on the data will be reflected after decryption as the nonce obtained would have changed. Any mismatch would raise an alarm in the system.

Splicing attacks are detected at the first level of verification itself. The node ID, n_i, of the fetched parent chunk is matched against the parent ID, p_i, of the child node. The node number is also matched with the address being requested (address = tree base address + node number). This ensures that there is no address mismatch.

Replay attacks are prevented due to the property of the nonce that is unique to each location. If an address is replayed, the nonce values of the replayed version and the current version will not match. Thus, the attack would be detected at the first non-replayed node. If the entire tree has been attacked, the last verification step of matching the root node with the on-chip counter will trigger the alarm.

VII. Experimental Results and Discussion

Our earlier implementations were evaluated only on FPGAs that had limited memory and no caches. For this work, we have implemented and evaluated our framework on a system simulator, Multi2Sim. For our evaluations, we used 12 benchmarks from the SPLASH-2, PARSEC, and SPEC-CPU2006 benchmark suites on Multi2Sim [20]. The Multi2Sim simulator integrates a number of useful features, such as separate functional and timing simulation, multithreading and multiprocessor support, and cache coherence. Multi2Sim is an application-only tool intended to simulate x86 binary executable files. Our experiments were undertaken using a single-core system, with a single fully associative cache utilizing a write-back policy and a least recently used (LRU) replacement policy. We will present the results for six different cache configurations, i.e., 256-kb, 1-Mb, and 4-Mb cache with 64- and 128-byte cache lines. The main memory latency has been used with 100-cycle latency, 1.4 GHz, and 8 B/cycle. As memory authentication contributes to significant overhead in the execution time, we measured the time required to perform integrity checking, i.e., encryption/decryption and verification. In addition to comparing our work with the work in [5], since we modify their basic framework, we have even compared our work with the work in [12] to highlight the effectiveness of this paper. To the best of our knowledge, [12] is the only work other than ours that tackle the problem of tree size by taking advantage of memory access patterns.

We ran the benchmarks for three different tree architectures: 1) BTEC [5]; 2) *ASSURE* [12] which is adapted to the TEC tree format; and 3) proposed dynamic skewed tree (denoted as Proposed). Note that all the implementations are based on the TEC tree format to ensure fairness in the evaluations. In addition, block encryption with cache line granularity is applied to all the methods for fair evaluations. AES encryption/decryption latency has been considered to be 40 clock cycles. In order to ensure fair comparisons, all the trees adopt an arity of 2.

A. Authentication Time

The authentication time is directly proportional to the number of tree levels accessed for the integrity verification of a leaf node. We have measured the authentication time for six different cache configurations with varying cache size and cache line sizes. The time taken to perform the integrity check is minimum for the proposed dynamic skewed tree for all the benchmarks, as shown in Fig. 5. The timing gain achieved varies for each application based on its memory usage pattern. From the results, it is evident that moving the nodes closer to the root reduces authentication time as both ASSURE and Proposed design performed better than BTEC. The proposed method has, on average, 35% lower integrity check runtime over BTEC and 18% lower runtime over ASSURE. The proposed dynamic skewed tree performs better than ASSURE due to its higher degree of adaptability where the placement of each leaf node is based on the individual count value of the leaf node rather than

