6-2022

Memory Protection with Dynamic Authentication Trees

Matthew T. Millar
Rochester Institute of Technology

Marcin Łukowiak
Rochester Institute of Technology

Stanisław Radziszowski
Rochester Institute of Technology

Recommended Citation

M. Millar, M. Łukowiak and S. Radziszowski, "Memory Protection with Dynamic Authentication Trees," 2022 29th International Conference on Mixed Design of Integrated Circuits and System (MIXDES), Wrocław, Poland, 2022, pp. 202-207, doi: 10.23919/MIXDES55591.2022.9838004.

Memory Protection with Dynamic Authentication Trees

Matthew Millar
Department of Computer Engineering
Rochester Institute of Technology
mxm1898@rit.edu

Marcin Łukowiak
Department of Computer Engineering
Rochester Institute of Technology
mxleec@rit.edu

Stanisław Radziszowski
Department of Computer Science
Rochester Institute of Technology
spr@cs.rit.edu

***Abstract*—As embedded devices increase in use and handle more critical information and functionalities, the importance of security grows even greater. Defense against bus attacks such as spoofing, splicing, and replay attacks is of particular concern. Traditional memory authentication techniques, such as hashes and message authentication codes, require significant amounts of on-chip memory and introduce a large performance impact when protecting off-chip memory during run-time. Balanced authentication trees such as the well-known Merkle tree or TEC-Tree can be used to reduce this cost. This work proposes a new method of dynamic authentication trees, which updates a tree structure based on a processor's memory access pattern. An HDL model for use in an FPGA has been developed as a transparent and highly customizable AXI-4 memory controller. The performance of our tree design is comparable to that of the TEC-Tree in several memory access patterns. Speedup over the TEC-Tree is possible to achieve when applied in scenarios that frequently access previously processed data.**

***Index Terms*—embedded security, memory authentication, FPGA**

I. Introduction

Embedded devices that process sensitive data and provide essential services have become increasingly common as modern digital infrastructures have grown at a rapid pace [1]. This raises security concerns as there are more reasons than ever for a malicious party to try to exploit these systems. In an attempt to provide security, encryption methods are oftentimes applied to any sensitive data to provide confidentiality [2]. However, data confidentiality itself is not enough to fully protect a system from an attacker. For example, erroneous data may be injected into the system in an attempt to disrupt the normal functionality of the device. In order to protect against such attacks, it is necessary to verify that any data the device processes is provided by the expected source. In addition, there needs to be a method of ensuring that the data has not been tampered with. Methods of authentication are then used to confirm the integrity of data processed in the system. Existing authentication methods, such as hashes and message authentication codes (MACs), are able to provide the intended protection; however, some of these methods are costly in terms of the device resources and performance overhead required to implement them.

In this paper we present a new method of dynamic authentication trees, which update a tree structure based on a processor's memory access patterns. An AXI-4 based framework is developed as a transparent and highly customizable memory controller. This design is then synthesized onto an FPGA and verified. While this design is motivated by vulnerabilities in embedded systems, the method presented is scalable and may be applied to any computing system. Different configuration options are provided allowing for the design to be used in applications with different constraints and requirements.

The remainder of this paper is organized as follows. Section II details background information on the threat model and current authentication techniques. Section III outlines the proposed design. Section IV analyses the design and presents the results of the implementation. Section V provides overall conclusions.

II. Background

A. Threat Model

In many applications, the physical security of the computing device is either too costly or too difficult to properly achieve. This is especially true of applications that require the use of small-scale embedded systems. In our threat model, it is assumed that the CPU in such a system is secure and trusted as presented in Figure 1. Additionally, it is expected that the OS running on the processor is trusted and that the on-chip caches cannot be monitored or tampered with.

Attacks on memory are performed when an adversary has the capability to read or modify any data traveling between the processor and off-chip memory. All data and addresses sent on this memory bus are exposed to examination and modification. Of particular concern are hardware man-in-the-middle (MITM) attacks such as spoofing, splicing, and replay attacks [3]. In the simplest of attacks, an adversary may probe the memory bus, reading and writing data as it is transmitted to and from RAM.

