Dynamic Skewed Tree for Fast Memory Integrity Verification

Saru Vig, Guiyuan Jiang and Siew-Kei Lam

School of Computer Science and Engineering, Nanyang Technological University, Singapore

***Abstract*—Memory authentication techniques often employ an integrity tree as a countermeasure against replay, spoofing and splicing attacks. However, the balanced memory integrity trees used in existing approaches lead to excessive memory access overheads for runtime verification. In this paper, we propose a framework to dynamically construct a customized integrity tree based on the data access patterns to reduce the overhead of runtime verification. The proposed framework can adapt the memory integrity tree structure at runtime such that the nodes that correspond to frequently accessed data are placed closer to the root. We validated the effectiveness of our approach on the Altera NIOS II processor with an external DRAM. Experimental results based on applications from widely used CHStone and SNU Real-Time benchmarks demonstrate that the proposed approach can lead to an average performance gain of 30% compared to the conventional means of using balanced memory integrity trees. In addition, to preserve data confidentiality, we implemented the encryption/decryption operations using custom instructions on the NIOS II processor to notably reduce the overall overhead of memory security.**

I. Introduction

Commercial processor-level security countermeasures typically define notions of trust zones or boundaries (or security perimeters) across the various on-chip hardware resources. However the protection schemes in current trusted computing platforms cannot extend fully beyond the processor core to guarantee security in external third party memories. This enables adversaries to mount active attacks on memories such as cold-boot attacks by exploiting memory remanence properties, passive bus snooping to infer secret cryptographic keys by monitoring data transfers between the processor and memory [1], or physical bus attacks. This is a major concern as the motive of almost all attacks is leakage or modification of information, making memory an obvious target.

A common attack to challenge the integrity of the content stored in external memories is through bus attacks. Such attacks replace external memory accesses with malicious content. Typical bus attacks are splicing, spoofing, and replay [2]. Hence, most memory protection schemes consists of some form of encryption and an integrity tree. Integrity tree based techniques split the memory into equal sized blocks which form the leaf nodes of the tree. The remaining nodes in the tree are created by recursively applying a primitive authentication function over the memory blocks until a single root node is left. The root is securely stored on chip and it represents the present state of the system. Thus any tampering can be detected if the root value does not match with the value stored on chip. For a leaf node to be verified and passed to the processor, authentication must be performed on all tree levels till the root. This mechanism has considerable time overhead due to accessing nodes on all tree levels for each memory access and thus utilizes the maximum fraction of energy consumed by embedded processors in memory intensive applications. This is often infeasible for embedded systems due to their tight energy constraints and performance requirements. As such, securing memory in embedded systems has been a long standing issue in trusted system design.

Prevalent techniques for of runtime memory integrity verification rely on a fully balanced memory integrity tree which suffers from significant performance overheads due to the need to traverse all levels of the tree for each memory access. In this paper, we propose a framework to construct dynamic skewed integrity tree based on the runtime memory access patterns of the application. In particular, the memory integrity tree will be dynamically restructured such that the more frequently accessed memory blocks lie in the shorter paths to the root. This will significantly reduce the number of verification steps for the frequently accessed memory blocks, leading to overall reduction in the overhead for memory authentication.

We evaluated the proposed approach for runtime data integrity verification on Altera NIOS II processor with external DRAM that stores the memory integrity tree. The approach can be adopted for code integrity verification as well. Using applications from well-known benchmarks, we show that an average runtime improvement of 30% can be achieved over conventional methods that use a balanced memory integrity tree. This clearly demonstrates that the overhead savings for runtime integrity verification of our approach far exceeds the processing overhead for constructing and maintaining the dynamic memory integrity tree. In addition, we implemented the AES algorithm for encryption/decryption using custom instructions on the NIOS II processor. This resulted in an additional 10x speedup, which further alleviates the bottleneck of runtime memory integrity verification. To the best of our knowledge, this is the first work to demonstrate the effectiveness of using dynamic memory integrity trees for memory authentication on an off-the-shelf platform using realistic applications.

This paper is organized as follows: Section II discusses related work. Section III describes the threat model and Section IV introduces the proposed memory integrity tree structure. Section V presents the proposed framework for constructing and maintaining the dynamic skewed memory integrity tree, and experimental results are shown in Section VI. We conclude the paper in Section VII.

