5-2021

Flexible Memory Protection with Dynamic Authentication Trees

Matthew T. Millar
mxm1898@rit.edu

Flexible Memory Protection with Dynamic Authentication Trees

Matthew T. Millar

Flexible Memory Protection with Dynamic Authentication Trees

Matthew T. Millar

May 2021

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

RIT | Kate Gleason College of Engineering

Department of Computer Engineering

Flexible Memory Protection with Dynamic Authentication Trees

Matthew T. Millar

Committee Approval:

Dr. Marcin Łukowiak *Advisor* Date
Department of Computer Engineering

Dr. Stanisław Radziszowski Date
Department of Computer Science

Dr. Cory Merkel Date
Department of Computer Engineering

Acknowledgments

I would like to thank my brother Kevin for being a role model for me, both in life and in my work. Without his encouragement and insight, I would not have made it as far as I have today. I'd additionally like to thank my parents for pushing me to achieve my full potential and unconditionally providing me with support when I need it the most.

My childhood friends have supported me through the ups and downs of my academic career, and I would like to thank them wholeheartedly for always being there for me. Whether it was to have fun and relax after a long day or help me through difficult times, they were always there for me earnestly and without question. Specifically, I'd like to thank Robert Downey, Dallas Cook, and Henry Imgrund for their unwavering support. I'd also like to thank my friends from RIT and Rochester which I have been incredibly fortunate to get to know. I'm especially grateful for the encouragement and support from Jarrett Wehle, Cameron Hudson, Peyton Vick, Carson Clarke-Magrab, Alexandra Capodicasa, and Jason Blocklove.

Samantha Jacobson deserves special recognition for being my partner in all things academic. She constantly pushed me to produce my best work and helped me prepare for all academic challenges. She is by far the most passionate and hardest working person I have ever met. In addition, outside of school, I could always rely on her for help with anything. Words cannot express how grateful I am for her help.

Lastly, I'd like to thank my academic advisors, professors, and committee members. Not only for teaching me in the classroom but helping teach me to enjoy and pursue lifelong learning. Dr. Łukowiak especially has been an incredible teacher and advisor, who has assisted me significantly throughout my research.

Abstract

As computing appliances increase in use and handle more critical information and functionalities, the importance of security grows even greater. In cases where the device processes sensitive data or performs important functionality, an attacker may be able to read or manipulate it by accessing the data bus between the processor and memory itself. As it is impossible to provide physical protection to the piece of hardware in use, it is important to provide protection against revealing confidential information and securing the device's intended operation. Defense against bus attacks such as spoofing, splicing, and replay attacks are of particular concern.

Traditional memory authentication techniques, such as hashes and message authentication codes, are costly when protecting off-chip memory during run-time. Balanced authentication trees such as the well-known Merkle tree or TEC-Tree are widely used to reduce this cost. While authentication trees are less costly than conventional techniques it still remains expensive. This work proposes a new method of dynamically updating an authentication tree structure based on a processor's memory access pattern. Memory addresses that are more frequently accessed are dynamically shifted to a higher tree level to reduce the number of memory accesses required to authenticate that address. The block-level AREA technique is applied to allow for data confidentiality with no additional cost. An HDL design for use in an FPGA is provided as a transparent and highly customizable AXI-4 memory controller. The memory controller allows for data confidentiality and authentication for random-access memory with different speed or memory size constraints. The design was implemented on a Zynq 7000 system-on-chip using the processor to communicate with the hardware design. The performance of the dynamic tree design is comparable to the TEC-Tree in several memory access patterns. The TEC-Tree performs better than a dynamic design in particular applications; however, speedup over the TEC-Tree is possible to achieve when applied in scenarios that frequently accessed previously processed data.

Contents

List of Figures

List of Tables

Acronyms

AES Advanced Encryption Standard

APSoC All Programmable System on Chip

AREA Added Redundancy Explicit Authentication

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

BRAM Block Random Access Memory

CBC Cipher Blocker Chaining

CPU Central Processing Unit

CTS Ciphertext Stealing

DAT Dynamic Authentication Tree

DDR RAM Double Data Rate Random Access Memory

ECB Electronic Codebook

FPGA Field-Programmable Gate Array

IoT Internet of Things

IV Initialization Vector

LUT Look up table

MAC Message Authentication Code

MITM	Man-in-the-middle
NONCE	Number-used-once
PAT	Parallelizable Authentication Tree
RAM	Random Access Memory
SoC	System on a Chip
TEC-Tree	Tamper Evident Counter Tree
VIP	Verification Intellectual Property

Chapter 1

Introduction

1.1 Motivation

With the recent rapid growth of modern cyber infrastructure, devices that process sensitive data and perform essential services have become extremely widespread. Instead of using expensive general-purpose computers, a wide variety of applications employ low cost and low power devices to perform specific applications. Appliances that do not require extensive amounts of computing power routinely employ smaller processing systems to act as a control unit. Often referred to as embedded systems, these devices usually consist of a microprocessor, a limited amount of off-chip memory, and methods of interfacing with a broader system. Embedded systems are present in various critical applications such as smart homes, medical devices, and automobiles [2]. These devices are less susceptible to traditional software attacks as their applications are comparatively simple and more difficult to exploit [3]. Therefore, attacks on the hardware of an embedded device are oftentimes more effective. Physical security is often too costly and difficult to achieve for many applications, leaving the embedded system vulnerable to examination by an attacker.

Embedded devices are employed in creating a system called the "Internet of Things" (IoT). The IoT consists of numerous devices that share a network in order to quickly pass information about their application to other devices on the network [4].

IoT networks have evolved to the point of controlling devices that can have significant impact on a consumer's life. For example, smart homes often use a number of connected computing devices to provide various impactful services to the user. There is a growing concern about the security of these devices as they may handle private information and can perform life saving functionality [5]. A security breach into a single device in an IoT can cause serious danger to an individual utilizing the network. The devices that create an IoT network are often extremely susceptible to memory attacks and it is imperative that their data and functionality is secured effectively.

In addition to embedded systems, other forms of small scale computing devices are often used in similar applications that may additionally require protection. A system on a chip (SoC) often consists of one or multiple hard-core processors and programmable logic. The same kind of programmable logic is also found in a field-programmable gate array (FPGA). A processor can interface with programmable logic in order for software applications to offload specific functionality to the hardware design. While hard-core processors are physically implemented on a silicon chip, additional soft-core processors can be generated as needed using the programmable logic of SoCs. Soft-core processors are able to run software applications and interface with the other on-chip components but are limited in resources by the FPGA fabric. These All Programmable SoCs (APSoC) use Random Access Memory (RAM) to help store software and data. An attacker with physical access to a SoC device is able to interface with this memory bus, allowing them to read or modify any data sent through the bus [6]. This allows for memory attacks such as spoofing, slicing, and replay attacks to be performed. Despite this vulnerability to physical attacks, many modern SoC FPGAs leave the memory unprotected, allowing attackers to access sensitive data or modify device behavior.

1.2 Objective

The goal of this work was to provide a transparent and simple to integrate design that can be utilized by those aiming to secure memory. While the design proposed in this work is concerned with the protection of an SoC that has available programmable logic, the concepts introduced can be extended to various embedded system platforms. The focus of the protection is the integrity of data transferred from the processor of an SoC to off-chip memory. Particularly, providing data confidentiality and protection against man-in-the-middle (MITM) attacks are the main concerns of the design. An authentication tree design is used to detect any invalid data modification before the data is forwarded to the central processing unit (CPU).

A new method of data authentication was developed by extending concepts of existing authentication tree designs. The performance and data overhead of this method was compared to current memory protection techniques. A Zynq-7020 APSoC was utilized for design prototype implementation and evaluation. The design is intended to be implemented in programmable logic and used as a memory controller that intercepts and modifies communication between the processor and memory. An AXI-4 interface protocol was used for the memory controller in order to interface with the Zynq processor and memory components; however, this design can be implemented in a variety of unique applications. Customizability and scalability are built into the design. For example, the data sizes and the algorithms used for authentication and encryption are interchangeable themselves. The relatively low overhead and easy to customize nature of the design provides value for a range of different operations, depending on the constraints of the application.

Chapter 2

Background

2.1 Threat Model

The device to be protected is assumed to be in an environment where physical access is possible, and the memory bus is exposed to an attacker. Attacks on memory are performed as the adversary has the capability to read or modify any data traveling between the processor and RAM. For the purposes of this threat model, it is assumed the device being protected is an APSoC that can interface with the processor and RAM. The processor itself is considered secure, implying that the on-chip caches contained within the processor are unable to be attacked. Additionally, it is assumed that the OS running on the processor is trusted and the on-chip caches cannot be monitored by the attacker.

As securing embedded devices remains the area of focus, traditional software attacks while still a threat are not as pertinent to defend against. Most embedded systems perform specific and simple applications that are not as susceptible to many common software attacks. However, many software threats do exist and must be heavily considered to protect against [7], but the work in this thesis focuses on hardware attacks. Physical attacks on embedded systems are often possible as the processing system of these devices is generally operating in insecure environments where physical access is possible. The methods of attack that this work is most concerned with are hardware MITM attacks that focus on the external memory of an embedded device. Off-chip memory is especially vulnerable to numerous attacks because the bus between the processor and memory can be exposed. Data and address values sent to external memory are able to be probed and can be read by an attacker [8]. An encryption algorithm is necessary to provide data confidentiality and protect sensitive data transmitted on this memory bus from exposure. Many existing data ciphers allow for adequate protection of data, such as AES [9]; however, data confidentiality is not enough to adequately secure the system. Data authentication is additionally required in order to prevent attacks that modify the data in ways that allow an attacker to manipulate the functionality of the device.

2.1.1 Spoofing Attacks

The focus of spoofing attacks is not to reveal protected data, but instead to disrupt the intended functionality of the device. In the most basic spoofing attack, an attacker has access to the bus between the processor and memory then waits for a memory fetch instruction to be sent. The attacker does not decrypt any memory blocks but instead sends fake information to the processor [10]. This fake memory block disrupts the normal program flow and can cause instability in the system if not detected.

2.1.2 Splicing Attacks

Splicing attacks are another type of bus attack in which the attacker replaces a memory block with a modified block. Instead of replacing the memory block with an arbitrary fake block, the block is replaced with a block of memory taken from a different address [8]. The fake block is ensured to be properly encrypted and contain valid data. Additionally, this adds complexity to defend against as the authentication must take memory addresses into account.

2.1.3 Replay Attacks

A memory replay attack consists of an attacker reading and saving a valid memory block for injection into the system at a later time. The memory block is ensured to contain valid data and can be injected into the same memory address [11]. These attacks allow for previous states of the system to be accessed, prolonging functionality, or allowing now illegal functionality to be repeated. Replay attacks are generally the most costly attacks to protect against and are the primary area of focus for authentication trees.

2.2 Memory Authentication Techniques

2.2.1 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 [12]. 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. This comparison is used to ensure that the data integrity has not been compromised.

2.2.2 Message Authentication Codes

Message Authentication Codes (MACs) are an authentication technique that utilizes a hash function and a secret key in order to generate a small block of data that is used for comparison. This method differs from simple hashing as the MAC itself does not need to be hidden from an attacker. Both the message and secret key are used to generate the code, as such only the secret key must be hidden [13]. Commonly, the MAC is appended to the end of a block of data, as the attacker cannot reproduce it themselves. If the block is corrupted, the data receiver's newly calculated MAC wouldn't match the expected MAC stored with the data.

2.2.3 Block-Level AREA Authentication

The block-level AREA Authentication scheme uses the diffusion proprieties of block-level encryption to additionally provide authentication. The principle of this system relies on utilizing the Added Redundancy Explicit Authentication (AREA) technique [14]. 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. This method relies on the Shannon diffusion property to work properly. If even a single bit on the ciphertext is modified, then it is statistically improbable that the correct nonce will be retained on decryption. 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 infinite error propagation, such as AES in the cipher block chaining (CBC) mode, must be used. This error propagation is used to ensure that if an error does occur that it is propagated to the rest of the ciphertext.

2.3 Authentication Trees

Authentication trees are a method of providing memory encryption and authentication during run-time. They are used as an attempt to limit authentication overhead compared to traditional authentication techniques. Oftentimes, the term “Integrity Tree” is used in place of “Authentication Tree”. The fundamental design of this authentication method is a tree structure that stores additional encrypted metadata that must be decrypted in order to access further memory blocks [15]. Authentication trees are used for protection against third-party intervention such as replay attacks and memory tampering. The main advantage of this technique over alternative authentication methods is the lower performance overhead; however, the speed of each memory transaction is still significantly negatively affected. The size of data stored off-chip is also significantly increased depending on the method employed. Table 2.1 contains a summary of the main features and overheads for each authentication method.

Table 2.1: Authentication Methods

	Hashes	MACs	AREA	Authentication 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
Provides Authentication	Yes	Yes	Yes	Yes

2.3.1 Merkle Trees

A Merkle tree is an authentication tree structure that requires a cryptographic hash function to be computed for each internal node of the tree [16]. It is a balanced tree that uses equal-sized memory blocks as the leaves of the tree. A hash function is applied to the concatenation of the values of both children nodes of an intermediate node. The resulting hash is used as the value for that intermediate node. A hash is calculated recursively for each node until the root of the tree is formed. Every node's value is subsequently dependent on the values of its children nodes as illustrated by the structure in Figure 2.1.

The authentication calculations start at the leaves of the tree, which contains the data being protected. During a read transaction, the target memory block's hash is computed then compared to the expected value contained in the internal nodes. This process is then repeated until the root of the tree is reached. If the calculated hash differs from the expected hash at any point, an exception can be raised to indicate that data may have been tampered with. While the internal nodes are able to be safely

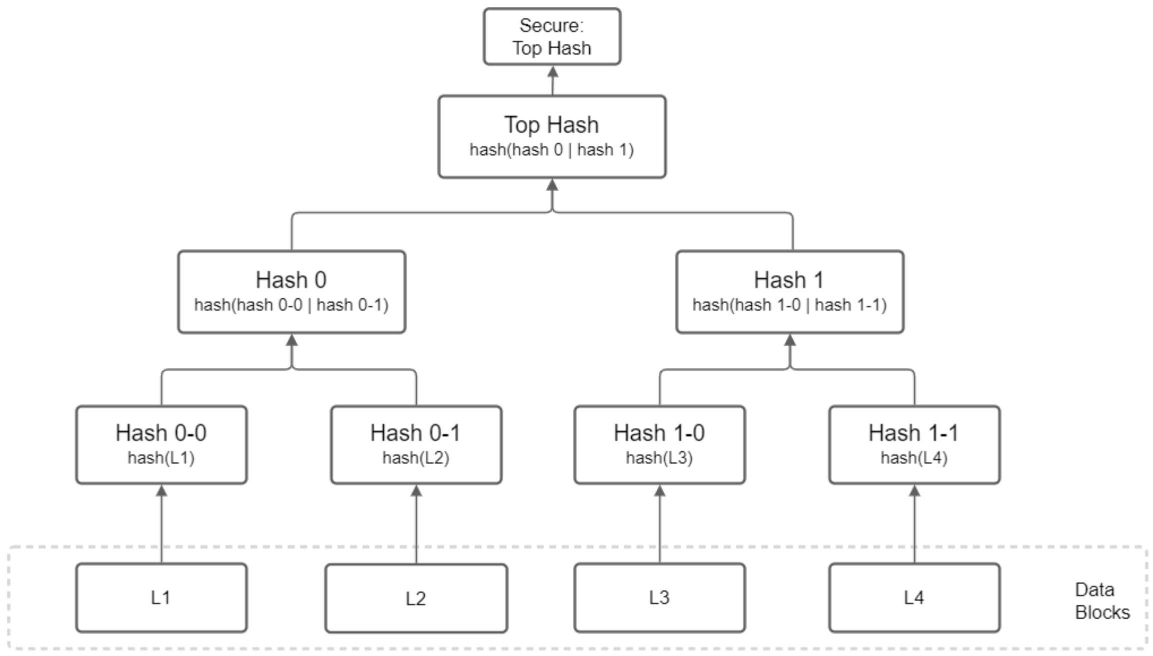

Figure 2.1: Merkle Tree Structure

stored off-chip, the root hash must be stored securely on-chip. Protection against replay attacks is provided as each time the contents of memory are modified, the root hash is updated. It is infeasible for an attacker to supply tampered data that will match the hash of the root node. Cryptographic hash functions are designed to be resistant to collisions, that is it is impractical to calculate two different messages that generate the same hash. The chances of two hashes colliding when using the SHA256 algorithm is statistically insignificant. Given the computing power of modern day processors, on average it would take longer than the lifetime of the universe to brute force search for a collision. Therefore it is safe to assume that an attacker would be unable to tamper the data in a manner that would match the root hash.

Merkle trees cause a significant memory and computation overhead as each memory block requires hash values to be stored. On a write operation, the data is read and authenticated, the data block is updated, and the hashes are recalculated for each node related to the data block. A secure cache can be utilized to store tree data in order to reduce the performance overhead of the authentication method. If the data stored in the cache is considered secure, an internal node can also act as a root node for data blocks. This is performed to reduce the number of operations required to validate memory [17].

2.3.2 The Parallelizable Authentication Tree

The Parallelizable Authentication Tree (PAT) is an attempt to increase the performance of the Merkle tree by allowing the hash computations to be run in parallel [18]. Each node of a Merkle tree is dependant on the value of their children's hashes requiring multiple dependant computations to be performed before retrieving the target hashes. PATs remove this dependence by instead generating numbers-used-once (nonces) and Message Authentication Codes (MACs) for each node. The nonce value can be generated randomly or deterministically, and it is updated each time the node is written to. The MAC value of a node consists of a hash value computed using the nonce value of both the node and its children nodes. Each node can then generate the MAC independently of other nodes because the nonce of each node is always available without computation. Data verification computations and tree updating computations for PATs do not rely on each other and can be parallelized to increase performance. As a nonce must be stored in addition to the hashes, the metadata size of PATs is greater than the metadata size of Merkle trees. This extra storage requirement adds additional memory costs to the authentication.