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

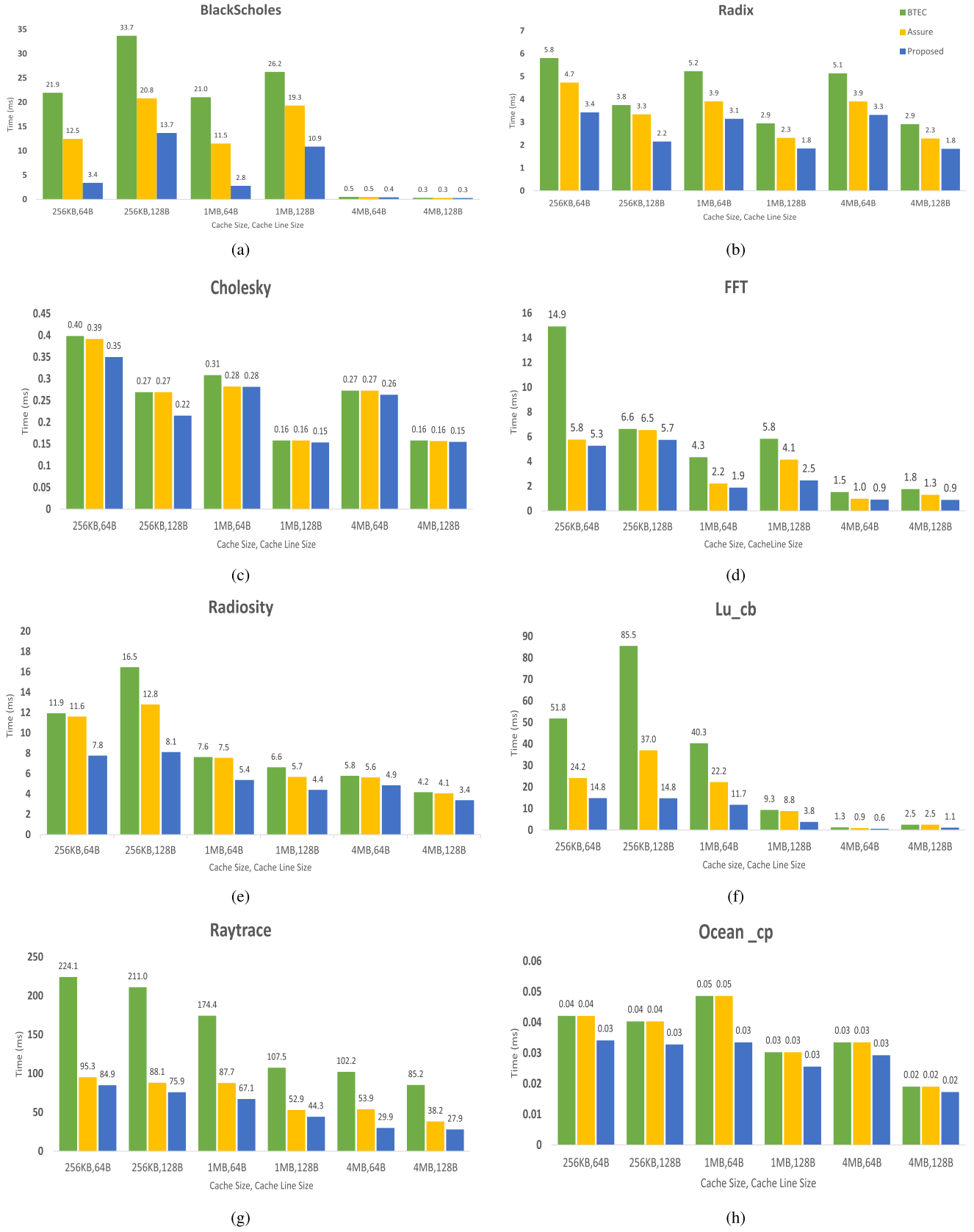


Fig. 5. Authentication time (in ms) for 12 different applications with varying cache configurations.

on the MBG as in the ASSURE architecture. As expected, it can also be observed that larger cache leads to improved performance.

