Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

5-2023

Memory Protection with Cached Authentication Trees

Andy Belle-Isle
atb1317@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Belle-Isle, Andy, "Memory Protection with Cached Authentication Trees" (2023). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Memory Protection with Cached Authentication Trees

Andy Belle-Isle

Memory Protection with Cached Authentication Trees

Andy Belle-Isle

May 2023

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

Memory Protection with Cached Authentication Trees

Andy Belle-Isle

Committee Approval:

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

Dr. Cory Merkel Date
Department of Computer Engineering

Dr. Stanisław Radziszowski Date
Department of Computer Science

Abstract

The use of embedded systems and the amount of data they process is rapidly growing in the modern information age. Given physical access to a device, an attacker can monitor the signals between the CPU and Memory to intercept, and possibly even inject new data into the system. A variety of attacks are possible including, replay, spoofing, and splicing attacks, each one threatening the safety of the system. Ensuring this data is intact is imperative, and as physical protection is difficult, data protection hardware is a must.

Protecting memory was researched in the past, and there are several methods of achieving it, with techniques such as memory encryption, memory hashes, and message authentication codes. While these achieve the desired effect, they do it at the cost of performance, memory usage, and additional hardware. To overcome these concerns, authentication tree designs have been proposed to protect memory with reduced overhead. For example, static tree designs such as TEC-Trees have been proven effective in the past, but have limited performance in certain access patterns (workloads). Most recently proposed dynamically balanced trees provide an additional solution with improved performance in certain workloads, however; with its own additional limitations. This research built on the top of the dynamic tree design by integrating tree node caches and evaluating the improved viability of the dynamic authentication tree (DAT) approach.

The design was implemented on a Xilinx Zynq-7000 SoC that used a hard processing system core to communicate with the fabric-based memory protection controller. The addition of caches to the dynamic authentication tree design increased the performance enough to perform similarly to TEC-Trees. As expected, in certain memory access patterns, such as those that repeatedly accessed a group of common memory locations, the cache-added DAT was able to outperform both the original design, and TEC-Tree based designs.

Contents

Signature Sheet **i**

Abstract **ii**

Table of Contents **iii**

List of Figures **vi**

List of Tables **vii**

List of Listings **viii**

Acronyms **ix**

1 Introduction **2**
- 1.1 Motivation 2
- 1.2 Objective 4

2 Background **5**
- 2.1 Physical Memory Attacks 5
  - 2.1.1 Replay Attacks 5
  - 2.1.2 Spoofing Attacks 6
  - 2.1.3 Splicing Attacks 6
- 2.2 Memory Protection 6
  - 2.2.1 Hashing 7
  - 2.2.2 Message Authentication Codes 7
  - 2.2.3 Authenticated Encryption with Associated Data 7
  - 2.2.4 Block-Level AREA Authentication 8
- 2.3 Authentication Trees 8
  - 2.3.1 Data Authentication Methods 9
  - 2.3.2 Authentication Tree Performance 9
  - 2.3.3 TEC-Tree 10
  - 2.3.4 Dynamic Authentication Trees 12
- 2.4 Authentication Method Comparison 19
- 2.5 Authenticated Memory Controller 20

3 Cached Authentication Trees 23
3.1 Authentication Tree Caching . . . 23
3.1.1 Memory Caching . . . 23
3.1.2 Keystream Caching . . . 24
3.2 Cache Reasoning . . . 26
3.2.1 Dynamic Authentication Tree Caching . . . 27
3.3 Cache Architecture . . . 27
3.3.1 Caching Indexing . . . 28
3.3.2 Cache Read Architecture . . . 30
3.3.3 Cache Write Architecture . . . 31
3.4 DAT Cache Architecture . . . 32
3.4.1 DAT Cache Refilling Example . . . 33
3.4.2 DAT Cache Addressing . . . 37
3.4.3 DAT Cache Population . . . 38
3.4.4 DAT Cache State Machine . . . 38
3.4.5 DAT Cache Implementation Decisions . . . 39
3.5 Additional Performance Improvements . . . 40
3.5.1 Request burst wrapping . . . 40

4 Memory Controller Framework 46
4.1 Memory Controller Security Model . . . 46
4.1.1 Memory Controller Encryption Model . . . 48
4.1.2 AMBA AXI4 Interface Protocol . . . 49
4.2 Memory Controller Pipeline . . . 50
4.2.1 Memory Encryption Pipeline . . . 50
4.2.2 Authentication Pipeline . . . 52
4.3 Hardware Testing Framework . . . 53
4.3.1 FSBL Modifications . . . 54
4.3.2 PL Binary Loading . . . 57
4.4 Results . . . 58
4.4.1 Authentication Tree Hardware Cost . . . 58
4.4.2 Authentication Tree Performance . . . 62
4.4.3 Summary . . . 68

5 Conclusion and Future Work 71
5.1 Future Work . . . 71
5.1.1 Tree Arity . . . 71

5.1.2 Compiler Assisted Rebalancing 72
5.1.3 Running Linux within Protected Memory 73
5.2 Conclusion 74

Bibliography **75**

List of Figures

2.1 TEC-Tree Architecture [1] 10
2.2 TEC-Tree Node Indexing [1] 12
2.3 DAT Node Metadata [2] 13
2.4 DAT Example Layout 13
2.5 DAT Unordered Restructuring Method 14
2.6 Restructuring Method 1 [2] 15
2.7 Restructuring Method 2 [2] 16
2.8 Restructuring Method 3 [2] 17
2.9 DAT Memory Layout 18
2.10 Memory Encryption Pipeline [3] 21
2.11 Memory Pipeline with Dynamic Authentication Tree [3] 21

3.1 Keystream Caching Architecture [4] 24
3.2 Cached TEC-Tree Pipeline 25
3.3 Tree-node Cache Ports 28
3.4 Tree-node Cache Indexing Scheme 30
3.5 DAT Cache Architecture 32
3.6 DAT Example Layout - Balanced 34
3.7 DAT Example Layout - Unbalanced 35
3.8 DAT Node Cache Read State Machine 39
3.9 Burst request with incorrect wrap parameters 41

4.1 Memory Authentication Pipeline Hardware Design 47
4.2 Cipher Block Chaining CTS Mode [5] 48
4.3 Memory Controller Encryption Pipeline 51
4.4 Encryption and Authentication Pipeline 52
4.5 FSBL State Machines 55
4.6 Sequential Memory Access Performance 65
4.7 Random Memory Access Performance 66
4.8 Memory Hotspot Access Performance 68

List of Tables

2.1 Memory Authentication Hardware Comparison 20
2.2 Authentication Tree NONCE sizes 20

3.1 DAT Cache Refill Node Access Numbers 36

4.1 Zynq XC7Z020 Memory Map 53
4.2 APSoC Resource Comparison 59
4.3 Synthesis Utilization Reports 59
4.4 Implementation Utilization Reports 61
4.5 XC7Z020 Resource Utilization Percentages 61
4.6 Simulation Parameters 63
4.7 Summary of Timing Results (µs) 69

List of Listings

4.1 Original FSBL Load Address Checking 57
4.2 Modified FSBL Load Address Checking 57

Acronyms

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

APSoC All Programmable System-on-Chip

DAT Dynamic Authentication Tree

DDR Double Data Rate SDRAM

DMA Direct Memory Access

DSP Digital Signal Processing

FF Flip-Flop

FPGA Field-Programmable Gate Array

FSBL First Stage Bootloader

IOT Internet of Things

LUT Look up table

MAC Message Authentication Code

NONCE Number-used-once

OCM On-Chip Memory

PL Programmable Logic

PS Processing Subsystem

RAM Random Access Memory

SDRAM Synchronous Dynamic Random-Access Memory

SoC System-on-Chip

TEC-Tree Tamper Evident Counter Tree

XOR Exclusive OR Operation

Chapter 1

Introduction

1.1 Motivation

With the continuous expansion of ”smart-devices”, the amount of user data processed in non-secure locations has been growing rapidly. These small, low powered, devices are generally known as embedded systems. Their purpose is to fulfill a set of tasks quickly and easily without wasting either space, power, or money. They are used everywhere in today’s computing industry: cameras, dishwashers, cars, wireless door locking systems, etc... Many of these deployments, while simple, are operation critical, and may contain sensitive data, meaning the security of which is of utmost importance. Generally, the software implementations are hardened for this reason, and many of the devices use proprietary communication protocols without access to the internet, making it difficult to illegitimately obtain the information processed on the device. However, given physical access to the device, obtaining data may become feasible.

Many of the implementations that make use of embedded systems are based on a system on a chip. A system on a chip (SoC) is an electronic component that implements a large portion of a computers typical components on a single chip. These components may include: processor, memory, I/O (input and output) and hardware acceleration support for data streams like: video, audio, or encryption. This work will

utilize as a case study an SoC that contains both a traditional processor-based setup, but also a reconfigurable hardware portion. In many implementations the SoC may interface with external storage and/or memory. Since these storage resources are not within the SoC, buses are used to transmit the data between the SoC and external memory. Since this data is transmitted to an off-chip location via a physically exposed bus, the data being transferred can be susceptible to either theft or tampering.

At the moment, some embedded system devices use encrypted flash to protect the device's code and persistent storage. This is a fairly trivial task given that flash access occurs infrequently, as well as the fact that many SoCs/FPGA devices support hardware-based flash encryption. Where these designs fall short are the lack of encryption for the device's random access memory (RAM). RAM is usually left unencrypted due to the nature of its function: fast, temporary data storage. Additionally, RAM is typically clocked slower than CPUs and is located off-die. This means that accessing data within memory already incurs a performance penalty, even before there is additional hardware used for encrypting data-in-motion. Placing dedicated encryption hardware directly within the memory controller adds protection against a variety of memory attacks; however, it introduces additional overhead, both design and timing based, neither of which are desirable for small, low-powered SoC-based devices. In an attempt to increase memory security, there has been research into methods of memory encryption that have a lesser performance overhead [6, 7].

While there has been research exploring the encryption of data-at-rest in memory, the topic of tampered data is another issue entirely. Exposed memory busses, or memory attacks such as rowhammer [8], can allow attackers to modify/corrupt the data within memory, to no knowledge of the SoC or the software running on it. A method to minimize such an attack vector is authenticating all data that travels to/from the memory. By authenticating all sensitive data-in-motion, both the CPU and memory are aware of any tampering that may have occurred while the data was

either in-motion or at-rest. Several memory authentication methods [3, 9, 10, 1] have been used in the past, but these implementations are neither simple to implement nor minimally adverse on performance.

1.2 Objective

The goal of this research was to provide a method of securing memory, both at-rest and in-motion, that is simple to use and makes use of commonly used interfaces. To achieve this, the work done built on top of previous research into dynamically balanced memory authentication trees (DAT) [2]. The primary goal was to improve the performance of DAT's by caching entries of the authentication tree. To evaluate the effectiveness of these improvements, both the FPGA resource utilization and performance impact of the design was compared to not only the dynamic authentication tree the design is improving, but also other implementations seeking to solve the same problem. Similarly to the dynamic authentication tree design this was based on, the improved design takes advantage of the AXI-4 protocol for interfacing with the processor and memory controller. The design was benchmarked using timing-accurate simulations, and verified using a Zynq-7020 SoC, which provided both the ARM CPU for running real-world workflows and the FPGA fabric that was used to create the cached dynamic authentication tree.

Chapter 2

Background

2.1 Physical Memory Attacks

Physical attacks are a more prevalent issue with embedded systems than traditional software-based approaches used in both enterprise and desktop software. This is because the software used in embedded systems is simpler and presents fewer attack vectors. Also, given that embedded systems may be deployed in unsecured locations, it makes them very susceptible to physical access/attacks. Once given physical access, the effort required to perform a physical memory attack can be substantially reduced in certain cases. For example, an attacker can monitor the CPU's memory bus lines to access the raw information being transferred to/from memory[6]. While there are memory encryption schemes that exist to prevent unencrypted memory access, they do not prevent the injection of malformed data into the CPU or memory[11, 4]. Not authenticating the data traveling on the CPU's memory bus can allow a potential attack to send any set of data to either the CPU or memory.

2.1.1 Replay Attacks

Memory replay attacks occur when an attacker observes the memory bus and records a subset of the transactions occurring. At a later time, the attack can use this information to "replay" data back over the bus to either overwrite it or remotely re-

execute specific parts of the system's firmware. Preventing this attack is very difficult and requires memory blocks to contain a one-use signature that can be authenticated on each read/write to ensure that the memory block wasn't replayed[12].

2.1.2 Spoofing Attacks

Spoofing memory is when an attacker attempts to disrupt the execution of the system by sending modified or malformed memory blocks back to the CPU or memory. If undetected, the CPU's execution may get disrupted or even cause a fault if the malformed memory presents an invalid instruction to the CPU. While there are methods to detect this type of fault, memory authentication would prevent the memory controller from forwarding the false data to the CPU at all [13].

2.1.3 Splicing Attacks

A memory splice attack is a combination of both a replay and a spoofing attack. The attacker will exploit the system memory by reading valid data blocks during a memory write event, and will later rewrite this same data to a different address within memory. Detecting and preventing this attack is done by authenticating a memory block based on both the contents of the block and the block's address within memory.

2.2 Memory Protection

Encrypting the contents of memory is a common method for protecting against unauthorized memory reads [3]. With plain encryption; however, memory isn't protected from the physical memory attacks listed above. Protecting against replay, spoofing, or splicing attacks requires memory authentication to verify the integrity of data. On their own, many plain encryption schemes don't provide memory authentication unless paired with an authentication scheme. As the focus of this research is on complete memory protection, memory contents must be both encrypted and authenticated.

2.2.1 Hashing

One method of verifying the integrity of data is via the use of hashing. Hashing algorithms generate a constant length hash of a variable length data input. The advantage of data hashing is the flexibility of the data it can operate on. Hashing can occur on either cryptographically secure or plaintext data. Given the variable length input, it can be configured to authenticate the most efficient data block size for either the hardware or software targets. A memory controller can stream data-in-motion through a hashing algorithm to generate a data block hash and compare it to the one stored securely on chip. The downside of this method is the large on-chip storage required for storing data block hashes.

2.2.2 Message Authentication Codes

Similar to a hash, a message authentication code is a unique, constant length, tag, generated from a variable length message. A cryptographic MAC primary differs from a hash function in that it also uses a secret key for tag generation. As the value of the tag depends on both the secret key and the plaintext, the generated tag can be stored alongside the corresponding message as long as the key is kept secret. The advantage this provides for hardware-based implementations is that only the key(s) have to be stored within trusted on-chip memory. Compared to hashing functions, this reduces the on-chip storage requirements by a large magnitude.

2.2.3 Authenticated Encryption with Associated Data

Authenticated Encryption with Associated Data (AEAD) is a subset of authenticated encryption in which both the data itself and the location of the data is authenticated. Verifying both the data and its location protects not the validity of the data, but it also ensures that it hasn't been duplicated. Both associated data (AD) and location information is bound to the target ciphertext to provide these protections. The

associated data is only authenticated and is not encrypted. AEAD ensures that two messages with identical plaintext will not yield the same ciphertext, increasing the diffusion properties of the secured data.

2.2.4 Block-Level AREA Authentication

Block-Level AREA Authentication provides block-based data encryption and authentication by applying the AREA technique [14] to a block of data. The AREA technique specifies a method for authenticating data through the use of the diffusion property of block-ciphers. As an alternative to implementing authentication features, data authentication can be performed given the cipher mode used to protect the target data block possesses infinite error propagation, such as CBC (cipher block chaining). The AREA technique operates as follows: an authentication tag L is appended to the end of plaintext message P. Once encrypted, L is prepended to the beginning of the ciphertext C. If the message both begins and ends with identical blocks of L after the decryption process is complete, the integrity of the message is intact. The benefit of using the AREA technique over small data blocks is simplified hardware, because the only thing required is an encryption engine. No additional hardware is required to implement an additional authentication scheme to the underlying encryption, AES-GCM for example. This can also provide better performance since the authentication is done through error propagation instead of through the generation of an authentication tag or hash.

2.3 Authentication Trees

Of the previous authentication methods discussed, each one has a common detriment to its hardware implementation: on-chip storage cost. Each of these methods requires trusted storage (on-chip) for at least one associated string or key per block. While the size of each block can be increased to decrease the total number of tags required,

this can cause large performance degradation as the entirety of each block must be read or written for authentication purposes. The on-chip storage cost can be reduced while retaining smaller data block sizes by allowing multiple blocks to share a single trusted element. With this technique, only the trusted element must be stored within trusted memory. To increase the cryptographic strength of the data, each block must use a unique tag, or NONCE, derived from a parent tag. These tags can be arranged in a tree-like fashion, with the trusted element being the root of the tree.

2.3.1 Data Authentication Methods

Many authentication tree implementations contain two types of nodes: counter nodes and data nodes [15, 2, 1, 16]. Data nodes are the (encrypted) blocks of data being stored in memory, and counter nodes are used to store tree ”metadata”. Metadata contains the structural information of the tree, including access counts, address maps, and authentication information. In these trees, the authentication and encryption is performed using the Block-Level AREA Technique. The entire plaintext and NONCE block is encrypted. Upon decryption, the integrity of the data can be verified by examining the value of the decrypted NONCE. If the value of the NONCE has remained the same through the encryption/decryption process, then the data has not been modified in any way.

