Customizing Skewed Trees for Fast Memory Integrity Verification in Embedded Systems

Saru Vig, Tan Yng Tzer, Guiyuan Jiang and Siew-Kei Lam
School of Computer Science and Engineering, Nanyang Technological University, Singapore

***Abstract*—Memory integrity in embedded systems has been a longstanding issue in trusted system design. Existing schemes perform runtime integrity verification using memory integrity trees in order to secure untrusted external memories from malicious attacks e.g. replay, spoofing, and splicing. However, the balanced memory integrity trees used in existing approaches lead to excessive memory access overheads during runtime verification. In this paper, we proposed a framework to construct customized integrity trees based on the memory access patterns of the application. The framework relies on an offline process to analyze the frequency of data accesses and utilizes the package merge algorithm to generate a skewed memory integrity tree based on the frequency pattern. To the best of our knowledge, our work is the first to propose an automated approach for generating customized memory integrity trees. We validated the effectiveness of our approach on the Altera NIOS II processor with an external DRAM. Experimental results based on applications from widely used CHStone and SNU Real-Time benchmarks demonstrated that the proposed approach can lead to an average performance gain of 18% compared to the case where balanced memory integrity trees is used. To provide for further performance improvement in integrity tree verification, we implemented the encryption/decryption operation using custom instructions on the NIOS II processor. This resulted in an additional 10x performance improvement for the applications considered.**

I. Introduction

Embedded systems have become an integral part of our everyday lives. Due to our high dependability on such systems it is becoming more crucial that they are secure and cannot be tampered with. Recent foray of security features in commercial embedded processors reflect the growing concerns of security. However, these security features cannot extend fully beyond the processor core to guarantee secure storage in external third party memories, therefore exposing the vulnerability of the system to memory attacks. This is a major concern as the motive of almost all attacks is leakage or modification of information, making memory an obvious target. Memories are also very vulnerable as they are hard to audit and almost never visible to the software.

A common way to tamper external memories is through Bus Attacks. Such attacks replace external memories with malicious content. Typical bus attacks are splicing, spoofing, and replay [1]. Another type of attacks that are prevalent today is Side Channel Attacks [2]. Such attacks are mounted to observe the system during run time for minute details such as power consumption, timing etc. By analyzing these statistics , the attacker can decipher cryptographic keys used to encrypt the data, hence leading to loss of data confidentiality.

Almost all techniques used for memory protection include some form of encryption and an integrity tree as part of the basic framework. The common assumption made is that anything stored on chip cannot be tampered with. Memory encryption deals with encrypting the contents of the main storage with a cryptographic key that is securely stored on-chip. Integrity tree based techniques (see example in Fig. 1) split the memory into equal sized blocks $Data_i$, where $i = 1, 2, ..., n$ is the node number on the tree. Data blocks form the leaf nodes of an A-ary tree. The remaining nodes are created by recursively applying a primitive authentication function over the memory blocks until a single root node CC_1 is obtained. The root CC_1 is securely stored on chip and it represents the present state of the system. Thus any tampering can be detected at runtime if the root value does not match the value stored on chip.