The research described in this paper has been supported by the Academic Research Fund (AcRF) Tier1, Ministry of Education, Singapore under grant number RG166/15.

978-3-9819263-0-9/DATE18/©2018 EDAA

II. Related Work

Memory protection from replay, spoofing and splicing attacks can be achieved through runtime integrity verification [2]. An integrity tree is vital to such mechanisms due to the limited on-chip storage. Authentication methods like hash function, MAC, Block-level AREA are used to realize the integrity trees [3]. The root value of the tree is stored on chip and is assumed to be safe and resistant to tampering. Existing integrity trees include HASH trees, PAT trees and TEC tree [3]. A number of commercial CPUs like IBM Secure Blue [4], and XOM architecture [5] processor also make use of such integrity trees.

Intel's SGX incorporates a memory encryption engine [6] that uses an integrity tree with a tweaked version of AES to provide authentication. The PoinsonIvy processor uses integrity trees with speculative execution of instructions [7]. AEGIS, which provides protection against both software and physical attacks employs a balanced hash tree structure for its memory protection [8]. TEC-tree and Merkle trees, both use fully balanced integrity trees with different authentication techniques [3]. Merkle trees process data in batches to save on-chip storage and the authentication procedure of TEC-tree is parallelizable as each node is unique [9].

All of the above-mentioned schemes make use of a fully balanced memory integrity tree structure and they do not take advantage of the memory access patterns to reduce the verification overheads. As such, the memory protection schemes are often computationally intensive as they account for excessive memory accesses [10]. In [11], a skewed memory integrity tree that is based on the frequency of memory accesses is introduced for run-time verification but the tree is constructed off-line and remains static throughout the operation. Hence, this method is only applicable to scenarios where the memory access patterns of the applications are known beforehand, which is highly unlikely in practical applications.

A. Main Contribution of this Work

The main contribution of this paper is a framework that dynamically restructure the memory integrity tree based on the runtime memory access patterns. In particular, our approach places data elements with the same memory access frequency together in a single node/set, leading to efficient tree storage. As the memory access patterns change, nodes in the tree are repositioned based on their frequency of access. Nodes with higher frequency are placed closer to the root. This adaptive approach for creating a skewed tree leads to significant overhead reduction compared to the balanced tree approaches in [3]. In addition, unlike [11] the proposed method do not assume that the memory access patterns are known a-priori. We demonstrate the effectiveness of the proposed approach for runtime verification on the Altera NIOS II processor with an external DRAM. The AES encryption has been implemented using custom instructions to further accelerate the runtime verification process.

III. Threat Model

Similar to previous works, our threat model assumes that anything stored on chip is secure and cannot be tampered with. We are concerned with bus attacks that tamper the memory and/or processor bus and thus are able to observe and inject manipulated data. Side channel and leakage attacks are beyond the cope of this work. Typical bus attacks are [3] :

- Spoofing: Attacker exchanges a memory block with a tampered one.
- Splicing: Attacker replaces the memory block at address A with a memory block at address B where $A \neq B$.
- 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.

IV. Proposed Integrity Tree Representation

The proposed memory integrity tree structure uses nodes as $sets$ that store a group of memory blocks rather than individual memory block (as adopted in conventional approaches [9]). Each node (or set) will store all the memory locations that have the same frequency of access. Therefore, unlike the previous methods, the number of nodes in the proposed memory integrity tree is equal to the number of memory access frequencies (instead of number of memory locations). Clearly, as the memory access patterns change during runtime, the tree must be dynamically restructured.

