# DISSECT: Dynamic Skew-and-Split Tree for Memory Authentication

Saru Vig*, Rohan Juneja† and Siew-Kei Lam*

* Nanyang Technological University, Singapore, †Qualcomm, India

*Abstract*—**Memory integrity trees are widely-used to protect external memories in embedded systems against replay, splicing and spoofing attacks. However, existing methods often result in high-performance overhead that is proportional to the height of the tree. Reducing the height of the integrity tree by increasing its arity, however, leads to frequent overflowing of the counters that are used for encryption in the tree. We will show that increasing the tree arity of a widely-use integrity tree from 2 to 8 can result in over 200% increase in memory authentication overhead for some benchmark applications, despite the reduction in tree height. In this paper, we propose DISSECT, a memory authentication framework which utilizes a dynamic memory integrity tree that can adapt to the memory access patterns of the application by progressively adjusting the tree height and arity in order to significantly reduce performance overhead. This is achieved by 1) initializing an integrity tree structure with the largest arity possible to meet the security requirements, 2) dynamically skewing the tree such that the more frequently accessed memory locations are positioned closer to the tree root (overcomes the tree height problem), and 3) dynamically splitting the tree at nodes with counters that are about to overflow (overcomes the counter overflow problem). Experimental results undertaken using Multi2Sim on benchmarks from SPEC-CPU2006, SPLASH-2, and PARSEC demonstrate the performance benefits of our proposed memory integrity tree.**

## I. Introduction

Numerous schemes have been reported for mitigating memory attacks [1], [2]. These methods inevitably make use of some form of encryption and a memory integrity tree. In an integrity tree, the memory contents are stored as leaf nodes of the tree after encryption/hashing. The tree structure is constructed by recursively applying a primitive authentication technique (e.g., MAC, hash) from the leaf nodes to the root. During authentication, a validation is performed at each level of the tree until the root, which necessitates multiple memory accesses. As such, memory authentication incurs a performance overhead that is proportional to the height of the tree. The works in [3], [4] discuss methods to reduce the height of the integrity tree by increasing the tree arity. For example, [3] designed a tree with 128-arity to reduce the height of integrity trees for securing large memories. For protecting a 16 GB memory, they were able to reduce the tree height to 4 levels.

In order to construct shallow trees with larger arity, [3], [4] employ varying encryption counter configurations. Such approaches, however, has their limitations. Firstly, as reported in [3], increasing the tree arity is not a strategy that can be adopted universally. For example, they cannot be applied to Message Authentication Codes (MAC) trees whose arity is limited to 8 irrespective of the counter design in the tree node. The arity for MAC trees depends on the MAC per entry and the cache line size. Secondly, the success of methods such as [3], [4] is dependent on the counter nodes being cached efficiently, as this contributes to reducing the number of counter writes and overflows on higher levels of the tree. However, introducing a dedicated cache for the counter nodes is not always possible, especially for embedded systems with tight constraints. Moreover, approaches that rely on caching the tree nodes have reported overall performance degradation due to cache contention [5]. Storing the metadata of the tree nodes in caches for large trees consumes substantial memory space. [5] 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.

In systems that require strong protection, when a counter overflows, it is essential that a new key is generated and the entire tree is re-encrypted using the new key. Without this preventive measure, the tree contents will not be secure against replay attacks if counters are re-initialized. As such, the use of larger arity can inadvertently lead to higher performance overheads due to more frequent counter overflows. For example, the technique proposed in [3] employs a 128-ary integrity tree which is made possible through the use of a single sizable major counter per cache line and several smaller minor counters in the tree nodes. As mentioned in their work, a non-optimized design packing of 128 counters per cache line results in 3-bit minor counters that can overflow in just 8 writes. As each overflow requires 256 extra memory accesses, this will result in significant performance degradation. Even with their optimized design, the counters tend to overflow frequently especially for certain data streaming applications, leading to high performance overheads. The experiments conducted in [3] demonstrated cases where the higher performance overheads due to counter overflow offset the gains obtained due to the reduction in tree height.

Reducing counter size to increase arity, reduces the security against replay attacks. The probability of a successful replay attack depends directly on the size of the counter. If an attacker manages to guess the counter value, the system security will be compromised. Thus, increasing arity by reducing counter size is only an option for systems that do not require a high level of protective measure [6], or if other mechanisms to detect replay attacks are available.