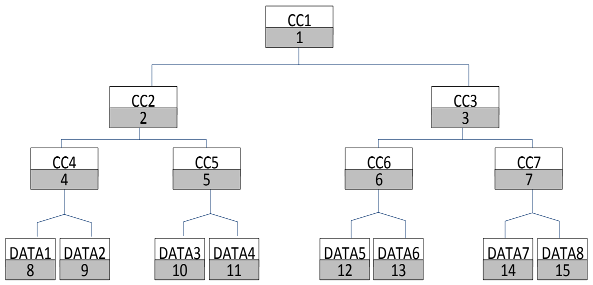


Fig. 1: Fully balanced memory integrity tree

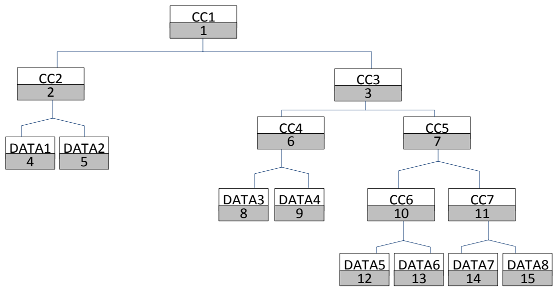


Fig. 2: Skewed memory integrity tree

The tight energy constraints imposed on battery-powered embedded devices presents a significant challenge for adopting strong memory security schemes. For a fully balanced integrity tree the number of checks to verify the integrity of each memory block during read accesses and the number of tree updates for each memory block during write accesses correspond to the number of tree levels, which is $\log_2 n$ for a fully balanced integrity tree. The additional memory accesses required for runtime verification for a fully balanced integrity tree therefore result in high computational overhead and contribute to excessive energy consumption particularly when the data size n is large.

In this paper, we proposed a framework to construct skewed integrity trees based on the memory access patterns of the application. It is a well known observation that applications running on embedded systems spend 90% of its time on 10% of the code [3]. As such, most embedded applications

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

have deterministic execution patterns which can be leverage upon to generate application-specific skewed integrity trees. To the best of our knowledge, our work is the first to propose an automated approach for generating customized memory integrity trees based on the application characteristics. The proposed framework relies on an offline process to analyze the frequency of accesses. Using the example in Fig. 1 and 2, let's assume that the offline profiling process reveals that $Data_1$ and $Data_2$ are accessed more frequently. The package merge algorithm is then employed to generate a skewed memory integrity tree based on the frequency of access. In the example shown in Fig. 2, the most frequently accessed data are placed at higher levels of the integrity tree. Since runtime verification for the frequently accessed $Data_1$ and $Data_2$ require lesser number of memory accesses (due to less number of levels to reach the root node $CC1$), the computational overhead of runtime verification is significantly reduced compared to the balanced tree in Fig. 1.

In order to demonstrate the effectiveness of our approach, we have implemented runtime data integrity verification on the Altera NIOS II processor with an external DRAM which stores the memory integrity tree. The approach can be adopted for code integrity as well. Experimental results using applications from widely used benchmarks show that the proposed method achieves an average performance improvement of 18% compared to conventional approaches employing a balanced memory integrity trees. Also, we implemented the AES algorithm for encryption/decryption using custom instructions on the NIOS II processor. This resulted in an additional 10x speedup, which further alleviates the bottleneck of runtime memory integrity verification.

This paper is organized as follows: Section II discusses related work in memory security. Section III describes the threat model and Section IV provides details of the modified TEC tree. Section V describes the proposed framework, and Section VI presents the experimental results. We conclude the paper in Section VII.

II. Related Work

Commercial processor-level security countermeasures typically define notions of trust zones or boundaries (or security perimeters) across the various on-chip hardware resources [4][5]. However the protection schemes in current trusted computing platforms cannot extend fully beyond the processor core to guarantee secure code/data storage in external third party memories. This enables adversaries to mount active attacks on memories such as cold-boot attacks by exploiting memory remanence properties (to extract confidential information), or perform replay, spoofing, and splicing attacks [6] (to influence program execution or reveal data by changing memory contents). Other attacks include passive bus snooping to infer secret cryptographic keys by monitoring data transfers between the processor and memory.

Successful memory security measures involve physical protection (to ensure memory hardware and media are not stolen or damaged), minimizing the risk and implications of error, failure or loss (e.g. developing a resilient back-up strategy), applying appropriate user authentication (e.g. employing strong password scheme), as well as implementing the encryption of sensitive data and/or cryptographic schemes for memory authentication. In order to mitigate the security threats on memories, secure architectures have been proposed with the underlying assumption that the processor chip is secure and off-chip devices (e.g. memories) are insecure [7]. To counter physical attacks, all sensitive information that needs to be transmitted from the secure processor (e.g. to external memory) are encrypted. A number of memory encryption schemes have been proposed in [8]. These methods typically employ a symmetric key cipher to encrypt data.

Memory protection from replay, spoofing, and splicing attacks can be achieved through memory authentication schemes that perform runtime integrity verification [6]. An integrity tree is vital to such mechanisms due to the limited storage space on-chip. Authentication methods like hash function, MAC, and Block-level AREA are used to realize the integrity trees [1]. The root value of the tree is stored on chip and is assumed to be safe and resistant to tampering. Existing integrity trees techniques include HASH trees, PAT trees, and TEC tree [9]. Implementing such integrity trees steeply increases the run time and memory usage of the application.