The benefits of skewing a tree are application-dependent. In particular, the gains are mainly affected by the number of memory writes. This is due to the fact that the counter values

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

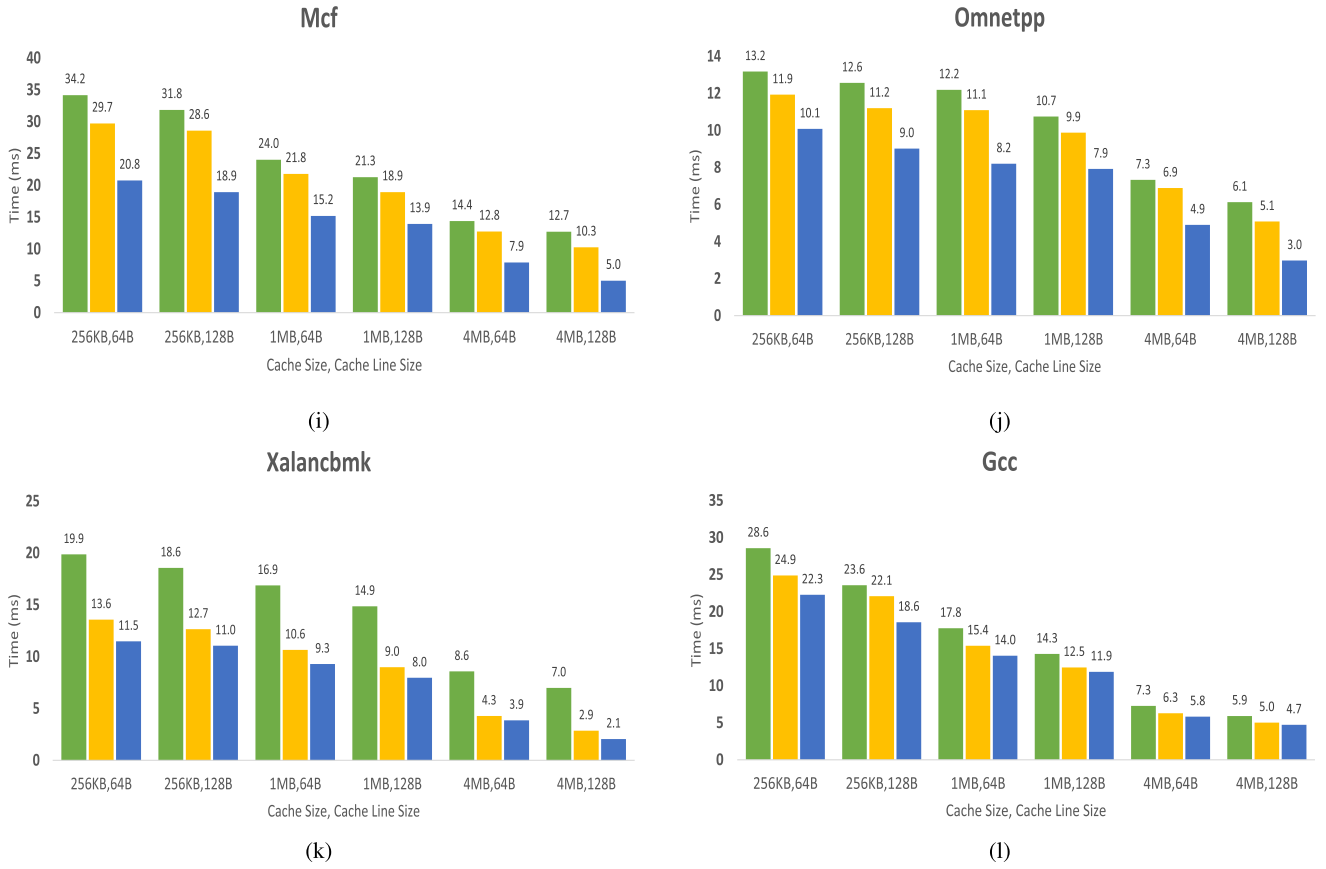


Fig. 5. *(Continued.)* Authentication time (in ms) for 12 different applications with varying cache configurations.