Another means of reducing the tree height during memory authentication is by skewing the tree based on static or run-time characteristics of the application. The work in [7] proposed to skew the integrity tree based on the static memory profiling results of the application. At run-time, the structure of the tree remains static throughout. ASSURE [8] employs smart MACs in an integrity tree, which can reduce the authentication time by dynamically creating a smaller sub-tree within a standard binary tree. The authors in [9], [10] propose approaches to dynamically skew trees based on memory access patterns, where the tree is restructured at run-time to shift memory locations that are frequently accessed closer to the root, thereby reducing the number of levels to be verified during authentication. Although the authors in [10] report performance gains as a result of dynamically skewing the integrity tree, their design was limited to a 2-ary tree.

### A. Main Contributions

In this paper, we propose a memory integrity tree structure and the corresponding authentication framework that simultaneously addresses the problems associated with tree height and counter overflow. We call the proposed framework DISSECT, which stands for Dynamic Skew-and-Split Tree for Memory Authentication. DISSECT enables the height and arity of the proposed memory integrity tree structure to be dynamically adjusted based on the run-time memory access patterns in order to reduce the performance overhead for authentication. This is achieved by 1) initializing an integrity tree structure with the largest arity possible to meet the security requirements, 2) dynamically skewing the tree such that the more frequently accessed memory locations are positioned closer to the tree root, and 3) dynamically splitting the tree at nodes with counters that are about to overflow.

DISSECT leverages on the complementary benefits of reducing tree height and increasing arity. This is based on the observation that nodes which tend to encounter counter overflow are also the nodes that are most frequently accessed (the counters associated with each memory location are incremented on the memory write access). DISSECT overcomes the tree height problem by shifting the nodes that are associated with the most frequently accessed memory locations closer to the root (to reduce the number of authentication levels of these nodes), and restrict the splitting process to these nodes when their counters are about to overflow.

We performed a detailed analysis on existing methods to show the effects of varying the counter size on authentication time, and demonstrate that increasing arity for large memory integrity trees can negatively impact the performance of the system due to the counter overflow problem. Experimental results undertaken using Multi2Sim on benchmarks from SPEC-CPU2006, SPLASH-2, and PARSEC demonstrate that the combined benefits of the proposed skewing and splitting scheme provides a scalable solution for low overhead memory authentication, particularly when the data to be protected is large.

## II. Proposed Tree Structure

### A. Overview

Our framework begins with a balanced integrity tree with $a$-ary. An example of 4-ary tree is shown in Fig. 1a. In this work, we have demonstrated the proposed approach on the Tamper Evident Counter (TEC) tree [11]. Note that DISSECT can be also applied to other memory integrity trees (e.g. Hash Trees, Merkle Tree). TEC tree provides security utilizing Block AREA (Added Redundancy Explicit Authentication) strategy. The tree nodes are partitioned into two classes, Data Chunk (DC) and Counter Chunk (CC). The orange and blue blocks in Fig. 1a are the CCs and DCs respectively. A nonce, which is unique to every chunk, is added to each tree node before encryption. Nonce includes a count concatenated with the memory address location of every node. Count is the number of write requests performed on each node, and is added to detect replay attacks. During authentication, the nonce is checked with its immediate parent node. Both DC and CC are encrypted and therefore provide confidentiality at no extra cost. This is a significant advantage of TEC tree over its counterparts.

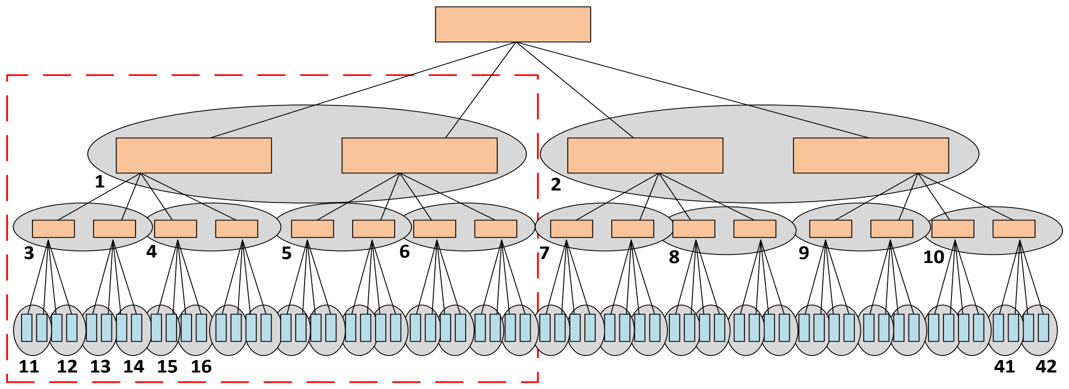
In DISSECT, initially a balanced tree is divided into different groups with each of them comprising of $a/2$ nodes. The numbers in Fig. 1a indicate the group numbers. The tree arity is determined based on the system's cache and security requirements, i.e., cache line size and encryption block. At run-time, the tree is skewed according to the memory access patterns. We have magnified the left portion of the 4-ary tree in Fig. 1b (i.e. the red bounding box in Fig. 1a) to illustrate an example of skewing. In this example, Group 15 has been switched with its uncle group, Group 3, since the former was accessed more frequently. As a result, Group 15 has been elevated one level closer to the root. Subsequent access to Group 15 will require a lesser number of authentication steps.