2.3.3 Tamper Evident Counter Tree

While previous methods only incorporate authentication into the design, the Tamper Evident Counter Tree or "TEC-Tree" [19] provides both data confidentiality and authentication. The TEC-Tree aims to reduce the memory overhead of PATs and provide data encryption while allowing for the same performance benefits. The TEC-Tree employs the block-level AREA scheme for data authentication. On a write operation,

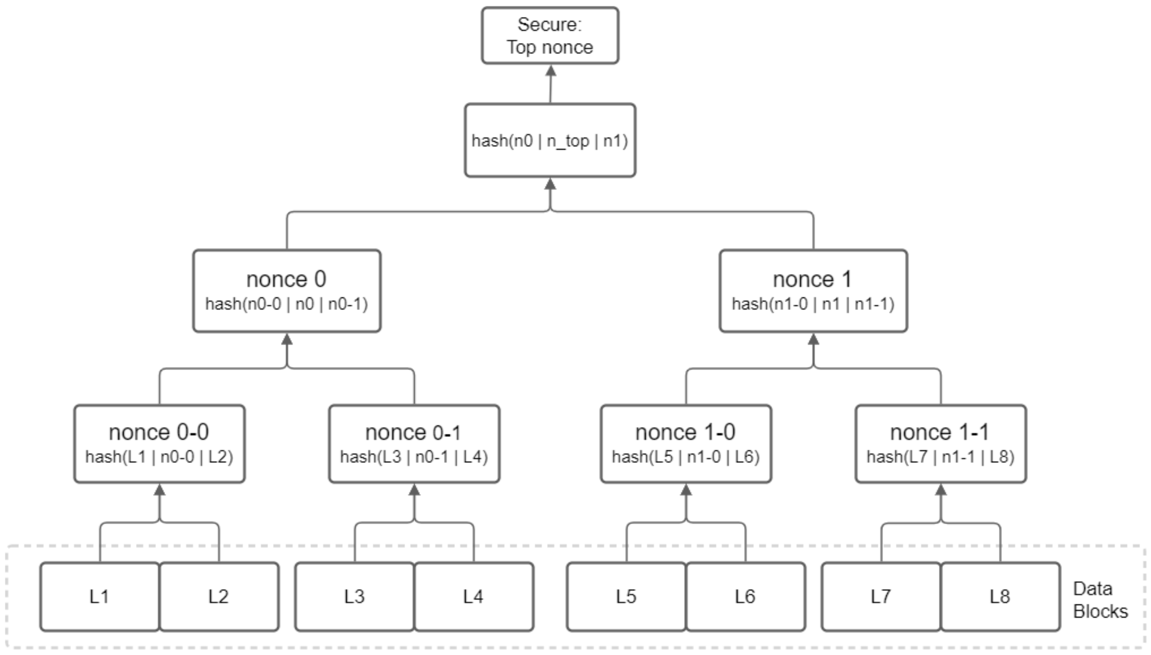

Figure 2.2: Parallelizable Authentication Tree Structure

a nonce consisting of the memory address and a counter value is concatenated to the data block being encrypted and written to memory. This nonce allows for a unique tag to be checked each time memory is accessed to provide the authentication. To reduce memory overhead and allow for parallelization, the counter values used in the nonces are then encrypted and stored off-chip together in their data block. The counter blocks are decrypted and utilized for verification when a value from memory is read. The only value necessary to store securely on-chip is the counter value used to create new nonces for each block of memory. A tree structure is formed as displayed in Figure 2.3.

The counter value stored on-chip acts as the root, and the counter blocks serve as the children nodes of the root. Leaves of the tree consist of ciphertext blocks that contain the original data being protected and the additional nonce encrypted with the block. As the nonce is encrypted in each block, Shannon's diffusion property states that any change at all in the encrypted block will cause the nonce value to not decrypt properly. Any modification of the nonce will subsequently cause the authentication

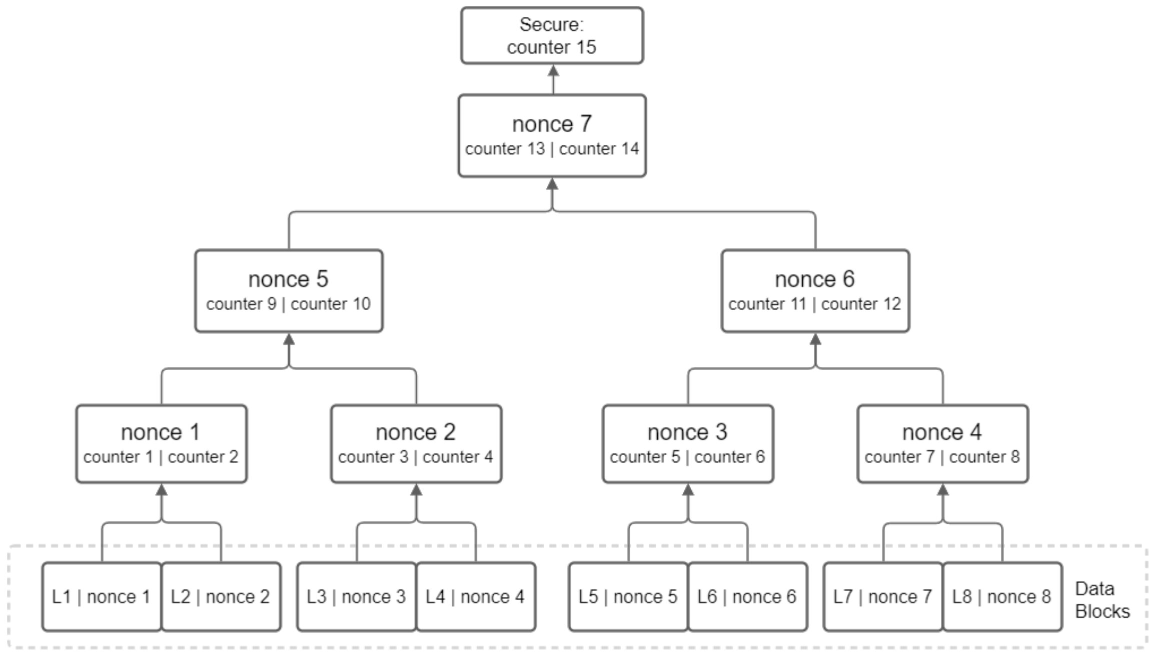

Figure 2.3: TEC-Tree Structure

to fail. When a read operation is performed, multiple instances of a decryption engine can be used to decrypt information in parallel, consequently reducing the performance overhead of this method. MITM attacks are able to be detected after the first decryption of the counter block as the nonce value will not match what is expected. A cache for the tree's most recently accessed nodes can be added in order to avoid reading tree data from memory to increase read speeds. An additional benefit of the TEC-Tree is the ability to change the arity of the tree structure to accommodate the requirements of the system it is being utilized in. Increasing the arity adds a larger additional memory overhead, but reduces the amount of time required to access the leaf nodes.

2.3.4 Dynamically Skewed Authentication Tree

2.3.4.1 Method One

Traditional authentication tree methods rely on static balanced tree structures to protect memory. A balanced tree approach introduces excessive overhead for real time memory verification based on different memory access patterns. The Dynamically Skewed Authentication Tree attempts to increase the performance of traditional authentication tree methods by reorganizing the structure of the tree dynamically. Nodes that were more frequently accessed are placed closer to the root of the tree during runtime[20]. Shifting frequently accessed nodes to higher levels allows for less tree traversal time to authenticate those data blocks by reducing the number of verification computations and intermediate node accesses. Less frequently accessed data blocks take significantly longer to authenticate; however, the time saved by reducing authentication time of more frequently accessed nodes with typical memory access patterns outweighs the negative performance impact of the lower weighted nodes. Similar to the TEC-Tree, data block nodes are concatenated to the plaintext implementing the block-level AREA technique providing memory encryption and authentication. Nonces are generated in the same fashion as in the TEC-Tree, a simple counter is stored securely on-chip to ensure each nonce is unique and never reused. Conventional approaches store a group of individual memory blocks in a leaf node, while this dynamic approach instead stores groups of memory blocks as sets. Each node stores the locations in memory that share the same frequency of access. Memory elements are stored on the lowest branch of the tree as leaf nodes and are referred to as data chunks. A structure represented by Figure 2.4 is established.

Each data chunk stores a unique nonce that additionally contains the data necessary to traverse to the root of the tree. Unlike the TEC-Tree, the first memory block accessed is the leaf node of the tree. The data is authenticated up from the leaf to the root of the tree. Traversal in this fashion impacts performance because the intermediate counter chunk data must be fully read before the next parent node can be accessed. The TEC-Tree does not suffer from this problem as the node protection addresses stay the same every time memory is accessed. A look-up table is required to map a node number to which data element is being protected. This differs from

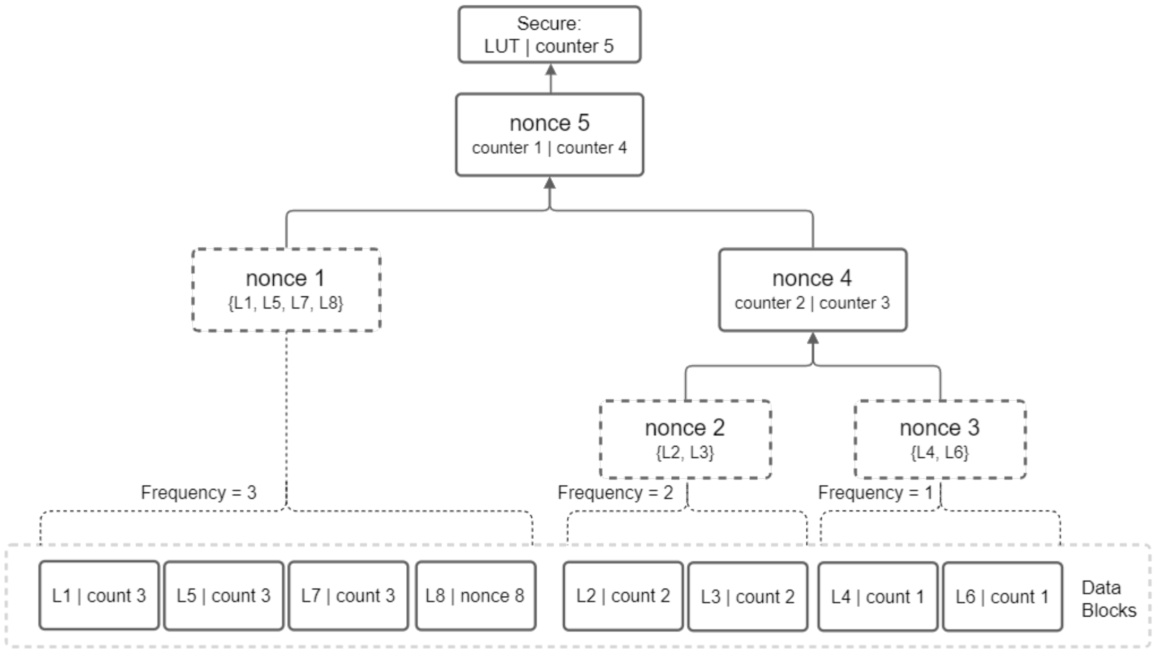

Figure 2.4: Dynamically Skewed Authentication Tree Structure: Method 1

the balanced tree whose natural ordering is used to determine which data block is being protected by a node. An index for each data block is stored, along with the corresponding node number protecting that data block. An element number is additionally required to indicate which element in the node's set represents the data block. This look-up table incurs an additional on-chip memory cost as it must be stored securely.

Dynamic restructuring of the tree requires two operations, set migration and rebalancing. Set migration occurs when the frequency of a data block is increased and needs to move to a different set. Since data blocks with the same frequency are grouped, whenever a block is accessed it must be moved to a set that is one frequency count higher. The algorithm for set migration is described in Algorithm 1.

Algorithm 1: Dynamically Skewed Authentication Tree Set Migration [20]

```
Data: $a$ : data element
$Q, P$ : pointers to Nodes;
$P \leftarrow$ find($a$);
$Q \leftarrow$ find($P' s frequency + 1$);
if $Q \neq \varnothing$ 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 $P \neq \varnothing$ then
        remove $P$ from the tree;
    else
        Shift Up($P$'s Sibling)
    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 $P \neq \varnothing$ then
        remove $P$ from the tree;
    else
        ShiftUp($P$'s Sibling);
        ShiftUp ($T$);
    end
end
```

Rebalancing must also occur when the nodes with a higher probability to be accessed are lower down in the tree than a node with a lower probability of access. In this case, nodes are shifted to fit the results of a Huffman encoding algorithm, as described in Algorithm 2. This algorithm is further discussed in Section 2.4.1.

Algorithm 2: Dynamically Skewed Authentication Tree Rebalance [20]

```
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
```

2.3.4.2 Method Two

A second and similar method of implementing dynamically skewed authentication trees is proposed in [21]. The most significant difference between this implementation and the first is that data nodes are no longer stored in sets. Alternatively, a single data node protects a single data element in memory. Eliminating the method of storing data in sets allows the look-up table to no longer be necessary and eliminates the added on-chip overhead costs associated with it. The block-level AREA concept is still applied, with each data element being protected by a nonce. The nonce contains the count of data accesses of that node and additional information to allow upwards tree traversal. Leaf nodes of the tree are capable of protecting a configurable number of data elements in order to reduce the storage overhead of the design. The structure generated by this method is depicted in Figure 2.5

With the removal of the look-up table and data element sets, the restructuring condition of set migration described by Algorithm 1 is no longer necessary. Instead, all restructuring is performed by the rebalance operation outlined in Algorithm 2. As with the previous method, on a write operation, the leaf node's count is incremented

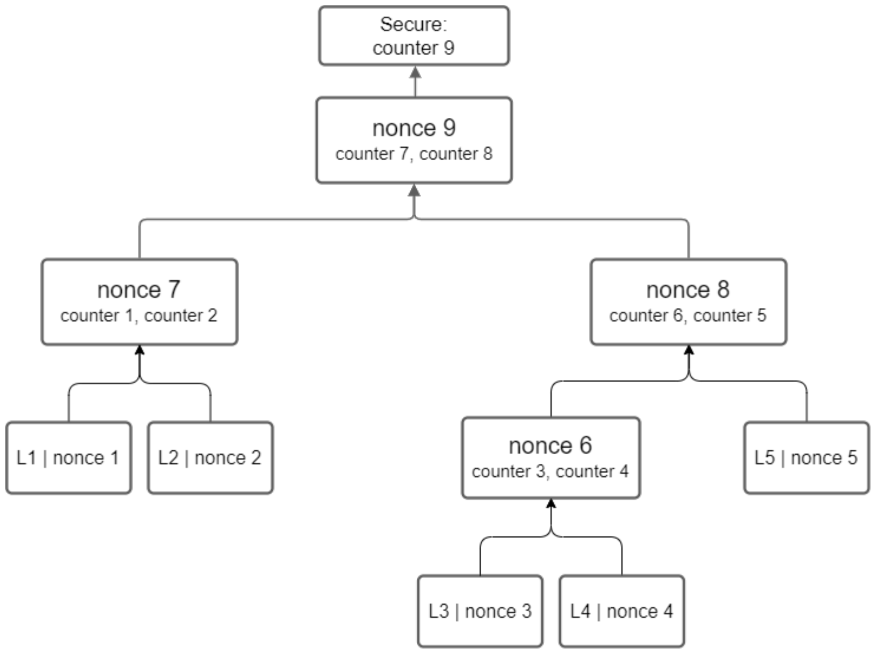

Figure 2.5: Dynamically Skewed Authentication Tree Structure: Method 2

and the tree is rebalanced if necessary. This is applied recursively until the root of the node is reached. If at any point, the nonce information does not correctly match the expected count, then an authentication error is raised.

2.4 Static Optimal Binary Search Trees

2.4.1 Huffman Encoding

Huffman encoding is a technique of constructing an optimal binary search tree that is widely used in the generation of lossless data compression using prefix codes [22]. Each leaf node of the tree consists of a weight used to guide the construction of the tree. Compared to similar methods, this tree is uniquely generated from the bottom up, ensuring optimality. The fundamental principle of the construction of a Huffman tree is the recursive combination of a pair of nodes that will result in the lowest weight. This new weight is then used to establish a new intermediate node of the tree, which is then added to the pool of nodes for the subsequent combination. Combinations are continued until the root of the tree is formed, and the result is an optimal binary search tree. Figure 2.6 demonstrates the construction of an optimal tree with the Huffman algorithm.

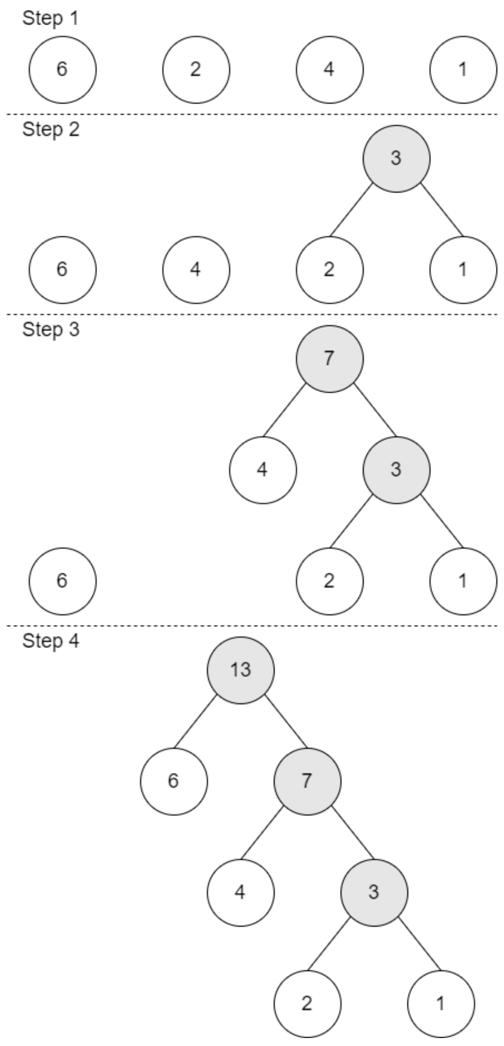

Figure 2.6: Huffman Tree Construction

In this example, the tree construct begins with an initial set of terminal nodes with varying weights. The pair of nodes resulting in the lowest weight (in this case, nodes with weights 1 and 2) are combined to create an intermediate node with a weight of three. Due to the unordered nature of the Huffman tree construction, the leaf nodes can be rearranged to construct this node. In step 3, the pair resulting in the lowest weight uses the terminal node with weight 4, and the newly constructed intermediate node. Finally, the last node is combined with the node created in step 3 and the root of the tree is formed.