2.3.2 Authentication Tree Performance

The plus-side of authentication trees is that they have a low hardware overhead, reducing the complexities of CPU hardware implementation. Conversely, one of the largest downsides is the high off-chip overhead created by authentication trees. This cost comes from the amount of extra metadata that must be stored in memory to keep track of the tree's state. The performance overhead of the tree is not negligent. Despite the tree structure allowing for efficient node access, there is an additional

overhead gained from generating tree node requests and navigating the tree that raw memory access does not have.

2.3.3 TEC-Tree

One of the authentication tree's this research was based on, and used for benchmarking purposes, was tamper-evident counter (TEC) trees. TEC-trees are statically arranged trees of a configuration arity. TEC-Trees are made up of the two standard node types used in authentication trees: counter and data nodes. Counter nodes contain access counts for all the children nodes, which may be either data or counter nodes. As TEC-trees are statically arranged, accessing a node is efficient, as its position will always be static in memory and does not require tree traversal to locate. Data nodes in TEC-trees are authenticated by appending NONCE's to the end of each data node. If the decrypted plaintext NONCE of a node matches the NONCE expected by its parent, there are no memory errors present. The architecture of a TEC-Tree is displayed in Figure 2.1.

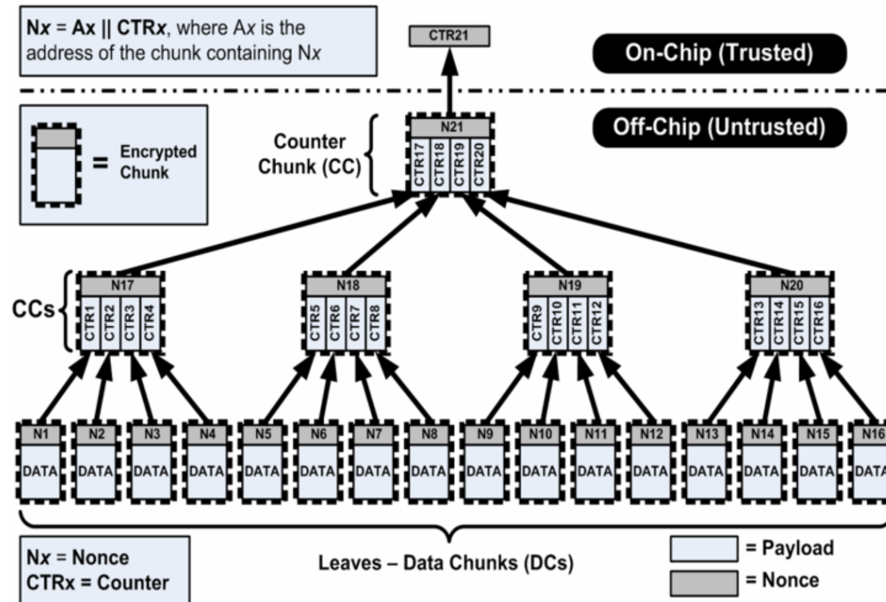


Figure 2.1: TEC-Tree Architecture [1]

2.3.3.1 TEC-Tree Cryptographic Properties

The TEC-Tree performs its authentication by verifying the integrity of the NONCE stored alongside each tree node. The NONCE stored with each node is generated using both the address of the data node and the access count of the current node. This allows the parent node to authenticate the child node using a similar technique to that in subsection 2.2.4. The exception to this authentication structure is the root of the tree, which is stored as a single counter on-chip. As the root node is considered trusted, it does not require associated data to authenticate against.

2.3.3.2 TEC-Tree Performance

TEC-Trees provide numerous benefits over purely authenticated memory data. Since each tree root is able to protect multiple data blocks, only a very small portion of on-chip memory is required to authenticate a single tree. Therefore, multiple trees can be used to protect larger portions of memory. Not only does this increase performance by reducing tree depth, but it allows for multiple simultaneous tree accesses across different trees. Space complexity wise, TEC-Trees scale using Equation 2.1.

$$O_{\text{TEC}} = \frac{l_p + nA}{l_p(A-1)} \tag{2.1}$$

The space overhead required for a single TEC-Tree is somewhat minimal as each data node, l_p bytes long, only requires a single configurable sized (n bytes) NONCE. Each counter contains A (tree arity) counters as well as a NONCE of the same size as previously configured. Since the tree is statically arranged, each node does not need to store any structural information regarding node locations as they can be calculated on the fly. The indexing of TEC-Tree nodes as depicted for a 4-ary tree in Figure 2.2 is done by numbering the root node as '0', and increasing counts across each tree level.

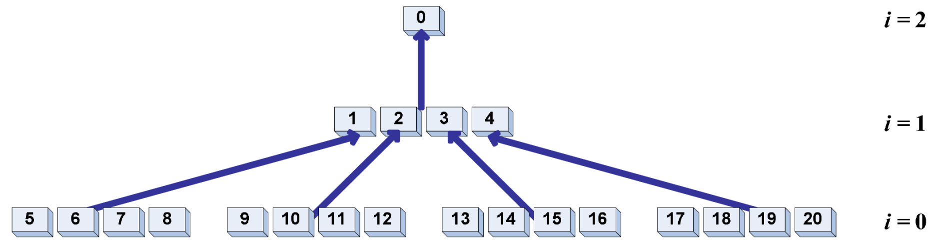


Figure 2.2: TEC-Tree Node Indexing [1]

The index of a parent node can be calculated by subtracting one from the index of the current node and dividing by the tree arity. Given that the arity of the tree is a power of two, this can be performed efficiently in hardware using bitwise shifting. If the parent node is calculated to have a negative index, the root of the tree has been reached, and child node authentication must happen using the root counter from trusted memory.

The downside of this approach however, is related to the static arrangement of the trees. The guarantee that each data block must exist at the lowest level of the tree requires full tree traversal for every data access.

2.3.4 Dynamic Authentication Trees

An authentication tree design that attempts to improve the performance of the TEC-Tree is the dynamic authentication tree (DAT). As the name implies, the dynamic authentication tree employs methods of tree re-balancing in order to place more frequently accessed nodes towards the top, or root, of the tree. This change reduces the performance overhead of balanced trees by offering a dynamically re-constructed tree to reduce tree traversal time. This is done by comparing the access counters of nearby nodes to determine which has been accessed the most. Each counter node also contains three counters of its own. The first counter is used to track the amount of accesses this counter node has. The second and third counters are left and right counters, these keep track of the access counts to the left and right child nodes respectively. Figures 2.3 and 2.4 show the layout of both the counter and the data nodes

for the dynamic authentication tree, as well as how they are arranged in an example tree.

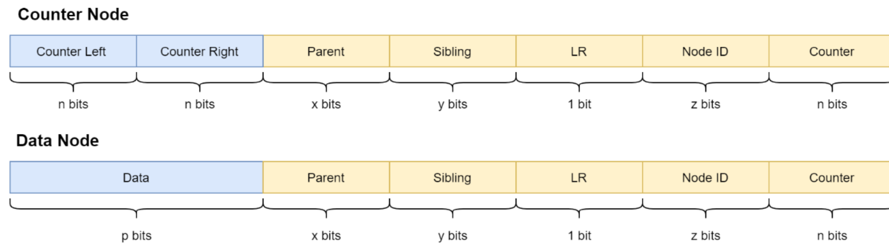


Figure 2.3: DAT Node Metadata [2]

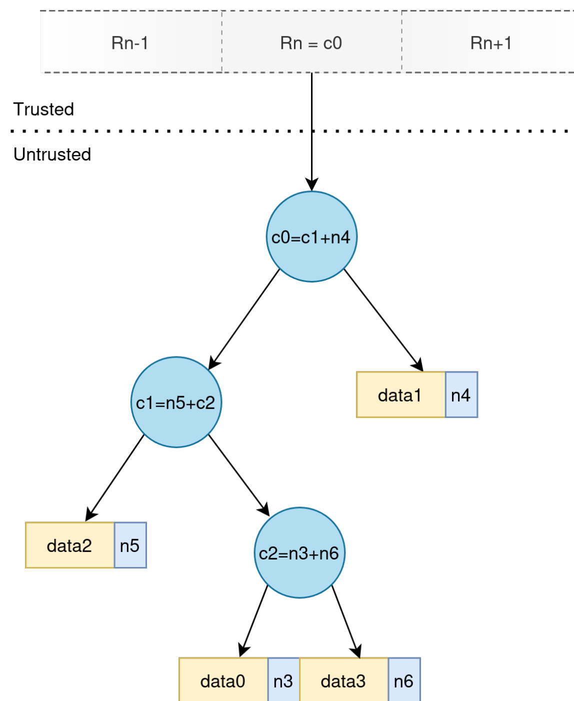


Figure 2.4: DAT Example Layout

DAT nodes are highly flexible and configurable. The size of each counter or parent node index can be configured to require the least amount of memory space required for proper operation. The memory location where each node is stored is static; therefore, rebalancing doesn't modify the physical memory address of data blocks or counter nodes. All rebalancing happens virtually, with modifications happening to the parent or sibling index fields of node metadata. This allows tree rebalancing to occur without data copies.

2.3.4.1 Unordered DAT Rebalancing

Dynamic authentication trees have two methods for rebalancing trees. The Huffman encoding scheme is used for the first rebalancing method. Huffman generates an optimal binary tree based on the weights of the leaf nodes in the tree. This also means that when weights are updated (node writes), the same algorithm can be rerun to reorganize the tree into an optimal layout. Huffman encoding doesn't guarantee the retention of node orders though, so any re-ordering using this algorithm may cause the nodes to become unordered. Nodes becoming unordered may cause non-optimal tree architectures. An example of an unordered rebalancing operation is demonstrated in Figure 2.5.

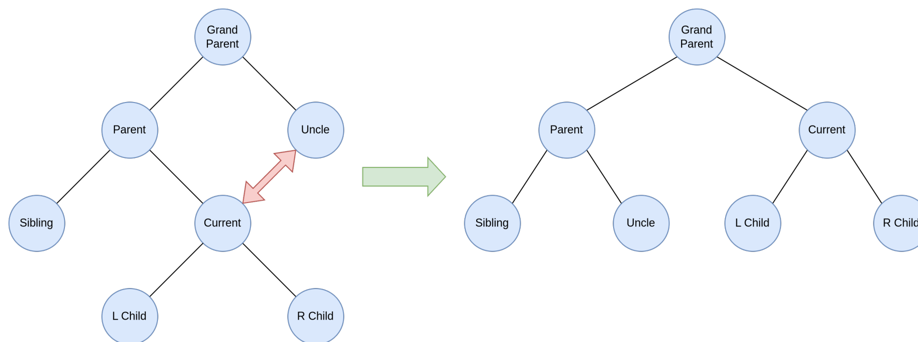


Figure 2.5: DAT Unordered Restructuring Method

Rebalancing nodes in an unordered tree is done by comparing the weights, or access counts, of a node and its uncle. If the access count of a node is higher than that of its uncle, the two nodes are swapped logically. Logically swapping nodes only updates the metadata of the two swapped nodes, their siblings, and their parents. The updated metadata points to the new parent and sibling nodes, as well as the updated access counts of the new relatives. Swapping two nodes also swaps their children since the metadata of the children still point to the unique node ID of the parent, which is not updated. Performing logical swaps also means the physical location of each node within memory remains the same during the reordering process.

2.3.4.2 Ordered DAT Rebalancing

The second method for rebalancing generates ordered trees. The data nodes within an ordered tree must always remain in the same order, even after rebalancing. Retaining the order of a tree's leaf nodes means a higher chance of the tree being optimal. Optimal trees are typically easier to traverse, as the positions of leaf nodes relative to each other is always constant. The dynamic authentication tree presents three different methods for rearranging ordered trees for performance efficiency.

Method 1 - The first rebalancing method uses the Parent, Sibling, Uncle, and current nodes for rebalancing the tree. Rebalancing is performed when the current node's weight is greater than that of the Uncle node. When this happens, the current node is shifted up to the same level of its own Parent node, and the Uncle replaces the current node under the Parent. Figure 2.6 shows the tree movements that make this occur.

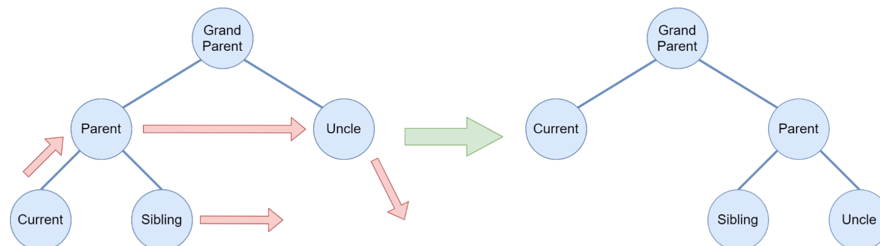


Figure 2.6: Restructuring Method 1 [2]

The issue with this restructuring method is that it does not modify the case in which the left or right relation to the Parent of Uncle and current nodes do not match. This causes the order of the leaf nodes to be invalid, requiring the implementation of a second method for handling this situation.

Method 2 - The second method uses the same data as the first, but it adds the weights of the Grand Uncle and Grand Parent into the restructuring calculation.

This method of restructuring occurs when the current node and its Uncle share the same weight, but the weight of the current node and its Grand Uncle do not match. When these conditions are met, the current node is moved upwards and becomes a Sibling of its Uncle. Following this, the Parent node moves upwards to replace the Grand Uncle, and the Grand Uncle replaces the current node under the Parent. The visualization of this restructuring is shown in Figure 2.7.

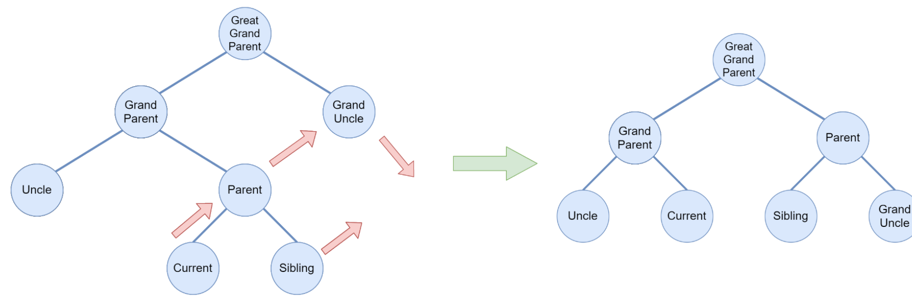


Figure 2.7: Restructuring Method 2 [2]

Restructuring using this method fixes the problem specified in the previous method; however, it creates a fresh problem in the case that the current node's weight is equal to both its Uncle's and Grand Uncle's weight. In this situation, the leaf node order is not retained, so a third restructuring method is needed to fix this case.

Method 3 - The final method for restructuring the tree is similar to that of method 2; however, it occurs when the current node's weight is equal to that of the Uncle and Grand Uncle. This method does not modify the current node itself, but it does move the parent up in the tree to replace the Grand Parent's location. Following this, the Grand Parent is moved over to replace the Grand Uncle, which becomes a child of the Grand Parent, as well as the Uncle. This movement is demonstrated in Figure 2.8.

These restructuring methods are run dynamically during the execution of the

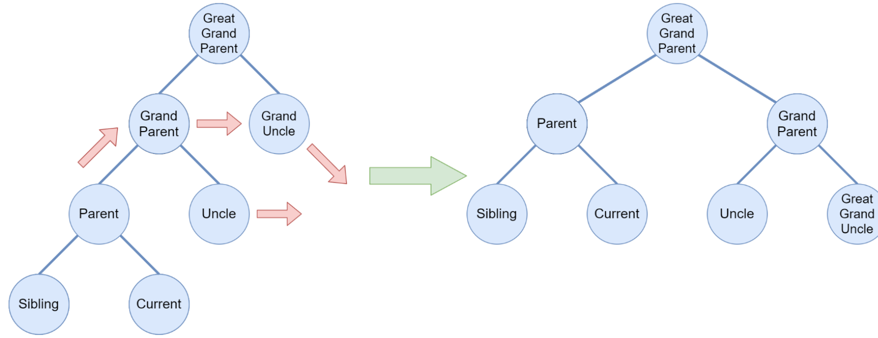


Figure 2.8: Restructuring Method 3 [2]

memory controller pipeline. Performing these checks and rebalances during execution provides an automatic performance boost to the code execution without the programmer needing to perform any special actions for memory structuring. The negative result of performing restructuring live, is the performance hits that occur during the restructuring itself. While restructuring the tree nodes, the memory controller is halted and cannot perform tree accesses until the rebalancing is completed.

2.3.4.3 DAT Memory Layout

DAT Nodes are stored sequentially in memory, with both the payload and metadata stored together. Data nodes are stored starting at the beginning of the protected memory region. The storage scheme for memory protected by two trees, each with four data nodes, is illustrated in Figure 2.9.

The node groupings for each tree are stored sequentially, similarly to the individual nodes within. The grouped data nodes make up the protected memory area. Due to the related metadata, the total space used to store data nodes is larger than the protected memory area they represent; however, only the payload itself is accessible via the virtual address space. Tree metadata, or counter nodes, are stored following the end of the protected memory region, and are not mapped in the virtual memory space; therefore, cannot be accessed by the host system.