Intel's SGX makes use of an integrity tree as part of its Memory Encryption Engine to provide memory security [10]. It uses a slightly tweaked counter mode encryption along with MAC as its primitive. Although the memory integrity tree has been tailored to the architecture, there is still a need to traverse all the levels of the tree for each read and write operation to verify and match the tags due to the used of a balanced tree. PoisonIvy has a similar architecture as Intel's SGX in terms of encryption and integrity tree but it makes use of speculation to improve the performance[11]. The approach tracks the data and addresses that are sent to the memory speculatively and ensures that no unverified data escapes the chip. Merkle integrity trees are used in[12] where data are processed in batches to reduce the on-chip storage, thus improving on performance. The Merkle trees are built using entire memory blocks and concatenation is employed to increment the hash function for building the internal nodes.

While the above-mentioned methods have adopted methods to improve the performance of runtime memory integrity verification, they still rely on a fully balanced memory integrity tree which suffers from significant performance overheads due to the need to traverse all levels of the tree for each memory access. As discussed in [6], the utilization of balanced memory integrity tree account for excessive memory accesses which contribute to the largest fraction of energy consumed in embedded processors. As such, these methods are often infeasible for embedded systems with tight energy constraints. Thus, run-time integrity checking has been identified as one of the main open issues for trusted computing in mobile devices.

A. Main Contribution

The work in[13] discusses the advantages of using a skewed tree to improve performance. But to the best of our knowledge no effort has been undertaken to construct skewed memory integrity trees in a systematic and application aware manner. The main contribution of this paper is a framework that can automatically generate an application-specific skewed memory integrity tree based on the application characteristics. We demonstrate that runtime verification using the proposed approach outperforms an existing balanced memory integrity tree approach on the Altera NIOS II processor with external DRAM. In addition, we show that by exploiting custom instructions to accelerate the encryption/decryption operation, we can substantially reduce the performance overhead of runtime integrity verification. It is noteworthy that the proposed methods can complement those adopted in [12][6] for improving the performance of memory integrity verification.

III. Threat Model

The main focus of this work is to prevent bus attacks. Similar to previous works, our threat model assumes that data stored on chip is secure and is resistant to all kinds of attacks. We are concerned with attacks that tamper the memory and/or processor bus and thus are able to observe and inject manipulated data. Side channel and leakage attacks are beyond the scope of this work. The attacker can choose to tamper the data with any of the following active attack mechanisms:

- Spoofing: Attacker exchanges a memory block with a tampered one.
- Splicing: Attacker replaces the memory block at address A with a memory block at address B where $A \neq B$.
- Replay Attacks: Attacker records data at an address and inserts it at the same address at a later point in time. Thus the present value of the data is replaced by an older value.

IV. Customized TEC Tree

In this section, we will introduce the modifications made to the TEC-tree [8] to enable the construction and deployment of a customized skewed memory integrity tree. We have adopted the TEC tree in this work as it provides for data confidentiality unlike other memory integrity trees such as PAT and Merkle trees [9]. It is worth mentioning that the proposed method for customizing memory integrity tree can be applied to all existing memory integrity trees. We will provide security analysis for the modified TEC tree in this section.