2.4.2 Hu-Tucker

The Hu-Tucker algorithm generates an optimal binary search tree in a similar fashion as the Huffman encoding algorithm with one key difference, the order of the leaf nodes is preserved. Order preservation allows for nodes with both a frequency of access count and an additional weighted constraint to remain ordered and still construct an optimal search tree [23]. Historically, these ordered trees had been used for alphabetic search trees. The weighted priority of the algorithm could potentially be capitalized on to increase the speed of authentication trees. The construction of a Hu-Tucker tree consists of three phases: combination, level assignment, and recombination. An initial set of terminal nodes acts as the leaves of the tree and are combined to construct the intermediate nodes. Each node contains a value and a weight associated with it. A greedy algorithm is implemented to combine the two neighboring nodes that result in the lowest weight into a single node. This process is repeated until all nodes are combined and the root of the tree is formed. If there is more than one pair of nodes with an equal combined weight, the pair with the leftmost node is selected. Figure 2.7 demonstrates the construction of an optimal tree with the Hu-Tucker algorithm.

In a process similar to Huffman encoding, the tree construction begins with an initial set of weight terminal nodes. The key difference in this algorithm is that the order of these terminal nodes must be preserved in the tree. Therefore, when constructing the first intermediate node in step 2 the lowest weight pair would be a node with weights two and one. However, these nodes are not directly next to each other causing that pairing to be invalid. Instead, the lowest weighted pair of nodes that are neighboring each other would be nodes 4 and 1. This pairing process is applied until the root of the tree is formed.

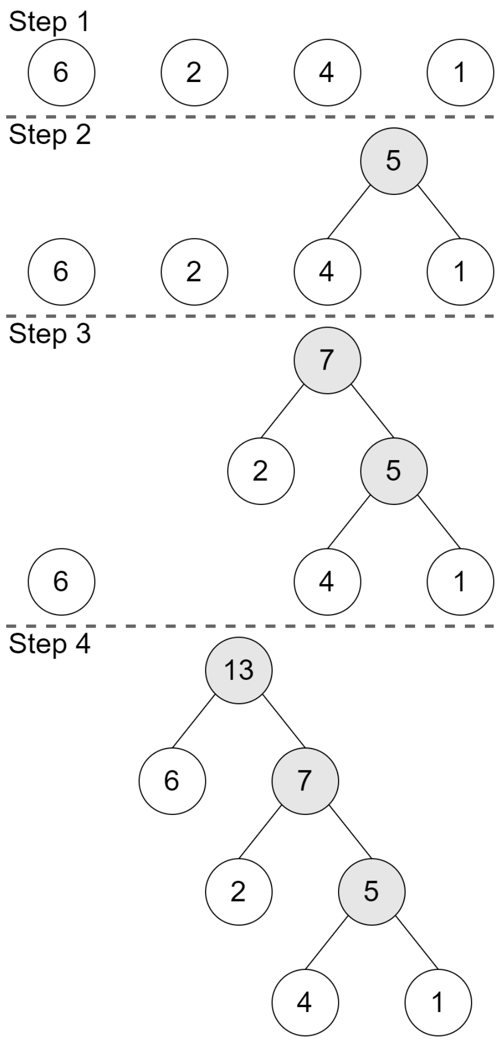

Figure 2.7: Hu-Tucker Tree Construction

Chapter 3

Flexible Ordered Dynamic Authentication Tree

3.1 Data Protection

3.1.1 Applied Block Level AREA

The data protection scheme employed in this authentication method relies on the diffusion property proposed by Shannon in [24]. Diffusion expresses the statistical relationship between the input plaintext and the resulting ciphertext of a given encryption cipher. Particularly, if a single bit of the input plaintext is changed while using the same key, the resulting ciphertext on average changes half of the ciphertext's bits. Routinely used modern ciphers, such as AES, are extensively studied to ensure the diffusion properties are sufficient enough for the cipher to be secure. For this data protection method, it is assumed any cipher in use meets this diffusion requirement. In both the TEC-Tree and the Dynamic Authentication Tree designs, a nonce is appended to any plaintext to be encrypted and stored in memory. On decryption, this nonce is compared to a previously stored nonce value that should match if data has not been tampered with. If even a single bit of the data is modified in an encrypted data block, on decryption the nonce has a negligibly low chance of remaining the same. The number of possible plaintext blocks with the same nonce resulting from the decryption of a tampered ciphertext is equal to 2^{b-n}, where b is the size of the data block in bits, and n is the size of the nonce in bits. The probability of the nonce remaining the same after decryption if memory is tampered with is then $\frac{2^{b-n}}{2^b} = \frac{1}{2^n}$ [19]. With this fact, the size of the nonce can then be increased to increase security. Intuitively, increasing the nonce size will also increase the extra memory overhead of the authentication. This nonce size can then be tuned to specific applications in order to balance security concerns versus extra storage costs.

3.1.2 Example

Considering an example case, assume we are using 128-bit blocks. A data block before encryption is composed of a 96-bit payload and a 32-bit nonce. The nonce contains an arbitrary value for this example, and in a real application, a larger nonce value can be used with deterministic single-use values. The hex value 0xAD is being written to memory, and the entire data block must be accessed in order to write this value. The original value of the data block is all zeros. The data is encrypted using AES in ECB mode with a secret key of 0x0. Figure 3.1 displays a case in which the ciphertext data is not tampered with and the authentication succeeds.

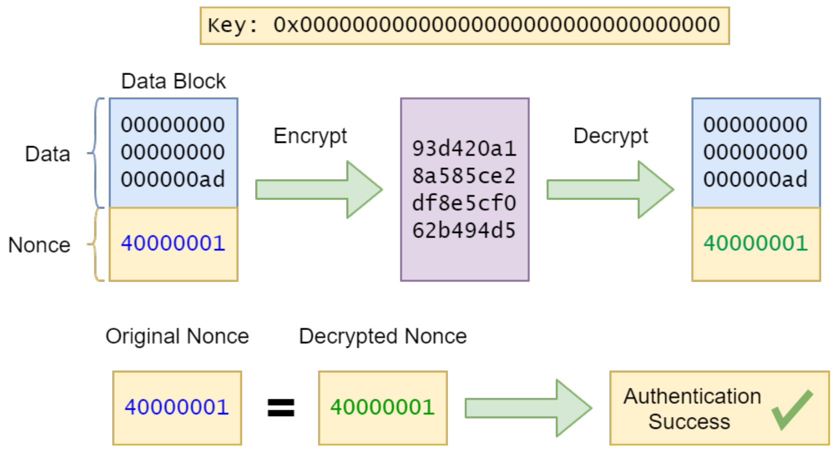

Figure 3.1: Block-Level AREA Example: Pass

First, the original 128-bit data block is encrypted, producing a ciphertext that is secure from the attacker's analysis. Once this data block is necessary to be read again, the data is decrypted, and as expected the original nonce matches the decrypted nonce value. Since the nonces match, it is safe to assume the data is untampered and the authentication is successful. The data would then be forwarded to the CPU for further processing as usual. In the opposite case, Figure 3.2 displays an authentication failure using the block-level AREA technique.

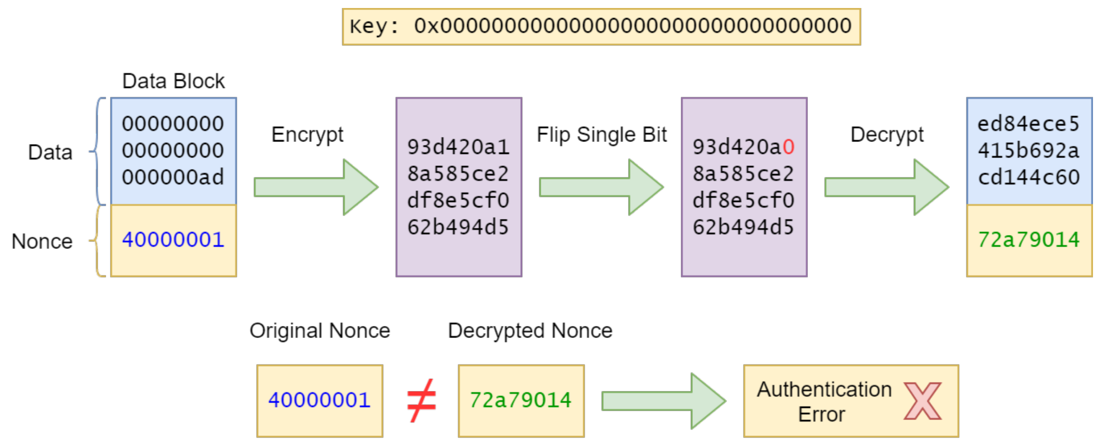

Figure 3.2: Block-Level AREA Example: Failure

Just as in the previous example, the original data block is encrypted and produces the same ciphertext. Here an attacker flips a single bit in an attempt to disrupt the system. The now corrupted ciphertext is decrypted, and the resulting plaintext does not match the original data block. Most importantly the decrypted nonce is different than the original nonce. These nonces are then compared and the authentication fails because of the mismatch. An authentication error can be raised, and the CPU is notified that the data has been corrupted.

3.2 Tree Nodes

The leaf nodes of the authentication tree structure contain the data blocks that are directly being protected by the tree. For the rest of this thesis, the leaf nodes will be referred to as data nodes. Numerous ways to relate the data blocks being protected to the leaves of the tree are possible. For example, the TEC-Tree [19] originally proposes that a 192-bit data block is used for each leaf node. 128 bits of this node store the original data to protect, while the additional 64 bits are used to store a nonce for authentication. AES with a 128-bit key in ECB mode then serves as the encryption engine for the design. While this is a viable option, the memory overhead for this method is considerable. Since the nonce used is half the size of the original data, this implementation requires a significant storage overhead of 1.5 times the original data size. The implementation suggested in [21] employs a similar design, recommending a 256-bit data block. 128 bits of that block would contain the original data while the other 128-bits contain a nonce storing tree metadata. This design requires an even larger 2 times storage overhead. In an attempt to address this issue, the approach used in this authentication tree design uses an implementation similar to that applied in [25]. Instead of limiting the data block size to be aligned with a cipher's size requirements, different cipher mode implementations are used to allow for variable data block sizes. Applicable modes of operation are further discussed in Section 4.4. This customization option allows for a more flexible design that is able to address various use cases. Intermediate nodes are generated using the frequencies of access for each data node as specified in Section 3.6. Due to the fact that the counts for each child node are stored in each intermediate node, these nodes are referred to as counter nodes. Figure 3.3 displays the general structure of a data node and counter node as used by the dynamic authentication tree.

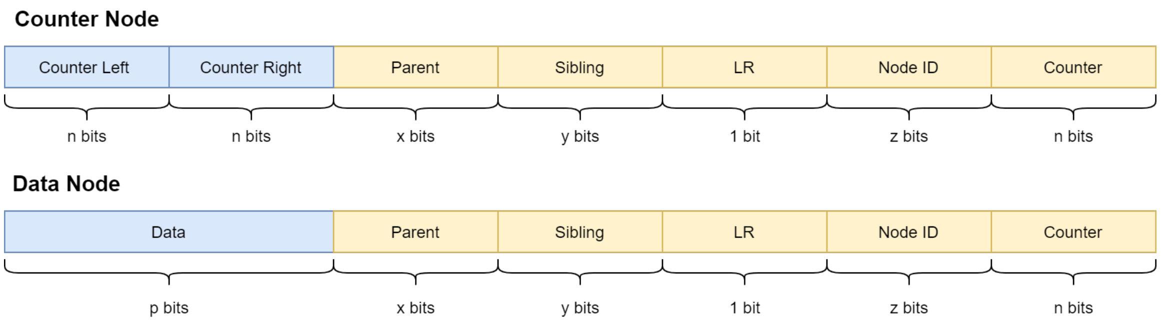

Figure 3.3: Data and Counter Node Contents

The data node is 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. As for the metadata, the size requirements are less flexible. Both the counter node and data node share the same metadata that stores the relationship between other nodes. When data is accessed, the authentication begins at the bottom of the tree with the data nodes. In order to traverse the tree from the bottom up, the parent node ID is required to be stored by each node except the root node. This ID is represented by "Parent" field in the metadata. The "Sibling" field represents the sibling ID of that node and is utilized in checking rebalancing conditions for dynamic restructuring of the tree. The ID of the node itself is also stored in the "Node ID" field, but this is an optional field depending on the implementation of this design. The "Node ID" field is included to allow for easier rebalancing but is not necessary if performed differently. As the frequency of access is required for the block-level AREA scheme, this is also stored in the metadata of each node as the "Counter" field. Whenever the tree is traversed, the node's current count is compared to its parent's stored count. The parent node must be a counter node that stores both the left and right children counts in the "Counter Left" and "Counter Right" fields. There are two counters stored per counter node and the "LR" is used to specify which counter must be compared. This value is stored to indicate whether or not the current node is a left or right child node to its parent. As the block-level area scheme utilizes the metadata as a nonce, additional data may be added to the nonce if a design requires a larger level of security. For most cases, simply comparing the counts of the node provides adequate authentication. For applications with a large security risk, it may be desirable to increase the nonce size and compare more information.

3.3 Tree Data Addresses

It is required by the block-level AREA technique for the data node metadata to be stored in the same area of memory as the data it is protecting. This causes the actual location of the data stored in memory may vary from what the processor expects when it sends requests. The memory controller must account for the data size differences and provide the processor with the appropriate information. In this case, the originally requested address that the processor expects is referred to as the virtual address. The offset memory address where the data is actually stored is referred to as the physical address. Every time memory is accessed, the entire memory block needs to be read to properly decrypt the data; therefore, a memory block address needs to be calculated. When a request is generated for a specified data block, the data modification or data read is handled by the Data Modifier and the Data Filter components specified in 4.3, respectively. The virtual address for the start of each data block can then be calculated using Equation 3.1 and the physical address is calculated using Equation 3.2. In this tree, the size of protected memory is N bytes and the size of data blocks is D bytes, the size of a counter is C bytes, and the size of the metadata is M bytes.

$$virtual\ address = address - (address \pmod{D + M}) \tag{3.1}$$

$$physical\ address = virtual\ address + \left\lfloor \frac{virtual\ address}{D + M} \right\rfloor * (D + M) \tag{3.2}$$

To simplify memory address calculations, it is recommended to store the counter node information together at the end of the protected memory area. The start address of all counter node data would then merely be the memory size N. With this method, each counter node address can be calculated using Equation 3.3.

$$Counter\ Node\ Address = N + (2C + M) * Node\ ID \tag{3.3}$$

3.4 Ordered Dynamic Authentication Tree

A dynamic authentication tree is a 2-arity tree consisting of data nodes as the leaf nodes and counter nodes as the intermediate nodes. The structure of the tree depends on the weighted frequency accesses of each data node. As previously discussed, the block-level area scheme is applied to each node of the tree when data is accessed. Data nodes consist of the underlying data being accessed and an injected redundant nonce, containing metadata required for the traversal of the tree from the bottom up. In the block level AREA scheme, the nonce data is considered redundant as the nonce must already be capable of being generated before the memory block is decrypted. This redundancy is required for the comparison of the generated nonce with the original nonce in the encrypted data block. When a memory access is performed, the entire data block including the nonce is first read and decrypted. Then the metadata contained in the nonce is used to determine which counter node is used as the data node's parent. The parent node is then also read and decrypted, and the parent node holds the access count of its child 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. The root counter node's count is compared to this secure root counter for authentication purposes. If at any point the authentication fails due to a counter mismatch, the processor is 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.

On 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' weight. The parent's frequency access count is incremented in addition. All operations performed on decrypted plaintext are performed in secure programmable logic where an attacker is incapable of observing the operations and plaintext data. When a write occurs, the tree additionally is inspected to ensure that the new node weight is not higher than the weight of the nodes above it. If the lower node's weight is higher, then the tree is rebalanced and restructured. While a number of algorithms provide a valid dynamic authentication tree, the focus of this work is a new dynamic algorithm based on an ordered binary search tree. The rebalancing condition and restructuring algorithm are detailed in Section 3.6. The generic dynamic authentication tree structure is represented identically to the description in Figure 2.5. Figure 3.4 displays the data structure of the authentication tree that these specifications create with additional metadata information shown.