Figure 2.9 also demonstrates the node indexing scheme of DAT nodes (via the

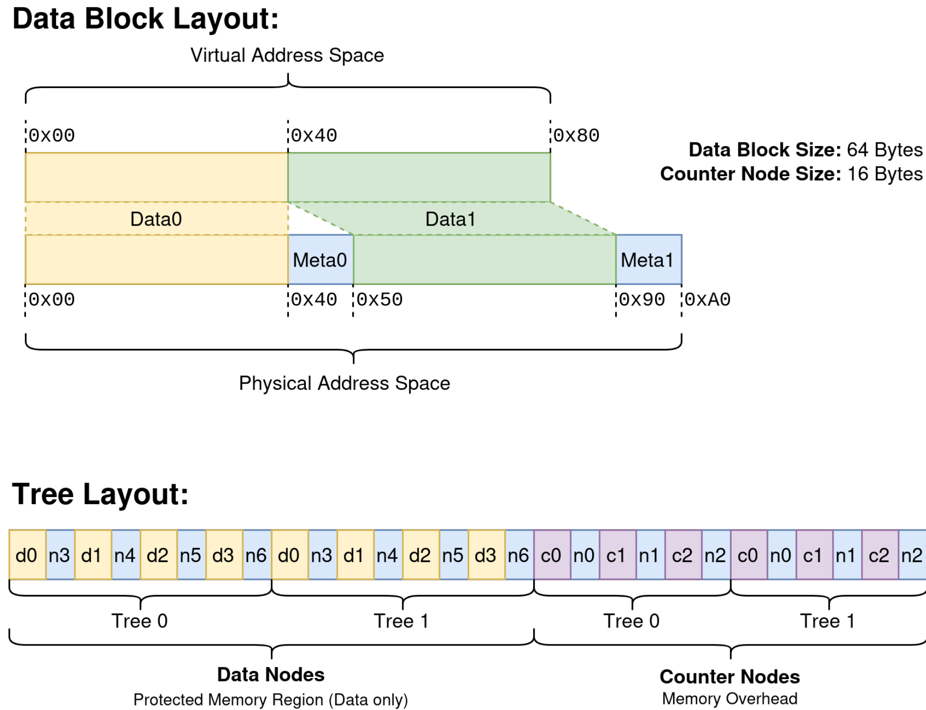


Figure 2.9: DAT Memory Layout

NONCE indexes). The indexing scheme used is identical to that of the TEC-Tree; however, given the dynamic nature of the tree, the position of each index relative to the root is not guaranteed to remain the same.

2.3.4.4 DAT Cryptographic Properties

Authentication within DATs are similar to that of TEC-Tree. The access counter of a node is compared to that of its parent; however, the tree structure must be taken into account. As it is possible for child nodes to be swapped during rebalancing, each child contains a field denoting its relation to the parent: `LR`. `LR` is a single bit field that specifies if the node is either left, value of '1', or right, value of '0', of the parent. After reading the value of the child's `LR` field, the parent will compare the corresponding counter with the child's counter and report the authentication result.

2.3.4.5 DAT Performance

As opposed to statically arranged trees, like TEC-Trees, DATs require tree structure information to be stored within each node's metadata. This causes a variety of issues for the hardware design. Not only does this increase storage complexity of DAT designs, but it also increases the design complexity, and therefore fabric utilization. Additionally, it mandates the parsing of all tree nodes and the traversal of additional states related to rebalancing. These states are used for determining whether a rebalance is needed and/or rebalancing the tree if required. The space complexity required for DATs can be calculated from Equation 2.2.

$$O_{\text{DAT}} = \frac{l_p + n \cdot 2}{l_p(2-1)} \tag{2.2}$$

This equation is identical to that of the TEC-Tree; however, the arity of the DAT is locked at 2, and the size of each NONCE is significantly larger as it contains tree data. A comparison of the NONCE sizes for each tree can be found in Table 2.2. Given the binary nature of the rebalancing methods used, only 2-ary DATs are supported. Reducing the amount of children each parent node can have increases the depth of the tree, creating a performance bottleneck as the average node depth will be deeper.

2.4 Authentication Method Comparison

Given the previous knowledge regarding memory authentication methods, Table 2.1 compares the hardware impacts of each one.

Providing full memory protection requires both memory encryption and authentication. Of the option's discussion in section 2.2, the only solution that provides full memory protection without severely impacting performance is authentication trees.

These equations demonstrate the additional overhead required to store tree information within the metadata of DAT nodes. DAT node metadata requires the storage

Table 2.1: Memory Authentication Hardware Comparison

	Performance Overhead	Memory Overhead	On-chip Overhead	Authenticated	Encrypted
Block Encryption	Medium	None	None	No	Yes
Hashes	High	None	High	Yes	No
MACs	High	High	High	Yes	No
AEAD	High	Medium	Low	Yes	Yes
AREA	High	Medium	Low	Yes	Yes
Authentication Trees	Medium	High	Low	Yes	Yes

Table 2.2: Authentication Tree NONCE sizes

Tree Type	Memory Overhead	NONCE Size	Average Node Depth
TEC	$\frac{l_p+n\cdot A}{l_p(A-1)}$	$c+\log_2 D$	$\log_A D$
DAT	$\frac{l_p+n\cdot 2}{l_p(2-1)}$	$c+3\cdot\log_2 D$	$< \log_2 D$

of three node indexes as opposed to the single index required for TEC-Trees. The advantage of the dynamic authentication tree comes with the average node depth. Since more frequently used nodes are placed near the top of the tree, the average node depth is much higher in the tree than that of the TEC tree.

2.5 Authenticated Memory Controller

Many previous implementations of memory authentication and encryption used a custom memory controller to perform these actions that were capable of interfacing with widely used memory busses (e.g. DDR3). [3, 2]

Figure 2.10 depicts the architecture of a memory controller that implements memory encryption [3].

The design of this memory controller was the base design of the Dynamic Memory Authentication tree. The benefits of a controller similar to the one in Figure 2.10 is

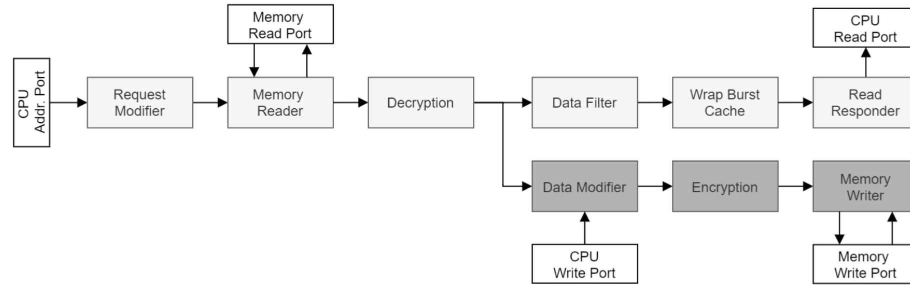


Figure 2.10: Memory Encryption Pipeline [3]

the seamless memory encryption and decryption that is completely obscured from the CPU. Obscuring this from the CPU keeps the CPU memory stage execution fast, but it can still provide the necessary security. Since this controller implements encryption and decryption, the basis for authentication is already implemented, making authentication tree implementation easier.

The modified design that implements the Dynamic Authentication tree is shown in Figure 2.11.

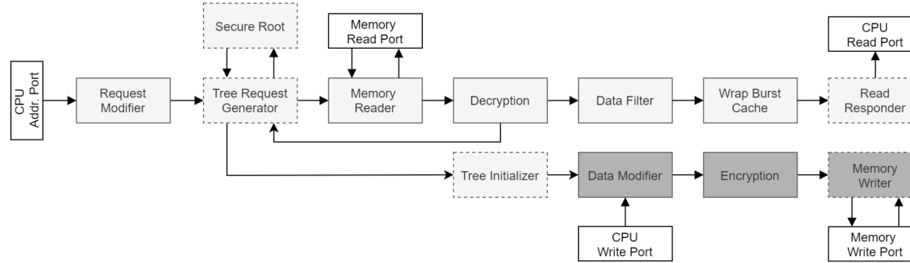


Figure 2.11: Memory Pipeline with Dynamic Authentication Tree [3]

The modifications made to the controller are shown with dotted outlines around both existing and new blocks. The most notable changes include the addition of a tree request generator and a tree initializer. These new blocks are used to generate memory address locations and decryption information for the decryption engine. These tree request blocks are where the subject of this research takes place. While adding these tree request blocks was necessary for the implementation, they introduce a major performance penalty for certain memory accesses.

As previously mentioned, the Dynamic Authentication Tree is able to restructure

the tree during execution on its own. Given that this happens during memory access cycles, it delays memory access and causes extra CPU stalls. The tree request generator also causes a performance hit. When given a memory request, the tree request generator is the pipeline stage that calculates the address of both the tree and data node containing the target data, as well as the corresponding NONCEs used to authenticate them.

This request generator may cause a stall in the controller's AXI pipeline as it needs to traverse through the tree in order to authenticate the requested data node. While the dynamically structured tree helps mitigate this issue, it will always be unavoidable for the access of certain nodes, namely those further down the tree. This is solely based on the mechanics of a tree-based data structure. To shorten the delay caused by tree traversal and reduce the length or number of pipeline stalls, a cache can be used to store tree node information to avoid traversing through the entire tree structure.

Chapter 3

Cached Authentication Trees

3.1 Authentication Tree Caching

Previous encrypted memory implementations took advantage of a small local cache to store encryption data, such as keystreams or authentication data (counter nodes, NONCEs, etc...) [9, 4]. The goal of these caches was not to store the data blocks themselves, but to store cryptography related data as a means of reducing the performance impact of cryptographic operations.

3.1.1 Memory Caching

The Von Neumann, or Princeton, Architecture is a commonly used design for computing systems. The design principals specify a singular memory used for both instruction and data that resides as a separate unit from the CPU. The shared data bus that exists between the CPU and memory limits the maximum possible performance of the CPU. This design only allows for either an instruction fetch or data operation, not both at the same time. The off-chip memory has both a relatively slow clock speed and a long bus length, so accessing data off-chip incurs a heavy latency. This latency can be prevented by caching memory data directly on the CPU die. CPU caches, while small compared to off-chip memory, are high speed and are considered trustworthy in the scope of this research.

The issue of memory latency with Von Neumann architectures are only exacerbated by the addition of cryptographic engines directly into the memory controller. The same logic for the addition of CPU caches can be applied to the use of caches for storing cryptographic primitives within memory controllers containing cryptographic hardware. An on-chip cache used for encryption/decryption metadata would not violate the integrity of the underlying algorithm because the purely on-chip location is considered trustworthy.

3.1.2 Keystream Caching

Caching encryption data provides an elegant way to increase the performance of an encrypted memory architecture without losing any of the security features. Figure 3.1 demonstrates a memory architecture used for implementing a keystream caching mechanism [4]. Using an architecture similar to this allows for such a large speedup because each encryption operation is reduced to a singular XOR operation.

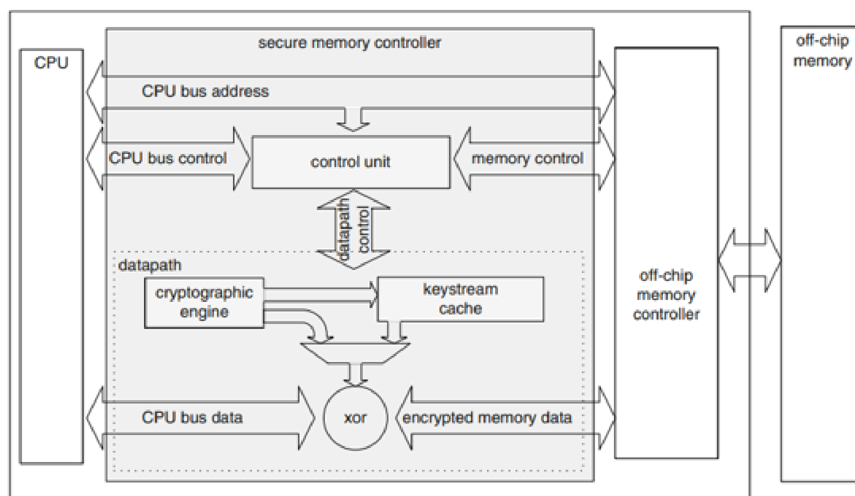


Figure 3.1: Keystream Caching Architecture [4]

This concept can be extended further for authentication tree architectures. The counter nodes used for authentication can also be used as the counter (CTR) input if the authentication tree implements a block cipher using counter mode. Caching

the keystream for each tree root or counter node would reduce each memory encryption/decryption operation into a singular XOR operation.

3.1.2.1 TEC-Tree Caches

As a method of improving performance, TEC-Trees can implement node caches to prevent off-chip memory access for authentication data. Data caches are stored on-chip, and therefore are trusted, so the node data can be stored in plaintext. Not only does this prevent off-chip data access, but it also bypasses the deployed cryptography scheme, increasing performance benefits.

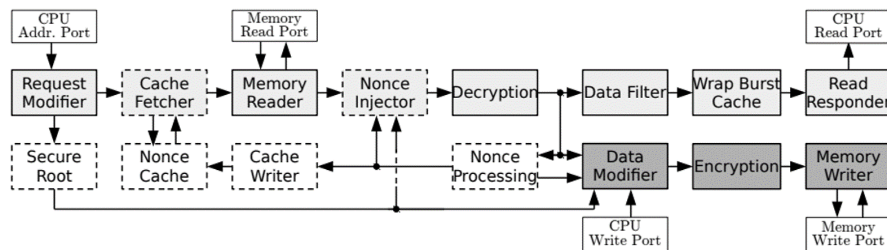


Figure 3.2: Cached TEC-Tree Pipeline

The TEC-Tree implementation in Figure 3.2 that was studied for this research implemented a NONCE cache. The NONCE cache improved the read performance of the TEC-Tree pipeline as it bypassed the need to fetch node NONCEs. Storing only NONCEs within the on-chip cache provided a higher node coverage rate compared to a metadata cache as every node within a TEC-Tree contains a NONCE. The cache accessing scheme used in the specific TEC-Tree implementation studied was a directly mapped cache, using the lowest significant bits from the NONCE's RAM address as the cache index.

The NONCE cache being placed directly within the pipeline provides the following benefits:

1. Tree generation blocks remain unmodified and don't have to be aware of the cache existing.

2. Cache access uses the same memory requests as standard memory access. This means the cryptographic engine used won't need to be changed.

However, since these caches are not write-through, they do not improve write performance of the TEC-Tree pipeline. This can be done by decreasing the amount of writes that occur per data node access. TEC-Trees have a configurable tree arity (subsection 2.3.3), which when configured to a higher number, will reduce the total depth of the tree. This reduces the number of nodes that must be written to during the write-back stage of the authentication process.

As this research focuses on the introduction of caches into authentication tree architectures, a cached TEC-Tree implementation provides a benchmark for the before and after effects of caches on authentication trees performance.

3.2 Cache Reasoning

As mentioned previously (section 2.4), authentication trees have improved performance over pure data authentication and encryption and uses less on-chip memory. However, this performance highly depends on the configuration of the tree, such as data block size, tree arity, and tree depth. This thesis focuses on an authentication framework that is flexible enough to fit a variety of use cases in which full memory security is desirable. This allows the structure of the tree to be configured to perform optimally for either the specific hardware or software workload used. However, the configuration chosen may introduce severe performance penalties in abnormal access patterns or large data transfers. Caching tree nodes lessens the performance impacts in these situations. The relatively low on-chip cost of authentication trees leaves a large enough overhead of on-chip memory for implementing node caches without sacrificing other portions of the design.

3.2.1 Dynamic Authentication Tree Caching

Dynamic access trees traverse the tree from the bottom up. Since data blocks are statically located within physical memory, their location can be calculated using the target virtual address. Authenticating data blocks is done by first reading the target data block and its metadata. The block's metadata contains the index number of the parent nodes. Using this information, the address of both the parent and sibling nodes can be calculated. However, only the parent node is used for authentication.

DAT architecture specifies that leaf nodes are *always* data nodes, and internal nodes are *always* counter nodes. This scheme guarantees that the parent node of any node is going to be a counter node. Since authentication is done by recursively reading every parent node until the root of the tree is reached, the latency of counter node reads are highly influential on the overall read/write latency of the tree. Therefore, reducing the access time of counter nodes provides a performance improvement during the authentication process.

To improve the access times of counter nodes (parent nodes), the value of these nodes can be cached within the memory controller hardware. This performs two functions, improving performance and avoiding the security implications of accessing off-chip data, the very action this research is securing.

3.3 Cache Architecture

The cache implemented for these authentication trees is a dual port, direct-mapped cache. Since the implementation uses a statically sized HDL array, synthesis tools will commonly place the cache entries into BRAM. Placing the cache in BRAM allows a larger cache size than one implemented within configurable fabric, but it still allows single cycle data access [17].

The cache is configured to use the BRAM's TDP (True Dual Port) mode. Using

the BRAM's TDP mode limits the maximum width of both the address and data ports, but it allows for reads and writes to operate independently of each other [17]. Figure 3.3 displays the cache ports and their directions assuming that the size of each counter node is 192 bits long, with a memory address width of 32-bits.