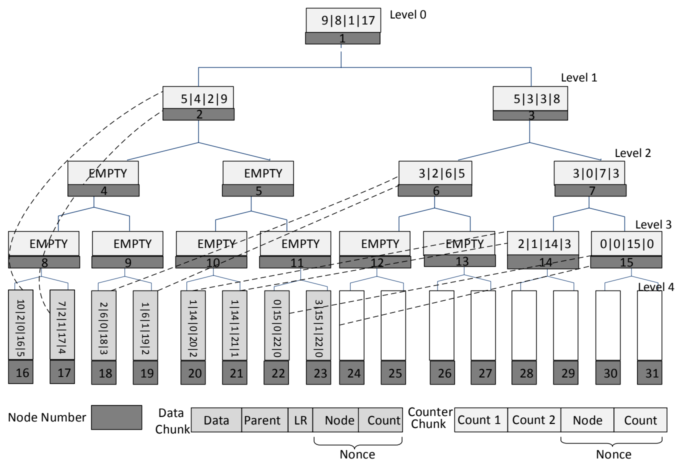


Fig. 3: Example of a modified TEC Tree

A. *Modified TEC Tree*

The original Tamper Evident Counter (TEC) Tree is a fully balanced memory integrity tree which uses as primitive, Block-level AREA, combining both encryption and authentication into one operation. A nonce (a number used only once), which consists of a counter value representing the number of writes to that location, is generated and concatenated onto every data block and the resulting block, called the Data Chunk (DC), is then encrypted using AES algorithm with a symmetric key cipher operating in Electronic Code Book Mode (ECB). The nonces are then combined into new blocks, called the Counter Chunks (CC), and the procedure is repeated up to the root, forming a tree structure. The final nonce value (root) is held on chip in secure storage. Thus the verification of a memory block traverses a path consisting of a DC and their parent CC's at different levels until it terminates at the root level in a final CC whose counter value is stored on-chip. In our work, we have fixed the arity, number of children of each node, as 2.

Note that a truly skewed tree implementation as in Fig. 2 increases the complexity of tree traversal since the determination of the location of the parent node (or their memory address) for child node i cannot be easily achieved with $\lfloor \frac{i}{2} \rfloor$. Our approach is to still maintain a physical balanced tree in the memory, but allow for empty nodes between the leaf nodes and their respective parents in order to emulate a skewed tree structure. Hence, in the modified TEC-tree implementation, the parent node may not necessarily be connected directly to the child nodes in the tree even though we are effectively utilizing a balanced tree structure. Specifically, we still place DC's on leaf nodes but while performing verifications, the parent node may be accessed by skipping some levels in the tree. This requires a different method for calculating parent node (address) of the leaf nodes. Essentially, our approach relaxes the restriction imposed on fully balanced trees that necessitates tree traversal to be performed at all levels of the tree for each memory block. Numbering the nodes in the same manner as a balanced tree simplifies the calculation of the parent address for any given node. Except for the leaf nodes, the parent of any node i can still be calculated as $\lfloor \frac{i}{2} \rfloor$. The parent node numbers of the leaf nodes are directly stored in the DC (as shown in Fig. 3). This enables the parent node of the leaf nodes that are separated by multiple levels to be accessed directly. We illustrate the modified TEC-tree using the example in Fig. 3 with 8 leaf nodes, whose parent nodes are located at different tree levels. It is evident that the runtime verification of Node 16 and Node 17 requires only two accesses of CCs (i.e. Node 2 and Node 1). The dotted line represents the first jump that nodes take to reach their respective parents. Compared to the fully balanced tree implementation, the modified TEC-tree enables the skipping of two levels of the tree during verification of Node 16 and Node 17.

In order to implement the modified TEC-tree, we need to introduce additional information in the DCs as shown in Fig. 3 (Note that the format of the CCs in Fig. 3 is the same as [8]). In particular, we have included two extra fields in DC i.e. $Parent$ and LR. The $Parent$ field specifies the parent node while LR indicates the child position i.e. which node is left or right child of its parent. The $Data$ field has the actual data. The $Node$ and $Count$ fields are concatenated to form the nonce. $Node$ is the node number of the tree and $Count$ represents the number of write operations on that particular node. As the node number is specific to each node of the tree the nonce for each node is unique. The CC has two fields $Count1$ and $Count2$ which store the $Count$ values of its right and left child respectively. The nonce of the CC is formed in the same manner as a DC. For example, Node 16 has 2 stored in its $parent$ field meaning that node 2 is its parent. Node 16 being the left child of Node 2 has its LR bit as 0 and similarly for node 17, also the child of node 2, the LR bit is 1. The $Count$ field at node 16 i.e. 5 is the $Count1$ placed at node 2 and similarly the $Count$ of node 17 i.e. 4 is placed at the $Count2$ field of node 2. The $Count$ value of node 2 represents the writes made on all of its children and hence, is set to 9.