It is important to note that because the tree's structure depends on the data node access frequencies, the structure displayed here contains example information to help describe them. This information is necessary to help explain the current balanced state of the tree. As this is a dynamic tree, the particular configuration of a tree given a memory access pattern is subject to change. Additionally, the dynamic nature of the tree does provide inconsistent resulting performances depending on the memory access pattern the tree is operating under, which is discussed further in Section 3.7.2. It is evident in this example that the tree structure does not allow a higher weighted node to be lower on the tree than a lower weighted node. This allows for more rapid access to a data node that contains a higher frequency of access. 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 traditional authentication tree methods which historically rely upon a

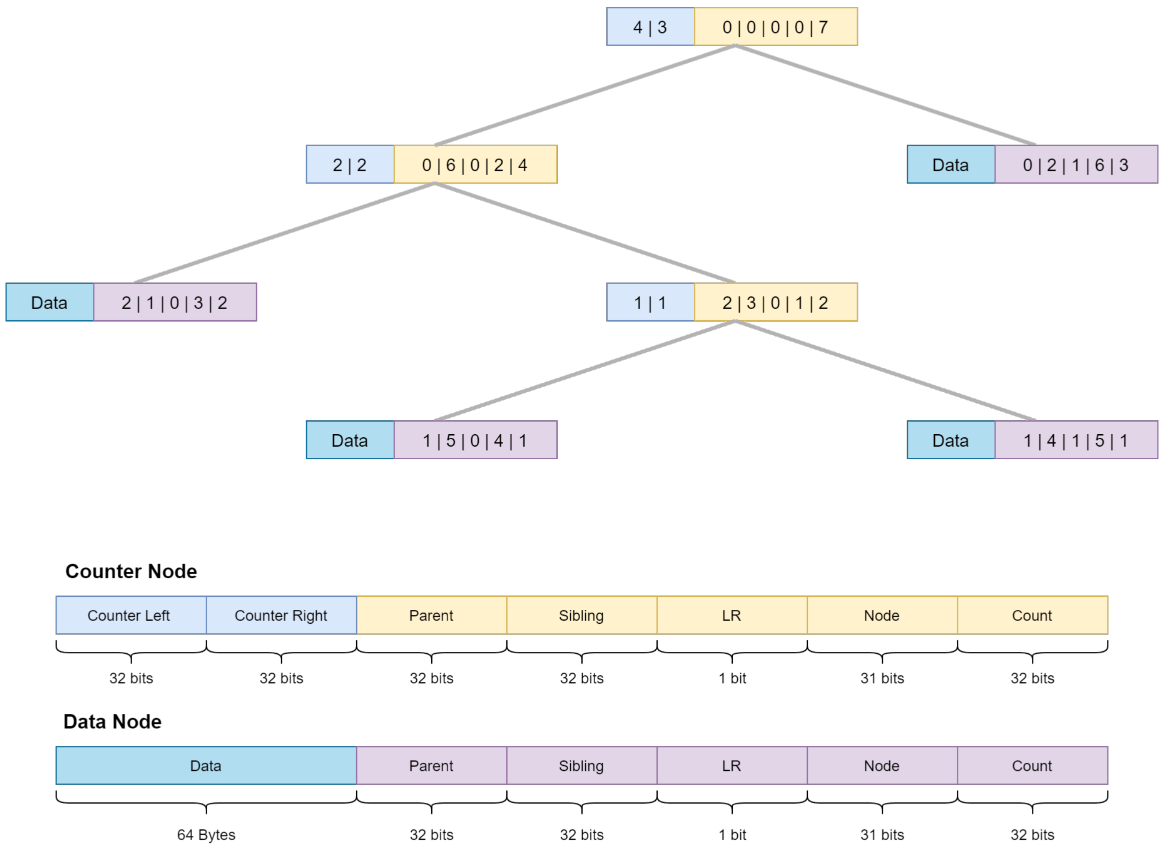

Figure 3.4: Ordered Dynamic Authentication Tree

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.

There are, however, downsides to this design compared to a design such as the TEC-Tree. Rebalancing provides two extra operations that increase the performance overhead of the design. Every time a write occurs, the tree must be checked for rebalancing conditions as the node frequency being accessed is incremented. This requires at least one additional node per level traversed to be read for a comparison of their counts. On top of this extra operation, if the tree does in fact need to be restructured, the counter nodes' metadata must be updated. Depending on the rebalancing condition, which is further outlined in Section 3.6, additional nodes may need to be read, updated, and written back to memory. If a rebalance condition was not met, these nodes would not need to be accessed. Compared to a static tree design, the extra rebalancing calculations provide a significant difference in performance for each memory write operation. In light of this fact, certain memory patterns will more favor a dynamic authentication tree structure versus a static authentication tree structure, which is discussed and analyzed further in Section 3.7.2.

3.5 Tree Initialization

Before any tree accesses can be performed, the tree requires an initial state in order to allow for dynamic restructuring or tree traversal to properly take place. The rebalancing algorithm implemented allows for zero weighted counts, which then requires only the metadata specifying the node's relationship with each other to be initialized. The tree begins as a balanced tree, with each node containing an equivalent count of zero. No memory addresses have been accessed yet and each node begins with the same probability of being accessed. Algorithm 3 contains an initialization routine for a dynamic skewed tree.

Algorithm 3: Dynamic Authentication Tree Initialization Routine

```
Data: $N[T]$ : Array of nodes of size $T$
Result: Dynamic tree nodes initialized

struct {
    parent;
    sibling;
    LR;
    ID;
    count;
} $Node$;

for $i \leftarrow 0$ to $T - 1$ do
    $N[i].ID \leftarrow i$;
    $N[i].LR \leftarrow \neg(i \pmod 2)$;
    $N[i].parent \leftarrow ((i+1)/2) - 1$;
    if $N[i].LR = 0$ then
        $N[i].sibling \leftarrow i + 1$;
    else
        $N[i].sibling \leftarrow i - 1$;
    end
    $N[i].count \leftarrow 0$;
end
```

In this case, to provide easy memory address calculations the counter nodes begin with the root ID as zero. The node IDs are filled in starting at the highest level of the tree downwards while assigning each node from left to right. With this method, the initial tree's nodes with an even ID will also be a left child node. The sibling node ID can then be set based on whether or not the node is a left or right child. A pattern emerges for the parent node ID; each node is originally a child of a parent with half their ID minus one.

3.6 Dynamic Restructuring

During a program execution, as memory blocks are accessed the tree structure is dynamically updated to place memory blocks with a higher frequency of access closer to the root of the tree. As each node is accessed, the count of that node is incremented and its parent node's left or right counter is also updated to ensure authentication checks function properly. When a node's weight is incremented, the new weight needs to be compared to the existing node's weights. This is done to ensure that nodes with a lower frequency remain further from the root than nodes with a higher frequency of access. As node data is stored encrypted in memory, it is infeasible to read and decrypt all tree nodes and check each weight value on each memory access. Performance of the memory accessed would be too greatly affected if each node were to be read; therefore, instead of accessing the entire tree, a small subset of the tree is validated each time memory is written to. A node contains the ID of that node's parent and its sibling node. Using this information, when a node is accessed, both its parent node and that parent's sibling are accessed. A parent's sibling node is referred to as an uncle node. The uncle node's weight is compared to the current node's new weight. If the current node's new weight is greater than the uncle node's weight, then the tree is flagged for rebalancing. The method of rebalancing in the tree design presented here consists of three different possible cases used in order to preserve the order of the data nodes. The first rebalancing case is described by Algorithm 4.

Algorithm 4: Ordered Rebalancing Method #1

```
Data: Parent, Uncle, Sibling, Current
if Current's LR $\neq$ Uncle's LR then
    // Shift Uncle down
    Uncle's Parent $\leftarrow$ Parent;
    Uncle's Sibling $\leftarrow$ Sibling;
    // Rotate Parent
    Parent's sibling $\leftarrow$ Current;
    Parent's LR $\leftarrow \neg$ Parent's LR;
    // Update and Rotate Sibling
    Sibling's sibling $\leftarrow$ Uncle;
    Sibling's LR $\leftarrow \neg$ Sibling's LR;
    // Shift Current Node Up
    Current's Parent $\leftarrow$ Parent's Parent;
    Current's Sibling $\leftarrow$ Parent;
end
```

If it is desired to retain the order of the data nodes, this rebalancing case is only valid if the current node's left/right position is different than its uncle node's left/right position. In this case, the current node being accessed is shifted up a level by shifting the parent node to the side and shifting the uncle node down a level. Figure 3.5 visualizes the steps in this process. To perform these operations, the parent and sibling relationship stored in each node is able to be updated with the new case. This allows the operation to be performed quickly and efficiently. It is necessary to read and decrypt the parent, uncle, sibling, and current node data from memory to perform these operations.

In the case that the uncle and current nodes do not match the left/right position, a second rebalancing method must be implemented. If the first rebalancing method were to be used, the leaf node order would be modified and invaliding the ordered design. The second rebalancing method is described by Algorithm 5 and visualized by 3.6.

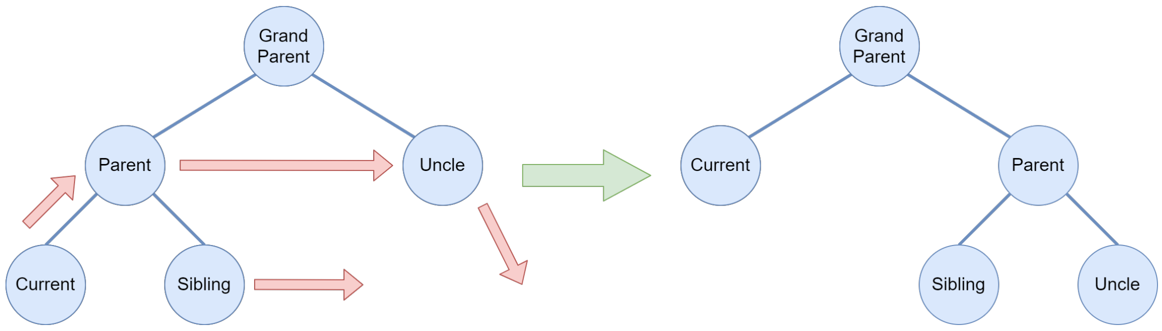

Figure 3.5: Ordered Rebalance Method 1

Algorithm 5: 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 $\leftarrow$ Grand Uncle's Parent;
    Parent's Sibling $\leftarrow$ Grand Uncle's Sibling;
    // Update Uncle
    Uncle's Sibling $\leftarrow$ Current Node;
    // Shift Grand Uncle Down
    Grand Uncle's Parent $\leftarrow$ Parent;
    Grand Uncle's Sibling $\leftarrow$ Sibling;
    // Update Grand Parent
    Grand Parent's sibling $\leftarrow$ Parent;
    // Shift Sibling Up with Current Node
    Sibling's sibling $\leftarrow$ Grand Uncle;
    Sibling's LR $\leftarrow \neg$ Sibling's LR;
    // Shift Current Node Up
    Current's Parent $\leftarrow$ Grand Parent;
    Current's Sibling $\leftarrow$ Uncle;
    Current's LR $\leftarrow \neg$ Current's LR;
end
```

The second rebalancing method requires the Grand Uncle and Grand Parent node's information in addition to the same information from the first method. When the current node is shift upwards, the parent node is not shifted downwards but is instead updated with the grand uncle replacing the current node as a child. To

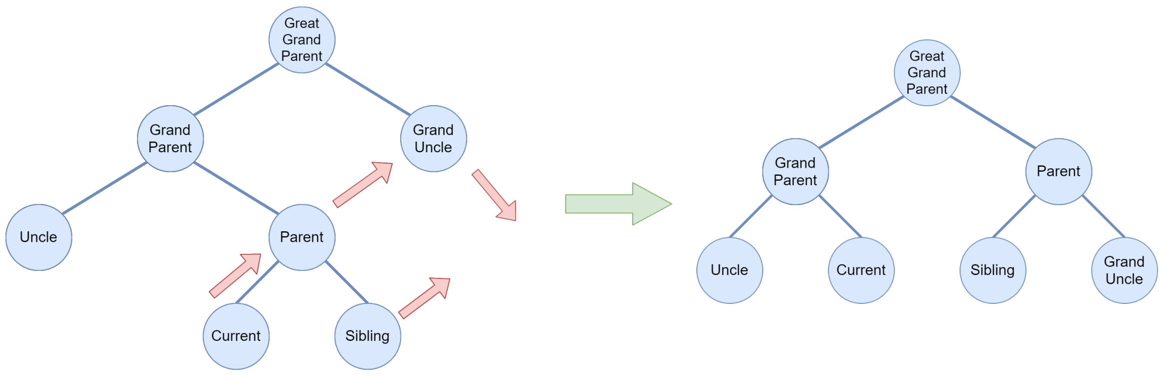

Figure 3.6: Ordered Rebalance Method 2

keep the order of the leaf nodes, the grand uncle and the sibling's left/right positions are swapped. Both of these methods do not retain leaf node order if applied to the case where 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. To address this final case, a third rebalancing method is introduced as described in Algorithm 6 and displayed by 3.7.

Algorithm 6: 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 $\leftarrow$ Grand Parent;
    Grand Uncle's Sibling $\leftarrow$ Sibling;
    // Update Grand Parent
    Grand Parent's sibling $\leftarrow$ Parent;
    Grand Parent's LR $\leftarrow \neg$ Grand Parent's LR;
    // Update Uncle
    Uncle's Parent $\leftarrow$ Grand Parent;
    Uncle's Sibling $\leftarrow$ Grand Uncle;
    Uncle's LR $\leftarrow \neg$ Uncle's LR;
    // Shift Parent Up
    Parent's Parent $\leftarrow$ Grand Parent's Parent;
    Parent's Sibling $\leftarrow$ Grand Parent;
end
```

Methods one and three are very similar, with the main difference being that the

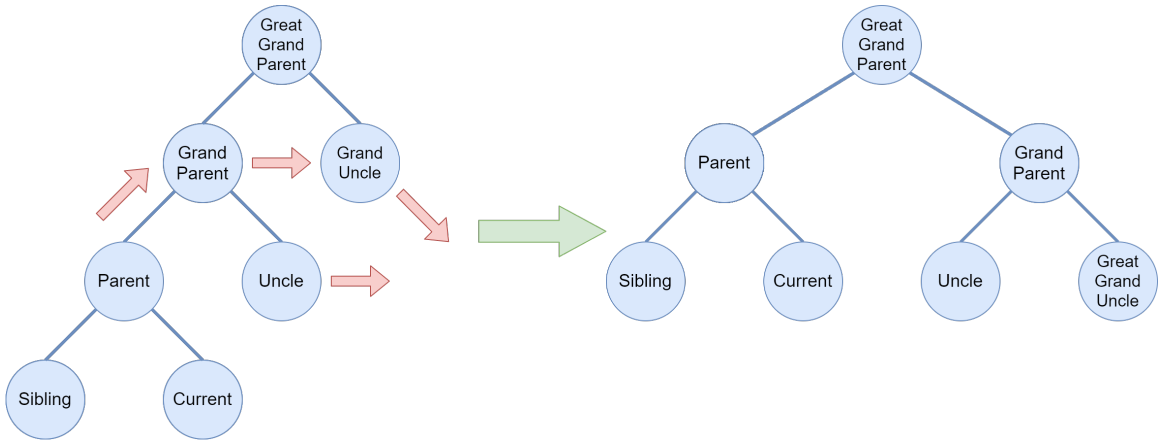

Figure 3.7: Ordered Rebalance Method 3

node operations are applied to one level above the current node. While the current node's information is not updated itself, the parent node is shifted upward, moving both the current and sibling node up a level as well. Considering the grand uncle must be accessed for this method as well as method two, when checking the rebalance condition an additional qualifier must be added to ensure that a great grand uncle exists and the root has not been reached. The final rebalance checking condition is then expressed in Equation 3.4.

$$\begin{aligned} &\neg((Current\ Node's\ LR = Uncle's\ LR) \wedge (Parent's\ Parent = 0)) \\ &\quad \wedge(Current\ Node's\ Weight > Uncle\ Node's\ Weight) \end{aligned} \tag{3.4}$$

3.7 Customizable Overhead Analysis

The authentication tree design presented here is highly customizable and is able to be modified in order to accommodate a specific application's needs. Real-time memory authentication is costly in both performance and memory overheads. However, if one aspect of an application contains larger performance or memory constraints, these parameters can be easily tuned to tailor the design to those needs. Generally, there is a trade-off between speed and storage requirements, the faster the authentication time the more extra memory overhead is required and vice versa. The overhead for this proposed design is compared against the overhead of the base TEC-Tree implementation described in [19] and the Dynamic Skewed Tree proposed by [21].

3.7.1 Memory Overhead

3.7.1.1 Off-Chip Overhead Cost

The additional off-chip memory costs associated with each design is as follows, where TEC is a subscript used to denote relation to the TEC-Tree, skewed denotes relation to the Dynamic Skewed Tree, and DAT denotes relation to the proposed Dynamic Authentication Tree design. Table 3.1 describes the variables contained in the following equations.

Table 3.1: Off-chip Memory Cost Summary

Variable	Description
O_{TEC}	Overhead of the TEC-Tree
O_{skewed}	Overhead of the original dynamic skewed tree
O_{DAT}	Overhead of the proposed dynamic authentication tree
l_p	The data payload size in bits
n	The bit length of nonces
A	The arity of the tree

$$O_{TEC} = \frac{l_p + n_{TEC}A}{l_p(A-1)} \text{ [19]} \tag{3.5}$$

$$O_{skewed} = \frac{l_p + n_{skewed}A}{l_p(A-1)} \text{ [21]} \tag{3.6}$$

$$O_{DAT} = \frac{l_p + n_{DAT}A}{l_p(A-1)} \tag{3.7}$$

Where l_p is the data payload size in bits, n is the bit length of nonces, and A

is the arity of the tree. The relationship between off-chip memory costs for each design turns out to be identical besides the variation in nonce sizes. Arity is also a large contributing factor, as a large arity increases the off-chip storage size. As the methodology of this design only contains algorithms for a tree arity of two, the comparisons will focus on all design utilizing an arity of two. It is possible to expand both the work presented in [21] and the work proposed in this thesis to allow for larger aritys, but for the scope of this design that is saved for future work. Each overhead equation can then be simplified as follows.

$$O = \frac{l_p + 2n}{l_p} \tag{3.8}$$

The total memory overhead then relies solely on the nonce size of each design. All authentication trees require a significant off-chip memory overhead, some exceeding even two times the original data size. The nonce sizes for each design are shown below.

$$n_{TEC} = c + \log_2 N \tag{3.9}$$

$$n_{skewed} = c + 3 * \log_2 N \tag{3.10}$$

$$n_{DAT} = c + 3 * \log_2 N \tag{3.11}$$

Where c is the bit length of the counters used for authentication, D is the total number of data nodes used in the tree. In the case of a dynamic tree, the design relies additional stored metadata in order to traverse the tree structure. The proposed method's off-chip memory cost is slightly increased versus the TEC-Tree, but remains the same as the Dynamic Skewed Tree. The optimal case for each node ID stored in the none metadata is a bit length of $\log_2 N$. However in cases where data size is not a significant issue, data alignment to 4 byte words can be more easily achieved by setting the node ID lengths to 4 bytes each.