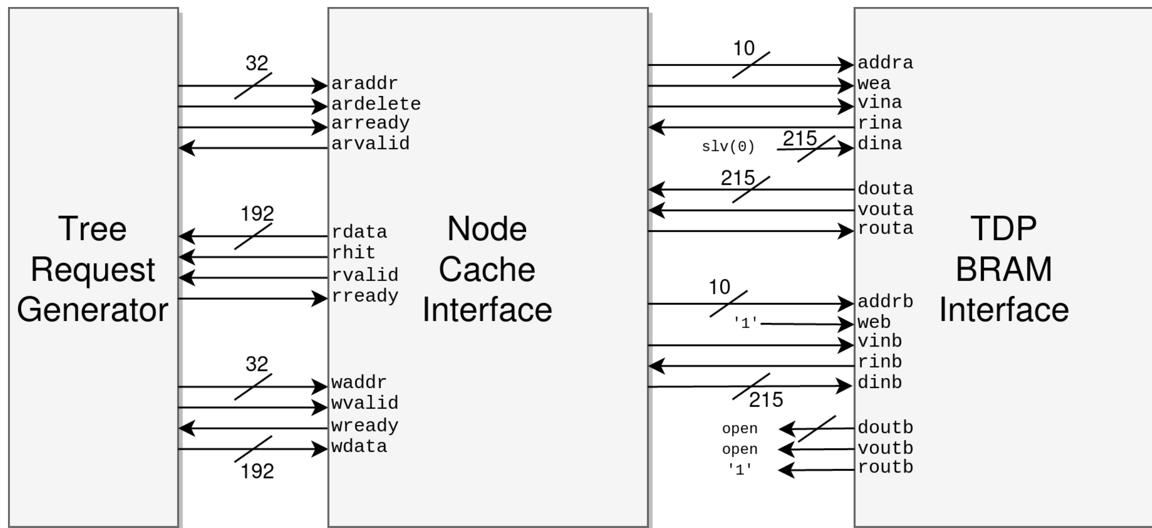


Figure 3.3: Tree-node Cache Ports

The tree-node cache is built up of three elements. First, the tree request generator generates the cache requests for accessing counter nodes. The second element is the node cache interface, which performs two functions: properly sizing cache addresses, and padding cache data with entry metadata. Finally, the last element is the TDP BRAM interface. This is a ram block that is designed to synthesize into on-chip BRAM in TDP mode. The description of the RAM port signals is described in the next few sections.

3.3.1 Caching Indexing

Since cache space is limited, the size of addressing required to index all entries is much smaller than the addressing used for main memory. For all implementations of this cache, the address width has been 32 bits. For the sake of consistency, this section will assume the cache in question uses a 32-bit address width, with 1024

cache entries, each with an entry size of 64 bits. In this scenario, only 10 bits of the address can be used for indexing the cache $2^{10} = 1024$ (Equation 3.1a). This leaves 22 bits of precision lost (Equation 3.1b). Using this 10-bit indexing scheme, there can be $2^{22} = 4194304$ different possible addresses represented by a single 10-bit index. Assuming all 32 bits of address space are used during operation, this means any cache index only has a $\frac{1}{2^{22}} = 2.38 \times 10^{-5}$ % chance of containing the correct data.

Cache sizing equations where:

$S_\#$: is the number of entries within cache

S_i: is the width (in bits) of a cache index

S_t: is the width (in bits) of a cache tag

S_e: is the width (in bits) of a cache entry

c_a: is the width (in bits) of the counter node address

c: is the width (in bits) of each counter node

$$S_i = \log_2 S_\# \tag{3.1a}$$

$$S_t = c_a - S_i \tag{3.1b}$$

$$S_e = 1 + c + S_t \tag{3.1c}$$

To mitigate these cache index conflicts, the 22-bit address tag is appended to the most-significant side of the cache entry. This means each 64-bit cache entry has a 22-bit tag overhead (and a single bit for validity), using a total of 87 bits (Equation 3.1c). The cache tagging, indexing, and data storage scheme are demonstrated in Figure 3.4.

The lowest N-bits of an address are used to generate cache indexes to make cache ”hits” more common. Two addresses in memory located within close proximity have a higher chance to share address bits on the lower end of the address word, and fewer

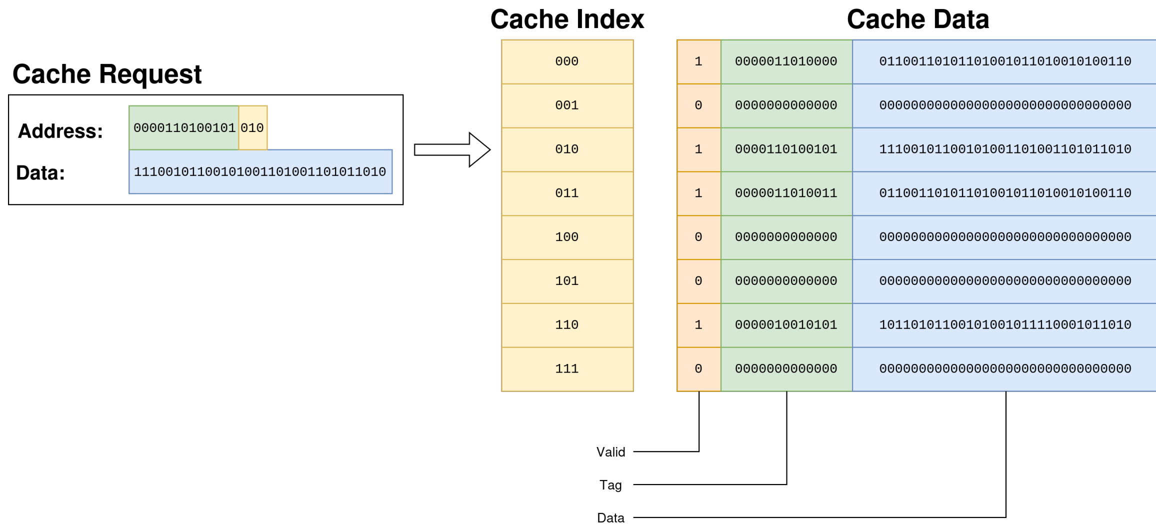


Figure 3.4: Tree-node Cache Indexing Scheme

bits on the higher end. Building the cache index from these lower bits leads counter nodes with close proximity, like those within the same or nearby trees, to be cached at the same time. This is beneficial for memory workloads that operate on either a single region of memory or access data sequentially.

The most significant bit of the data stored is the 'valid' bit. This bit is set to '1' when the data in that index is populated. Otherwise, it's '0' to denote the lack of data in that entry.

3.3.2 Cache Read Architecture

Since counter nodes are modified during every write operation, a 'delete' signal was added to the cache read ports. In tree architectures that allow for node restructuring, cache lines should be deleted before all restructure operations. When set high, the 'delete' signal informs the cache to mark an entry as empty if that specific entry was considered a 'hit' on a read access. A cache 'hit' refers to an entry in cache for which both of the following are true. Both the query address and original write address much match, and the data contained within the entry must be valid. The 'delete' line is always set high when a write operation is sent through the memory authentication

pipeline. Since each write operation guarantees a counter accumulation on each parent connecting the target data node to the root, clearing an entry on read access ensures the data won't be outdated on the next read.

Port 'a' of the block RAM's dual ports is used for read operations. Only the lowest $\log_2(\#\text{entries})$ bits of the address are used for cache indexing, while the rest are stored as a cache tag. As the BRAM hardware itself doesn't report 'valid' or 'invalid' signals related to the queried data, the 'hit' condition for cache data is calculated in the node cache interface. A hit condition is detected by using the request to entry process shown in Figure 3.4. If the 'valid' bit within the read entry is high, and both the tag and index of the entry and request match, the requested entry is a hit. Conversely, deleting data from cache is done using a clever trick with the BRAM's dual port architecture. While port 'a' is only used for cache reads, Figure 3.3 shows that a vector of all zeros is routed into the data in bus for port 'a'. If an entry deletion is required, setting port 'a's write enable (`wea`) pin high will clear the entry cache entry, including the valid bit.

3.3.3 Cache Write Architecture

As opposed to cache reads, all cache writes happen using the BRAM's second port, 'b'. Again, the same cache indexing scheme shown in Figure 3.4 is used within the node cache interface to convert write data to properly formatted and indexable cache entries. All cache writes are "write-back", meaning that all write operations to the cache are confined strictly to the cache and are not passed through to the primary computer memory. This ensures that all cache write operations are fast, since they do not require acknowledgment signals from the system's primary memory. However, the downside to this approach is that the off-chip memory may not contain up-to-date counter node entries.

If a memory controller pipeline would benefit from a "write-through" cache design,

implementing it is possible as the cache uses request lines separate from the rest of the pipeline.

3.4 DAT Cache Architecture

Since DATs only supports a tree arity of two in the current implementation, the size of the memory protected by each tree and the depth of the tree directly correlate. While it is possible to increase the number of tree roots, increasing the number of roots also increases the amount of BRAM used for the secure root module. Adding a counter-node cache provides a method for possibly preventing tree depth from crippling performance of authentication trees with DAT architecture.

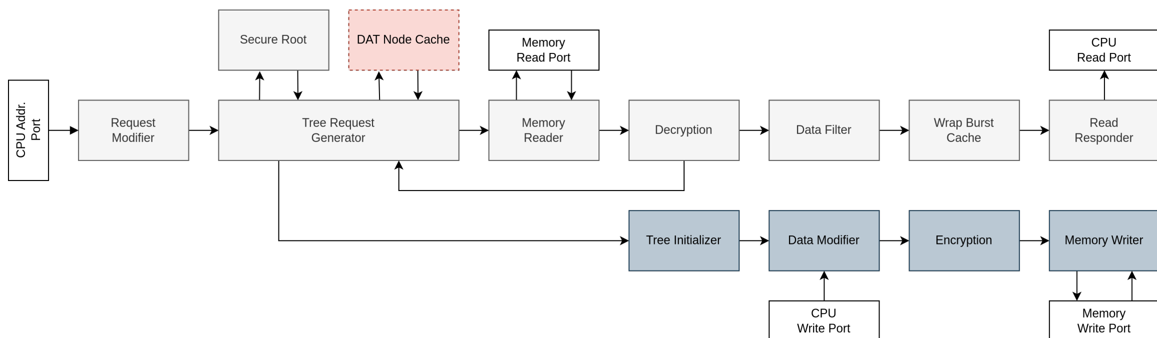


Figure 3.5: DAT Cache Architecture

As a means to maximize the performance gain of the counter node cache, the DAT request generator directly issues cache requests as opposed to issuing cache requests via the pipeline. The advantage to this architecture is the reduction in cache access bandwidth. Issuing cache requests through the pipeline allows the tree request generator to avoid the separation of request types, effectively allowing the cache to operate like a pass-through entity. All requests passing through the cache blocks would query the cache while in-motion, providing the ability to skip later blocks, such as memory access blocks. This implementation strategy would also have the smallest impact on resource utilization as it would take advantage of an already

existing request and handshake system and would operate inline with the rest of the memory pipeline.

The tree-node cache is implemented to store the value of counter nodes. Storing only the counter nodes within the DAT's tree-node cache allows a larger region of memory to be effectively cached than caching the data nodes themselves. In this instance, the tree-node cache acts as an intermediary step between the CPU's highest level cache, and the external memory. As the CPU cache is extremely effective at storing blocks of data, that does not need to be done within the DAT memory controller. Instead, the DAT's tree-node cache should reduce the amount of time required to refill CPU cache lines, by reducing latency. One of the structural characteristics of dynamic authentication trees is the static location of nodes in physical memory. Given that the address of each node is used as the cache access tag, the tags can remain in cache even after a tree rebalance. Since the DAT architecture ensures the write-back of all nodes after modification, a rebalance would modify the existing nodes in cache without leaving dirty nodes behind at old addresses.

3.4.1 DAT Cache Refilling Example

A standard CPU cache will always read, write, evict, etc... data to/from memory in blocks. CPU cache lines can vary in size, anywhere from 16 B to 2 KB [18]. Reading a single byte from memory will copy a block of memory matching the size of a cache line that contains the byte into cache. Since a single DAT counter node is responsible for protecting an entire data block beneath it, the same counter node will be access multiple times in succession during the sequential reads used for cache refilling.

Given a dynamic authentication tree that protects 16 KB of memory with 32 trees and a data bus width of 64 bits, data blocks 64 bytes wide means each tree is protecting 8 data blocks. Eight data blocks in the tree means there are seven counter nodes and a single root node. This gives a total tree depth of at most seven. Assuming

the tree is perfectly balanced, the tree is will only be three levels deep. Figure 3.6 and Figure 3.7 demonstrate the layout of the example tree and the two extreme configurations (balanced and unbalanced). In this example, it will also assume that the CPU is fetching 256 bytes of data at a time since each cache line is 256 bytes long.

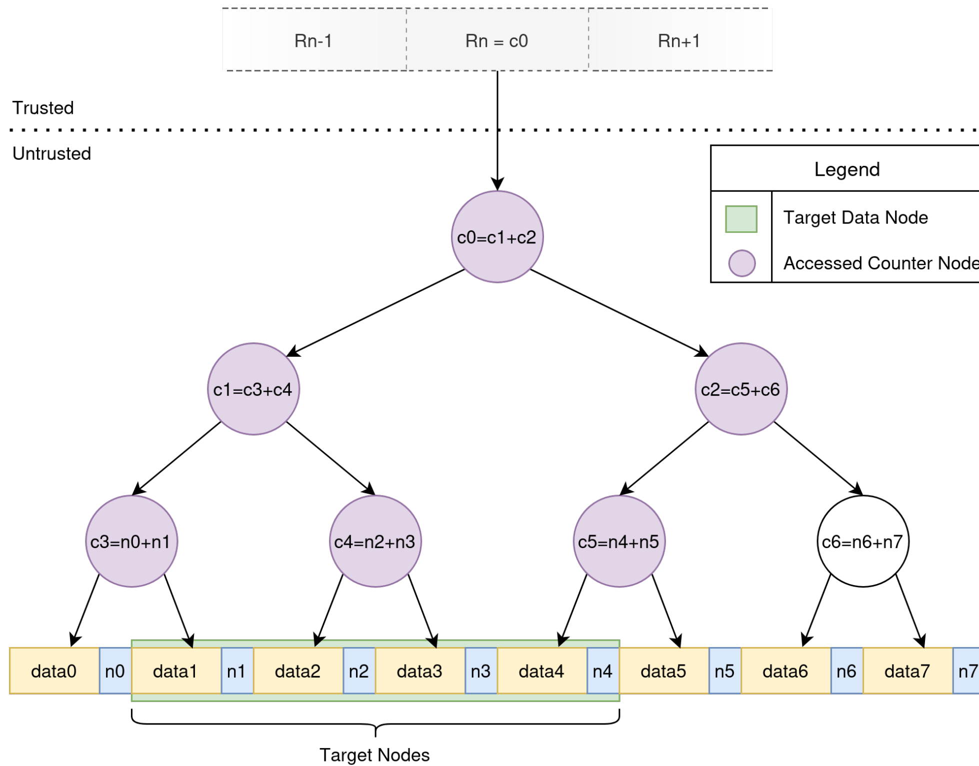


Figure 3.6: DAT Example Layout - Balanced

Using this example, a 64-byte block of memory can have all parent nodes cached using at most 7 cache entries and as little as 3 entries (the root node is stored securely in the secure root). Requests from the CPU cache for an additional 256-byte line would be split into 32, 64-bit transactions (Equation 3.2a), as the entire line can not be sent over the 64-bit bus. These 32 transactions are split across 4 separate data blocks

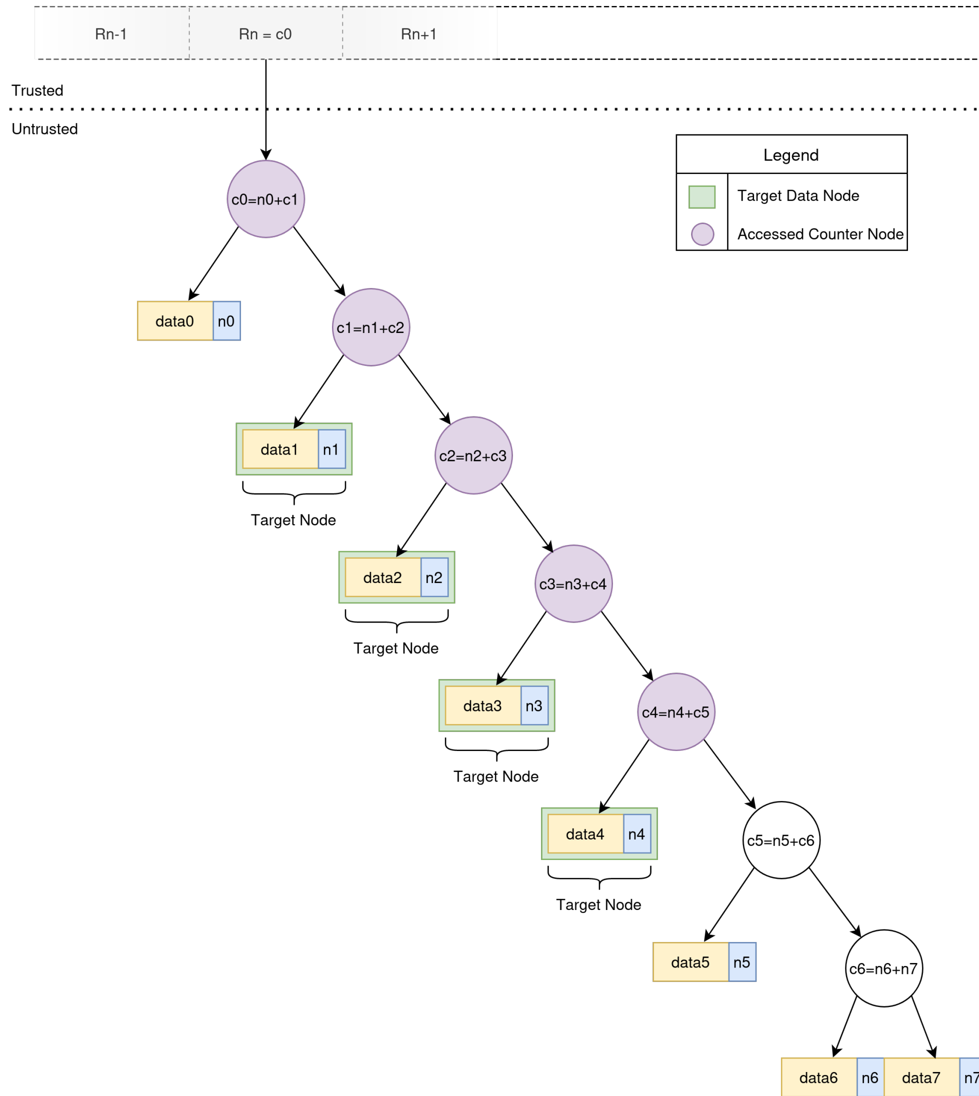


Figure 3.7: DAT Example Layout - Unbalanced

Table 3.1: DAT Cache Refill Node Access Numbers

	Balanced Tree	Unbalanced Tree
Average Node Depth	**3**	$\mathbf{3.5} = \frac{(2\cdot8)+(3\cdot8)+(4\cdot8)+(5\cdot8)}{32}$
Total Counter Node Reads	$\mathbf{96} = 3 \cdot 32$	$\mathbf{112} = 3.5 \cdot 32$
Unique Counter Node Reads	**6**	**5**
Cached Node Reads	$\mathbf{90} = 96 - 6$	$\mathbf{107} = 112 - 5$
Percent Cached Reads	$\mathbf{93.75\%} = 90 \div 96$	$\mathbf{95.54\%} = 107 \div 112$

(Equation 3.2b). Each of these 32 transactions would require decryption and data authentication of each counter node leading to the root of the tree (Equation 3.2c), with D_{depth} referring the number of counter nodes between the data node and the root.