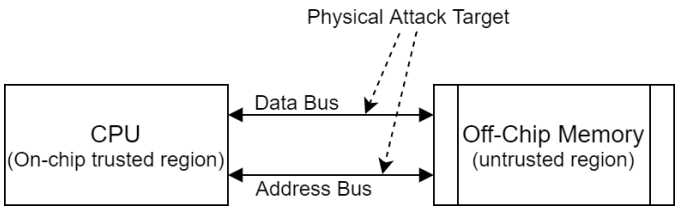

Figure 1. Region of Protection

B. Authentication

Protection against the previously mentioned MITM attacks requires the additional step of data authentication as well as encryption. A summary of a few existing authentication techniques is presented in Table I.

Table I
AUTHENTICATION METHODS

	Hashes	MACs	AREA	Auth Trees
Performance Overhead	High	High	High	Medium
Off-chip Overhead	None	High	Medium	High
On-chip Overhead	High	High	None	Low
Provides Encryption	No	No	Yes	Varies

Hashes. One of the simplest forms of data authentication can be performed using cryptographic hashes. A hash function creates a unique fixed-size output given a unique variable-sized input [4]. A block of data can be hashed, and that hash can be stored securely on-chip in order to prevent attackers from accessing it. Every time data is read, a newly computed hash of the data can then be compared with the hash stored on-chip.

Message Authentication Codes. Message Authentication Codes are an authentication technique that utilizes a keyed hash function. This is very similar to the previous technique, with the difference that only a secret key needs to be stored securely from the attacker [5].

Block-level AREA Authentication. Using the diffusion proprieties of block-level encryption [6], the block-level Added Redundancy Explicit Authentication (AREA) scheme is able to achieve encryption and authentication [7]. In the AREA technique, a unique nonce is appended to the end of a plaintext block, and the entire block is encrypted. On decryption, this added nonce is checked in order to determine if the data block has been corrupted. Traditionally, in implementations of a block-level AREA authentication technique, the nonce and the secret key must be stored securely while the ciphertext is visible to an attacker. An encryption mode that has an error propagation property, such as the AES in the cipher block chaining (CBC) mode, must be used.

Authentication Trees. Authentication trees are used as an attempt to limit authentication overhead compared to other techniques. The fundamental design of this method is a tree structure that stores additional encrypted metadata [8].

TEC-Trees. A TEC-Tree is a balanced authentication tree that protects data nodes using the Block-level AREA method. To ensure data integrity, each node contains a nonce that is verified after decryption. These nonces are generated using a secure on-chip counter [9].

Dynamic Authentication Trees. Most authentication tree methods rely on static balanced tree structures. A balanced tree approach may result in excessive overhead for access patterns that access the same address space frequently. A Dynamically Skewed Authentication Tree attempts to increase the performance of traditional authentication tree methods by reorganizing the structure of the tree at runtime. Shifting frequently accessed nodes to higher levels allows for less tree traversal time by reducing the number of verification computations and intermediate node accesses [10].

III. OUR DESIGN

A. Authentication

Modified block-level added redundancy explicit authentication (block-level AREA) scheme is employed in our ordered Dynamic Authentication Tree (DAT) method. Utilizing this approach has the benefit of providing encryption inherently with the authentication operations [7].

B. Tree Nodes

The leaf nodes of the authentication tree structure contain the data blocks that are directly protected by the tree. These nodes are referred to as "data nodes," and are separated into two parts: the data that is being protected and the nonce. The nonce contains the tree metadata required for tree traversal and a count of the number of times the node has been accessed. Each data node protects a block of data that can be any length specified by the application. Intermediate tree nodes contain the same metadata used for tree traversal, as well as the counts of each of their child nodes. These intermediate nodes are referred to as counter nodes. Bottom part of Figure 2 shows the general structure of a data node and a counter node as used by our ordered dynamic authentication tree (DAT).