The memory elements are placed as the leaf nodes on the lowest level of any tree branch, in the form of data chunks (DC). Let n_i denote the i^{th} node of the tree. p_i, s_i, LR_i, f_i denote the parent, sibling, side, frequency of n_i respectively. p_i stores the node number of the parent and s_i stores the node number of the sibling of n_i. LR_i denotes whether the node is a left child or a right child to its parent. f_i denotes the frequency of access for the elements stored in n_i. NE_i denotes the number of data elements in n_i. The weight of n_i is denoted by $w_i = NE_i * f_i$. Fig. 1 shows how the contents of the nodes in the proposed memory integrity tree are organized, and the format of the data and counter chunks. The (Node|Weight) concatenation in the counter and data chunks serves as the nonce (number use only once) as is unique to each node. The remaining nodes are formed by recursively combining the weights and are known as counter chunks (CC). The DC's and CC's are all encrypted using AES algorithm with a symmetric key cipher operating in Electronic Code Book Mode. We will explain the tree structure for the following data elements $\{0, 1, 1, 5, 6, 8, 9\}$ of the memory integrity tree as shown in Fig. 1.

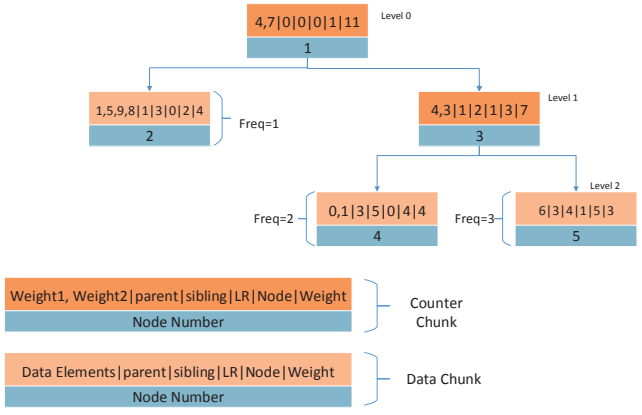


Fig. 1: Example of proposed dynamic integrity tree structure

For n_4, the attributes are: $p_4 = 3$, $s_4 = 5$, $LR_4 = 0$, $f_4 = 2$, $NE_4 = 2$, $w_4 = NE_4 * f_4 = 4$. Since the tree structure is different from a fully balanced tree where the location of the parent of a child node n_i can be easily computed by $\lfloor \frac{i}{2} \rfloor$, for the proposed integrity tree we need to explicitly store the parent and sibling node numbers for each node. CC stores the weight of its two children instead of data symbols. Its own weight is the sum of the weights of its children. In Fig. 1, n_3 is a CC. It stores the weight of its children (i.e. n_4, and n_5) and its own weight, w_3, is 7 (i.e $3 + 4$). The same goes for n_1 which is also a CC. Note that n_5 with $f_5 = 3$ is placed at level 2 as compared to n_2 with $f_2 = 1$, which is placed at level 1. This is because w_2 is greater than or equal to w_4 and w_5.

The proposed tree structure incurs an additional memory overhead in the form of a look-up-table (LUT) on chip as shown in Table I. This is required to identify where each element is stored. Unlike the fully balanced tree structure, where each element was stored in the same order as it occurs in the application following a natural sequence, this tree stores elements based on the frequency of accesses. The LUT stores the node number along with an element number (to indicate which element it belongs to in the particular set). If the number of nodes of the tree is n, the memory overhead of the LUT is $(3 * \log_2 n)$.

TABLE I: Look Up table

Index	Node No.	Element No.
1 {*0*}	4	1
2 {*1*}	4	2
3 {*1*}	2	1
4 {*5*}	2	2
5 {*6*}	5	1
6 {*8*}	2	3
7 {*9*}	2	4

V. Proposed Framework for Dynamic Tree Construction

The proposed framework for dynamic tree construction is adapted from [12] and is shown in Algorithm 5. The tree is initialized with a single node consisting of all the data symbols as its members and frequency 0. Whenever any data element is requested by the processor, its frequency increases by 1 which triggers the process of tree update. During the course of execution, whenever there is a memory request from the processor, verification steps are performed to ensure that the data read from the memory has not been tampered with. For a read request, we match the address with a LUT, as shown in Table I, and attain the node number corresponding to its position in the tree. This is followed by verifying all the nodes on the path in the tree to the root. Once verified, data is passed to the processor. Next, set migration is undertaken where the symbol is migrated to a new node set that matches its new frequency. A similar procedure is adopted for a write request. Once migration is complete, we check the tree for imbalance based on some criterion. If there is an imbalance in any part of the tree, we move the nodes to rebalance the tree [12].