Furthermore, when any node is about to exceed its maximum local counter value, the node is split. A new node is created, and half of its children are assigned to the new node. An example of the tree split is shown in Fig. 1c, where we assume that the local counter of node 1 has reached its maximum value. Consequently, the arity of node 1 has decreased from 4 to 3, resulting in extra counter bits for nodes in Group 15. We will discuss in detail each of the above steps in the following sections.

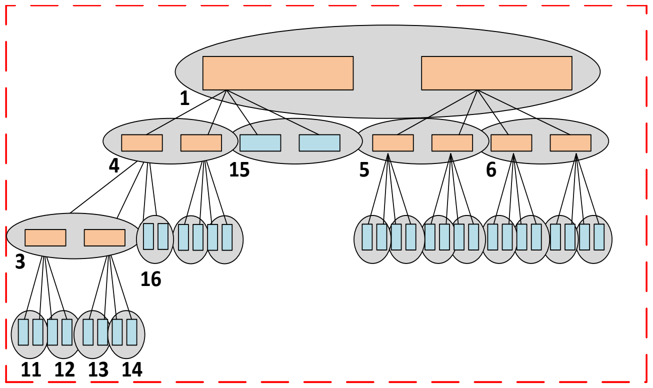
### B. Proposed Tree Node Structure

In order to achieve dynamic skewing and splitting efficiently, we need to redesign the nodes of the memory integrity tree. The protected data from the external memory is first divided into equal-sized blocks, and each block is used to create a single DC. DC comprises of 'data' concatenated with a nonce value. Nonce is created using a count value, $c_i$, which is equal to the number of write requests made to the DC. $c_i$ is concatenated with the node ID, $n_i$, making the nonce unique to each node. $p_i$, $s_i$, $LR_i$ are the other attributes of each DC as shown in Fig. 2a. $p_i$ 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 are important as they contribute to the skewing and spitting steps. In particular, for a conventional balanced tree, the parent's position can be easily inferred from the child's position. However, when skewing or splitting is undertaken, the position of the nodes can no longer be inferred easily, and hence it is vital to explicitly store the parent's information. The sibling's information will be used to check the skewing requirements, as will be discussed in Section III-B.

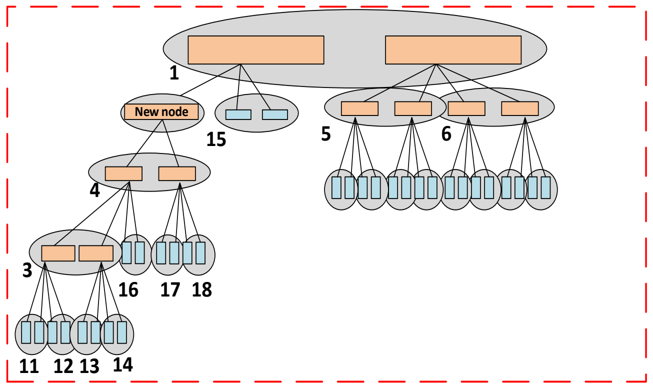
The counters, $c_i$, are also stored in the off-chip memory in a hierarchical tree structure format. In the rest of the paper, we refer to CC as the node where the counters are stored. Each CC also comprises of its own nonce and other attributes, similar to that of a DC. CC also store $Split_i$, which denotes whether the node or any of its children have split and thus have an increased



(a)



(b)



(c)