3.7.1.2 Memory Overhead Example

To put the overhead in perspective, consider a case where 1 KB of memory is to be protected, and the total number of trees protecting memory is 2. Data nodes have been configured to protect 64 bytes of memory each. Each tree then protects 512 B of memory with 8 data blocks per tree. For easy memory alignment, a non-optimal 32 bits per node ID stored is configured, as displayed in the example in Figure 3.4. A counter size of 32 bits has also been chosen for convenience. For any binary tree, the number of intermediate nodes is equal to the number of leaf nodes subtracting one. In this case the number of counter nodes is then 7. Figure 3.8 represents off-chip memory with these parameters.

Despite containing large counter sizes and node ID sizes, the additional overhead of this design is comparable to other authentication methods. The extra metadata stored per node is 16 bytes, and the counter nodes require an additional 8 bytes to store the left and right counters for comparison. Each tree produces an additional $15*16+8*7 = 296$ bytes. To protect 1 KB of memory with this method, an extra 592 bytes is required, resulting in a $(1024+592)/1024 = 1.578$ times memory overhead.

The on-chip memory cost is simple. The size of the counter value used for authentication and the root of the tree is customizable. Only a single counter per tree used to protect memory is required to be stored on on-chip memory. Therefore, the on-chip memory cost is equal to Equation 3.12. The on-chip memory cost is the same for the TEC-Tree and both dynamic tree designs. For this example with 2 trees and a counter size of 32 bits, the total on-chip memory cost would then be only 64 bits.

$$Root\ Storage\ Size = counter\ size * number\ of\ trees \tag{3.12}$$

3.7.2 Performance Overhead

A dynamic authentication tree requires additional computational overhead on each write transaction compared to the TEC-Tree. Every write requires that a recursive check for rebalancing is applied. An extra counter node must be read for each level in addition to the information that the TEC-Tree requires. If the tree does need to be rebalanced there are extra computations and writes to memory that must be performed. Generally, a dynamic tree rebalance does not happen frequently, especially with memory access patterns that frequently access certain areas in memory. This rebalance overhead is usually minimal and is explored further in the implementation results in Section 4.5.3. Due to the number of extra computations performed, it is expected that a memory access pattern that utilizes writes to memory more frequently than reads may perform worse. The worst-case scenario for a dynamic tree would involve a uniform random distribution of memory accesses through the entire protected memory address range. If each memory address were accessed with the same frequency, the tree would statistically diverge to a balanced tree. This tree would include worse write performance than a TEC-Tree due to the extra rebalancing calculations. Contrarily, the best-case scenario for this structure would be a memory design that most frequently accesses a single memory address, and that memory address is mostly read instead of written. Performance speedup occurs when a read or write transaction occurs on a data 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.

3.7.3 Counter Bit Length

Counter bit length is an important customizable part of the tree's design. Whenever a counter reaches the max value it is able to represent, the authentication tree cannot allow the counter to simply overflow. If the counter overflowed, a nonce may be repeated, and this can leak information to an attacker. In order to prevent this, each time the counter overflows the secret encryption key must be regenerated. All encrypted tree data must first be decrypted and re-encrypted with the new key, requiring a large amount of time to complete these operations. A counter length with a size large enough to prevent too frequent overflows must be chosen in order to minimize the impact of this performance. For many applications a counter size of 32 bits will take an extremely large amount of time to overflow; however, for applications that run for long periods of time, it could be a concern. Naturally, increasing the counter size to reduce the performance head will increase the size of off-chip storage utilized by the tree.

3.7.4 Leaf Node Data Block Size

The size of the data block per leaf node can be refined to provide a balance between performance and storage size. Whenever memory is accessed, the entire data block must be decrypted because the entire block is encrypted together. If only part of the block was decrypted or re-encrypted without the rest of the block, the encryption scheme would fail and the data would become corrupted. In light of this, increasing the data block size per leaf node will reduce the storage required, as fewer leaf nodes would be required to protect the entire portion of memory; however, the performance of each memory transaction would be reduced as more data must be accessed each time.

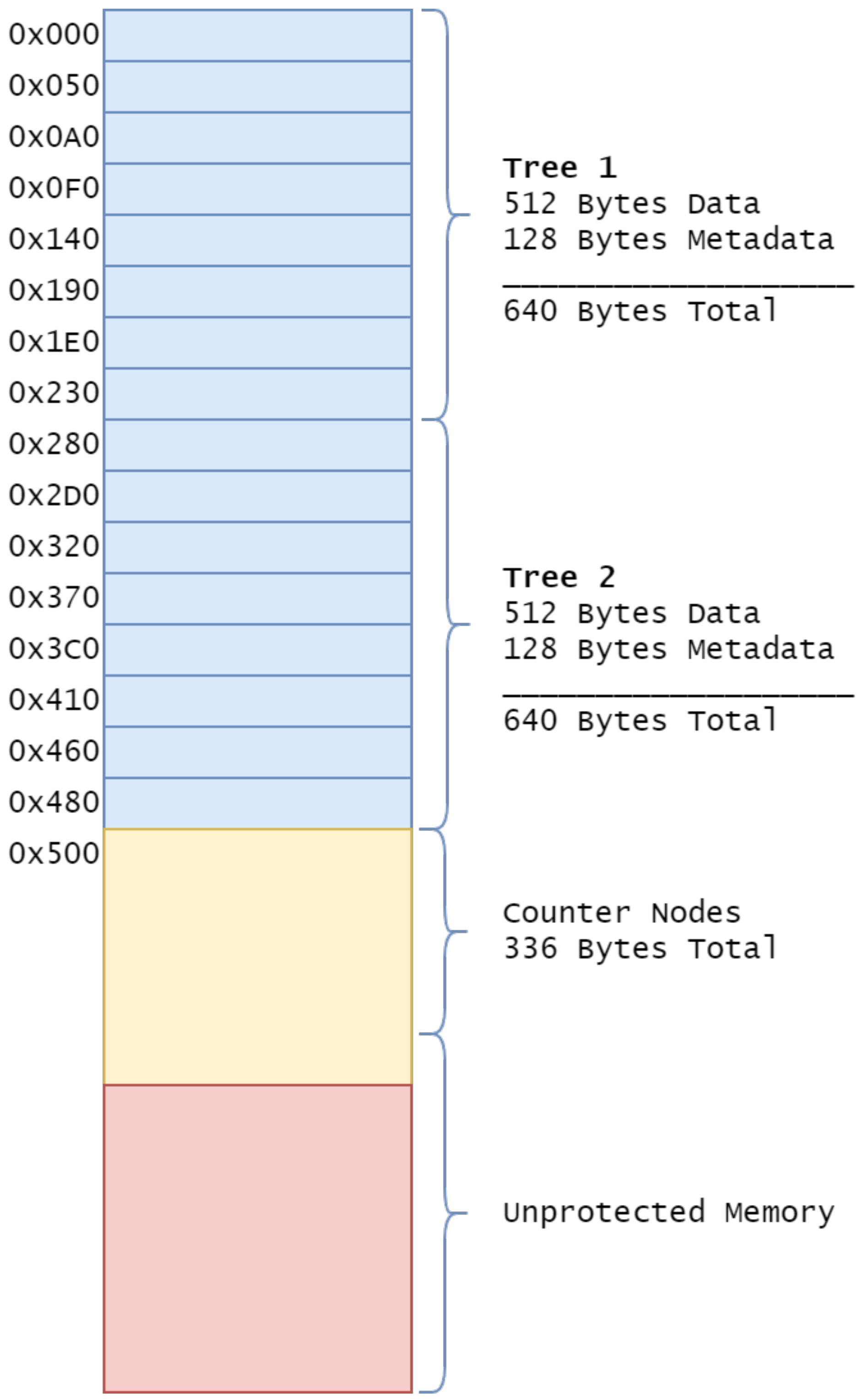

Figure 3.8: Off-Chip Memory Overhead Example

3.8 Optimality Analysis

Optimal binary search trees provide the lowest cost to access each leaf node of the tree given a set of weights for each node. An unordered optimal tree can be built using Huffman coding [22] while an ordered optimal tree can be constructed with the Hu-Tucker algorithm [23]. A software implementation written in the C programming language was developed for both the Hu-Tucker algorithm and the algorithm proposed in Section 3.6 for comparison of optimality. Due to the nature of the proposed algorithm, rebalancing conditions only consider adjacent nodes of the nodes being accessed. It is possible that nodes are accessed in a pattern that the tree is not optimal due to limitations of the algorithm not extending to check the entire tree. 10,000 different trees with different weights for the leaves and different memory accesses were constructed with both the Hu-Tucker and dynamic tree algorithms. The total cost of each tree was then calculated and compared to determine the optimality of the dynamic tree. As the Hu-Tucker tree is guaranteed to be optimal, these measurements were used as a baseline for optimality. In 10,000 different cases, only 20 Dynamic Trees were not optimal, leading to a 99.8% chance of the dynamic tree remaining optimal. The maximum difference in cost was 38 while the lowest was 1. The average difference in cost was 5.7; however, this average does not represent a significant statistic as the difference in cost for a tree with 10,000 memory accesses would be higher than a tree with 10 memory access. Concluding from these results, the difference in optimality from the proposed dynamic tree algorithm and the Hu-Tucker algorithm is minimal, with a 99.8% chance of remaining optimal with a sample size of 10,000. Furthermore, when there is a difference in optimality that difference is minuscule and insignificant as it may even be corrected in the future with further rebalancing.

Chapter 4

Memory Controller Framework

4.1 Security Model

Many embedded systems have an exposed memory bus line from the on-chip processor to off-chip memory such as DDR RAM. It is possible to observe transactions on this memory bus or even inject modified data that causes unwanted behavior from the system. As discussed in Section 2.1 the main focus of protection is bus attacks; in particular, protection against spoofing attacks, splicing attacks, and replay attacks are the main concern. Replay attacks are the most costly threat to defend against as it requires capturing an instance of time in which certain data is invalid. To combat this, the security model assumes our system's on-chip memory is secure against attackers and cannot be observed or tampered with. It is also assumed that the OS kernel can also be trusted and is resistant to attacks. Memory bus data observation and data injection is the only weakness that this model attempts to address. Data leakage and side-channel attacks such as differential power analysis attacks are beyond the scope of this model. DDR RAM is generally accessed very often and is significantly relied upon. It is very important that the design to protect memory does not provide large levels of performance overhead. Any additional performance slowdown is significant in the execution of a program. As described by Figure 4.1, a custom memory controller platform placed between the CPU and the DDR Memory facilitates the memory encryption and authentication. As speed is a priority, an FPGA's programmable logic is used to implement the controller in hardware. A ZedBoard™ development board with a Xilinx Zynq®-7000 All Programmable SoC is the target platform for this design. The design can be extended to any comparable platform that can make use of the full AXI-4 interface. As the ZedBoard is a low cost development kit, more sophisticated FPGA's may be able to run the design with higher clock frequencies and make even better use of the design.

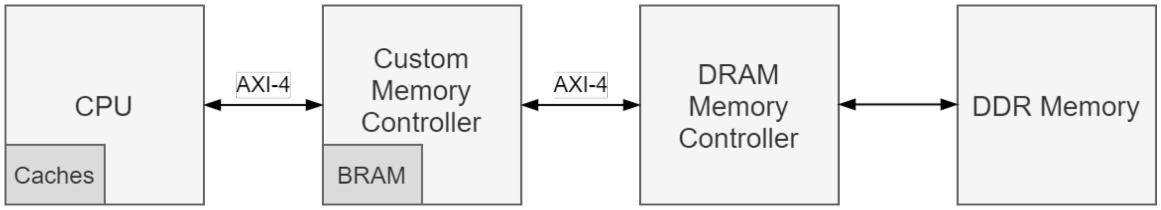

Figure 4.1: Target Security Model

4.2 AXI-4 Interface

The Advanced eXtensible Interface (AXI) is a widespread flexible high-performance bus interface that is commonly used in on-chip communication applications. Zynq processors incorporate high-performance full AXI-4 interfaces into their design in order to allow for easy communication with designs in the programmable logic. As a result of that fact, the AXI interface was chosen for the implementation of the authentication tree memory controller design. Many optional signals and features exist for the interface allowing for a versatile number of functionalities depending on the application [26]. AXI-Lite interfaces reduce the capabilities of AXI into simple 32-bit read and write transactions. Due to the complexity and versatility of the memory controller's design, a full AXI interface was instead chosen in order to provide support for a wider range of applications.

4.3 Memory Controller Pipeline

4.3.1 Encryption Pipeline

While individual components of the AXI-4 interface are not by themselves complicated, integrating a completely transparent memory controller design for each possible combination of AXI capabilities quickly increases the complexity. Aiming to reduce this complexity, the design presented here utilizes a heavily modified version of the AXI-4 transparent memory encryption pipeline provided in [25]. Figure 4.2 describes the original unmodified encryption only pipeline design [25].

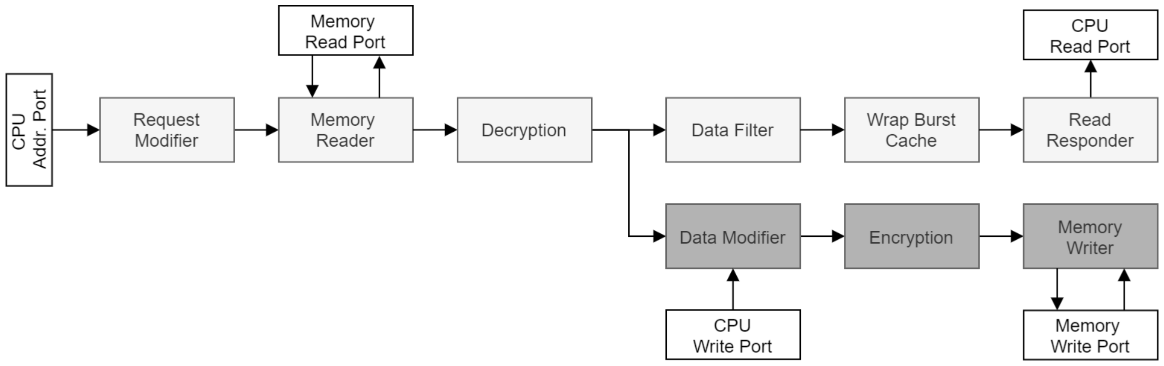

Figure 4.2: Transparent Memory Encryption Pipeline

The design incorporates an effective way to implement memory security for a variety of memory types with little effort required from a user. As the goal is to remain transparent to the user, the user of the secure memory will not have to perform any additional steps in order to perform memory transactions. A read-modify-write approach is followed. When a processor sends a transaction to the controller, first the controller will read and decrypt the requested data. If a write transaction was specified, the decrypted plaintext is then forwarded to a data modifier, which is responsible for merging the requested write data with the data retrieved from memory. Once the data block is properly modified with the written data, the entire block is sent to the encryption engine. After the data is properly encrypted depending on the cipher implemented, the final encrypted ciphertext is forwarded to memory to be written. If instead a read transaction was specified, after the decryption stage the decrypted plaintext is then forwarded to the processor. Before reaching the processor the plaintext data is filtered to remove any additional data that was necessary to store depending on the encryption scheme employed. Requests also support bursts as specified by the AXI protocol, which is handled by the wrap burst cache component in the pipeline.

4.3.2 Encryption and Authentication Pipeline

The encryption pipeline design provided by [25] was heavily modified in order to provide support for Dynamic Authentication Trees. Dynamic trees require additional tree metadata to first be read from memory in order to generate appropriate requests for new intermediate nodes. Therefore, the pipeline needs to return data read from memory to the component that generates requests. The modified pipeline including components required for authentication is described by Figure 4.3. The work presented in [25] also includes an implementation for a TEC-Tree design using a modified version of the encryption pipeline; however, due to the large difference in operation between a dynamic authentication tree and a TEC-Tree, the simple encryption pipeline was utilized as a base for the design instead. The TEC-Tree authentication pipeline is used as a performance comparison for the finished design.