An important question is when should the tree be rebalanced. The criterion used in [12] is:

Algorithm 1: Dynamic Skewed Tree

```
begin
    Initialize tree with single node set having all data elements
    while (1) do
        if read_request(addr) then
            n←LUT(addr)
            ReadNCheck(n)
            Set_Migration(n)
            Update LUT
        end
        if write_request(addr) then
            n←LUT(addr)
            WriteNUpdate(n)
            Set_Migration(n)
            Update LUT
        end
        Rebalance_flag ← rebalance_check(n);
        if Rebalance_flag then
            Shift_up(n);
        end
    end
end
```

Algorithm 2: Dynamic Tree

```
begin
    while (1) do
        if read_request(addr) then
            if addrv̇ulnerable Ī then
                ReadNCheck(addr)
            end
            read(addr)  Check criterion for add(addr) if true
            then
                addrv̇ulnerable Ī add_to_tree(addr)
            end
        end
        if write_request(addr) then
            n ← LUT(addr)
            WriteNUpdate(n)
            Set_Migration(n)
            Update LUT
        end
        Rebalance_flag ← rebalance_check(n);
        if Rebalance_flag then
            Shift_up(n);
        end
    end
end
```

if $(w_i > (w_{s_i} + 1) \wedge (\ w_i > w_{s_{p_i}})\ )$ **then** rebalance

The above criterion states that if the weight 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 procedures used in Algorithm 5 are described below in greater detail. For a read, $ReadNCheck$ is performed and for a write, $WriteNUpdate$ is performed.

- *ReadNCheck*: The DC is loaded from memory and decrypted while the corresponding parent CC is fetched to match the weights for verification. If the two values match, we proceed to the next level. The remaining branch authentication until the root is undertaken and in case of any weight mismatch, a warning is sent to the

processor else the data is passed to the processor. Once passed, the data is now migrated to a new set with higher frequency and the tree is checked for any imbalance.

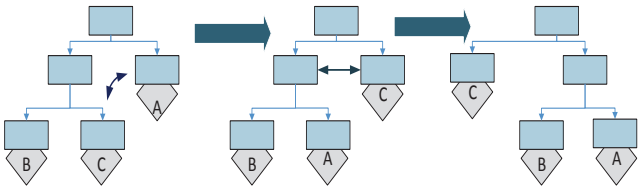
- *WriteNUpate*: The DC to be updated is first authenticated using the *ReadNCheck* process following which the DCs data symbol is updated. All loaded chunks are authenticated before being updated. Once updating is complete the new data is migrated to a new set with higher frequency. The tree is then rebalanced if necessary.

The dynamic tree algorithm performs two other operations to restructure itself: Set migration, and Rebalancing.

- Set Migration: Migration happens when a data element moves from one set to another. Essentially, our approach aims to group elements with the same frequency. Thus, whenever an element is accessed, it has to be migrated to a new set with a higher frequency (current frequency + 1). If such a set does not yet exist, a new set is created with that element being its first member. The algorithm is described in Algorithm 3.
- Rebalancing: The nodes are relocated whenever a region in a tree becomes imbalanced. If a certain node has higher weight than its neighbouring nodes then it must be relocated to a higher level in the tree. The algorithm is described in Algorithm 4.

Rebalancing shifts the nodes with higher probability of occurrence closer to the root. This is achieved in two steps as illustrated in Fig. 2:

- Exchanging sub tree with its uncle
- Rotation



(a) (b) (c)

Fig. 2: Shifting up sub tree C: a) Exchanging C with uncle A, (b) rotation, (c) final state

Each time an element is accessed, it's frequency increments and this triggers the migration process in Algorithm 2. If the rebalancing criterion is satisfied after migration, the shift up operation is performed. Note that in Fig. 2, A, B, and C are all sub trees and not single nodes. The operation of migration for an element has a total time complexity of O(lg k) in the worst case, where k is the cardinality of the set. The update procedure can also be done in logarithmic time as discussed in [12].