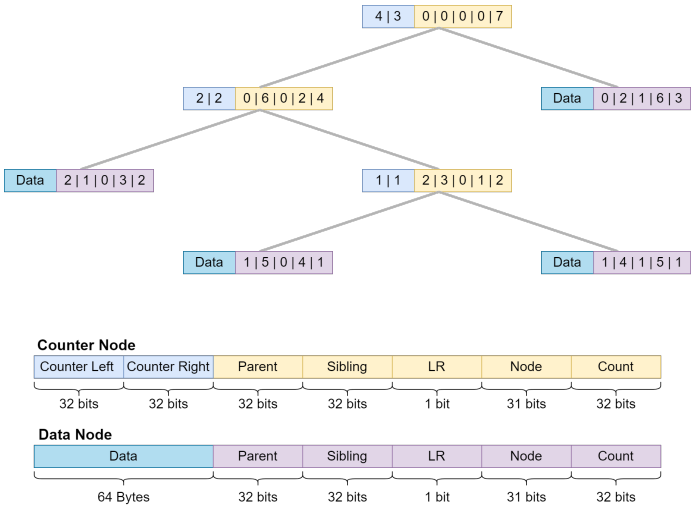

Figure 2. Ordered Dynamic Authentication Tree

C. Tree Structure

The structure of the tree depends on the weighted frequency of accesses to each data node. When a memory access is performed, the entire data block including the nonce, is first read and decrypted. The corresponding parent node is then also read and decrypted. This parent node holds the access count of its children for comparison to apply the block-level AREA scheme and provide authentication. This authentication process is repeated recursively until the final root of the tree is reached. Secure on-chip storage is used to store a master counter that is incremented on every write operation and used

to generate new nonces. For authentication purposes, the root counter node's count is compared to this secure root counter. If at any point, the authentication fails due to a counter mismatch, the processor can be alerted to an authentication error. Otherwise, the requested data operation is performed as usual. To prevent additional unnecessary performance overhead, data access frequencies are only updated on a write operation. In a write operation, the data node's frequency count itself must first be updated. As the tree is traversed upwards, each parent node's stored left and right counts are updated accordingly to their child nodes' weights. In addition, the parent's frequency access count is incremented. An example of a tree structure with additional metadata information is presented in Figure 2.

On a read operation for every level higher a data node is stored, it requires one less counter node to be read and compared before it is fully authenticated. This differs from other authentication tree methods which rely upon a balanced tree that is agnostic to the number of times a data node has been accessed. Given most memory access patterns, the more often a data block is accessed, it is more likely that the node will be accessed again in the future. Thus weighting data nodes and restructuring the tree as such should provide a substantial improvement in performance for data blocks that are accessed often.

D. Dynamic Restructuring

As memory blocks are accessed, the tree structure is dynamically updated to relocate memory blocks with a higher frequency of access closer to the tree's root. To ensure that authentication checks function properly, the count of each node is incremented, and its parent node's left or right counter is similarly updated as each node is accessed. When a node is accessed, it is accessed along with its parent, sibling, and uncle in order to perform the updates and navigate the tree. An uncle node refers to a parent's sibling node. The weight of the uncle node is compared to the new weight of the current node. The tree is flagged for rebalancing if the current node's new weight is greater than the uncle node's weight. There are two approaches to rebalancing. The first method permits the tree's leaf nodes to be rearranged, but their order is not preserved. The second approach adds the restriction of requiring the leaf node order to remain constant. The ordered tree design's rebalancing procedure consists of three different possible cases that are described below. The first rebalancing case is described by Algorithm 1.

Algorithm 1: Ordered Rebalancing Method #1

Data: Parent, Uncle, Sibling, Current

if *Current's LR* $\neq$ *Uncle's LR* **then**

```
// Shift Uncle down
Uncle's Parent ← Parent;
Uncle's Sibling ← Sibling;
// Rotate Parent
Parent's sibling ← Current;
Parent's LR ← ¬ Parent's LR;
// Update and Rotate Sibling
Sibling's sibling ← Uncle;
Sibling's LR ← ¬ Sibling's LR;
// Shift Current Node Up
Current's Parent ← Parent's Parent;
Current's Sibling ← Parent;
```