Fig. 1: An example structure of (a) 4-ary DISSECT (b) post Skew (c) post Skew-and-Split

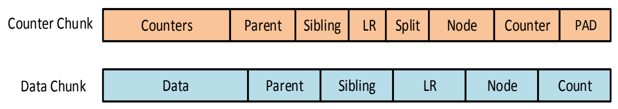
counter size. A *Split* flag is necessary for checking a given nodes counter width. An entire CC is encrypted as a single block before being stored off-chip in the external memory. This scheme is recursively applied to subsequent levels of the tree till we obtain a single CC, called the root node of the tree. The count of the root node is stored on-chip securely. Thus, the tree structure reduces the on-chip memory overhead to a single node while providing full memory integrity for multiple leaf nodes.

## III. Proposed Skew-and-Split Framework

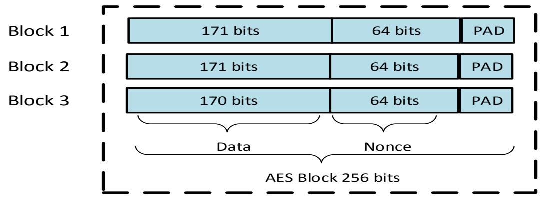
The proposed framework consists of the following steps: 1) initializing an integrity tree structure with the largest arity possible to meet the security requirements, 2) dynamically skewing the tree such that the more frequently accessed memory locations are positioned closer to the tree root, and 3) dynamically splitting the tree at nodes with counters that are about to overflow.

### *A. Initialize Tree Structure*

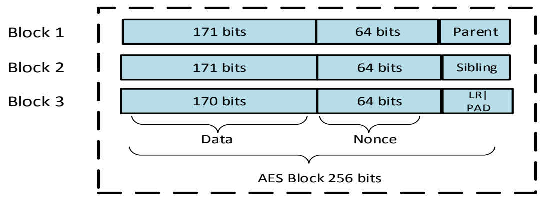
A memory block (data + nonce) loaded by the processor on a cache miss must be a multiple of the ciphered-block length. Therefore, the 'data' width for a given tree node is determined such that on a cache miss, a single memory block unit will forward 'data' of size equal to the cache line size upon verification. The structure for a 512-bit cache line with 256 cipher block length is shown in Fig. 2b. Based on Eq. 1,



(a)



(b)



(c)

Fig. 2: (a) Proposed Node Structure (b) memory block with 512-bit cache line for TEC tree, and (c) memory block of DISSECT

we can determine the arity and 'counter', where 'counter' is the bit width of a single local counter of each tree node.

$$arity = \frac{data}{counter} \tag{1}$$

We also improvised the structure of the DC introduced in Fig. 2a and split the additional attributes (i.e. $p_i$, $s_i$, $LR_i$) over the data chunks in a single memory block as shown in Fig. 2c to maximize the memory utilization. This is possible as on a cache miss, these data chunks of a single memory block have to be verified together. As such, the proposed structure avoids the redundancy of storing the parent/sibling information in each node individually.

As an example, for a 512-bit cache line, the minimum size of the nonce required for data to be equal to the cache line is one AES block (256-bit). The resulting memory block distribution is three AES blocks long, i.e. 768-bit as shown in Fig. 2b. To maintain 170-bit 'data', we calculate the 'counter' size to extend the arity to 8 using Eq. 1, which is 21-bit. The possibility that a replay attack will be successful is $1/2^{21}$. Following the same principle, we customize data/counter chunks of the proposed tree structure to fit the given cache line, as shown in Fig. 2b and 2c. Note that the counter width should be determined based on the security requirements against replay attacks, which is discussed in Section IV. Compared to the TEC tree with the same security strength, the proposed method results in a lower arity. To retain the 21-bit width of the counters, the proposed tree will only be able to accommodate six child counters instead of eight. Similarly, different counter sizes will result in different arity. For the TEC tree, arity must be a power of 2 to simplify the hardware computations of parent address [11]. Such a restriction does not apply to the proposed design as the parent node number is explicitly stored in all the chunks.

The extra attributes (i.e., $p_i$, $s_i$, $LR_i$, and $Split_i$) of each CC utilize extra bits, reducing the remaining bits to store counter values. For a 32-bit tree, the last level is made up of DC*s* and cannot be a parent to any node. Thus, $p_i$ requires 31-bits. $s_i$ stores sibling group number (each group has $a/2$ nodes, so there are half as many groups as parents) and requires 30-bits. $Split_i$ will hold information required to decide if a split has occurred, and if so, which of the left or right group of children was assigned to new nodes. $LR_i$ indicates if the node is a left or right child to its parent.