A. Implementation Example

We consider the example in Fig. 1 with the following data symbols: {0,1,1,5,6,8,9}. Suppose that the elements are accessed in the following order: {0,1,1,5,6,8,9,6,0,1,6,1,6}. Note that all the 7 elements are accessed once in the beginning. Thus we begin our tree design with a single set consisting of all the elements and frequency 1 as can be seen in n_2 on

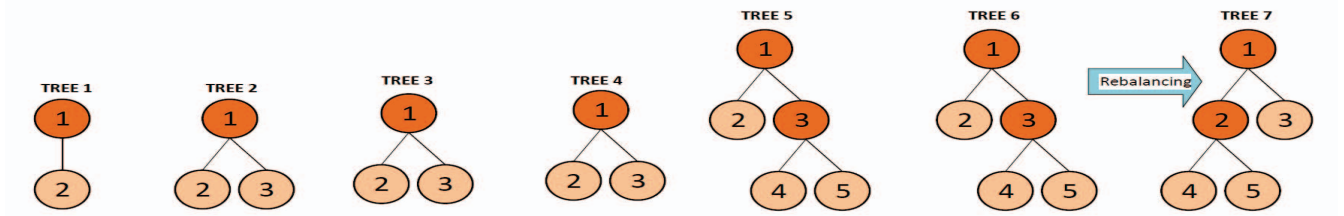
Algorithm 3: Set Migration $\{a$: *data element*$\}$

```
begin
    Q, P : pointers to Nodes
    P ← find (a)
    Q ← find (P's frequency +1)
    if Q ≠ ∅ then
        remove a from P's set
        P's weight = P's weight - P's frequency
        add a to Q's set
        Q's weight = Q's weight + Q's frequency
        ShiftUp (Q)
        if P =∅ then
            remove P from the tree
        end
        else
            ShiftUp(P's Sibling)
        end
    end
    else
        create a new node T
        T's right child is a new node N
        T's left child is P
        N's set =a
        N's weight =P's frequency +1
        N's frequency = P's frequency +1
        replace the old P in the tree by T
        remove a from P's set
        P's weight = P's weight -P's frequency
        if P =∅ then
            remove P from the tree
        end
        else
            ShiftUp(P's Sibling)
        end
        ShiftUp (T)
    end
end
```

Algorithm 4: ShiftUp $\{T$: *pointer to 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
```

Tree 1 in Fig. 3. It has a weight of 7, with 7 elements each having occurred once. From the access log we note the next element being called is {6}. As currently there is no set with frequency 2 we will need to create a new set. This is created in Tree 2. As {0} and {1} are called, they are migrated from set of frequency 1 to 2. This results in the tree structure shown as Tree 4 consisting of 3 nodes, n_3 with 3 elements and weight 6 and n_2 with 4 elements and weight 4. Next, {6} is called again. We will need to create another new node, this time



	Tree 1		Tree 2			Tree 3			Tree 4			Tree 5					Tree 6					Tree 7				
n_i	1	2	1	2	3	1	2	3	1	2	3	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Data Elements	0,7	0,1,1, 5,6,9,8	6,2	0,1,1, 5,9,8	6	5,4	1,1,5, 9,8	0,6	4,6	1,5, 9,8	1,0, 6	4,7	1,5, 9,8	4,3	0,1	6	4,8	1,5, 9,9	2,6	0	1,6	6,6	2,4	1,6	0	1,5, 9,8
p_i	0	1	0	1	1	0	1	1	0	1	1	0	1	1	3	3	0	1	1	3	3	0	1	1	3	2
s_i	0	0	0	3	2	0	3	2	0	3	2	0	3	2	5	4	0	3	2	5	4	0	3	2	5	4
LR_i	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	1	0	0	1	0	1	0	0	1	0	1
w_i	7	7	8	6	2	9	5	4	10	4	6	11	4	7	4	3	12	4	8	2	6	12	6	6	2	4
NE_i	-	7	-	7	1	-	5	2	-	4	3	-	4	-	2	1	-	4	-	1	2	-	-	2	1	4
f_i	-	1	-	1	2	-	1	2	-	1	2	-	1	-	2	3	-	1	-	2	3	-	-	3	2	1

Fig. 3: Tree Migration and Rebalancing Steps

Algorithm 5: Add to tree