DAT Cache node access equations where:

$C_{\#}$: is the number of counter node reads that occur

$$2048 \text{ bits} \div 64 \ \frac{\text{bits}}{\text{request}} = 32 \text{ requests} \tag{3.2a}$$

$$2048 \text{ bits} \div 512 \ \frac{\text{bits}}{\text{block}} = 4 \text{ blocks} \tag{3.2b}$$

$$C_{\#} = 32 \text{ requests} \cdot D_{\text{depth}} \tag{3.2c}$$

$$512 \ \frac{\text{bits}}{\text{block}} \div 64 \ \frac{\text{bits}}{\text{request}} = 8 \ \frac{\text{requests}}{\text{block}} \tag{3.2d}$$

As each data node is read 8 times in total (Equation 3.2d), the total number of parent node reads is very large. Table 3.1 demonstrates a comparison regarding the node caching results of the two tree layouts shown in Figure 3.6 and Figure 3.7.

Assuming the node cache is blank when the read process begins, the first time a unique counter node is accessed, it is written into the node cache. This results in each unique counter node being read from memory a single time. The following counter node reads for the following requests would happen straight out of node cache. In

this example, counter nodes are read from cache at least 93% of the time, regardless of tree layout.

3.4.2 DAT Cache Addressing

DAT allows for configurable sized counter nodes, and each counter node is stored packed in memory (zero bits between each entry). Counter nodes being packed in memory ensures that the address offset of each counter node is a multiple of the counter node size in bytes. As it is guaranteed that each counter node will be larger than a single byte, a few of the least significant bits of the node address will remain unused. These unused bits reduce the effective address width. In order to fully utilize the address space of the cache, each counter node address must be shifted right by the number of unused bits. The value of this shift can be calculated using Equation 3.3.

$$\text{Addr}_{\text{Shift}} = \log_2(\text{Addr}_{\text{Width}} \; \& \; \overline{(\text{Addr}_{\text{Width}} - 1)}) \tag{3.3}$$

3.4.2.1 DAT Cache Addressing Example

Assuming DAT uses 192-bit counter nodes (24 bytes), and each counter node is stored packed in memory (zero bits between each entry), the cache addresses can be reduced in size to improve cache hit percentages. Since each DAT counter node is 24 bits in length, represented as $\texttt{11000}_2$ in binary and $\texttt{0x18}_{16}$ in hexadecimal, the rightmost 3 bits in the address will always be zero. The counter node addresses in cache can be shifted right three bits. With a 10-bit cache address width ($\log_2 1024$), a non-shifted address only allows 128 entries to be accessed since the 3 least significant bits will always be zero, making the address effectively 7 bits in width. However, shifting the address right by 3 bits allows all 1024 entries in the cache to be accessed.

3.4.3 DAT Cache Population

The DAT cache operates in a write-through style. During the write-back stage of the DAT, all counter node writes are also simultaneously sent to the cache. Writing to the cache at the same time as memory avoids an additional stall while waiting for the cache handshake. Performing all counter node writes as write-through also ensures that both the cache and memory are constantly up-to-date.

To improve read performance, counter nodes read from memory (those non-existent in cache) are written to the node cache directly after read. This mechanism provides the option for future reads to take advantage of cached counter nodes instead of relying on write operations to populate the cache.

3.4.4 DAT Cache State Machine

To improve performance, the dynamic access tree's cache operates semi-asynchronously from the tree's primary state machine. Figure 3.8 shows the DAT cache's read state machine.

The DAT cache state machine performs cache pre-fetching based on the state of the primary DAT state. Whenever the tree either reads a data node or moves up a level in the tree, the current nodes parent node is read. Since the parent node is always a counter node, it is guaranteed to possibly exist in the cache. If the parent node exists in cache and the CPU is requesting a data write, the current node's uncle node is read from cache as well. However, if the parent node does not exist in cache, the uncle node is not read. The parent and uncle nodes are read from cache *before* the master state machine requests parent or uncle reads from primary memory.

This asynchronous, pre-fetched read does not inhibit the master state machine by forcing it to wait for cache reads to complete. When parent or uncle nodes are needed by the primary DAT module, it can nearly instantly check for locally registered parent or uncle node values before issuing memory reads for the same nodes. Not only does

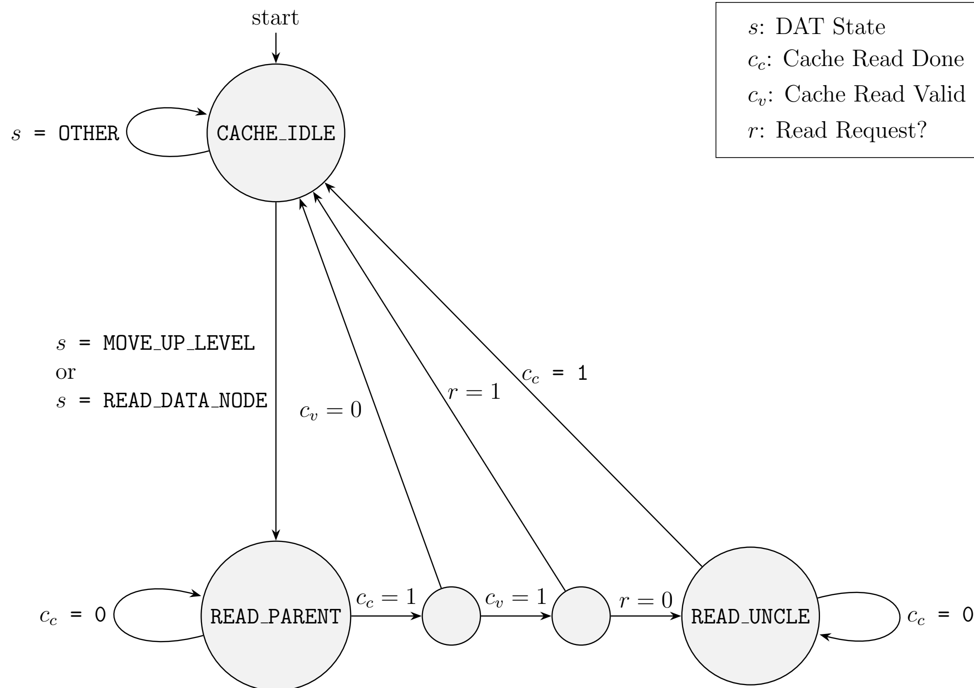


Figure 3.8: DAT Node Cache Read State Machine

this improve performance, but it reduces the overall impact of adding the node cache to the already-complicated DAT module.

To signal the master state machine whether the cached nodes have been read is done using two signals, `{parent/uncle}_cache_read` and `{parent/uncle}_cache_valid`. Upon moving up a level in the tree, both of these signals are set to '0'. When the cache asserts a cache read valid signal, the corresponding `_cache_read` signal is set high. The `_cache_valid` signal is **always** set to the same value as the cache read port's `hit`, signal. The corresponding `read` and `valid` signals are held at that value until the tree either moves up a level, or ends the current transaction.

3.4.5 DAT Cache Implementation Decisions

Various cache behaviors and architectures were examined during the design process. The decision regarding the finalized implementation was based on various levels of empirical testing using synthetic benchmarks. The choice to read uncle nodes from cache only if its sibling (target node's parent node) was present in cache is an exam-

ple of this process. Fetching the uncle node from cache only happens during write operations as part of the rebalancing process. If the parent node exists in cache, the cached node can be fetched and immediately used to calculate the uncle node's address. If the parent node does not exist in cache however; the cache read state machine must first wait for the memory read operation to complete before calculating the uncle node's address. The delayed uncle node cache request then causes a stall in the primary pipeline while the cache request is propagated through the cache controller. When this stall was observed, the likelihood of the uncle node also not existing in cache was extremely high. In this situation, the pipeline stall was wasted time in which the memory read process could have started. Since parent and uncle nodes are **always** read together during write operations, they are both written into cache together as well. The cache addressing (subsection 3.3.1) policy and their proximity in memory lessens the probability of only one of the two getting overwritten. These two aspects of the design lead to reduced performance if a cache read request for the uncle node was attempted after reading its non-cached sibling from memory. This reduction in performance drove the decision to only attempt reading the uncle node from the cache if its sibling was also existent within cache.

3.5 Additional Performance Improvements

3.5.1 Request burst wrapping

To improve performance, many bus protocols allow for burst transfers (section 4.1.2). Due to hardware addressing, these burst are limited to either certain address spaces or boundaries. During DAT initialization upon writes or data-block reading, large, contiguous chunks of memory are read from memory at once. AXI-4 for example, has 4 KB aligned wrap boundaries (section 4.1.2.1). It is not guaranteed that either a data-block or tree metadata will not cross a 4 KB memory boundary, and purposefully

preventing it wastes memory due to padding.

In the instance that a burst request crosses a burst boundary imposed by the bus protocol, the offending request must be split into multiple requests that avoid crossing the burst boundary. Figure 3.9 demonstrates an example of an incorrect burst request that crosses a 4 KB burst boundary.

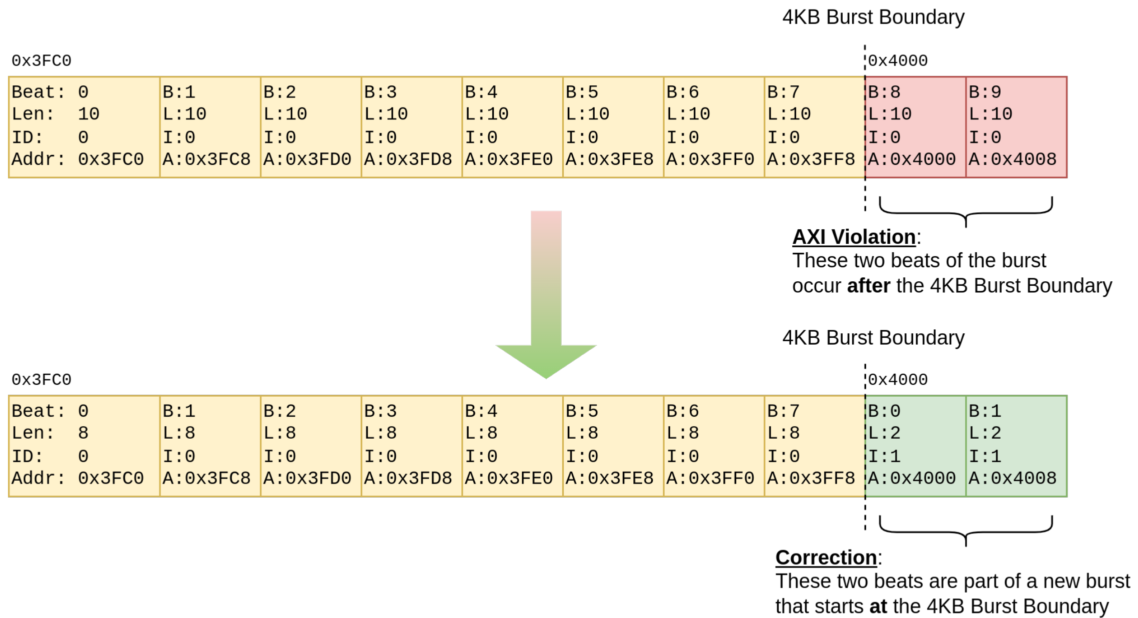


Figure 3.9: Burst request with incorrect wrap parameters

As the burst starts at `0x3FC0` and has a length of 10 beats, each with a length of 64 bits (8 bytes), the burst will end at address `0x4010`, crossing the 4 KB boundary at `0x4000`. One potential solution to this issue is to split the 10 different beats across 10 different requests. This would prevent the burst from crossing the 4 KB boundary. The downside to this approach; however, is that the pipeline must perform 10 full read/write handshakes, each one creating additional memory access latency. This issue presents the ideal solution to the problem mentioned above, creating multiple burst transfers, each maintaining proper burst boundary rules.

3.5.1.1 Write burst wrapping

Due to the tree's separate read/write pipeline behavior (Figure 2.11), separate burst wrapping blocks were required for read and write requests. DAT-based controllers only generate burst write requests during tree initialization while writing the initial counter values for the current tree. During tree initialization, the "Tree Initializer" block is responsible for taking write initializer burst requests and generating multiple independent beats, each with the proper initialization data. This process is seen in Algorithm 1.

Algorithm 1: Dynamic Tree Initialization Routine [2]

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

```
struct {
    parent;
    sibling;
    LR;
    ID;
    count;
} Node;
for i ← 0 → T − 0 do
    N[i].ID ← i;
    N[i].LR ← ¬(i( mod 2));
    N[i].parent ← ((i + 1)/2) − 1;
    if N[i].LR = 0 then
        N[i].sibling ← i + 1;
    else
        N[i].sibling ← i − 1;
    end
    N[i].count ← 0;
end
```

Since each request may contain unique data, separate beats must be generated to populate the data field of counter location. As separate beats are generated creating burst wraps is trivial. Using the example provided above (Figure 3.9), generating wrapped burst beats is as simple as subtracting the extra burst lengths from the

request's burst length. The original initialization write request contained a burst length of 10, so 10 individual requests are created one for each beat. The burst length reported within each beat request is the amount of remaining beats after the current one. The first beat has a burst length of 9, then 8, 7, etc... These burst lengths are used to notify later tree blocks, like the memory writer port, how many more requests are remaining, as well as the total burst length to place on the bus. In this initialization request example, the master interface of the memory controller will inform the memory that the following write sequence is a burst of 10 beats.

To correct the burst lengths of initialization write requests, the burst wrapper block calculates the start address and the ending address of that burst boundary. For efficient hardware implementation, this is done using bitwise operations and a single addition operation. The equations used to calculate these values are shown in Equation 3.4 and 3.6.

$$\text{AddrBound}_{\text{Start}} = (\text{Addr}_{\text{Start}} \mid \texttt{0x0FFF}_{16}) + \texttt{0x0001}_{16} \tag{3.4}$$

$$\text{Addr}_{\text{End}} = \text{Addr}_{\text{Start}} + \text{BurstLen}_{\text{Bytes}} \tag{3.5}$$

$$\text{AddrBound}_{\text{End}} = (\text{Addr}_{\text{End}} \mid \texttt{0x0FFF}_{16}) + \texttt{0x0001}_{16} \tag{3.6}$$

The variable $\text{AddrBound}_{\text{Start}}$ denotes the 4 KB boundary that cannot be passed given the bursts starting address. $\text{AddrBound}_{\text{End}}$ is the 4 KB burst boundary that connect be passed of the burst end address. If these values are the same, the burst will not wrap over a burst boundary. However, if they differ, the burst will pass over the burst boundary and must be corrected. Calculating the correct length of the current burst can be done using Equation 3.9.

$$\text{BurstLenWrapped}_{\text{Bytes}} = \text{Addr}_{\text{End}} - \text{AddrBound}_{\text{End}} \tag{3.7}$$

$$\text{Wraps}_{\text{Bool}} = \text{AddrBound}_{\text{Start}} \equiv \text{AddrBound}_{\text{End}} \tag{3.8}$$

$$\text{BurstLen}_{\text{Bytes}} = \text{Wraps}_{\text{Bool}} \begin{cases} \text{BurstLen}_{\text{Bytes}} & \text{True} \\ \text{BurstLenWrapped}_{\text{Bytes}} & \text{False} \end{cases} \tag{3.9}$$

The correct burst length of each of these beats is now calculated using Equation 3.9. Since each beat contains the amount of beats left in the burst, the beats will report burst lengths of 7, 6, 5, etc... down to 0; however, as there are still two beats left, Equation 3.9 will wrap the final two beats burst lengths back to 1, then 0.

Correcting the burst lengths of each beat creates two bursts over the course of 10 beats, a burst of 8 beats, followed by a burst of 2 beats starting at the burst boundary.