TABLE I
MEMORY ACCESS PATTERNS

Benchmarks	Reads	Writes
Blackscholes	80,242	19,549
Radix	3,212	140
Cholesky	3,418	165
FFT	15,81,279	5,42,979
Radiosity	6,777	12,05,435
Lu_cb	7,42,191	5,72,497
Raytrace	56,804,585	20,144,507
Ocean_cp	4,022	1,172
Mcf	1,30,857,287	36,484,603
Omnetpp	1,07,159,247	69,846,205
Xalancbmk	1,77,685,047	59,298,527
Gcc	95,761,851	74,480,861

are incremented on every write followed by a rebalancing check (see Algorithm 1). Thus, a large number of memory writes will lead to more frequent skewing that increases the overhead of verification.

The memory patterns of all the applications are shown in Table I. The proposed method achieved high gains for applications that are sufficiently large with a substantial number of memory accesses. Such applications will result in trees with many levels for the BTEC implementation. The proposed dynamic skewed tree can effectively lower the overhead in verification for these applications by reducing the average number of tree levels during authentication.

We have included *mcf, omnetpp, xalancbmk, and gcc* workloads from SPEC2006 in our evaluations, as they have high memory traffic ($\geq$1 memory access per 1000 instructions) and random data accesses. It can be observed that for these benchmarks, the proposed scheme leads to an average reduction of authentication time of about 33% and 19% over ASSURE and BTEC, respectively. Other interesting observations are applications, such as *Radiosity* that has a large number of memory accesses, that do not benefit as much from the proposed method. This is because the majority of requests are writes, which are substantially higher than the reads. In this scenario, the system spends significant time performing the rebalancing operation leading to higher overhead. It is noteworthy that even with this increased overhead, the proposed method still achieves a gain of 30% over ASSURE and 25% over BTEC. The remaining three applications, Radix, Cholesky, and Ocean_cp, are relatively smaller with fewer number of accesses made to the main memory. This results in a compact integrity tree with better cacheability of entries on-chip. If a part of the protected region is in the cache, it is considered safe and hence not verified. This results in fewer number of verifications. Such nonmemory intensive workloads with infrequent tree traversals have limited potential for reduction in authentication time. Thus, the benefits of skewing the tree are limited (on average 15% over ASSURE

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE II
4-MB BALANCED VERSUS 1-MB PROPOSED

Benchmarks	Execution Time (sec)		% Difference	Energy (Joules)		% Energy Saved
	4MB BTEC	1MB Proposed		4MB BTEC	1MB Proposed	
Blackscholes	647.85	677.23	-4.5%	3368.85	3047.55	9.57%
Radix	4.91	3.21	+34.5%	25.54	14.46	43.3%
Cholesky	0.62	0.63	-1.08%	3.27	2.86	12.52%
FFT	1060.10	1065.14	-0.47%	5512.57	4793.18	13.05%
Radiosity	229.8	235.38	-2.42%	1195.00	1059.24	11.36%
lu_cb	1750.83	1758.8	-0.48%	9100.71	7913.60	13.04%
Raytrace	3114.39	2550.99	+18.09%	16195.87	11479.46	29.11
Ocean_cp	1.31	1.34	-2.23%	6.84	6.05	11.52
Mcf	387.58	606.24	-4.8%	2015.43	1828.11	9.29%
Omnetpp	778.48	780.47	-0.25%	4048.13	3512.14	13.24%
Xalancbmk	1128.32	1152.85	-2.17%	5867.30	5187.86	11.58%
Mcf	1202.59	1237.25	-2.88%	6253.51	5567.66	10.96%