## B. Dynamic Skewing

For an $a$-ary integrity tree, the attributes must now incorporate the information of ($a$-1) siblings instead of just 1. To avoid storing this additional meta-information, we create 'group' of $a/2$ nodes and store a group ID as $s_i$. The benefits of dynamically skewing a tree are extensively discussed in [10], [9]. We have improvised the *ShiftUp* algorithm introduced in [9] for higher arity trees. The *ShiftUpGroup* procedure to perform dynamic skewing is applied to such groups instead of single nodes. The storage is then reduced to storing only a single sibling group number (similar to a 2-ary design where a single sibling node number is stored).

*ShiftUpGroup*: Let T be the group whose node is to be checked for shifting. And let Q be the parent node.

1) Check if total $c_i$ of T is greater than its corresponding sibling group, $s_i$, total count by 1, and is also greater than the total count of its uncle group, $s_{p_i}$.
2) Exchange T with its uncle group.
3) Exchange the new children node of Q, i.e. switch their positions.
4) Recursively perform steps 2-4 for all nodes on the path from T to the root node.

## C. Dynamic Splitting

**Algorithm 1:** Dynamic Split Algorithm

```
A, Q : pointers to Nodes
while A is not root do
    if $count_{p_A}$ = max then
        Q ← create new node
        $p_Q \leftarrow p_A$
        $s_{s_A} \leftarrow node_Q$
        $s_Q \leftarrow s_A$
        if $\sum_{i=my_group} c_i \geq \sum_{i=sibling_group} c_i$ then
            for all $n_i$ in my group do
                $p_i \leftarrow Q$;
                Update $s_i$;
            end
            Update $count_{Q,A}$
        else
            for all $n_i$ in my sibling group do
                $p_i \leftarrow Q$;
                Update $s_i$;
            end
            Update $count_{Q,A}$
        end
    else
        A ← $P_A$
    end
end
```

As shown in Algorithm 1, when a counter value reaches its (max-1) permissible limit, it is selected for splitting its children nodes. A new node is created and half the children is assigned to this node, thereby providing additional bits to their respective counter values to prevent overflowing. It is noteworthy that although splitting will push nodes one level down, the subsequent shift operation promotes the nodes to a higher level. Also, splitting is performed recursively, so it is highly likely that further splits will follow on the same nodes. As each split incurs an additional tree level, the tree size is bound to increase. However, the number of increased levels will be relatively small compared to the original tree height. This is due to the fact that nodes which undergo splitting are likely to have shifted closer to the root (since they are accessed frequently).

## D. Memory Authentication

Algorithm 2 describes the verification procedure for the proposed method.

*ReadNCheck*: This function is called when there is a read request sent to the protected data region and thus requires verification.

1) Call the requested DC'*s* parent CC from external memory.

**Algorithm 2:** Memory Authentication with DISSECT

```
begin
    Initialize tree with data elements as data chunks 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);
            Split_flag ← Split_check(addr)
        end
        if Rebalance_flag then
            ShiftUp(addr);
        end
        if Split_flag then
            Split(addr);
        end
    end
end
```

2) Verify decrypted CC with the child counter value.
3) Repeat above steps till the root node is reached.
4) 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 and thus requires verification.

1) Call the requested DC'*s* parent CC from external memory
2) Verify decrypted CC with the child counter value.
3) Increment $c_i$.
4) Check for *ShiftUpGroup* and *Split*.
5) Repeat above steps till the root node is reached.
6) If root node verifies correctly, return the requested data.

## IV. Security Analysis

The proposed method is able to mitigate spoofing, splicing, and replay attacks. We also maintain confidentiality by encrypting the data using the AES algorithm. We use a block size and key of 128 bits (probability of a successful attack is extremely low i.e., $1/2^{128}$). The key used for encryption is securely stored on-chip. The AES mode used is Electronic Code Block (ECB). This enables each block to be processed independently, reducing the granularity of integrity verification. Only one cipher block is loaded and decrypted for one load/store instruction. The only drawback is that it produces the same ciphered text each time for a particular data, but we overcome this limitation by using a nonce which makes each ciphered chunk unique.