3.5.1.2 Read burst wrapping

Burst requests are used within the controller's read pipeline as well; however, compared to the write pipeline, the burst requests are not split into individual beat requests. Since the burst requests are not split into individual beats, only a single request is sent down the pipeline. Wrapping the single request not only requires the same burst length correction from Equation 3.9, but it also requires the generation of a second burst request using the remaining burst length. When the adjusted request has a remaining beat count of zero, the second request is created by generating a beat with a size denoted by the total number of remaining beats in the transaction. Then, on each request handshake, the address of the returned request will be incremented by the size of the data, and the beat counter will be reduced until there are no more outstanding beats in the transaction.

The original burst request (Figure 3.9), is updated to a burst length of 8, followed

by the generation of a second request with a burst length of 2. The second request also has both the block and virtual addresses incremented by 64 (8 previous bursts, each with a size of 8 bytes), so the new burst leaves off where the other began. After all beats are accounted for by the burst wrapper, it sends a ready signal to accept the next read request.

Chapter 4

Memory Controller Framework

4.1 Memory Controller Security Model

The architecture of many embedded systems places large amounts of memory off-chip. A variety of protocols are used to access said off-chip memory, such as SPI, Quad-SPI, DDR, etc... This research focuses on the use of external DDR memory. To evaluate the cached designs, an Avnet ZedBoard with a Xilinx Zynq series AP SoC and 512 MB of external DDR3 memory was used as the target device.

Figure 4.1 displays how the memory authentication controller integrates into the hardware.

Since external memory is accessed very frequently during operation by the main CPU, it is imperative that both the bandwidth and latency of memory is maximized/minimized respectively. Implementing the memory controller within the Zynq's PL allows the memory controller to take advantage of specialized hardware and higher performance than a software based protection method.

Implementing the memory controller within the PL does not just offer higher performance, but also lower level access. As the Zynq's ARM CPU has direct access to the PL fabric, it can send and receive data to the memory controller. The AXI-4 protocol and bus is used as the interface between the PS and PL portions of the Zynq.

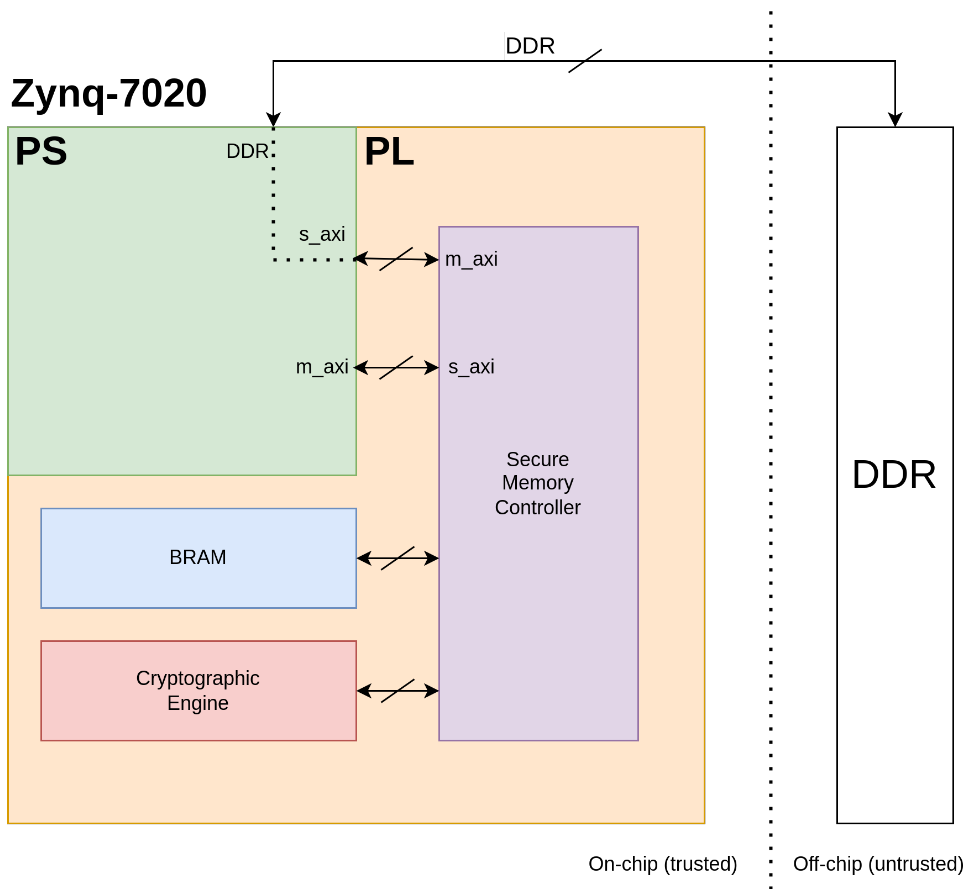


Figure 4.1: Memory Authentication Pipeline Hardware Design

4.1.1 Memory Controller Encryption Model

The memory authentication trees used within this thesis used the block-level AREA authentication model (subsection 2.2.4). One of the requirements for this authentication to properly detect data tampering is the requirement for the use of either a stream cipher, or block cipher mode that propagates errors between blocks. A block cipher mode of operation that exhibits this property is Cipher Block Chaining (CBC) mode. The memory controller presented in this thesis uses a hardware implementation of AES that operates in the CBC mode to fulfill this requirement.

Plaintext data is fed into the encryption engine directly, meaning that the length of each plaintext must be 128-bits in length. To meet this requirement, the payload of multiple requests contained within a burst are concatenated together until they reach a length of 128-bits. After the encryption/decryption process, the output data is once again split up and linked with the original request. Given the flexible nature of the pipeline architecture however, it is not guaranteed that the burst will have a total data length that is a multiple of 128-bits. To support these cases, the modification shown in Figure 4.2 was made to the CBC algorithm to allow for non-perfect data lengths to be encrypted/decrypted.

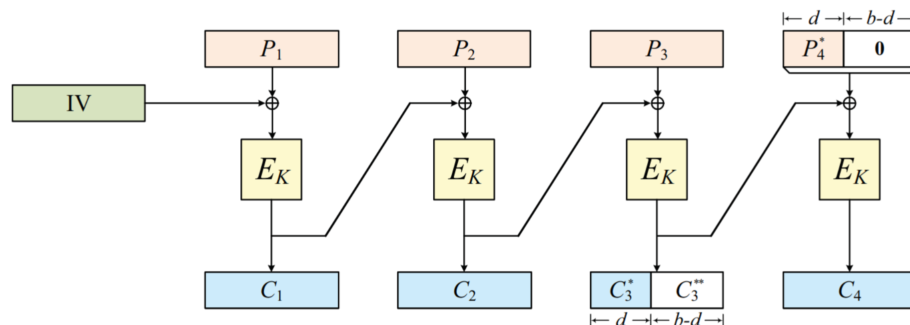


Figure 4.2: Cipher Block Chaining CTS Mode [5]

CTS, or ciphertext stealing, is a method for reusing the ciphertext of the last encrypted block to allow for the encryption of a plaintext block that is not aligned to 128 bit boundaries. By appending part of the previous ciphertext to the end of the

plaintext block that is shorter, the plaintext can then be encrypted using the standard encryption process. The appended length can be thrown out after the operation. The same procedure can be replayed while decrypting the data, negating the requirement for storing the additional padding in memory, thus reducing memory overhead.

4.1.2 AMBA AXI4 Interface Protocol

The AMBA (Advanced Microcontroller Bus Architecture) AXI-4 (Advanced eXtensible Interface 4) interface protocol is a flexible bus protocol developed by ARM for use by on-chip busses [19]. To communicate with the programmable logic, Zynq processors use full-speed AXI-4 interfaces. The transaction handshake system used by AXI-4 is composed of two signals, 'valid', 'ready'. The following handshake convention is used: the subordinate interface sets a 'valid' line high when the bus lines are set and ready for the next transaction. The manager interface then performs the transaction and sends a 'ready' signal back to the subordinate to let it know the transaction has been completed. In an effort to support a wide-variety of use cases, AXI deploys extra signals to denote special transaction types, such as user vs kernel transactions, large data transfers through a single request, DMA, etc... Additionally, AXI is an open standard, allowing any architecture to implement it royalty-free. As a result, AXI-4 was chosen as the communication protocol for the memory controller due to its versatility, speed, and prominent use in embedded systems.

On resource constrained system or low-bandwidth IPs, AXI can be used in either AXI-Lite or AXI-Stream mode to either ease the design challenges, or reduce PL utilization.

4.1.2.1 AXI Addressing Scheme

AXI-4 uses an addressing granularity of 4 KB, meaning that each AXI interface must begin at a 4 KB boundary in virtual memory. The flexible nature of the AXI protocol

lends itself well for use in a variety of hardware IP categories, such as GPIO controller, Ethernet, or even memory controllers. To support this wide variety of use cases, there are two addressing modes: register-based and memory addressing. Nothing differs between the two either physically or at the protocol level. The addressing mode of an AXI subordinate interface is reported by the IP definition containing the interface, and is used by software tools to determine how the IP can be used. Register-based addressing is similar to that of a memory-mapped peripheral on a microcontroller, where each control register is bound to a virtual memory address. The register addresses are both defined and handled within the IP in whatever way it chooses. The second addressing mode is memory-based. Using this mode, no registers are defined for the IP. The address range instead maps to a block of data, whether it be physically or virtually represented.

The IP defined within this thesis presents itself as a memory mapped AXI IP. This allows vendor tools, such as Xilinx's Vitis, to generate linker scripts that place either code or data within the memory range of the AXI IP.

4.2 Memory Controller Pipeline

4.2.1 Memory Encryption Pipeline

While the AXI-4 protocol was used for both CPU and memory transactions, inter-system communication within the memory controller used an interface independent request system. The architecture of the request system used was highly influenced by AXI however; as the handshake system uses the same 'valid', 'ready' scheme as AXI, and the requests contain the same transaction burst and data length metadata as AXI. Figure 4.3 displays the architecture of the memory controller pipeline, with the different bus types annotated.

At any point in which the internal request system is translated to/from the ex-

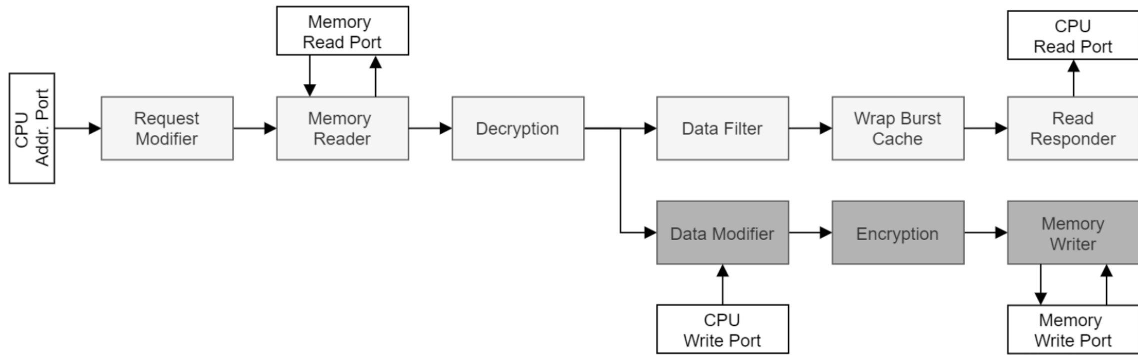


Figure 4.3: Memory Controller Encryption Pipeline

ternal communication bus, in this instance AXI, a bus de-serialization block is used. Keeping the internal request system independent of the processor/memory bus provides the opportunity for the memory controller to be implemented with relatively little change on a variety of memory bus types.

The encryption pipeline follows a read-modify-write approach. As the entire contents of memory in encrypted, any memory access must first be read into the pipeline, decrypted, then modified before write-back. As a bonus, the read-modify-write architecture provides a strong base for adding memory authentication to the pipeline. Since memory authentication relies on NONCE's linked to each data block for authentication, the read-modify-write approach can be configured to read the corresponding NONCE along-side the data at the specified address.

To improve performance of the design, the pipeline also includes burst support for the AXI protocol. AXI bursts allow multiple sequential memory read/writes to be made using a single request, reducing the amount of time spent generating and sending requests.

As discussed in subsection 2.2.4, the use of Block-Level AREA requires the entire data block (including metadata) to be read from/written to when any subset of data contained within is accessed. As the size of data blocks are somewhere in the tune of 64 bytes in length (at least), burst reads and writes are used to transfer the data to

and from memory without generating a multitude of individual read or write requests.

4.2.2 Authentication Pipeline

The most significant modification to the pipeline, illustrated in Figure 4.4, was the addition of the tree request generation block. The tree request generator is responsible for the following tasks:

1. Converting virtual memory addresses to physical memory addresses

2. Generating memory requests

3. Authenticating tree nodes

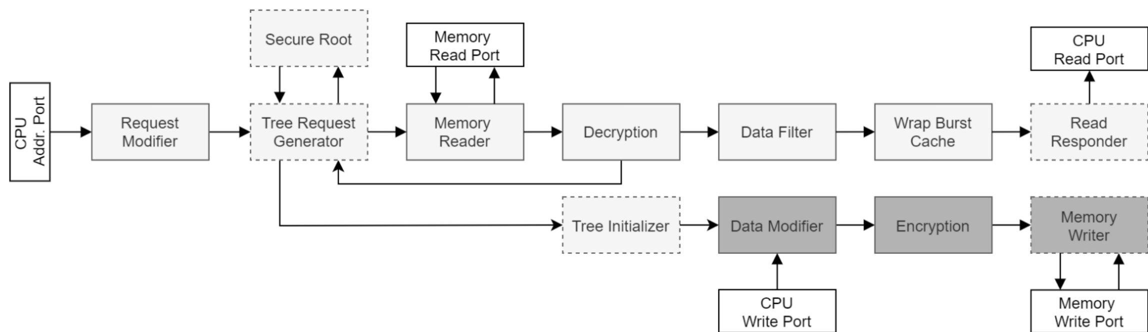


Figure 4.4: Encryption and Authentication Pipeline

To support both authentication and encryption, the pipeline was modified to return decrypted tree metadata back to the tree controller block for authentication. If a given node fails its authentication test, the tree request generator will create a read response request to inform the CPU of the failed authentication. Given the tree's ability to restructure leaf and counter nodes based on access frequency, it is required to examine, and copy all nodes in transit to determine the access patterns of each node. The request generator is able to determine the best possible restructuring method of the current working tree and generate additional requests to restructure the tree.

4.3 Hardware Testing Framework

As laid out in Figure 4.1, the memory authentication controller was able to access system memory by taking advantage of the Zynq's AXI subordinate port. The AXI subordinate port on the Zynq is translated to DDR requests the Zynq forwards to the off-chip DDR memory. This allows the PL to access the static system memory. For traditional IP that relies on the CPU for control and instructions, this architecture is very flexible and provides low level access to memory, with additional benefits/features such as speed and DMA (direct memory access). DMA is a hardware feature that allows hardware subsystems to access system memory independently of the CPU. DMA improves memory bandwidth to peripherals by reducing CPU utilization since it no longer acts as a middleman.

What separates this hardware design from traditional AXI-based IP designs is that both the memory controller and CPU depend on each other. As the point of this memory controller is to provide full memory protection or both the data and instructions, the CPU must be able to perform all memory accesses through the PL. The Zynq XC7Z020 has a memory map shown in Table 4.1.

Table 4.1: Zynq XC7Z020 Memory Map

Start Address	Size (MB)	Description
0x0000_0000	1,024	DDR DRAM and on-chip memory (OCM)
0x4000_0000	1,024	PL AXI subordinate port #0
0x8000_0000	1,024	PL AXI subordinate port #1
0xE000_0000	256	IOP devices
0xF000_0000	128	Reserved
0xF800_0000	32	Programmable registers access via AMBA APB bus
0xFA00_0000	32	Reserved
0xFC00_0000	64 MB - 256 KB	Quad-SPI linear address base address (except top 256 KB which is in OCM), 64 MB reserved, only 32 MB is currently supported
0xFFFC_0000	256 KB	OCM when mapped to high address space

As AXI memory addresses are mapped within the PL range, the PL must be initialized before any data can be read or written to the memory controller. Using Xilinx's standard first-stage bootloader (FSBL), this is not possible because it restricts program loading to the DDR address range, not the entire available address space.

4.3.1 FSBL Modifications

The FSBL is responsible for handling the initialization and boot process of the hardware. The default FSBL behavior is to load a 'BOOT.BIN' file off of either an SD card or flash memory. This 'BOOT.BIN' file contains the boot information for the FSBL, the PL bitstream, and as the primary program binary. The FSBL performs this action by reading each "sector" of the 'BOOT.BIN' file in order, directly copying the data from the ROM to the corresponding memory address stored within the sector. During the FSBL program loading phase, the DDR memory is not yet initialized, so the FSBL is loaded directly onto the Zynq's PS OCM. The OCM is large enough to store the FSBL and a small buffer of data, and its location within the PS makes it more difficult to exploit with physical attacks.