end

Algorithm 2: Ordered Rebalancing Method #2

Data: Parent, Uncle, Sibling, Current, Grand Uncle, Grand Parent

if *(Current's LR = Uncle's LR)* $\wedge$ *(Current's LR* $\neq$ *Grand Uncle's LR)* **then**

```
// Shift Parent Up with Current
   Node
Parent's Parent ← Grand Uncle's Parent;
Parent's Sibling ← Grand Uncle's Sibling;
// Update Uncle
Uncle's Sibling ← Current Node;
// Shift Grand Uncle Down
Grand Uncle's Parent ← Parent;
Grand Uncle's Sibling ← Sibling;
// Update Grand Parent
Grand Parent's sibling ← Parent;
// Shift Sibling Up with Current
   Node
Sibling's sibling ← Grand Uncle;
Sibling's LR ← ¬ Sibling's LR;
// Shift Current Node Up
Current's Parent ← Grand Parent;
Current's Sibling ← Uncle;
Current's LR ← ¬ Current's LR;
```

end

By shifting the parent node to the side and the uncle node down a level, the current node being accessed is shifted up a level. To perform these operations, the parent and sibling relationship stored in each node is able to be updated with the new case. It is necessary to read and decrypt the parent, uncle, sibling, and current node data from memory to access all of the information required to perform these operations. If it is not desired to retain the order of the data nodes, this rebalancing algorithm is the only one necessary to use. However, if the order of the nodes must remain the same, this case is only valid if the current node's left/right position is different than its uncle node's left/right position. If the first rebalancing method were to be used, the leaf node order would be modified, thus invalidating the ordered design. Algorithm 2 shows the second rebalancing method used for handling this additional scenario. The second rebalancing method requires the information from the grand uncle and grand parent nodes in addition to the information from the first method. When the current node is shifted upwards, the parent node is updated with the grand

Algorithm 3: Ordered Rebalancing Method #3

Data: Parent, Uncle, Sibling, Current, Grand Uncle, Grand Parent

if *(Current's LR = Uncle's LR)* $\wedge$ *(Current's LR = Grand Uncle's LR)* **then**

```
// Shift Grand Uncle Down
Grand Uncle's Parent ← Grand Parent;
Grand Uncle's Sibling ← Sibling;
// Update Grand Parent
Grand Parent's sibling ← Parent;
Grand Parent's LR ← ¬ Grand Parent's LR;
// Update Uncle
Uncle's Parent ← Grand Parent;
Uncle's Sibling ← Grand Uncle;
Uncle's LR ← ¬ Uncle's LR;
// Shift Parent Up
Parent's Parent ← Grand Parent's Parent;
Parent's Sibling ← Grand Parent;
```

end

uncle node rather than being shifted downwards. The current node would then be replaced as a child by the grand uncle. To keep the order of the leaf nodes, the left/right positions of the grand uncle and sibling are swapped. When the current node's left/right value, the uncle node's left/right value, and the grand uncle's left/right value are all equal, neither one of these methods is able to maintain leaf node order. To address this final case, a third rebalancing method, as shown in Algorithm 3, is introduced. Methods one and three are very similar, with the main difference being that the node operations are applied to a node one level higher than the current node. While the current node's information is not updated, the parent node is shifted upward, causing both the current and sibling nodes to move up a level. Given that the grand uncle must be accessed for both this method and method two, an additional qualifier is required when checking the rebalance condition to ensure that a great grand uncle exists and that the root has not been reached.

E. Encryption and Authentication Pipeline

Aiming to reduce design complexity, our implementation utilizes a modified version of the AXI-4 transparent memory encryption pipeline provided in [11]. In order to generate appropriate requests for new intermediate nodes in dynamic trees, tree metadata must first be read from memory. As a result, the pipeline must return the data read from memory to the component that generates requests. Figure 3 depicts the modified pipeline, which includes authentication components. The pipeline is divided into two paths: a read request pipeline and a write request pipeline. The Tree Request Generator component possesses a master state machine to determine which tree nodes must be accessed and updated. The root of the tree is checked to see if the data node has been accessed yet. The root is stored in BRAM, which is embedded in the programmable logic, and can be considered secure against an attacker with physical access. If the root has not yet been accessed, the tree's data must first be initialized. To correctly populate the initial tree data, a Tree Initializer component has been added.