TABLE III
1-MB BALANCED VERSUS 256-KB PROPOSED

Benchmarks	Execution Time (sec)		% Difference	Energy (Joules)		% Energy Saved
	1MB BTEC	256KB Proposed		1MB BTEC	256KB Proposed	
Blackscholes	671.30	705.49	-5.09%	3067.86	2941.93	4.10%
Radix	8.55	5.73	+32.9%	39.08	23.91	38.8%
Cholesky	0.96	1.00	-4.07%	4.31	4.17	5.02%
FFT	1096.98	1108.04	-1.00%	5013.20	4620.58	7.83%
Radiosity	363.13	364.98	-0.50%	1659.53	1521.97	8.28%
lu_cb	1936.06	1922.24	+0.71%	8847.83	8015.77	9.40%
Raytrace	3753.56	3239.53	+13.69%	17153.79	13509.52	21.24%
Ocean_cp	1.46	1.56	-6.43%	6.69	6.50	2.88%
Mcf	446.58	451.58	-1.11%	2009.62	1883.10	6.29%
Omnetpp	799.46	796.84	0.32%	3597.59	3322.83	7.63%
Xalancbmk	1183.69	1197.83	-1.19%	5326.62	4994.97%	6.22%
Gcc	1259.46	1262.58	-0.24%	5667.59	5264.97	7.10%

and 20% over BTEC). Among these three applications, Radix seems to benefit the most from the proposed method. This can be attributed to its memory access pattern, where accesses have a random nature without much temporal locality. This results in considerable memory traffic during integrity tree traversal as there is limited reuse of the cache entries. Applications, such as Radix, can attain high benefits from dynamically skewed trees due to the reduced number of authentication levels. It can be observed that Radix achieves on average a gain of about 38% over BTEC.

Finally, Fig. 5 shows the effect of varying the cache 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.

B. Energy

This is the first time that we perform energy evaluations for our work. We perform energy evaluations using multicore power, area, and timing (McPAT). McPAT is an integrated power, area, and timing modeling framework for multithreaded, multicore, and manycore architectures. The simulations are performed using the same system configurations and technology as Multi2Sim.

The results in Tables II and III compares the total runtime and energy consumption of the eight applications when they are executed using the BTEC and the proposed method for memory authentication. Each row in Tables II and III shows the runtime and energy consumption for the BTEC running on a system with a larger cache size compared with the proposed method. For example, in Table II, the BTEC method is executed on a system with 4-MB cache, while the proposed method is executed on a system with 1-MB cache. Similarly, Table III compares the runtime and energy consumption when the BTEC is executed on a system with 1-MB cache, while the proposed method executes on a system with only 256-kB cache. The fourth and seventh columns in Tables II and III report the percentage reduction in runtime and the percentage energy savings of the proposed method over BTEC. It can be observed that for all the applications considered, the proposed method has almost a similar ($\leq$6% decrease) or better performance over BTEC even though the former is executed on a system with the significantly smaller cache. For Radix and Raytrace, the proposed method achieve almost 30% and 15% lower runtime, respectively. For all the cases considered, the proposed method leads to energy reduction due to the