The modified FSBL behavior (shown in Figure 4.5), loads the FSBL and bitstream as normal; however, it initializes the hardware in a different order. As the memory controller both encrypts and authenticates all data passing through it, the format and organization of data within physical memory is handled by the tree. Bypassing the memory controller and writing the program binary directly to RAM would lead to hardware panics, as the required tree metadata would be missing for authentication. This would cause memory authentication errors to be thrown on the first data access. There are two solutions to this issue:

1. Use a software implementation of the deployed authentication tree to create a modified version of the program binary. This would both encrypt and format the binary's data in a way that contains the required tree metadata as it would

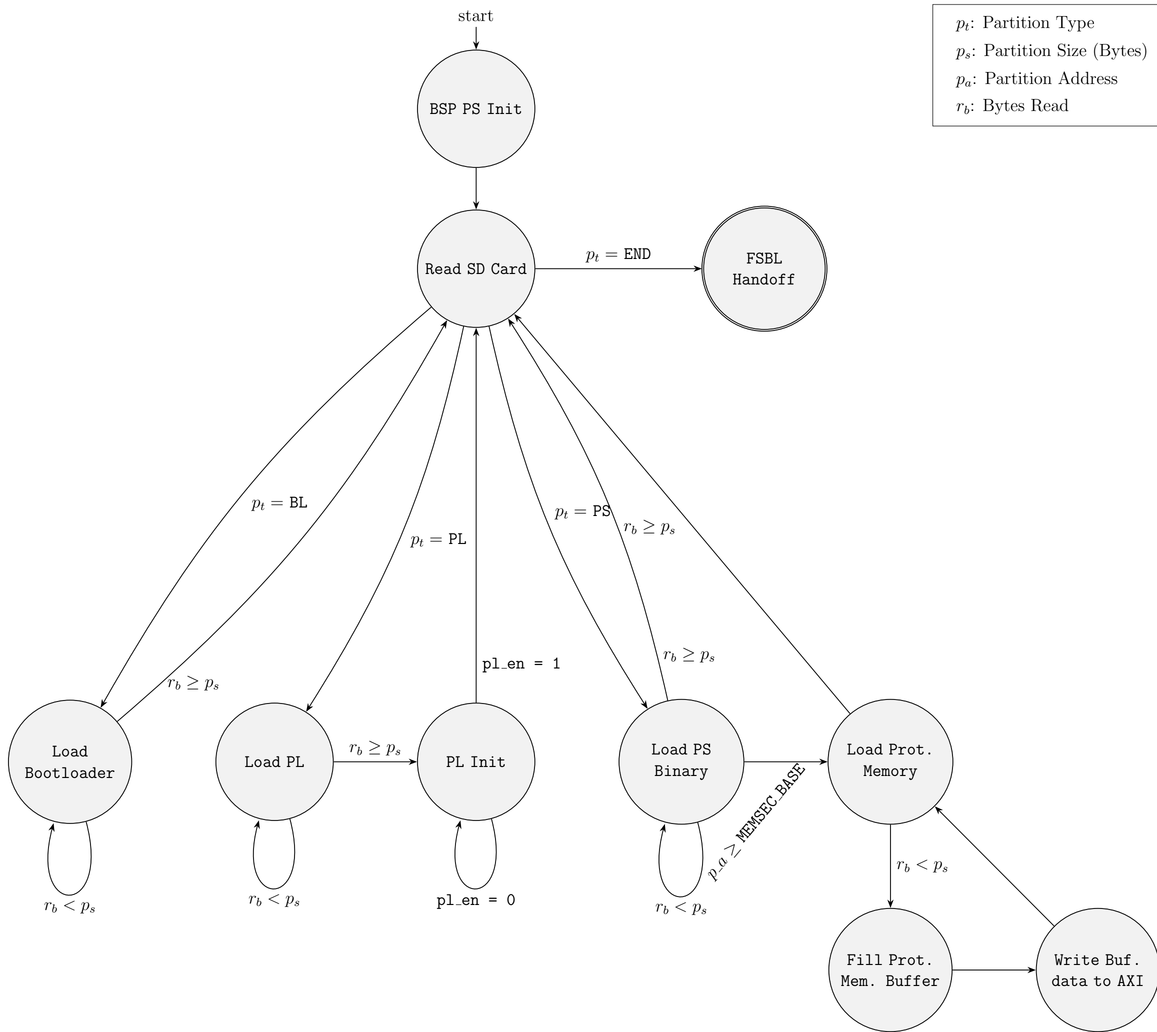


Figure 4.5: FSBL State Machines

exist in memory.

2. Keep the original program binary format and use the FSBL to load it into memory through the authentication tree.

The first solution would require the least amount of hardware effort; however, it would also reduce hardware flexibility. The hardware implementations explored by this research are very flexible, as they allow developers to configure the IP to fit their needs the most. Since the memory controller lives in PL, it also means that software can be written and compiled to a single binary then run on any configuration of the IP. Reformatting the binary to include metadata for the authentication tree would require a different binary for every single configuration of hardware, not just for different host processor architectures.

The second solution requires more engineering on the hardware/driver side. To make sure the program binary is encrypted and authenticated properly for any configuration of the memory controller, it can be fed through the controller to be placed in RAM with the correct associated metadata and formatting. Since the PL must be initialized for the authenticated memory controller to function, the PL must be initialized before the program binary is loaded into it. This presents a problem as the Xilinx provided FSBL doesn't initialize the PL until after all binaries have been loaded, and directly before transferring execution to the program binary.

To allow the program memory to be loaded onto RAM *through* the authenticated memory controller, three modifications of the FSBL were made (shown in Figure 4.5).

1. Allow loading binaries into memory locations outside DDR

2. Modify the binary copying code to support AXI-based transactions

3. Initialize the PL immediately after loading it to support PL use during FSBL

4.3.2 PL Binary Loading

The FSBL's 'BOOT.BIN' file contains multiple partitions. Each partition stores a binary with a different purpose. The partition header specifies information about each sector, including what it is, and what address it should be written to. By default, the FSBL will reject any partition that has a destination outside the Zynq's DDR range (Listing 4.1).

```
if (PSPartitionFlag && (PartitionLoadAddr > DDR_END_ADDR)) {
    fsbl_printf(DEBUG_GENERAL,
                "INVALID_LOAD_ADDRESS_FAIL\r\n");
    OutputStatus(INVALID_LOAD_ADDRESS_FAIL);
    FsblFallback();
}
```

Listing 4.1: Original FSBL Load Address Checking

To allow loading memory in PL, the code in Listing 4.1 was updated (Listing 4.2) to allow code to be loaded **if and only if** it existed within the memory range of the `MEMSEC` block. The `MEMSEC` block addressing is defined as `XPAR_MEMSEC_1_(BASE/HIGH)ADDR` because the IP is named `memsec_1` within the hardware block design.

```
if (PSPartitionFlag && (PartitionLoadAddr > DDR_END_ADDR)) {
    if ((PartitionLoadAddr >= XPAR_MEMSEC_1_BASEADDR) &&
        (PartitionLoadAddr <  XPAR_MEMSEC_1_HIGHADDR)) {
        fsbl_printf(DEBUG_GENERAL, "PL_LOAD_ADDRESS\r\n");
    } else {
        fsbl_printf(DEBUG_GENERAL,
                    "INVALID_LOAD_ADDRESS_FAIL\r\n");
        OutputStatus(INVALID_LOAD_ADDRESS_FAIL);
        FsblFallback();
    }
}
```

Listing 4.2: Modified FSBL Load Address Checking

This code provides flexibility as the addressing of the `MEMSEC` block is not hard-coded, allowing the hardware designer to map the AXI addressing of the memory

controller to whatever range fits the needs of the design. This change only allows PL writes within the `MEMSEC` data block because writing to an uninitialized AXI address will cause the CPU to hang while waiting for an AXI response.

In order for this change to work properly, the code to initialize the PL was moved from directly before the program handoff to the callback function called directly after loading the PL binary: `FsblHookAfterBitstreamDload()`.

4.4 Results

As the primary goal of this research is to discover the viability of dynamically weighted authentication trees with node caches, it must be compared to similar designs in terms of both performance and hardware costs. Given that the design of the cached DAT was based upon the original implementation of an unordered DAT [2], the two were compared to measure the performance improvements brought by the node cache. The unordered DAT was chosen as the basis of the cache design over the ordered DAT because the unordered design utilized significantly less FPGA resources with nearly identical performance [2]. Additionally, the design of the DAT was heavily influenced by that of the TEC-Tree, which uses a similar AXI-based memory pipeline to both the cached and uncached DAT implementations, allowed for the use of a unified test suite.

4.4.1 Authentication Tree Hardware Cost

It is important to consider the physical implementation of the researched trees while addressing viability concerns. The hardware target, a Xilinx Zynq XC7Z020, imposed a fabric clock timing constraint of 50 MHz due to design routing [3]. The available programmable logic resources of both the target hardware and similar devices are displayed in Table 4.2.

Table 4.2: APSoC Resource Comparison

	Zynq-7007S	Zynq-7020	Zynq-7100
Logic Cells	23k	85k	444k
LUTs	14400	53200	277400
Flip Flops	23800	106400	554800
DSP Slices	66	220	2020
# 36Kb Block Ram	50	140	755

4.4.1.1 Synthesis Results

Each design was synthesized using Vivado v2021.1. The same parameters were used to synthesize the TEC-Tree, cached DAT, and original DAT implementations. The trees were configured to use 64-byte data blocks with 8 tree roots, and 128 node cache entries. As it is important to gauge the hardware utilization of each tree's "full suite of features", the trees were each synthesized twice: once with the encryption engine enabled, and again with it disabled. This is done for two reasons: one being that the encryption engine used between the two tree classes differs (DAT uses AES-128, and TEC-Trees use ASCON-128), and the other being that the encryption engine contributes to a large portion of the fabric utilization, as demonstrated in Table 4.3.

Table 4.3: Synthesis Utilization Reports

	TEC-Tree	Cached DAT	DAT
Encryption			
LUTs	9240	13816	10857
Flip Flops	4416	7296	5743
DSP	0	15	15
BRAM Tiles	2.5	7.5	1
No Encryption			
LUTs	4846	11369	8233
Flip Flops	3429	6643	5095
DSP	0	15	15
BRAM Tiles	2.5	6.5	0

Comparing the resource utilization between the two tree types displays the sig-

nificant resource savings that can be achieved by using a TEC-Tree based design. This was the expected outcome given that the TEC-Tree architecture is inherently simpler than that of the DAT. The static arrangement of both counter and data nodes make TEC-Trees very efficient to implement since the leaf node depth and position is known at compile time. DAT on the other hand, requires a large state machine to track the current position in the tree, as well as support for comparing node counts, issuing reorder requests, and authenticating the reordered nodes. This additional logic requires roughly 1.7 and 1.5 times the amount of LUTs and FFs as the TEC-Tree design, respectively. Adding the node cache logic on top of this brings the additional LUT and FF usage to 2.5 and 1.9 times respectively.

Something else important to note is the additional use of digital signal processing (DSP) units for the DAT. DSP blocks are used to perform logic-heavy mathematic operations, such as multiplication, with minimal latency by taking advantage of dedicated arithmetic hardware separate from the PL's logic slices. Within DAT, DSP blocks are used for the calculation of node write-back addresses. If the use of DSPs is either desired or required, due to lack of physical DSP blocks in target fabric for example, they can be either omitted through the use of synthesis flags, or block sizes can be changed to allow for simple bitwise operations for write-back address calculations.

4.4.1.2 Implementation Results

The synthesized design was sent through Vivado's (version 2021.1) implementation engine with default optimization and routing settings targeting the Zynq XC7Z020. The resulting utilization report is contained in Table 4.4.

As expected, running the synthesized designs through Vivado's implementation engine reduced the resource utilization of the designs by a small margin. Something to note is the introduction of an additional row in the table, LUTs used for memory. In the case of TEC-Tree, the secure root module was written such that the root values

Table 4.4: Implementation Utilization Reports

	TEC-Tree	Cached DAT	DAT
Encryption			
LUTs	9029	13736	10780
Flip Flops	4395	7279	5725
DSP	0	15	15
BRAM Tiles	2.5	7.5	1
LUT (Mem)	44	0	0
No Encryption			
LUTs	4771	11283	8188
Flip Flops	3409	6624	5082
DSP	0	15	15
BRAM Tiles	2.5	6.5	0
LUT (Mem)	44	0	0

of each tree were actually stored within the fabric itself instead of BRAM. Another thing to mention is the higher BRAM usage for DAT caches over TEC-Tree caches. This is solely because of the extra metadata required within DAT counter nodes. This does limit the maximum number of DAT node cache entries more than it does the TEC-Tree, however; observing Table 4.5, shows that neither design is close to hitting the entry limit with the current configuration.

Table 4.5: XC7Z020 Resource Utilization Percentages

	TEC-Tree	Cached DAT	DAT
Encryption			
LUTs	16.97%	25.82%	20.26%
Flip Flops	4.13%	6.84%	5.38%
DSP	0.00%	6.82%	6.82%
BRAM Tiles	1.79%	5.36%	0.71%
LUT (Mem)	0.08%	0.00%	0.00%
No Encryption			
LUTs	8.97%	21.21%	15.39%
Flip Flops	3.20%	6.23%	4.73%
DSP	0.00%	6.82%	6.82%
BRAM Tiles	1.79%	4.64%	0.00%
LUT (Mem)	0.08%	0.00%	0.00%

The percent of utilized resources proves the efficiency of the designs in hardware. Even with encryption enabled, the largest usage of any individual component is the use of LUTS, at just over 25% when implementing the cached DAT. The efficiency of the designs keep a large enough collection of resources free that other components can be implemented in the same fabric. As previously mentioned, while the extra size required for DAT counter nodes reduces the maximum node cache size, 128 cache entries only utilize 4.64% of the onboard BRAM. This points to a maximum potential DAT node cache size of 2700 entries using the same hardware. If more node cache entries or a higher clock speed is desired, the designs can be implemented onto higher end parts with newer manufacturing processes. This would yield both more BRAM and a faster fabric clock as the overall routing delay would be reduced.

4.4.2 Authentication Tree Performance

The three methods of memory authentication were run through a set of identical synthetic benchmarks for an accurate account of performance across different workloads. Both clock frequency and bus widths were chosen based on constraints set by the host SoC. The Zynq XC7Z020 used for implementing the design was only able to route the designs in such a way that the maximum clock speed for the fabric was 50 MHz. The XC7Z020 has a manager AXI interface that uses a 32-bit (4-byte) data width, clocking the fabric at 50 MHz provides a maximum throughput of 200 MB per second. Table 4.6 lists the hardware configuration used to test the authentication tree performance.

While the cache line size of the ZedBoard's host CPU is 32 bytes in width, a block size of 64 bytes was chosen because it was found to be more performant for authentication trees in previous research [3]. It is also important to note that, if the target CPU has a configurable cache line size, matching the data block size to the width of a cache line will result in lower latency memory access through the

Table 4.6: Simulation Parameters

	Value
Protected Memory Size	16 KB
Manager Data Bus Width	32 bits
Subordinate Data Bus Width	64 bits
Address Bus Width	32 bits
Data Block Size	64 Bytes
Tree Roots	8
Tree Arity	2
Cache Entries	128
Sequential Memory Read/Write	8 KB
Random Memory Read/Write	1024 Beats
Memory Hotspot Reads/Writes	2400 Beats

authentication tree. Larger data blocks will not only reduce the depth of the tree, but they will also reduce the relative storage overhead of tree metadata.

The specific version of ASCON integrated into the TEC-Tree pipeline had a latency of 6 cycles with 64-bit blocks, and the AES implementation used for DAT had a latency of 12 cycles with 128-bit blocks. The differing cryptographic engines between the TEC-Tree and two DAT methods tested were standardized by replacing the cryptographic logic with registers for the testing process. The flexible nature of the pipeline allows for the use of a variety of block ciphers, as long as they can operate in a stream mode with error propagation. The trees were configured to protect 256 MB of memory; however, the benchmarks assumed the protected memory size was 16 KB. This was an artificial limitation imposed solely by physical memory limitations of the host performing the simulations.

Authentication trees require some form of initialization to create the nodes in memory. The actual method of initializing nodes is tree dependent, but as a general rule, it happens when either a leaf node in a tree is modified, or when any node is modified for the first time and has an uninitialized parent. These tree creation procedures are time-consuming and only need to happen once after the system is powered on. Typically, this happens in the bootloader when the data and instruction

sections of the executable are loaded into the tree. To simulate this behavior, a mock bootloader ran before any other test during the simulation process. The mock bootloader forced all tree nodes to get initialized by writing the first word of each data block in protected memory. Ensuring that all nodes were initialized before the benchmarks began provided the most accurate metrics as execution time was not spent initializing tree nodes.

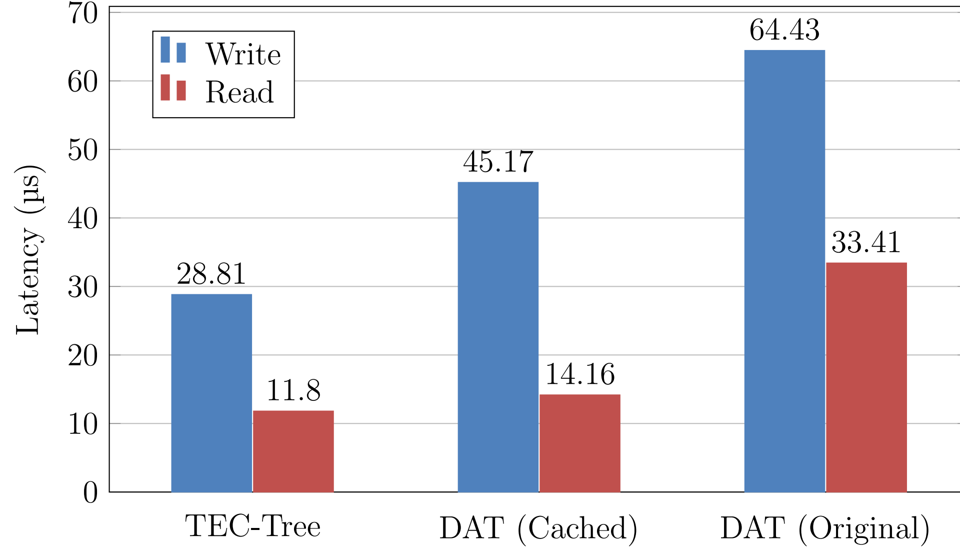
Synthetic benchmarks were performed using the Vivado v2021.1 Simulator. Three different types benchmarks were used to validate the performance of the trees in different scenarios. As a means of simulating the cost of accessing off-chip memory, an artificial memory access delay of 35 cycles was added to the testbench. Each benchmark consisted of memory writes, followed by the same number of memory read beats.