IV. Analysis & Results

In this section, we compare the results of the proposed design's hardware implementation and two similar designs: the TEC-Tree and the Dynamically Skewed Tree. All three were integrated within the previously stated pipeline framework described by Figure 3.

A. Memory Overhead

The off-chip memory overhead is presented in Equation (1), where the variables l_p, n, and A are the data payload size in bits, the bit length of nonces, and the arity of the tree respectively. For the arity of 2, the relationship between off-chip memory costs for our method, TEC-Tree, and Dynamically Skewed Tree is identical apart from the variation in nonce sizes [9], [12].

$$O = \frac{l_p + n \times A}{l_p(A-1)} \tag{1}$$

The nonce sizes n for each method are shown below using Equations (2), (3), and (4). Where c is the bit length of the counters used for authentication, and D is the total number of data nodes used in the tree.

$$n_{TEC} = c + \log_2 N \tag{2}$$

$$n_{skewed} = c + 3 \times \log_2 N \tag{3}$$

$$n_{DAT} = c + 3 \times \log_2 N \tag{4}$$

The proposed method's off-chip memory cost is slightly increased relative to the TEC-Tree, but remains the same as the Dynamically Skewed Tree. This increase is due to additional stored metadata that is needed in order to traverse an unbalanced tree structure. The size of the counter value used for authentication and the root of the tree are customizable. Only a single counter per tree is required to be stored in on-chip memory. The on-chip memory cost is the same for all three methods.

B. Dynamic Tree Performance

A dynamic authentication tree requires additional computational overhead on each write transaction compared to the TEC-Tree as a recursive check for rebalancing needs to be applied. A counter node must be read for each level in addition to the information that the TEC-Tree requires. Generally, a dynamic tree rebalance does not happen frequently, especially with memory access patterns that focus on particular ranges of the whole memory address space. Due to the number of extra computations performed, it is expected that a memory access pattern that utilizes writes to memory more frequently

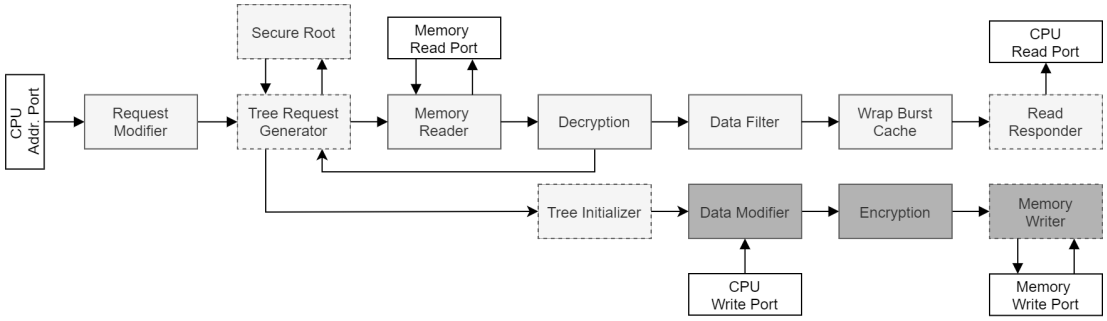

Figure 3. Modified Memory Encryption and Authentication Pipeline [11]

than reads may perform worse. Performance speedup occurs when a read or write transaction occurs on an address that has been shifted higher in the dynamic tree than it would be in the corresponding balanced tree structure. In order for a dynamic tree to perform better than the TEC-Tree design, this speedup must outweigh the slowdown caused by the additional tree calculations.

C. Implementation Results

The TEC-Tree, the Dynamically Skewed Tree, and our design were all implemented targeting the ZedBoard development kit with Xilinx Zynq xc7z020clg484-1 part, and Xilinx Vivado version 2020.1 tools. The TEC-Tree design was provided by [11]. The Dynamically Skewed Tree was manually developed and modeled in VHDL based on the framework outlined in [12]. The performance was evaluated with a clock frequency of 50 MHz. For programmable logic utilization comparisons, the total number of resources available for various Xilinx devices is displayed in Table II. Part xc7z2020 is the SoC used by the ZedBoard. Compared to some other FPGA parts available, the resources provided by the Zedboard are relatively small. The resulting utilization is presented in Table III.