```
begin
    Q, P, R : (pointers to Nodes)
    R ← New (sub tree)
    P ← find last added (sub_tree)
    if P.sibling == 0 then
        P.sibling ← R
    else
        Q ← New parent node
        Q.child ← P
        Q.child ← R
    end
end
```

with frequency 3. An additional step needs to be performed i.e. we need to build a new counter chunk and the new node of frequency 3 will be one of its children as can be seen on Tree 5. Next $\{1\}$ is called again. It will now migrate to n_5. The tree at this point can be seen as Tree 6 in Fig. 3. It now needs rebalancing as n_5 with weight 6 satisfies the shift up criterion. After performing the tree shift up steps as shown in Fig. 2, Tree 7 is obtained. In the final state, it can be seen that the element $\{6\}$ and $\{1\}$ which are being called the maximum number of times are placed closest to the root to reduce the time required for runtime verification.

B. Custom Instructions

The NIOS II processor offers the capability of extending the basic instruction set using custom instructions which are realized as hardware accelerators that augment the ALU. In order to reduce the latency of encryption/decryption during memory integrity verification, we have implemented the 128-bit mix-column AES algorithm as custom instructions. 32-bit inputs are passed in a sequential manner to form 128 bit AES inputs. AES encryption can be broken into the following major functions: 1) Byte Substitution, 2) Skip Row, 3) Mix Column, 4) Add Round Key, 5) Key Expansion, and 6) Inverse Key Expansion. The Altera tool chain created macros for all the above routines.

C. Security Analysis

Spoofing attacks are detected by making use of the block AREA scheme [13]. 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 makes sure that any change on the data will be reflected after decryption as the nonce obtained would have changed.

Splicing attacks are detected during the first stage of verification. As the node number bits are stored in our nonce, any mismatch associated with the node address used to fetch the chunk and the bits extracted from the chunk would raise an alarm.

Replay attacks are prevented due 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 data block. If the entire tree is replayed, the last verification step of matching the root node with the on chip counter will trigger the alarm.

VI. Experimental Results

We evaluate the performance benefits of the proposed method using the Altera DE2 board and Qsys tool in Altera Quartus II, v12.1. The system consists of NIOS II processor operating at 50MHz, On-Chip Memory, JTAG-UART for connection between the system and the board, SDRAM Controller for using the SDRAM, Clock Series for DE-series Board Peripherals, and performance counter unit for measuring the performance statistics such as time elapsed and number of clock cycles.

In our experiments, we ran six applications from the CH-Stone and SNU Real Time testbenches with data set size varing from 16-64. The applications are tested: 1) using a balanced TEC tree), and 2) static skewed tree [11], 3) dynamic skewed tree. The amount of performance gain achieved while running the application will directly depend on the decrease in the number of levels accessed for runtime verification. Fig. 4(a) shows number of tree levels that are accessed for the various methods considered. It can be observed that the proposed dynamic memory integrity tree results in the least number of levels accessed. On average the number of levels reduced by 35% compared to a balanced tree and 25% compared to the static skewed tree. The performance overhead of the proposed

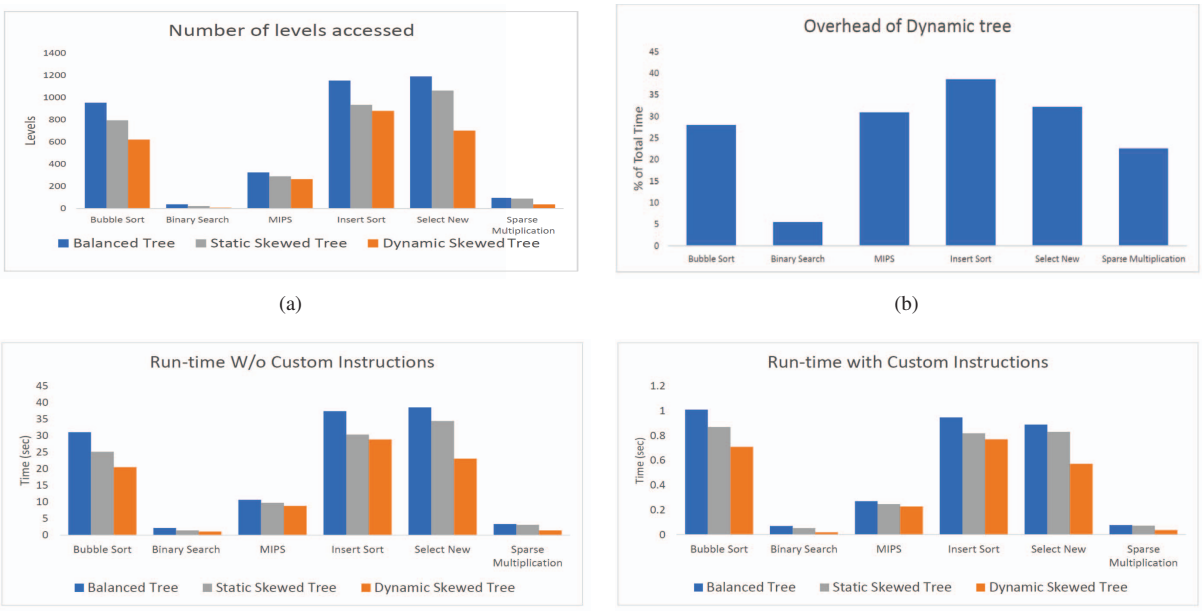