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

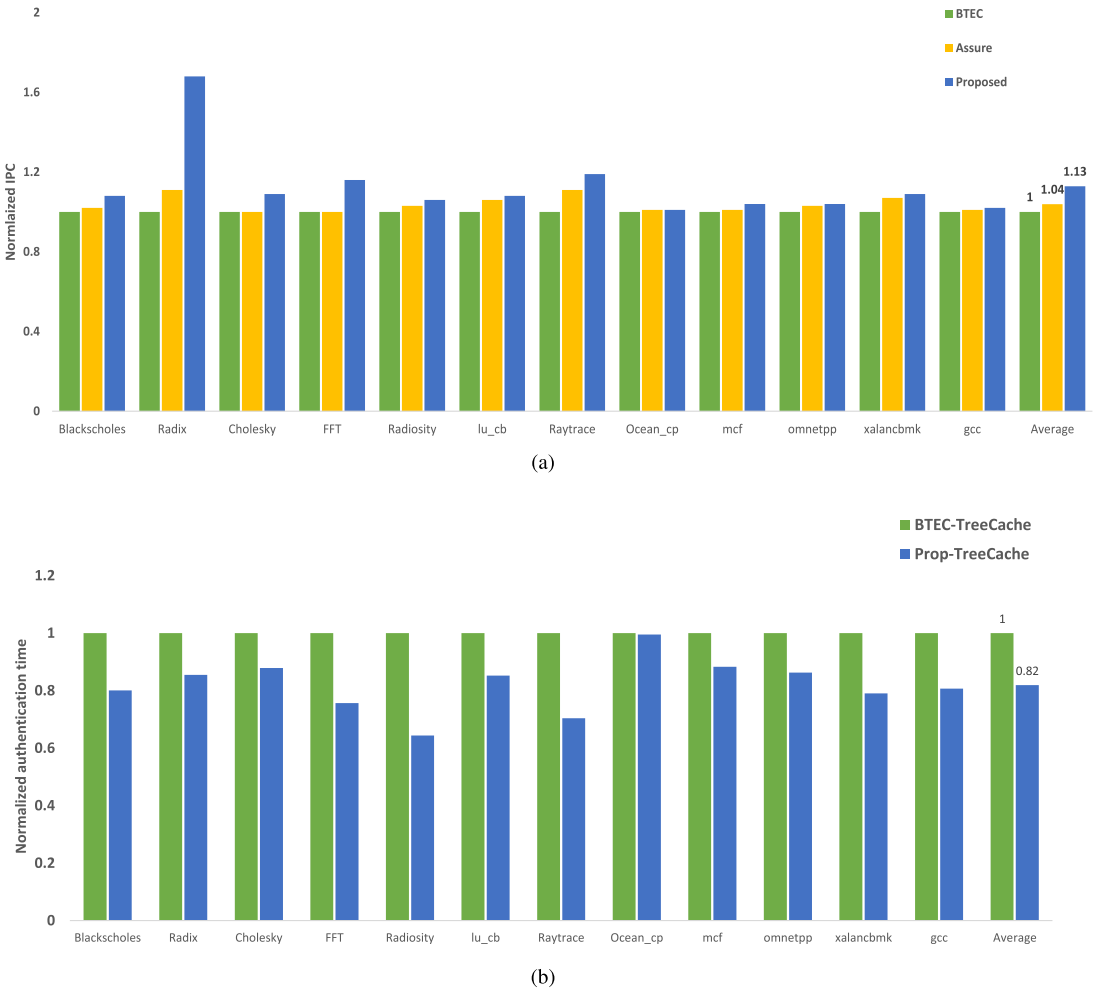


Fig. 6. (a) IPC Comparison. (b) Authentication time (in ms) for integrity trees with TreeCache.

use of a smaller cache. This reduction can be over 43% in certain applications. Similar results are also observed when the proposed method is compared with ASSURE. Although the gains are not as significant, the proposed dynamic skewed tree was still able to achieve energy savings at the cost of slight or no performance loss.

The combined advantages of the proposed integrity tree structure (which was designed in a cache-aware manner) and our proposed mechanism to dynamically skew the tree result in comparable performance with the existing memory authentication methods that require almost double the cache size. Hence, our memory authentication method provides the advantage for choosing systems with smaller caches to minimize energy consumption without compromising performance.

C. Instruction Per Cycle

Fig. 6(a) shows the impact of memory verification on the overall application performance in terms of IPC for a system with 256-kB cache and 64-byte cache line. For all the different benchmarks, the IPC of the tree schemes is shown with AES implemented as custom instructions. It can be observed that the proposed method achieves an average IPC improvement of 13% over the balance tree and 10% over ASSURE. The maximum gain is close to 50% for Radix. Smaller applications, such as Ocean_cp, show the minimum gain.

D. TreeCache Sensitivity Analysis

As discussed in Section II, the authentication overhead of the memory integrity trees can be reduced by using a dedicated TreeCache for caching the tree nodes. In order to study the impact of using TreeCache, we implemented the proposed design with a TreeCache and compared its performance with the BTEC model that also has a TreeCache. In the presence of a TreeCache, authentication of the leaf node stops as soon as we encounter a tree node that is cached. For our evaluations, we used a system with 256-kB data cache and 64-byte cache line. The size of the TreeCache is 16 kb with the 64-byte cache line. It can be observed in Fig. 6(b) that the authentication time of the proposed method is reduced by approximately 20% compared with BTEC.