Table II
REFERENCE RESOURCES AVAILABLE FOR A SAMPLE SET OF XILINX FPGAS

Part	LUTs	Flip Flops	DSPs	BRAM Tiles
xc7z2020	46,200	92,400	160	95
xczu9eg	274,080	548,160	2,520	912
xcvu9p	1,182,240	2,364,480	6,840	2,160
xcu250	1,341,000	2,749,000	11,508	1,766

As the simplest design, the TEC-Tree uses the fewest number of each component. Most of the differences in utilization can be accounted for by the added components for the dynamic tree request generation and initialization. While the TEC-Tree design can simply generate all necessary requests without reading data from memory, the other two designs require logic to both store read data and translate the read data to appropriate requests for tree traversal and modification. The Dynamically Skewed Tree design also requires fewer resources

Table III
AUTHENTICATION TREE IMPLEMENTATION RESULTS

	TEC-Tree	Dynamically Skewed Tree	Ordered DAT
No Encryption			
LUTs	3270	8394	11649
Flip Flops	2893	5651	7582
DSPs	0	15	21
BRAM Tile	1	1	1
Encryption			
LUTs	9018	11251	14506
Flip Flops	4122	6419	8350
DSPs	0	15	21
BRAM Tiles	1	1	1

to implement than our ordered DAT. This can be attributed to the additional tree structure and rebalancing complexity that comes with the added condition of maintaining terminal nodes' ordering. While there is only one reblancing algorithm used in the Dynamically Skewed Tree design, the ordered design requires three different algorithms to be implemented based on the current state of the tree to be restructured. However, the overall usage for the ordered DAT is still notably low if compared to any of the FPGA models listed in Table II.

D. Memory Controller Performance

All three designs were evaluated for correctness and performance using the Xilinx AXI Verification IP (AXI VIP). The verification IPs ensure signal behavior and timing are appropriate for the AXI-4 interface. They do not, however, ensure data written to and read from memory represent proper values. A wrapper testbench was written in SystemVerilog to both control the AXI VIP's and ensure the memory contained the correct data on a read and write operation. A master AXI VIP was used to simulate the processor sending initial read or write requests to the memory controller. A second AXI VIP was configured for slave responses in memory-mapped mode, allowing the IP to simulate a BRAM with appropriate timings. The designs for all three methods were evaluated for performance in various conditions with different memory access patterns. Each designs' customizations were standardized to make direct performance comparisons easier. The master interface of the pipeline was configured to accept a

data length of 32 bits, while the slave interface was configured for a data length of 64 bits. The size of the memory to be protected was configured for 256 MB and a data block size of 64 bytes for each design.

Because of the dynamic nature of the proposed tree designs, multiple scenarios were required to thoroughly investigate the various performances of the design. First, the designs were tested utilizing a fully random test suite. As each memory address is random, there is a randomly distributed spread of accesses throughout the entire tree. While this type of memory access pattern is very unrealistic in practical applications, this type of access pattern is the worst-case scenario for a dynamic tree which is important to note.

The second test applied to each authentication tree was designed to test the best-case scenario for a dynamic tree. For this test, only a single data node is accessed to allow the dynamic tree to restructure it to the top of the tree.

The third test was run to provide a realistic memory access pattern. In this test, contiguous memory was accessed in order to simulate reading and writing to a large array in memory. The previous test may provide an accurate representation of accessing a smaller array in memory, but it would require the array to remain within the size range of the tree's data block configuration. Figures 4 and 5 contain the timing results of each test.