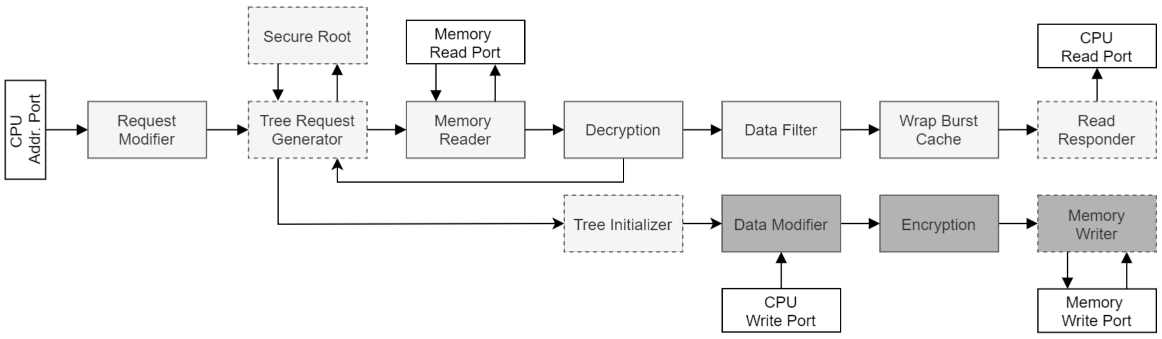

Figure 4.3: Transparent Memory Encryption and Authentication Pipeline

Due to the fact that read requests now need to be returned to the request generator, the pipeline has been further separated into two paths, a read request pipeline and a write request pipeline. A master state machine is used in the Tree Request Generator component in order to determine which tree nodes must be accessed and updated. True to its name, the Tree Request Generator then generates the appropriate requests for tree node processing into the read and write pipelines. When a request is received from the processor, the root number for the memory address is calculated. The root of the tree is checked to see if the data node has been accessed yet. The root itself is stored in BRAM that is embedded in the programmable logic, as it is can be considered secure against an attacker with physical access. If the root has not yet been accessed then the data for that tree needs to be initialized. A Tree Initializer component has been added to correctly populate the initial tree data with the data specified in Algorithm 3. The Tree Request Generator specifies whether or not a request is an initialization request. If this is the case, then the Tree Initializer calculates the appropriate data values based on the memory address. Otherwise, normal requests are simply forwarded through the initializer with no modification. Both the read and memory write responders were modified to ensure that the master AXI signal response was generated to the processor only when the original request was processed. This is necessary because, in the original encryption pipeline, only the original data block is accessed. The dynamic tree requires multiple intermediate requests to first be sent to process the tree data before processing the original request. Figure 4.4 displays the state machine used to process read requests.

The state machine remains at an idle state until a read request is received. Once a request is issued by the processor, the tree that is being requested to access is checked to ensure it has been accessed before. If it has not, the tree is initialized to a balanced tree state. Once the initialization is finished, the data node that contains the requested data is first read. The data node is then decrypted and the stored

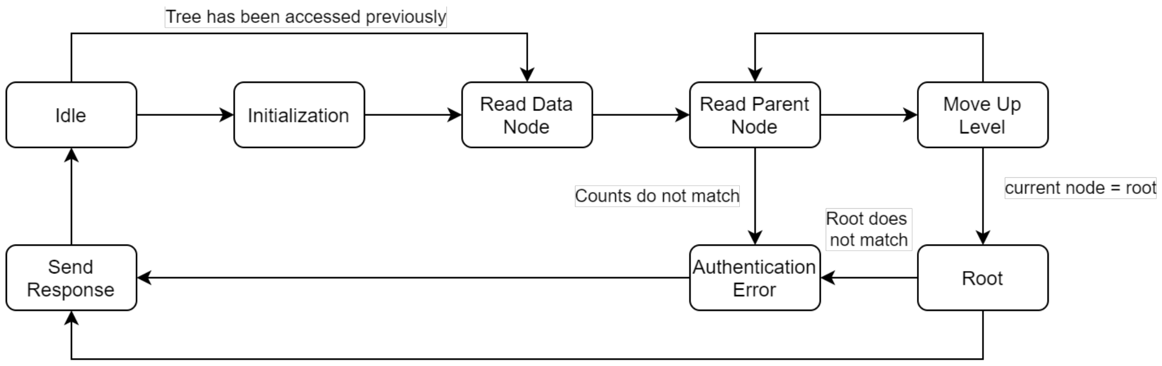

Figure 4.4: Read Request State Machine

metadata is used to determine which counter node must be read next as the parent node. That parent node is then read, and the store counts are compared to ensure data has not been tampered with. If the counts do not match, an authentication error is raised and sent to the processor. Otherwise, the process is repeated recursively, and the next parent node is read until the root of the tree is reached. After reaching the root node, the count contained in the root node is then compared to the secure root stored on-chip. If these counts do not match, the same authentication error process is completed and the processor is notified. Finally, if everything authenticates properly, the original data that was requested is extracted from the original data node and forwarded to the processor.

4.4 Ciphers and Modes of Operation

Customizability is one of the primary benefits of this design, and in that spirit, any cipher can be used in the encryption pipeline. The design tested here uses AES as the encryption engine. AES is a widely adopted standard cipher with quick timings that is able to be efficiently implemented on an FPGA. The only limitation on the cipher to be used is that the cipher mode of operation must propagate errors to later ciphertext blocks. Block cipher modes such as Electronic Code Book (ECB) and Counter Mode (CTR) do not propagate errors to ciphertext blocks beside the current block being decrypted. Cipher block chaining (CBC) mode was chosen for the test implementation as each block of ciphertext relies on all plaintext blocks that have been previously processed. As the payload of a data node is decrypted first, any change in the payload will result in changes to the metadata, which will then cause the authentication to fail. A significant drawback of the CBC mode is that each encryption and decryption operation is sequential, relying on the previously processed data block before the next block's processing can begin. Parallelization of the cryptographic operations is then not possible if it is desired to speed up the encryption or decryption of data.

CBC mode still requires the data size to be aligned with the limitations of the block cipher's size. In the case of using AES as a cipher, the data block must be a multiple of 128 bits. The implementation example provided in this thesis utilizes AES for both the data nodes and the counter nodes. In this case, the data nodes have a size of 80 bytes, which is aligned properly with this cipher, but the counter nodes have a size of 24 bytes which is misaligned by an extra 8 bytes. The CBC mode was then modified to take advantage of the ciphertext stealing (CTS) operation that allows for the processing of data blocks of any size. This is achieved by padding the last incomplete block with a portion of the second to last's block ciphertext and encrypting the padded block. Because the padded block's plaintext then includes the partial ciphertext of the second to last block, it is not necessary to store the full block in memory as it will be recovered later on decryption. Figure 4.5 demonstrates the behavior of a ciphertext stealing mode.

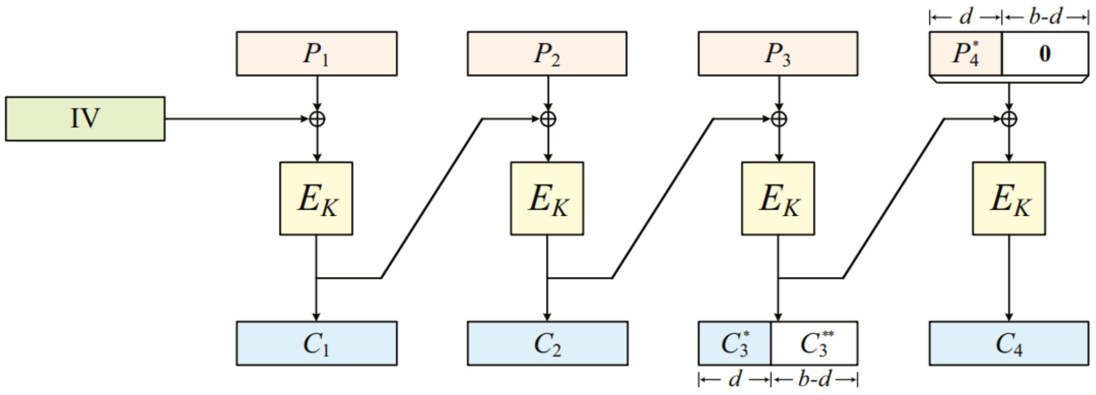

Figure 4.5: Ciphertext Stealing [1]

4.5 Results

The results presented in this section are based on an implementation of the proposed design written in the VHDL hardware description language. Synthesis was performed with Xilinx Vivado version 2020.1 and targeted the ZedBoard development kit (part xc7z020clg484-1) using the Xilinx Zynq-7000 All Programmable SoC. Due to the complexity of the programmable logic design and the limitations of the ZedBoard, the design was evaluated with a clock frequency target of 50 MHz. This limitation is due to the routing delay in the critical path of the implementation. The FPGA used is an entry-level model with a lower speed than more sophisticated platforms. Running this design on a higher level model will allow for a significant increase in clock frequency. For programmable logic utilization comparisons, the total number of resources available for various Xilinx FPGAs is displayed in Table 4.1. Part xc7z2020 is the SoC with FPGA used by the ZedBoard. Compared to the other FPGA parts available, the resources provided by the Zedboard are relatively small.

Three different designs were implemented for comparison. First the TEC-Tree was tested as it is currently one of the lowest overhead methods of providing encryption and authentication. The original dynamic skewed tree was also chosen for evaluation

Table 4.1: 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 performance for the proposed design should be similar. Each test performed was run with the same configuration of 1 MB of memory protected by 2048 trees with each leaf node protecting 64 bytes of memory. The Zynq processor uses an AXI data width of 32 bits, while the memory interface uses an AXI data width of 64 bits to allow for faster memory transactions. The memory controller handles the difference in memory sizes and generates the appropriate responses for both the master and slave AXI devices. For the scope of this design, a tree arity of two was tested for each configuration. The proposed dynamic tree algorithm currently does not support higher arities, but this can be expanded in future work.

The performances of the dynamic designs were evaluated using AES-128 in CBC mode with ciphertext stealing to match data block alignments. The TEC-Tree was evalued using the lightweight ASCON cipher in CBC mode, instead of AES. The particular version of the AES cipher implemented requires only 12 cycles for decryption and encryption operations on 128-bit data blocks. The ASCON cipher requires 6 clock cycles for its operation. AES has been chosen for evaluation as it is a standard and widely used cipher that lends itself well to an HDL implementation. Any cipher may be chosen for use, for example using a lightweight cipher will increase performance by requiring fewer clock cycles for each read and write operation. It is important to note however that the choice of cipher and mode of operation is important to ensure proper security. ECB and CTR modes are discouraged as the lack of error propagation eliminates the effectiveness of the block-level AREA authentication technique.

The 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. Figure 4.6 displays the configuration of the test environment for these performance tests.

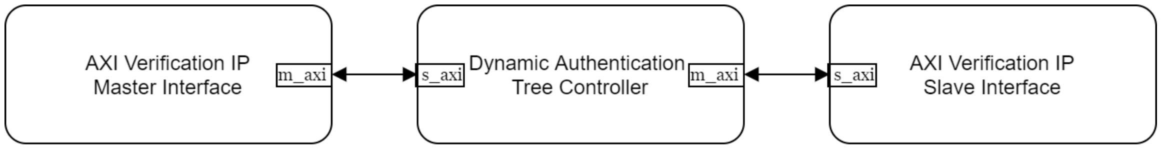

Figure 4.6: Test Environment Configuration

4.5.1 Synthesis Results

The design was synthesized in Xilinx Vivado version 2020.1 and compared to the implementation of the TEC-Tree [25] and the Dynamic Skewed Tree framework [21]. The TEC-Tree implementation uses an Ascon cipher for the encryption and decryption of data, while the Dynamic Trees both use AES-128 for their encryption engine. As displayed in Table 4.2, the method of encryption contributes to a large portion of the programmable fabric usage. In order to better compare the designs, the usage was measured with and without the encryption.

As expected, the TEC-Tree design utilizes the least number of each component as it uses the simplest design. Most of the differences in utilization can be accounted for by the added components for the dynamic tree request generation and initializa-

Table 4.2: Authentication Tree Synthesis Results

	TEC-Tree	Unordered DAT	Ordered DAT
No Encryption			
LUTs	3582	8521	12404
Flip Flops	3064	5953	8475
DSPs	0	15	21
BRAM Tile	1	1	1
Encryption			
LUTs	9398	11438	15321
Flip Flops	4349	6738	9260
DSPs	0	15	21
BRAM Tiles	1	1	1

tion. The state machine itself requires a significant number of flip-flops and LUTs to manage state transitions and state outputs. While the TEC-Tree design can simply generate all necessary requests without reading data from memory, both DAT designs require logic to both store read data and translate the read data to appropriate requests for tree traversal and modification. The unordered DAT design also requires fewer resources to implement than the 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 unordered DAT design, the ordered design requires three different algorithms to be implemented based on the current state of the tree to be restructured. Additionally, the ordered DAT's state machine contains a larger number of states that need to be implemented. The unordered algorithm is only concerned with the current node, the sibling node, the parent node, and the uncle node. While the ordered algorithm may need to access the grand-parent node, the grand-uncle node, and even the great grand-parent node's data. This results in an additional state for both generating the requests and reading the node data back after the data has been retrieved and decrypted. Additionally, each potential node requires more flip-flops to store the metadata while it is processed and modified. For perspective, Table 4.3 contains the utilization percentage of each component available in the xc7z2020 SoC.

Table 4.3: Authentication Tree Synthesis Results Percentage Utilized

	TEC-Tree	Unordered DAT	Ordered DAT
No Encryption			
LUTs	7.75%	18.44%	26.85%
Flip Flops	3.32%	6.44%	9.17%
DSPs	0%	25%	35%
BRAM Tiles	1.05%	1.05%	1.05%
Encryption			
LUTs	20.34%	24.76%	33.16%
Flip Flops	4.71%	7.29%	10.02%
DSPs	0%	25%	35%
BRAM Tiles	1.05%	1.05%	1.05%

The number of LUT's used for all implementations is between 7% to 27%, and the number of Flip Flops used is between 3% to 10%. While the DAT's utilization is considerably higher than the TEC-Tree implementation, the overall usage is still fairly low as there are many more LUTs and Flip Flops available for additions to the design or other programmable logic. This programmable logic utilization is notably low if compared to any of the higher grade models listed in Table 4.1. Interestingly, the DAT design both use DSPs for their synthesis that the TEC-Tree does not require. The DSPs are used in the calculation of the write-back addresses for each node after their metadata has been modified. These address calculations are simple addition and multiplication operations using the initial tree start address and the node metadata. If the use of DSPs is not desired, the HDL can be modified or the synthesis tools can be specified to not use DSPs.

4.5.2 Implementation Results

The design's netlist generated from synthesis was then implemented for the xc7z2020 SoC. Default Vivado 2020.1 implementation and optimization settings were used. The resulting utilization is contained in Table 4.4.

Table 4.4: Authentication Tree Implementation Results

	TEC-Tree	Unordered DAT	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

As expected, the utilization results are very similar to the synthesis results. Overall, the component usage is slightly less than that of synthesis due to the optimizations that took place. Of particular note, the proposed ordered DAT design's synthesis reports 1,2404 LUTs and 8,475 flip flops used, but the implemented design only requires 1,1649 LUTs and 7,582 flip flops. Additionally, the percentage of components used was calculated and displayed in Table 4.5. Naturally, the percentage utilization for each component is reduced as well, with the Ordered DAT still requiring the most resources. Despite this large utilization, the Ordered DAT design uses a fairly low number of components given the number of resources available. While the synthesis and implementation results for all three designs are limited to 50 MHz, it is worth noting that a target device with more resources and better timings will be able to achieve higher frequencies and better performance overheads.

Table 4.5: Authentication Tree Implementation Results Percentage Utilized

	TEC-Tree	Unordered DAT	Ordered DAT
No Encryption			
LUTs	7.08%	18.17%	25.21%
Flip Flops	3.13%	6.12%	8.21%
DSPs	0%	25%	35%
BRAM Tiles	1.05%	1.05%	1.05%
Encryption			
LUTs	19.52%	24.35%	31.40%
Flip Flops	4.45%	6.95%	9.04%
DSPs	0%	25%	35%
BRAM Tiles	1.05%	1.05%	1.05%

4.5.3 Performance

The designs for all three methods were evaluated for performance in various conditions with different memory access patterns. The TEC-Tree design tested was provided by [25], while the dynamic tree implementations were manually implemented and written in VHDL based on the framework outlined in Section 4.3. As previously discussed, due to timing limitations in the implementation on the target xc7z2020 SoC, each design was run at 50 MHz. 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 memory to be protected was configured for 256 MB and a data block size of 64 bytes for each design

The dynamic nature of the proposed tree designs requires different scenarios to fully explore the different possible performances the design may have. It is important to test the worst and best case scenarios for the design, in order to help determine which use cases it may perform better in. 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 makes it important to note. The results of this performance are displayed in Figure 4.7. Each test was run with 10,000 write and read access to dynamically construct the tree and test timings. While the TEC-Tree read and write timings remain consistent throughout the tests, the dynamic trees timings consistently change. In order to provide valid results, the total average read and write times were calculated for comparison.

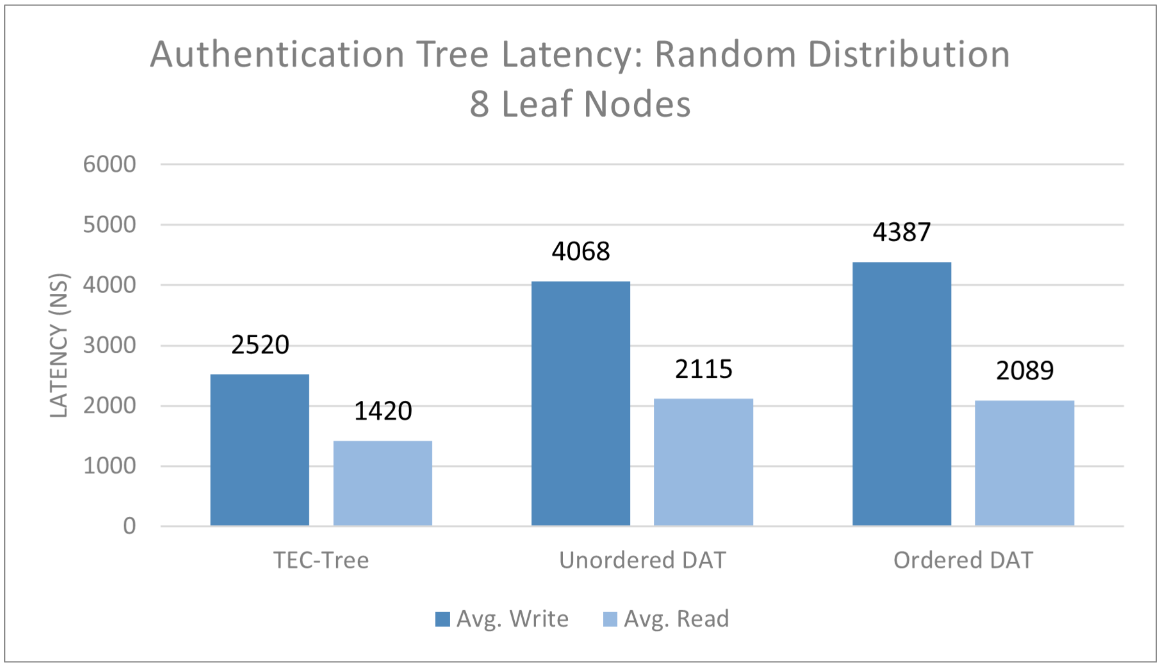

Figure 4.7: Authentication Tree Latency: Random Distribution with 8 Leaf Nodes

As expected, the dynamic trees here perform worse compared to the TEC-Tree's performance. As a result of the uniform randomly distribution of memory accesses, the law of large numbers states that each data node is accessed approximately equally. The end result would then be a balanced tree that has additional data accesses and authentication computations compared to the TEC-Tree. The additional rebalancing operations causes significant slowdown in write operations for both dynamic authentication trees, as each rebalance will ultimately be reversed as the tree is uniformly accessed in the future. Write operations for the unordered dynamic tree were an average approximately 1.61 times slower than a static TEC-Tree, while the ordered dynamic tree's write accesses were 1.74 times slower. Despite not including extra authentication calculations, read operations were slower for both dynamic trees as well. This can be accounted for by both the varying levels of data nodes as the tree is rebalanced and the fact that dynamic trees require additional metadata to be read for each read. Due to the additional metadata that must be accessed to traverse the tree, even if a data node in the dynamic tree structure maintained the same level of access as the TEC-Tree, the read would be slightly slower than the TEC-Tree's read. The average read speed for the unordered dynamic tree was 1.49 times slower than the TEC-Tree's read speed and the average read speed for the ordered dynamic tree was 1.47 times slower. These read speeds aren't as slow as the average write speed because each read does not need to additionally access the uncle node for rebalancing checks as each write operation requires.