VIII. Conclusion

We presented a memory authentication framework for dynamically skewed integrity tree, which combines

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

architecture-specific optimization of the integrity tree structure with runtime mechanisms to minimize the authentication overhead. A new node structure is proposed, which enables the integrity tree to be restructured dynamically with low-performance overhead. Compared with the existing integrity tree methods that are also based on exploiting memory access patterns, the proposed integrity tree is able to adapt more effectively to the dynamic memory access patterns by maintaining a separate counter for each leaf node. Experiments show that the proposed method achieves the highest performance gains over the existing methods for reasonably large applications where the majority of the memory accesses are reads. We also show that the proposed framework is able to select a suitable cache configuration to minimize energy consumption. Extensive simulations on systems with varying cache configurations using applications from SPEC-CPU2006, SPLASH-2, and PARSEC benchmarks show that the proposed framework leads to a significant reduction in authentication time compared with the existing methods. For our future work, we plan to develop a programmable MC that consists of processors with custom ISAs that can be adapted to the memory workloads in order to reduce energy consumption.

REFERENCES

[1] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres, "Hardware mechanisms for memory authentication: A survey of existing techniques and engines," in *Transactions on Computational Science IV*. Berlin, Germany: Springer, 2009, pp. 1–22.

[2] D. Lie *et al.*, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[3] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors OS-and performance-friendly," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2007, pp. 183–196.

[4] S. Gueron, "A memory encryption engine suitable for general purpose processors," IACR Cryptol. ePrint Arch., 2016, p. 204.

[5] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Berlin, Germany: Springer, 2007, pp. 289–302.

[6] G. Saileshwar, P. Nair, P. Ramrakhyani, W. Elsasser, J. Joao, and M. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 416–427.

[7] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing paging overheads in SGX with efficient integrity verification structures," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, Mar. 2018, pp. 665–678.

[8] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proc. 9th Int. Symp. High-Perform. Comput. Archit., (HPCA)*, Feb. 2003, pp. 295–306.

[9] J. Lee, T. Kim, and J. Huh, "Reducing the memory bandwidth overheads of hardware security support for multi-core processors," *IEEE Trans. Comput.*, vol. 65, no. 11, pp. 3384–3397, Nov. 2016.

[10] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 381–391, 2007.

[11] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, 2010.

[12] J. Rakshit and K. Mohanram, "ASSURE: Authentication scheme for SecURE energy efficient non-volatile memories," in *Proc. 54th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2017, pp. 1–6.

[13] S. Vig, T. Y. Tzer, G. Jiang, and S.-K. Lam, "Customizing skewed trees for fast memory integrity verification in embedded systems," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2017, pp. 213–218.

[14] S. Vig, G. Jiang, and S.-K. Lam, "Dynamic skewed tree for fast memory integrity verification," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 642–647.

[15] T. Unterluggauer, M. Werner, and S. Mangard, "MEAS: Memory encryption and authentication secure against side-channel attacks," *J. Cryptograph. Eng.*, vol. 9, no. 2, pp. 137–158, Jun. 2019.

[16] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," *IACR Cryptol. ePrint Arch.*, vol. 169, pp. 1–47, Jun. 2002.

[17] Y. Tsunoo, "Crypt-analysis of block ciphers implemented on computers with cache," in *Proc. ISITA*, Oct. 2002.

[18] S. Pigeon and Y. Bengio, "A memory-efficient adaptive Huffman coding algorithm for very large sets of symbols," in *Proc. Data Compress. Conf. (DCC)*, Mar. 1998, p. 568.

[19] C. E. Shannon, "Communication theory of secrecy systems," *Bell Syst. Tech. J.*, vol. 28, no. 4, pp. 656–715, 1949.

[20] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," in *Proc. 21st Int. Conf. Parallel Architectures Compilation Techn. (PACT)*, Sep. 2012, pp. 335–344.