Fig. 4: Performance Evaluation: (a) Number of tree levels accessed (b) Time Overhead for running Dynamic Tree (c) Comparison without Custom Instructions (d) Comparison with Custom Instructions

method for constructing and maintaining the dynamic tree is shown in Fig. 4(b). On average the overhead constitutes to about 30% of the total overall runtime.

Next, we measured the run time for all the six applications on the different methods. The measurements were done both with and without custom instructions. It can be observed that despite the overheads of using the dynamic integrity tree, we attained maximum runtime advantage compared to the existing approaches. In particular, we observe on average an improvement of 30% over a full balanced tree and an average improvement of 20% over the static skewed tree, for cases with and without custom instructions. The results are shown in Fig. 4(c) and Fig. 4(d). The amount of gain varies with each application and its memory usage pattern. Applications using data sets with larger variance in the memory access frequencies are expected to result in larger performance benefits. We should note here that the run time has reduced considerably using custom instructions, which is expected as AES algorithm takes the bulk amount of time.

VII. Conclusion

The paper proposes an algorithm to construct a dynamic skewed memory integrity tree. The tree is dynamically restructured based on runtime memory access patterns. Data symbols that have the same frequency are placed together in a single node on the tree. Nodes are relocated at runtime to make sure that the nodes with higher frequency are placed closer to the root. This reduces the time required for runtime verification of the data accessed from the external memory. Experimental results show improvement in runtime of 30% compared to the conventional balanced tree approach.

References

[1] Oksana Cherednichenko, AA Baranov, and TI Morozova. Side-channel attack. 2013.

[2] Ross Anderson. *Security engineering*. John Wiley & Sons, 2008.

[3] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B Lee, Nachiketh Potlapally, and Lionel Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. In *Transactions on Computational Science IV*. Springer, 2009.

[4] Ronald Mraz. Secure blue: an architecture for a scalable, reliable high volume ssl internet server. In *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual*, pages 391–398. IEEE, 2001.

[5] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[6] Shay Gueron. A memory encryption engine suitable for general purpose processors.

[7] Tamara Silbergleit Lehman et al. PoisonIvy: Safe speculation for secure memory. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, oct 2016.

[8] G Edward Suh et al. Aegis: A single-chip secure processor. *Information Security Technical Report*, 10, 2005.

[9] Reouven 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.

[10] Siddhartha Chhabra and Yan Solihin. Green secure processors: towards power-efficient secure processor design. In *Transactions on computational science X*. Springer, 2010.

[11] Saru Vig, Tan Yng Tzer, Guiyuan Jiang, and Siew-Kei Lam. Customizing skewed trees for fast memory integrity verification in embedded systems. In *VLSI (ISVLSI), 2017 IEEE Computer Society Annual Symposium on*, pages 213–218. IEEE, 2017.

[12] Steven Pigeon and Yoshua Bengio. A memory-efficient adaptive huffman coding algorithm for very large sets of symbols. In *Data Compression Conference, 1998. DCC'98. Proceedings*, page 568. IEEE, 1998.

[13] Claude E Shannon. A mathematical theory of cryptography. *Memorandum MM*, 45, 1945.