These limitations of the dynamic tree design cause a uniform random distribution of accesses to be the absolute worst-case scenario for authentication performance. Luckily, in reality a processor with a completely random memory access pattern is extremely rare. If this access pattern was employed in a real-world scenario, it would not be recommended to utilize a dynamic tree for authentication, but instead rely on a balanced tree design.

The previous test was performed with a total of eight leaf nodes protecting memory. With this configuration the TEC-Tree's leaf nodes would be at level $\log_2 N = \log_2 8 = 3$ where N is the number of counter nodes and with the 0th level being the root of the tree. Both dynamic tree's leaf node levels vary between the 1st level and the level equivalent to the number of data nodes in the tree minus one in the most extreme case. In order to test the impact that increasing the number of data nodes and levels of the tree might have, the same test was rerun with the trees' configurations set to include 16 leaf nodes per tree. The results of that this test is contained in Figure 4.8.

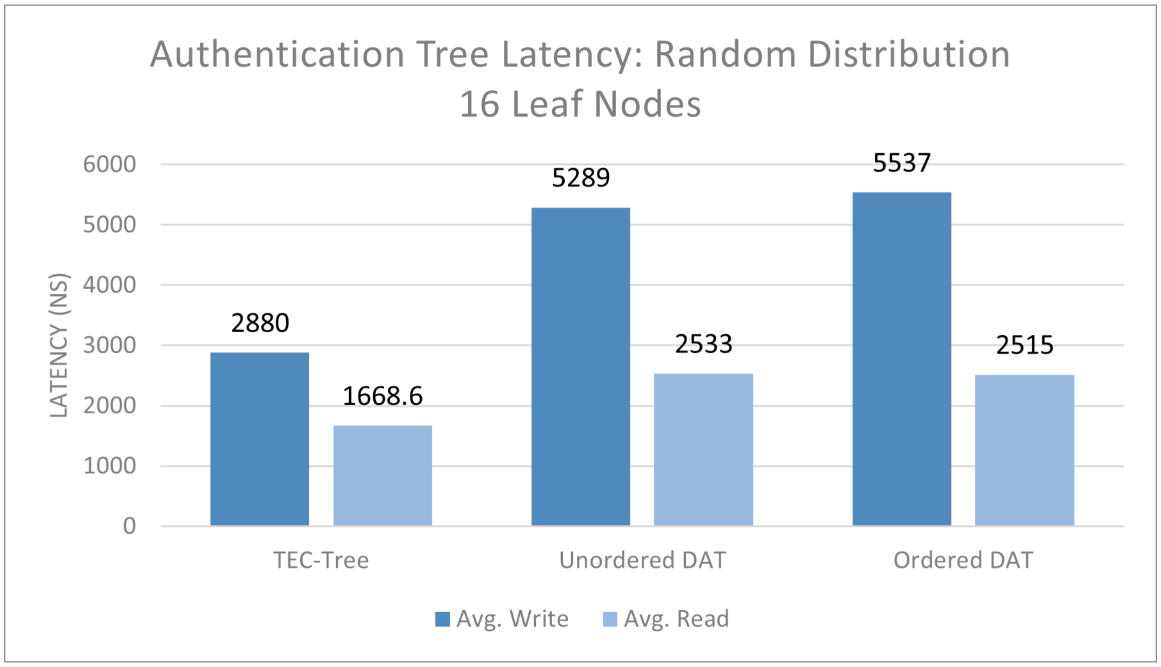

Figure 4.8: Authentication Tree Latency: Random Distribution with 16 Leaf Nodes

Doubling the number of leaf nodes of a balanced tree will increase the number of levels of the tree by one, as shown by $\log_2 16 = 4$. In a dynamic tree, the maximum level increases significantly more as the max level for a dynamic tree is $N - 1 = 16 - 1 = 15$. It can be then be inferred that the dynamic tree's performance in both the extreme worst and best-case scenarios is further heightened by the number of leaf nodes in the tree. In the worst-case scenario, the dynamic tree adds an additional $15 - 4 = 11$ extra levels for the tree traversal. Oppositely, in the best-case scenario, the dynamic tree reduces the number of levels to traverse by $4 - 1 = 3$. While the worst-case scenario does indeed add a significant number of levels to access compared to the number reduced by the best-case scenario, due to traditional memory access patterns the worst-case scenario will almost never happen. The weighted nature of the tree and memory access means that the nodes that are stored that far down the tree will be very infrequently accessed compared to those nearer to the root of the tree. The time saved by accessing these nearer nodes then may outweigh these extra performance overheads. As Figure 4.8 demonstrates, increasing the number of nodes increases the write time of the TEC-Tree by 1.14 times and increases the read time by 1.18 times. The unordered dynamic tree's write time is increased by 1.3 times and the ordered is increase by 1.26 times. For read times, the unordered is increased by 1.197 times, and the ordered is increased by 1.2 times. As expected, the dynamic trees' worst-case scenario performance is impacted more significantly more than the TEC-Tree's performance when increasing the number of leaf nodes in the tree structures.

The second test applied to each authentication tree was designed to test the best-case scenario for a dynamic tree, as opposed to the first test that introduces the worst case. For this test, only a single data node is accessed to allow the dynamic tree to restructure it to the top of the tree. 10,000 reads and writes were applied to this single node and the average read and write times were recorded for each design, as displayed in Figure 4.9.

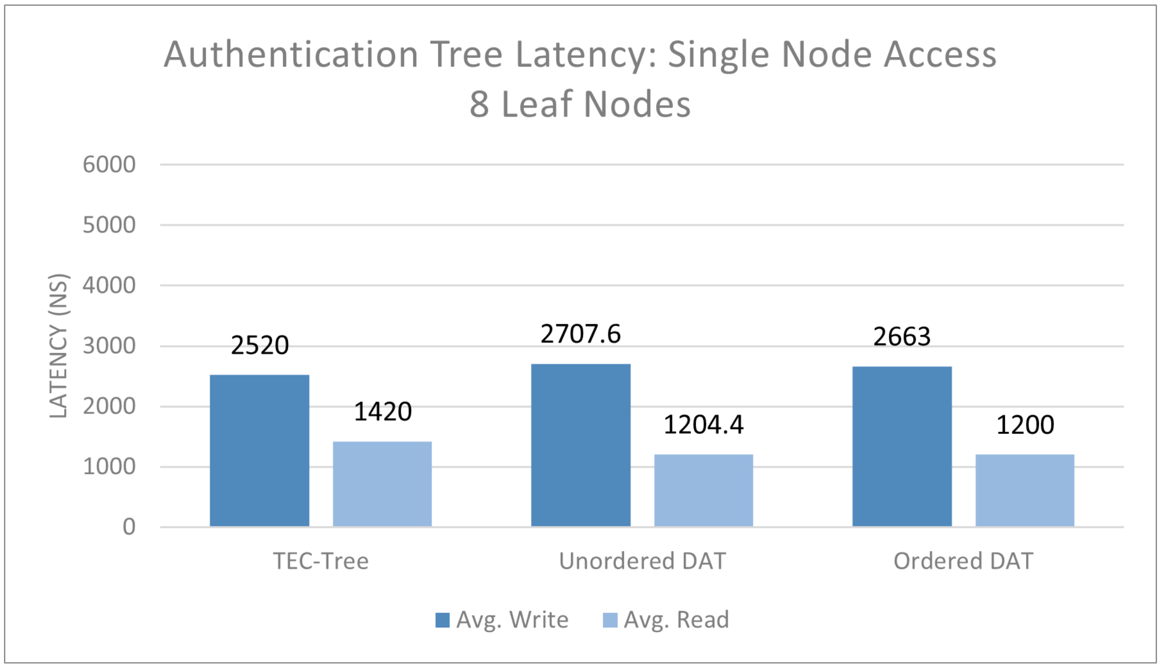

Figure 4.9: Authentication Tree Latency: Single Node Access with 8 Leaf Nodes

As the TEC-Tree's tree remains balanced, read and write access times remain consistent no matter the memory access pattern. The read and write latencies are then equivalent to those recorded in the first test case. The dynamic trees conversely increase in performance significantly, with the read speeds beating the read speed of the TEC-Tree in both dynamic tree designs. For the unordered DAT, the write latency achieves a 1.5 times speedup compared to the original test case, and the read latency gains a speedup of 1.76 times. Similarly, the ordered DAT achieves a write speedup of 1.65 times and a read speedup of 1.74 times. In this scenario, the write speeds for each dynamic tree are only slightly slower than that of the TEC-Tree, while the read speeds are significantly less. The performance of the dynamic trees would then be favored over the TEC-Tree in scenarios where the number of read accesses outweighs the write operations. Although write-heavy patterns may suffer slightly. The experiment was also repeated with 16 leaf nodes to measure the effects of increasing the number of possible tree levels on each tree type. Figure 4.10 contains the results of this test.

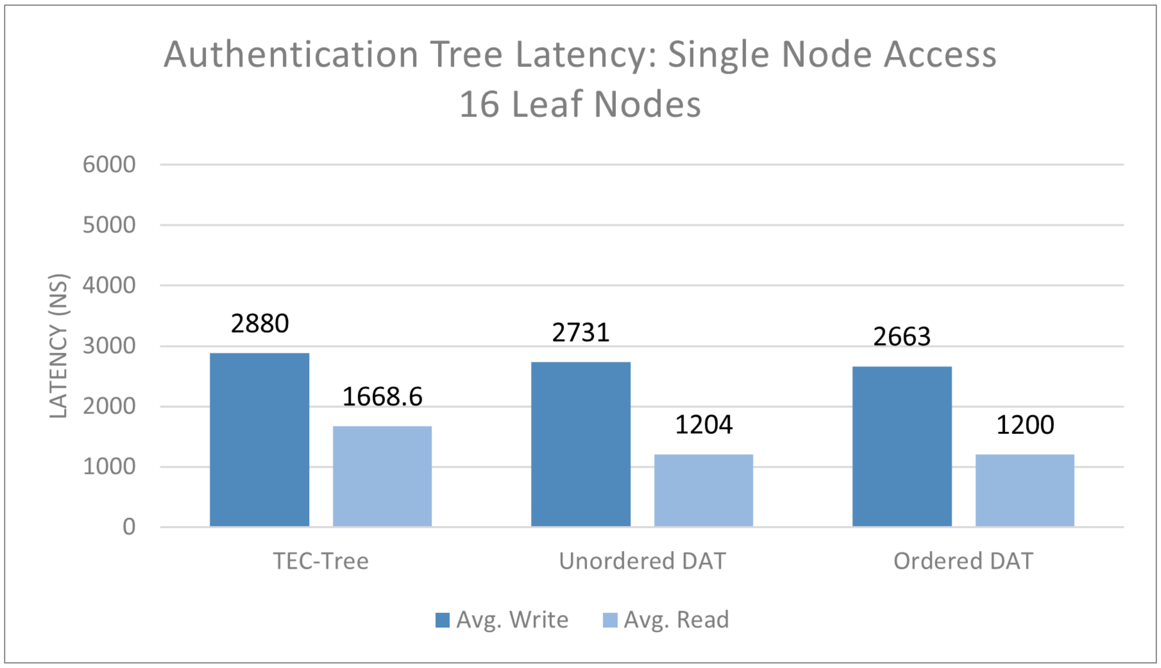

Figure 4.10: Authentication Tree Latency: Single Node Access with 16 Leaf Nodes

The TEC-Tree's read and write speeds are the same as the read and write speeds in the 16 leaf node random distribution test. This behavior is expected because the TEC-Tree does not rebalance. The dynamic authentication trees' performance remains the same as the previous test with eight nodes. This result can be attributed to the fact that when accessing a single node, it will quickly be rebalanced to the top level of the tree. Even though adding additional nodes will add an extra rebalance operation for the node to reach the top of the tree, with a large enough number of memory accesses this additional overhead is quickly averaged out and becomes insignificant. Both the unordered and ordered DAT beat the TEC-Tree's performance in read and write operations as the TEC-Tree must access four levels each time while the DATs only need to access two, the node itself and the root. In this case the unordered DAT gains a speedup of 1.05 times on write operations and a speedup of 1.39 times on read operations. The ordered DAT gains a speedup of 1.08 times on write operations and a speedup of 1.39 times on read operations. While the previous two tests represent the worst and best-case scenarios for the dynamic tree's performance, a 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. Figure 4.11 demonstrates the results of accessing a large array in memory that spans the entire tree size.

Once again, the read and write speeds of the TEC-Tree remain consistent with previous tests. Naturally, the performance of both dynamic trees falls in-between the cases presented by the previous two tests. Unfortunately, both designs do not quite achieve the same level of performance as the TEC-Tree. The unordered DATs write speeds are 1.44 times slower and the write speeds are 1.23 times slower. The ordered DAT achieves an average write speed that is 1.5 times slower and an average read

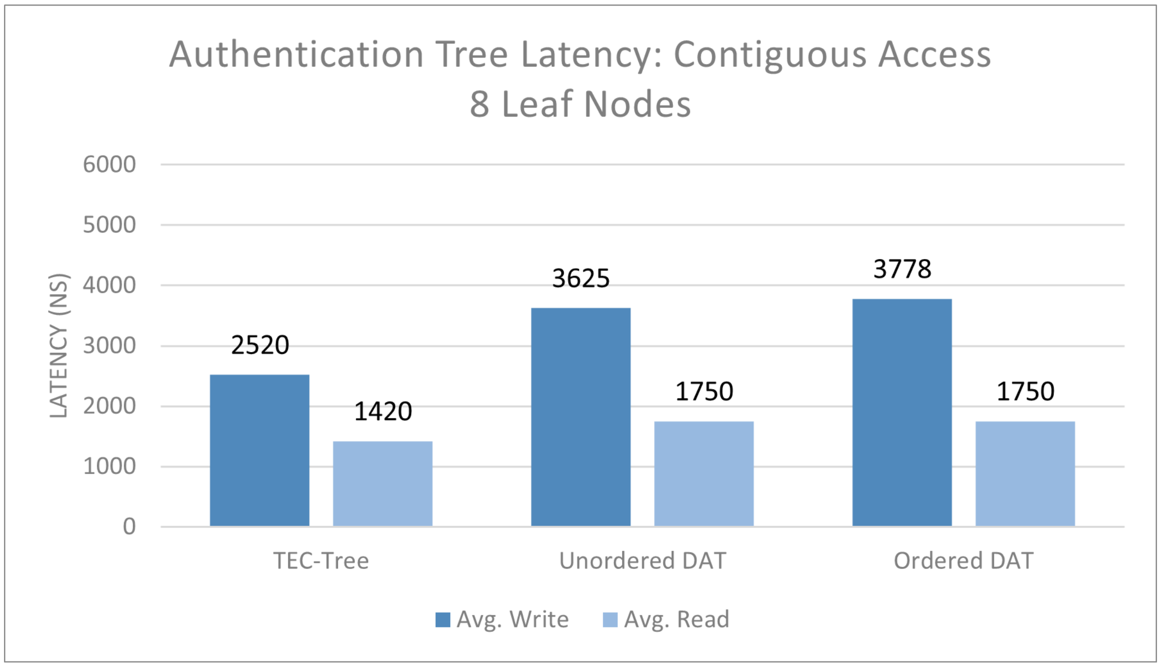

Figure 4.11: Authentication Tree Latency: Contiguous Access with 8 Leaf Nodes

speed that is also 1.23 times slower. This difference in performance can be attributed to the fact that in this memory access pattern when the first data block is being accessed it quickly shifts the rest of the data blocks further down in priority than the current one. Once the next block is accessed, this happens again until the data block size is exceeded and the process is repeated. However, if this array were to be accessed again in the future, the performance would be increased as the area of memory has been accessed before. To compare the performance impact of increasing the number of nodes in the dynamic tree, the test was run again with 16 leaf nodes, and the results are displayed in Figure 4.12.