Data is protected by making use of the block level AREA scheme. This scheme makes use of Shannon's diffusion property [12] to add some redundant data to the actual data before encryption and to check it each time after decryption. This is the motivation for adding a nonce to the data to form a data chunk. Once the chunk is encrypted, the data and nonce cannot be differentiated. For a $a$-bit nonce, the probability that the last $a$ bits remain the same after tampering is $1/(2^a)$.

The nonce consists address and count i.e. $a$-bit nonce = $d$-bits of address + $r$-bit of count. This makes sure that the nonce is unique for each location. The probabilities of a successful bus attack for our given threat model are as shown in Table I

TABLE I: Security Limitations

| Attacks | Spoofing | Splicing | Replay |
|---|---|---|---|
| Time (Sec) | $1/2^a$ | 0 | $1/2^r$ |

Spoofing attacks are detected by making use of the block AREA scheme. The nonce is checked during the verification step after decryption. Any change on the data will be reflected, and the last $a$ bits obtained would have changed. This mismatch would trigger an alarm, and the data will not be passed to the processor. The probability is derived directly from the use of block AREA scheme as explained earlier.

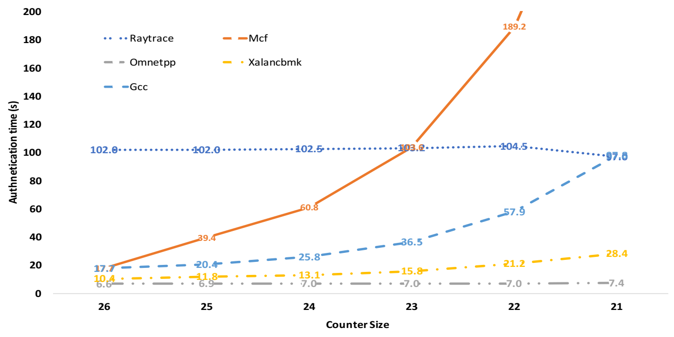


Fig. 3: Analysis of counter overflow problem

Splicing attacks are detected during the first stage of verification. As the address bits are stored in our nonce, if there is a mismatch between the address used to fetch the chunk and the bits extracted from the chunk, then the data would not be processed further, and an alarm would be raised. Thus, a 32-bit address space is completely protected from attacks if we allocate 32 bits to the address segment of the nonce.

Replay attacks are prevented due to the property of uniqueness of the nonce. If an address is replayed, the count values of the replayed and the current version will not match. The probability for a successful attack is directly dependent on the length of the count used in the nonce. The attack would be detected at the first non-replayed data block. If the entire tree is replayed, an alarm would be raised at the last verification step of matching the root node with the on-chip counter.

## V. Performance Evaluations

For our evaluations, we simulated applications from SPEC-CPU2006, SPLASH-2, and PARSEC benchmark suites on Multi2Sim [13]. Multi2Sim is an application-only tool intended to simulate x86 binary executable files. We perform experiments on a single-core system. We ran the benchmarks for two different tree architectures: 1) Balanced TEC tree [11] implementation (BTEC) 2) DISSECT. Similar to the example explained in Section III-A, we assume a system with 512 cache line.

In order to understand the counter overflow problem in existing methods, we first show the authentication time for write-intensive applications, and thus have a high probability of overflowing, (i.e., *Raytrace*, *Mcf*, *Gcc*, and *Omnetpp*) implemented with BTEC in Fig. 3. Authentication time is the time spent by the processor to perform integrity check on