4.4.2.1 Sequential Memory Access

Whether it be uploading system crash reports, logs, or applying firmware updates, embedded systems commonly access large regions of memory sequentially. While it does not perfectly replicate real behavior, this benchmark (results shown in Figure 4.6) simulates these access patterns. Specifically, the 8 KB region of memory is broken into 2048 different beats, each one the native data bus size of 32 bits.

Sequential memory access under and authentication tree generates a series of similar tree traversal requests. With 64-byte data blocks and 32-bit requests, each set of 16 sequential beats accesses the same data block. These 16 specific accesses traverse the tree using the same path. Caching all-of the parent nodes during the first traversal will prevent any further memory reads outside the original data request. The simulation results demonstrate the latency improvements the node cache provide DAT architectures. While the original DAT had read latency roughly 2.8 times longer than that of the TEC-Tree, the cached DAT only had a read latency 1.2 times, or

Figure 4.6: Sequential Memory Access Performance



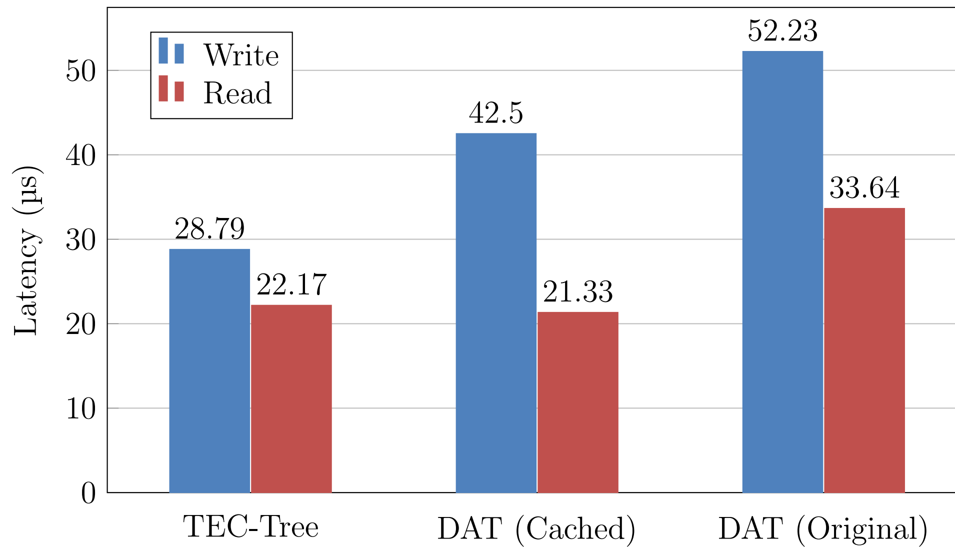
20% longer. This shows a 2.36 times speed up over the original DAT implementation. Write performance, on the other hand, improved by a factor of nearly 30%.

While the dynamic authentication tree read performance improves dramatically with node caches, the performance of write operations is hindered for two reasons: the write-through architecture of the node cache (subsection 3.3.3), and the rebalancing that occurs within the tree. The improved write performance introduced by caches stems from the cached reads of nearby nodes that occur during the rebalancing process. Despite the fact that each data node will be accessed an equal amount, the DAT is not able to predict this access pattern and will assume otherwise. This is what limits the maximum write performance of DAT's during sequential access. Nodes are rebalanced towards the top of the tree during access; however, when the sibling node is accessed a few beats later, it too is moved towards the top of the tree. This process happens continuously, causing issues with write latency.

4.4.2.2 Random Memory Access

While also not a realistic representation of memory access, accessing completely random words within memory demonstrates the worst case performance for reads. Random data block access guarantees a higher rate of node cache miss penalties because of the lack of frequent, similar, tree traversal paths. This is shown in Figure 4.7.

Figure 4.7: Random Memory Access Performance



Compared to the 2.36 times improvement to read speeds for sequential access, the DAT node cache only improves random access read latency by 1.5 times (36.6% speedup). The DAT node cache improved the performance of memory reads enough to be competitive with 2-ary TEC-Trees. The random read performance of the cached DAT was 3.8% faster than that of the TEC-Tree; however, given the random selection of memory addresses, this speed is within the margin of error to consider the two trees of equal speed. Similar to the sequential access benchmark, write speeds of the cached DAT still lag behind that of the 2-ary TEC-Tree because of the rebalancing overhead.

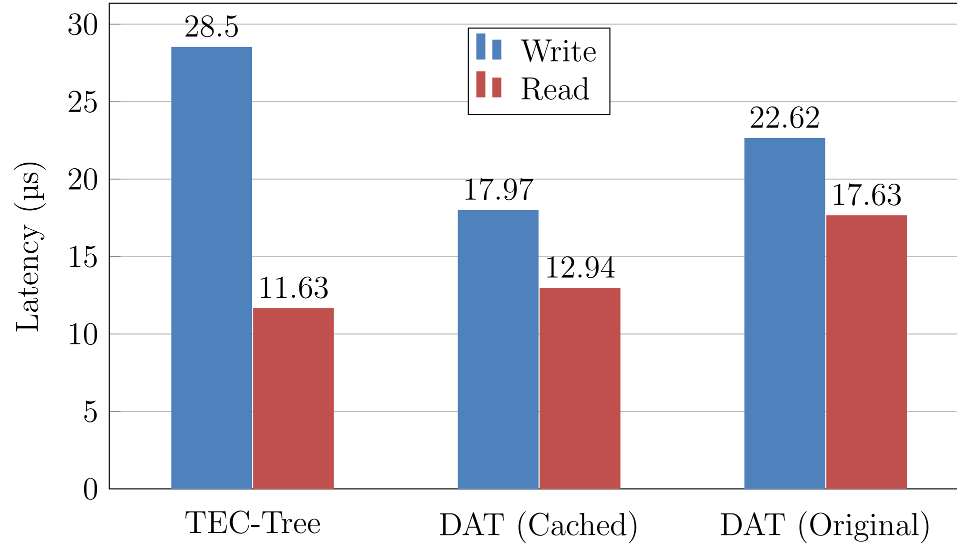
4.4.2.3 Memory Hotspot Access

A more realistic memory access pattern was created that simulates the existence of memory hotspots. The firmware running on many embedded systems frequently accesses a subset of the variables in the system. If the system is being managed by an RTOS (real time operating system), every context switch will update the task control block of the current task. Context switching is an expensive process on an embedded device; therefore, moving the data blocks containing either the top of the task stack or task control block to the top of the tree will reduce the overall latency of a task switch. This is something DAT-based designs can help mitigate, as they move commonly accessed blocks towards the top of the tree.

This behavior was simulated by generating a set of randomly sized "structs" (referring to C structures) within protected memory. Each "struct" ranged in size anywhere from 4 to 128 bytes. These structures were randomly and uniformly distributed throughout protected memory. The entirety of each of these structures was read from and written to a random number of times until there were 2400 beats both read and written. Figure 4.8 shows the access latency of all three trees using this access pattern. It is important to note that the placement, size, and access order of each structure was determined ahead of time and used for all three tree methods.

The rebalancing of dynamic authentication trees provides a sizeable decrease in write latency with workflows that have heavily "hot-spotted" memory. DAT rebalancing will place frequently used nodes closer to the top of the tree. This reduces the number of nodes that need to be authenticated, and also reduces the number of write-backs that occur on the "path to root" counter node updates. In other workflow scenarios, the rebalancing methods caused slowdowns during write operations. Given that rebalancing moves the most frequently accessed blocks towards the root, the rebalancing algorithm does not have to be run on every write. Once the target data block is already at the highest possible location in the tree, rebalancing calculations

Figure 4.8: Memory Hotspot Access Performance



no longer have to be performed while accessing this node. This is demonstrated by the fact that both the cached and non-cached DAT have lower write latency than the 2-ary TEC-Tree, with speedups of 1.56 and 1.26 times, respectively. Memory hotspots also improve the read latency of DAT data nodes. The original DAT implementation without cache had a read latency improvement of 47.6%. The cached dynamic authentication tree on the other hand had a relatively small read latency improvement of 8.6%. The latency of the TEC-Tree is nearly identical to that of its sequential performance since, while randomly distributed throughout memory, the "structs" themselves are read from memory sequentially.

4.4.3 Summary

The addition of a node cache to DAT designs proved to be an effective method of reducing both the read and write latency of data in protected memory (Table 4.7). The largest speedup occurred during sequential reads and writes with 2.36 and 1.43 times latency reduction, respectively, over the non-cached alternative. Compared to 2-ary TEC-Trees, the added node cache brings the DAT-based designs into a nearly identical performance space for memory reads, with reads being **at worst** 20% slower,

and in certain scenarios, a little (3.8%) faster than the TEC-Tree.

Table 4.7: Summary of Timing Results (μs)

Access Pattern	TEC-Tree		DAT (Cached)		DAT (Original)	
	Write	Read	Write	Read	Write	Read
Sequential	28.81	11.80	45.17	14.16	64.43	33.41
Random	28.79	22.17	42.50	21.33	52.23	33.64
Hotspot	28.59	11.63	17.97	12.94	22.62	17.63

Despite vastly improving write speeds, the speed of node reads were not able to catch up to that of the TEC-Tree because of the extra storage overhead required for DAT counter nodes. As discussed previously, Table 2.2 explains that the counter nodes used in the DAT architecture require 3 additional fields on top of the TEC-Tree counter nodes due to the extra required metadata. Each node cache miss in a DAT design requires extra read beats to read the additional metadata, and since this NONCE is also appended onto data nodes, data nodes within DAT protected memory require extra read beats as well. This limits the maximum read performance of DAT's even while bypassing memory reads through the use of caches.

Overall, node caches helped improve the performance of DAT-based designs significantly. The improved architecture explored in this thesis still lags behind similar architectures such as TEC-Trees in some scenarios, such as random access and sequential access. However, a more "real world accurate" memory access pattern that favors frequent access of specific regions are much more efficient using DAT designs. The continuously reordering design of DAT's inherently caps the maximum write performance of the tree. Limiting the frequency of both performing and calculating rebalances could reduce the performance impact while still allowing for improved performance on heavily trafficked data nodes. On systems deploying authentication trees over a large region of physical memory, the design is likely to be limited by the amount of BRAM available to the reconfigurable fabric. In these instances, a trade-off has to be made between the size of the node cache, and the number of trees

to split the data across. The more memory stored under a single tree, the larger the number of leaf nodes. Trees spanning a larger portion of memory with a greater number of tree nodes would benefit from a dynamically restructuring design as the average depth of commonly used data nodes would be reduced.

Chapter 5

Conclusion and Future Work

5.1 Future Work

While this thesis contains a complete implementation of multiple memory protection mechanisms, many improvements can be made to both the designs and the toolchain used to deploy them.

5.1.1 Tree Arity

The largest limiting performance factor of dynamic authentication trees is the limit to 2-ary trees. Only having binary tree support increases the depth of the tree, making both reads and writes much slower. Increasing the arity of the tree provides a few benefits. It can be used to reduce the average depth of a tree, or retain the same tree depth while reducing the number of tree roots. In a resource constrained environment, reducing the number of tree roots could increase the viability of the use of authentication trees.

The greatest challenge with implementing flexible tree arity in DATs is the hard-coded binary design and re-ordering algorithm. The current DAT design uses a hard-coded navigation system that assumes a single sibling, uncle, and grand uncle node. Supporting flexible tree arity would require these node types to be indexable instead of assuming there is only a single sibling, uncle, etc... It would also require the modi-

fication of the node encoding algorithm to use an n-ary Huffman, or similar, encoding scheme. Supporting increase arity would present a few drawbacks too.

Increased arity also requires a larger memory footprint to support the additional node relationships within node metadata, and rebalancing would be more costly, both in resource utilization and processing time. Comparing against multiple sibling and uncle nodes for rebalancing would slow write operations considerably. A possible option for negating this additional cost is reducing the frequency of rebalance checks.

5.1.2 Compiler Assisted Rebalancing

To maximize the possible performance of specialized programs, modern CPU ISAs implement cache prefetch instructions for preloading soon-to-be used data into cache to prevent cache misses from stalling the CPU's execution pipeline. If a chunk of data is processed within a loop, the data processed in future iterations may be prefetched a few loop cycles early. A similar mechanism can be adopted for dynamic authentication trees. A control interface can be added to the memory controller for providing direct instructions to the tree's request generation block. Giving the CPU control, or even allowing it to issue restructuring suggestions to the tree, could maximize the possible bandwidth of the protected memory. For example, during an operating system context switch, the compiler could generate tree reordering instructions that would reorder the data block containing the top of the next task's stack, closer to the top of its tree. With the top of its stack near to root of the tree, the currently executing task would have much better performance than if it was placed at the bottom of a statically allocated tree.

A more basic approach could be used that splits the protected memory region into different segments, each one beginning at the start of a tree. These segments could be used by the linker to spread data across memory, using all the configured trees equally. This could improve the performance of dynamically balanced trees as

a modified linker could spread access probability such that each tree was accessed an equal number of times. If trees are accessed equally, that could possibly indicate that highly trafficked data structures are spread across memory such that each tree contains one, or at most, a couple. Data nodes containing these highly accessed locations would be balanced towards the top of the tree, improving performance. Spreading highly accessed structures across trees prevents them being contained in a small memory region under a single tree, reducing the likelihood of two nodes "fighting" for better placement within the parent tree.

5.1.3 Running Linux within Protected Memory

As part of this research, a hardware toolchain was developed for deploying memory authentication onto hardware devices. The toolchain handles the memory protection designs all the way from code, to the creation of a bootloader and hardware bitstreams with as little as a single command. The hardware bitstream and FSBL can be imported into Vitis to create a bare metal program running both data and instructions through protected memory. However, this limits the total scope of the memory protection hardware. The Zynq device targeted in this research, the XC7Z020, contains an integrated dual-core ARM Application processor.

Application processors integrate memory-management units (MMU) for the conversion of virtual to physical memory address mapping. This increases the security and flexibility of both the software and the hardware it runs on by segmenting memory on a per-application basis. An MMU is also required, in most cases, to run an advanced operating system on a processor. Specifically, Linux requires an MMU to be present on the target CPU to run. The extra utilities and libraries provided by Linux make it an ideal step-up over a bare-metal RTOS system for many embedded hardware targets.

While it is possible to deploy the hardware developed during in this research onto

bare metal targets, running Linux within the protected memory is not possible at the moment. Xilinx's embedded Linux flavor, PetaLinux, was run on the target system; however, it was not run within the protected memory region. Running Linux within the protected memory region requires modifications to both the Linux bootloader, U-Boot, and the kernel itself. It is also likely that the device-tree definition of the ZedBoard will have to be updated to remove the address map of the off-chip DDR. Only the address space of the protected memory controller will be listed as valid system memory.

Running Linux within protected memory will not just allow for stronger physical memory security, but it will also increase software security because of the kernel's permission system.

5.2 Conclusion

As the number of embedded systems rapidly grows in a world becoming increasingly interconnected, the importance of physical hardware security grows along with it. The work presented in this thesis introduced the use of caches into existing memory authentication methods as a means of improving performance. The introduced caches proved to be an effective method for improving the viability of existing memory authentication schemes in certain workloads. As an addition to the performance enhancements made, a toolchain suite was developed and provided that assist in the creation and deploying of memory authenticated hardware. A user is able to use this toolchain to integrate the design seamlessly into both their hardware and their software stack without needing to learn the specifics of the underlying implementation.

Bibliography

[1] 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.

[2] M. Millar and M. Lukowiak, "Dynamic authentication trees," in *RIT Academic Thesis*, 08 2021.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutli, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *IEEE*, 2014, p. 12.

[9] 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.

[10] T. Unterluggauer, M. Werner, and S. Mangard, "Meas: Memory encryption and authentication secure against side-channel attacks," in *Springer Journal of Cryptographic Engineering*, 01 2018, p. 22.

[11] J. Schaad, “Use of the advanced encryption standard (aes) encryption algorithm in cryptographic message syntax (cms),” RFC3565, USA, Tech. Rep., 2003.

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

[13] 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.

[14] C. Fruhwirth, “New methods in hard disk encryption,” in *Institute for Computer Languages Theory and Logic Group*, 2005, p. 116.

[15] 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.

[16] 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.

[17] *Zynq-7000 SoC Data Sheet: Overview*, Xilinx.

[18] *ARM Reference Manual: ARMv7-A and ARMv7-R edition*, ARM Limited, 110 Fulbourn Road, Cambridge, England CB1 9NJ.

[19] *AMBA AXI and ACE Protocol Specification*, ARM Limited.