Interestingly, because the array is accessed enough times to maintain the same tree level configuration for the nodes protecting the array's memory, the performance of the read and write operations for the dynamic trees remains the same as the previous test with eight leaf nodes. While in this scenario the dynamic trees still perform worse than the TEC-Tree, if the node count is continued to be increased the dynamic trees will overtake the TEC-Tree in performance. Intuitively, this makes sense as the

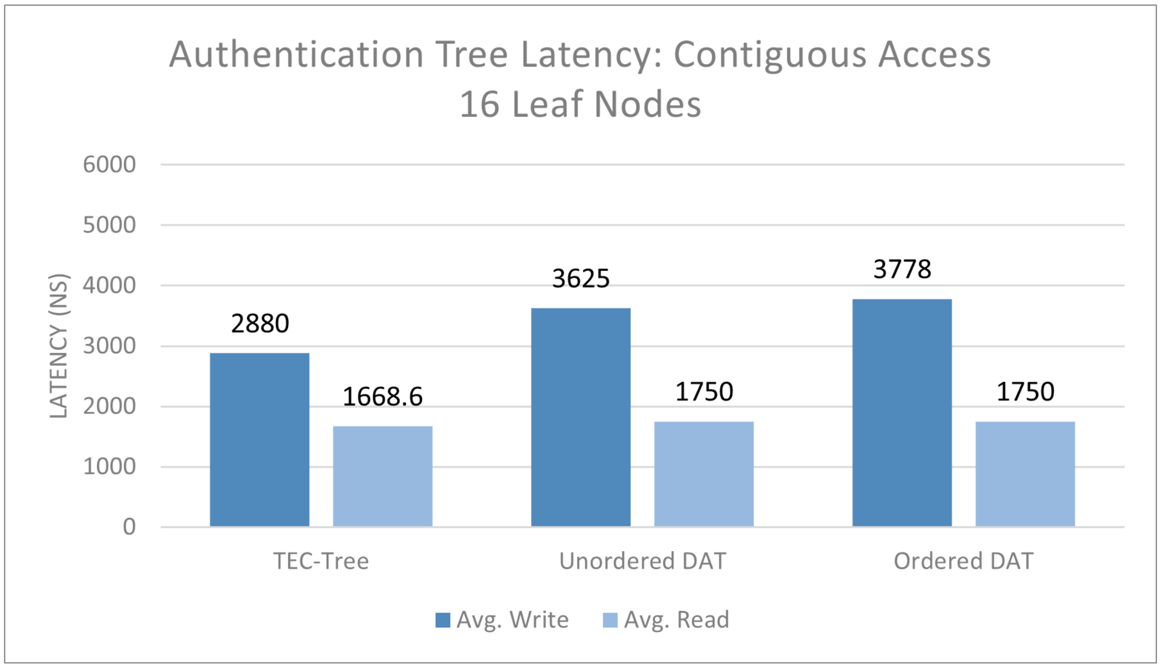

Figure 4.12: Authentication Tree Latency: Contiguous Access with 16 Leaf Nodes

TEC-Tree's nodes are equally level, which means nodes that are never accessed are as equally weighted as those that are accessed the most frequently. Increasing the node count for the dynamic tree allows the nodes that are never accessed to be much further down on the tree than the TEC-Tree's unaccessed nodes.

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. As processors generally are more likely to access memory that has already been accessed, this dynamic structure has the potential to massively outperform TEC-Trees. According to the results of the tests presented previously, the DAT has a larger advantage over the TEC-Tree in configurations with a larger number of leaf nodes per tree. The application that an authentication tree is used in can then be evaluated for memory access patterns, storage requirements, and memory performance requirements in order to tailor a specific tree type to that application. Additionally, the design presenting in this thesis is a proof of concept with much room for improvement. Further optimizations and the addition of tree node caches may allow the dynamic tree structure to overtake the performance TEC-Tree in general use applications. For easier comparison, a summary of the results of the each timing test is contained in Table 4.6 and Figures 4.13and 4.14.

Table 4.6: Summary of Timing Results

Access Pattern	Leaf Nodes	TEC-Tree (ns)		Unordered DAT (ns)		Ordered DAT (ns)	
		Write	Read	Write	Read	Write	Read
Test 1	8	2520	1420	4068	2115	4387	2089
	16	2880	1668.6	5289	2533	5537	2515
Test 2	8	2520	1420	2707.6	1204.4	2663	1200
	16	2880	1668.6	2731	1204	2663	1200
Test 3	8	2520	1420	3625	1750	3778	1750
	16	2880	1668.6	3625	1750	3778	1750

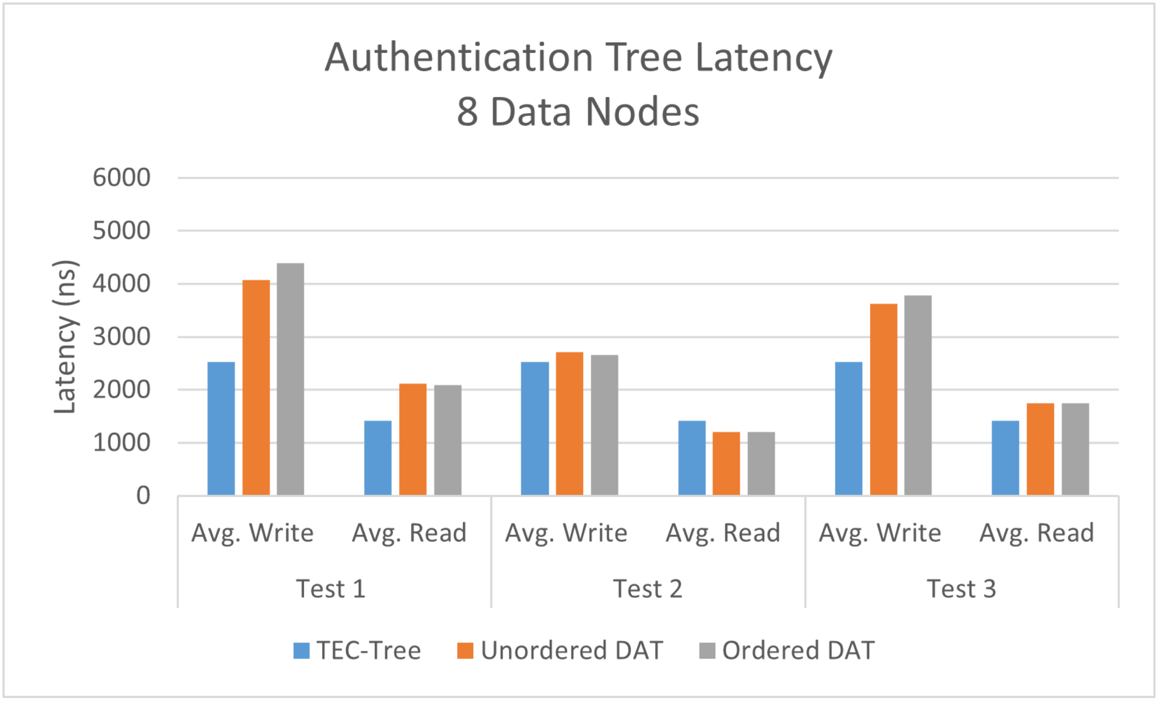

Figure 4.13: Summary of 8 Node Timing Results

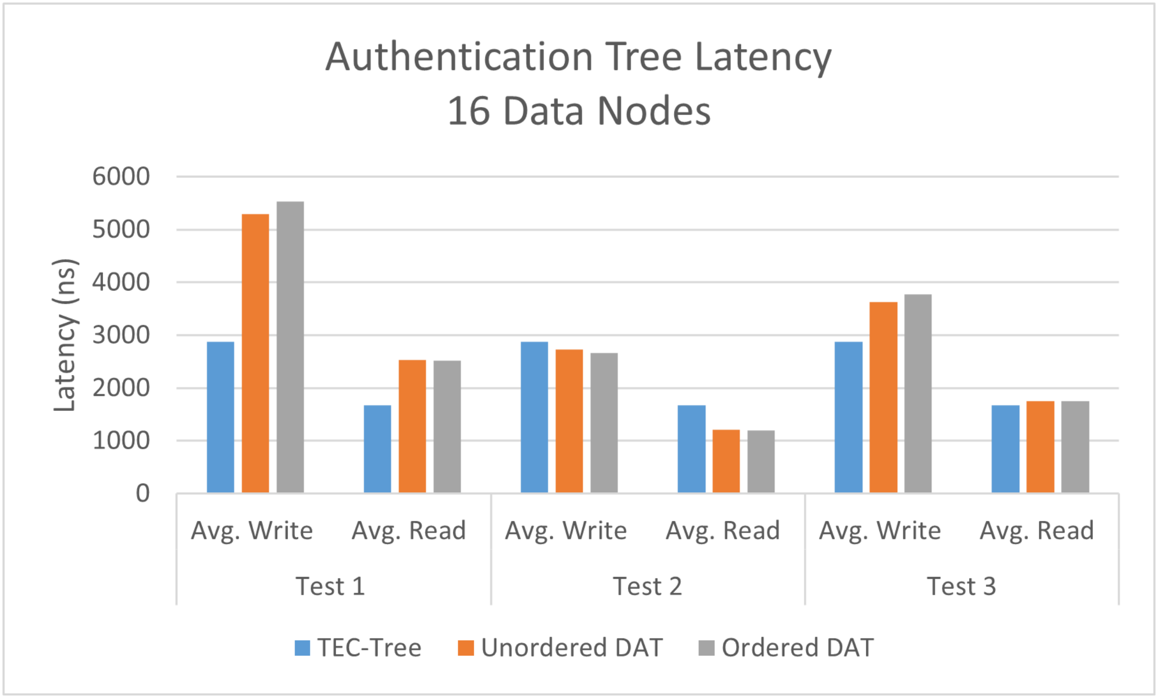

Figure 4.14: Summary of 16 Node Timing Results

Chapter 5

Conclusions and Future Work

5.1 Future Work

While the work presented in this paper is complete for multiple configurations and thoroughly tested, there are still many improvements and optimizations that can improve upon the design significantly. This framework is a proof of concept for a new customizable encryption and authentication pipeline based on adding improvements and options for scalability and customizability of previous works. As detailed in the previous analysis, the new dynamic ordered algorithm focused on in this work is best used in specific applications However it can certainly be improved upon in order to provide more use cases, optimality, and flexibility in future designs.

5.1.1 Tree Arity

Perhaps one of the greatest limiting factors of both the unordered and ordered dynamic tree algorithms is the arity of the constructed tree. The largest challenge to overcome would be to provide more rebalancing algorithms for each case that might be possible if retaining the order of the leaf nodes is important. While it is definitely possible to extend the algorithms for customizable arities, the number of checks added towards restructuring the tree would increase the overhead for write operations significantly. For example, if the arity of the tree was increased to four instead of two, there would be three more additional uncle nodes that must be read from memory, decrypted, and compared for rebalancing. In addition, when a rebalance does occur, additional performance slowdown would be incurred in order to update multiple node relations per counter node. As for metadata storage, there would be more metadata stored for the position of each node, instead of just storing a single bit for whether the node is a left or right node. It would be expected that increasing the tree's arity would overall decrease the average height of each node, thus decreasing the total number of levels that need to be accessed for both read and write operations. All read operations would then be significantly increased in speed; however, write operations would be slowed down with the additional tree rebalancing checks and operations. Depending on the memory access patterns, this could increase overall operational speed or the extra overhead might be too significant to be useful. As with any increase in authentication tree arity, the overall memory storage would also be increased due to the added relationships that need to be tracked for each node.

5.1.2 Encryption and Decryption Parallelization

In its current form, the design framework presented only utilizes a single instance of a cipher's encryption and decryption engines. It is possible to instantiate multiple of these components in order to encrypt and decrypt multiple data blocks at the same time if there are enough resources available in the programmable logic fabric. Unfortunately, the block level AREA technique employed in this design relies on error propagating cipher block modes of operation. These error propagating modes rely on previously processed data blocks in order to begin processing the next block, allowing for only sequential operation and not enabling parallelization. Separate nodes could be encrypted and decrypted in parallel if that information was available to the design at the same time. Generally, the encryption and decryption process is already quicker to perform than it is to retrieve additional information from memory. A dual-port

RAM could potentially be utilized to read or write multiple tree nodes at once, and then multiple cipher components could be used to process this data at the same time.

5.1.3 Caches and Leaf Node Ordering

Caches have widely been used to store frequently accessed data to enable faster retrieval of that data to speed up memory operations. This technique has been expanded to multiple forms of authentication trees in the past, including the work presented in [19]. This framework currently does not support caching for the dynamic tree information, but this can be implemented in the future. There are multiple different types of caches that could be utilized to speed up the overall design's functionality. Tree information itself could be stored in a cache in order to prevent the need to reread more frequently accessed nodes from memory every time the tree is accessed. As most of the slowdown in the dynamic tree design is caused by requiring the information from nodes to be read to traverse the tree, caching these nodes would provide a considerable increase in performance. A second type of cache could be implemented that caches payload data. The payload data would be the actual data requested to be written to memory by the processor. This second cache could be implemented instead of or in conjunction with tree node caching. This would help increase performance significantly as this data would be more readily available and external memory would not have to be accessed every time data is read or modified.

Considering that processors are generally more likely to access data that has already been accessed, a dynamic tree design with caches could be used to provide extremely minimal performance overhead compared to traditional authentication tree methods. Additionally, the ordered dynamic algorithm proposed in this paper could potentially increase cache performance by providing the caches with information about node order and relationships. For example, a cache could predict that a large array is being accessed in the running program. The nodes are ordered and the cache could preemptively store the sequential data blocks that will be accessed in the future.

5.1.4 Compiler Assistance

Instead of initializing each authentication tree to a balanced tree on startup, the compiler could communicate memory access patterns to the programmable logic or a bootloader before the program begins operation [27]. In the case of a dynamic tree, this would allow improvements in authentication speeds to be available from the very beginning of the program's execution as opposed to needing some run-time for the restructuring to begin. While the dynamic algorithm here performs a form of dynamic analysis for memory access patterns, a compiler can perform a static analysis of the program to be run at compile time. This static analysis can then be used to initialize the tree based on expected memory accesses. In the case that the compiler's static analysis does not provide a good prediction, the dynamic tree algorithms can then be used in order to correct the tree's optimization on the fly. Additionally, this analysis could potentially prevent unnecessary rebalancing overhead by telling the dynamic tree to not rebalance on conditions that will need to be rebalanced back to the previous configuration.

5.2 Conclusions

The importance of embedded security continues to rise as devices with important functionality steadily increases in usage. The work presented includes an authentication tree design that improves upon existing techniques to provide an increase in performance while providing adequate protection against attackers with physical access to a device. While current memory authentication techniques are expensive, this design allows for further customizations to further suit the specific application it is to be employed in. Additionally, a transparent memory controller that allows for encryption and authentication is provided. The memory controller contains a large number of customization and support for platforms with different memory speed and size requirements. A user is able to integrate the design in their system without a large amount of prior knowledge or experience.

Bibliography

[1] P. Rogaway, M. Wooding, and H. Zhang, "The security of ciphertext stealing," in *Fast Software Encryption*, A. Canteaut, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 180–195.

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

[3] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in Embedded Systems: Design Challenges," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, p. 461–491, Aug. 2004.

[4] R. Want, B. N. Schilit, and S. Jenson, "Enabling the Internet of Things," *Computer*, vol. 48, no. 1, pp. 28–35, 2015.

[5] S. Ziegler, "Internet of Things," in *Internet of Things Security and Data Protection.* Springer, Cham, 2019.

[6] R. Vaslin, G. Gogniat, J.-P. Diguet, E. Wanderley, R. Tessier, and W. Burleson, "A security approach for off-chip memory in embedded microprocessor systems," *Microprocessors and Microsystems*, vol. 33, no. 1, pp. 37–45, 2009, selected Papers from ReCoSoC 2007 (Reconfigurable Communication-centric Systems-on-Chip).

[7] T. Stapko, "Chapter 5 - Embedded Security," in *Practical Embedded Security*, T. Stapko, Ed. Burlington: Newnes, 2008, pp. 83–114.

[8] R. Vaslin, G. Gogniat, J.-P. Diguet, E. Netto, R. Tessier, and W. Burleson, "A Security Approach for Off-chip Memory in embedded Microprocessor Systems," *Microprocessors and Microsystems*, vol. 33, pp. 37–45, 02 2009.

[9] J. Schaad, "Use of the Advanced Encryption Standard (AES) Encryption Algorithm in Cryptographic Message Syntax (CMS)," RFC3565, USA, Tech. Rep., 2003.

[10] E. Leontie, O. Gelbart, B. Narahari, and R. Simha, "Detecting memory spoofing in secure embedded systems using cache-aware FPGA guards," in *2010 Sixth International Conference on Information Assurance and Security*, 2010, pp. 125–130.

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

[12] 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.

[13] 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.

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

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

[16] R. C. Merkle, “Protocols for Public Key Cryptosystems,” *1980 IEEE Symposium on Security and Privacy*, pp. 122–122, 1980.

[17] B. Gassend, D. Clarke, M. van Dijk, S. Devadas, and E. Suh, “Caches and Merkle Trees for Efficient Memory Authentication,” in *High Performance Computer Architecture Symposium*, 09 2002.

[18] W. E. Hall and C. S. Jutla, “Parallelizable Authentication Trees,” in *Selected Areas in Cryptography*, B. Preneel and S. Tavares, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 95–109.

[19] 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.

[20] 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.

[21] 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.

[22] D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[23] T. C. Hu and J. D. Morgenthaler, “Optimum alphabetic binary trees,” in *Combinatorics and Computer Science*, M. Deza, R. Euler, and I. Manoussakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 234–243.

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

[25] 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.

[26] ARM, "AMBA AXI and ACE Protocol Specification," Arm Holdings, developer.arm.com, Tech. Rep., 2019.

[27] V. Nagarajan, R. Gupta, and A. Krishnaswamy, "Compiler-Assisted Memory Encryption for Embedded Processors," in *High Performance Embedded Architectures and Compilers*, K. De Bosschere, D. Kaeli, P. Stenström, D. Whalley, and T. Ungerer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 7–22.