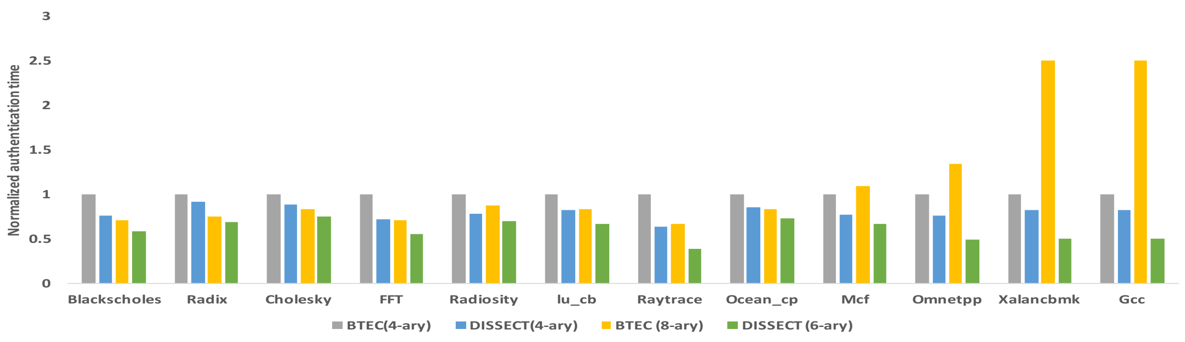


Fig. 4: Normalized authentication time

the leaf nodes. The authentication time is directly proportional to the number of tree levels accessed during integrity check. The counter width is varied from 26 to 21. For counter width of 22-26 bits, BTEC can maintain 4-arity based on Eq. 1. When the counter width is reduced to 21, it is possible to increase arity to 8. We can observe that the authentication time increase as the counter size decreases. With a single bit reduction in the counter width, the authentication time will increase exponentially for certain applications. In scenarios where the counter overflows, the cost of re-encrypting the entire tree will depend on the data size being protected. *Mcf* and *Gcc* have the largest tree size in terms of data region being protected. Thus, the overhead due to overflow for these applications is more pronounced. For *Omnetpp*, although it overflows, the overhead is not significant as only a relatively smaller data size needs to be protected. For *Raytrace*, we can observe that increasing arity is still able to improve performance despite counter overflows. This is because *Raytrace* overflows the least amount of time compared to the other applications in Fig. 3.

We have presented results in Fig. 4 to compare the normalized authentication time. For practical scenarios, a 32-bit counter is a safe size. Hence, we start with a 4-ary BTEC and 4-ary DISSECT with 32-bit counters; and then increase the arity to 8-ary BTEC and 6-ary DISSECT with smaller 21-bit counters. Note that the security strength for the 4-ary BTEC and 4-ary DISSECT, as well as the 8-ary BTEC and 6-ary DISSECT are the same respectively. It can be observed that for the same security strength, DISSECT has significantly lower authentication overhead than BTEC. 6-ary DISSECT, even with a lesser arity than 8-ary BTEC, shows on average 20% improvement in terms of reduction in authentication time over BTEC. For memory-intensive applications (e.g., *Mcf, Omnetpp, Xalankbmk, Gcc*) the counter overflow overhead for 8-ary BTEC actually causes the applications to perform even worse than the same tree with 4-arity. The results demonstrate that for the same security strength, DISSECT will outperform BTEC with a higher arity tree, as it is capable of reducing the overhead due to counter overflow.

## VI. Conclusion

In this paper, we have presented DISSECT, a skewed integrity tree which can be customized to increase arity depending on the security and performance requirements. The tree is dynamically skewed during run-time to place frequently accessed nodes closer to the root to reduce tree height. Increasing arity by reducing counter width can cause overflows. We propose a technique to tackle local counter overflows, by dynamically splitting only those nodes whose local counter is about to exceed its maximum permissible value. These techniques have been evaluated on Multi2Sim using various benchmarks to demonstrate their benefits.

## 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*. Springer, 2009.

[2] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural support for copy and tamper resistant software,” *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[3] 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 *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 416–427.

[4] M. Taassori, A. Shafiee, and R. Balasubramonian, “Vault: Reducing paging overheads in sgx with efficient integrity verification structures,” in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 665–678.

[5] B. Gassend *et al.*, “Caches and hash trees for efficient memory integrity verification,” in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE, 2003.

[6] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, “Improving cost, performance, and security of memory encryption and authentication,” in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 179–190.

[7] S. Vig, T. Y. Tzer, G. Jiang, and S.-K. Lam, “Customizing skewed trees for fast memory integrity verification in embedded systems,” in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2017, pp. 213–218.

[8] J. Rakshit and K. Mohanram, “Assure: Authentication scheme for secure energy efficient non-volatile memories,” in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6.

[9] S. Vig, G. Jiang, and S.-K. Lam, “Dynamic skewed tree for fast memory integrity verification,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 642–647.

[10] S. Vig, R. Juneja, G. Jiang, S.-K. Lam, and C. Ou, “Framework for fast memory authentication using dynamically skewed integrity tree,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[11] R. Elbaz *et al.*, “Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2007.

[12] C. E. Shannon, “A mathematical theory of cryptography,” *Memorandum MM*, vol. 45, 1945.

[13] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, “Multi2sim: a simulation framework for cpu-gpu computing,” in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2012, pp. 335–344.