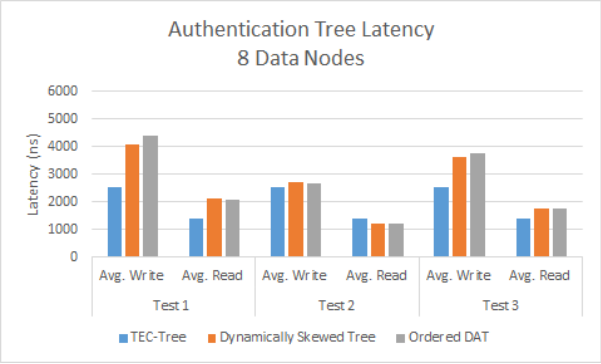

Figure 4. Summary of 8 Node Timing Results

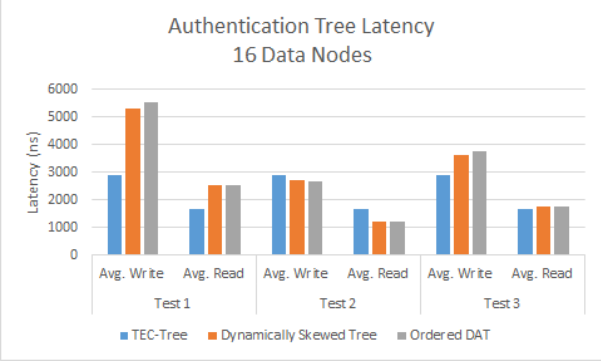

Figure 5. Summary of 16 Node Timing Results

Overall, the TEC-Tree outperforms the dynamic tree designs in scenarios where memory access patterns do not favor portions of memory over others. The dynamic tree implicitly contains higher overheads on write operations as there are more calculations that must be done in order to ensure that the tree is properly balanced. However, the read and write performance on nodes that are frequently accessed have the potential to increase the overall memory operation speed in certain scenarios.

V. Conclusions

The necessity of embedded security grows as devices with crucial functions become more common. In this paper we presented a new ordered Dynamic Authentication Tree (DAT) method, which updates the tree structure based on memory access patterns. As expected, initial performance tests with simple synthetic benchmarks show that our method is comparable to TEC-Tree in some access patterns and it has potential to outperform it when it is necessary to frequently access previously processed data. Our future work will focus on adding tree caches to avoid high cost computations and memory accesses for frequently accessed memory cells, development and adopting verity of additional benchmark applications, optimizing the design to allow for using higher frequency clock, and to allow the framework to work with trees that have arities larger than 2.

References

[1] S. Parameswaran and T. Wolf, "Embedded Systems Security - An Overview," in *Design Automation for Embedded Systems*, September 2008.

[2] M. Kurdziel, M. Lukowiak, and M. Sanfilippo, "Minimizing performance overhead in memory encryption," *Journal of Cryptographic Engineering*, vol. 3, 06 2013.

[3] L. Hars, "Security against memory replay attacks in computing systems," US Patent US201 414 340 294 20 140 724, 2016.

[4] C. Malinowski and R. Noble, "Hashing and data integrity: Reliability of hashing and granularity size reduction," *Digital Investigation*, vol. 4, no. 2, pp. 98–104, 2007.

[5] M. Rogobete and O. Tarabuta, "Hashing and Message Authentication Code Implementation. An Embedded Approach," *Scientific Bulletin "Mircea cel Batran" Naval Academy*, vol. 22, no. 2, pp. 296–304,296A, 2019.

[6] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[7] C. Fruhwirth, "New methods in hard disk encryption," *Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology*, 08 2005.

[8] O. Shwartz and Y. Birk, "Distributed Memory Integrity Trees," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 159–162, 2018.

[9] 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 *Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier and I. Verbauwhede, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 289–302.

[10] S. Vig, G. Jiang, and S. Lam, "Dynamic skewed tree for fast memory integrity verification," in *2018 Design, Automation Test in Europe Conference Exhibition*, 2018, pp. 642–647.

[11] M. Werner, T. Unterluggauer, R. Schilling, D. Schaffenrath, and S. Mangard, "Transparent memory encryption and authentication," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–6.

[12] S. Vig, R. Juneja, G. Jiang, S. Lam, and C. Ou, "Framework for Fast Memory Authentication Using Dynamically Skewed Integrity Tree," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 10, pp. 2331–2343, oct 2019.