The memory access overhead caused due to the CC's and DC's has been discussed in [8], that is $m = \frac{l + ((d + count) * Arity)}{(l * (Arity - 1))}$, where l is the size of data and d is the number of bits used to represent that address as can be seen in [8].The additional storage required for a skewed tree over a balanced tree is $m' = (\log_2(n) + 1) * d$, n being the number of data elements. As the table is being stored in external memory, this additional storage requirement does not pose a serious concern.

The following steps are performed for reading and writing to the memory blocks of the modified TEC-tree. For a read, the $ReadNCheck$ function is performed and for a write, the $WriteNUpdate$ function is performed.

$ReadNCheck$: When a DC is read from the memory, it is first decrypted and the corresponding $parent$ is extracted. Based on this, the parent CC is fetched and decrypted to obtain the corresponding counter value $Count$. If the $Count$ of the CC and DC matches, the authentication will be repeated at the next level. From this point onwards, the parent node of i can be determined using $\lfloor \frac{i}{2} \rfloor$. The verification is performed up till the root if there is no mismatch in the $Count$ values at any level. Should there be a mismatch we terminate the read access. If the $Count$ at the root node matches with the value stored on chip, the authentication is successful.

$WriteNUpdate$: During a memory write operation, the DC to be updated is first authenticated with the help of the $ReadNCheck$ function described above, and then updated by overwriting the data. For updating the block it is first decrypted and the data part of DC is extracted and replaced with the new data. The $Count$ value in the DC is then incremented by one. Updating of $Count$ values is recursively performed from the parent CC up till the root.

B. Security Analysis

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

Data is protected by making use of the block level AREA scheme. This schemes makes use of Shannon's diffusion property[14] to add some redundant data to the actual data before encryption and to check it each time after decryption. This is the motivation for adding a nonce to the data to form a data chunk. Once the chunk is encrypted, the data and nonce cannot be differentiated. For a a-bit nonce, the probability that the last a bits remain same after tampering is $1/(2^a)$.

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

TABLE I: Security Limitations

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

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

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

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

V. Proposed Framework

The framework consists of two parts: (i) offline profiling for weight calculation and tree formation, (ii) online verification. In this paper, we demonstrated the proposed method for data integrity protection. It is noteworthy that the method can be easily extended for code integrity protection through assembler modifications for loading the code integrity tree into memory.

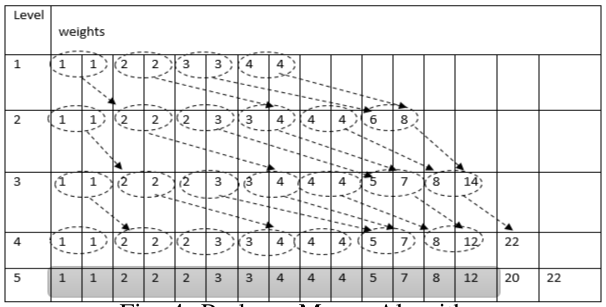


Fig. 4: Package Merge Algorithm

A. Profiling

Application profiling is performed by running the application to extract all the data memory addresses whenever a read or write operation is performed on the data. Through this process we calculate the frequency of accesses for each memory address. Based on this information we can calculate the weights of each data using Equations (1) to (3).

B. Weight Calculation

Weights are calculated based on the amount of time it requires to perform a read and write operation. For a read, verification is performed at each level. Thus the time required depends on L. Furthermore, at each level we need to extract the $Count$ value of both the node and its parent to perform verification, this requires decryption to be performed for both the nodes. Both these sums up to form the $ReadNCheck$ weight in Eq (2). $WriteNUpdate$ can be divided into two parts: 1) ReadNCheck, and 2) Updating. A write operation on any memory location is performed only after it has been ensured that the location has not been tampered with before, requiring an additional $ReadNCheck$ step. Updating can further be broken down into two parts: 1) Updating the DC, and 2) Updating the CC. Updating the DC requires overwriting the $data$ and $Count$ of the node. Thus we need to first decrypt the node and extract the required fields, update them, and encrypt the node. Updating the CC has to performed at all the remaining levels i.e. $L-1$. To update the $Count$ value of a CC we also need to extract the updated $Count$ value from its child, so as to make sure they match. Thus an extra decryption is performed for this step at each level. The time spent at each level during a read and write can be calculated using the formulae given below. L - number of levels of the balanced tree for a given number of data; t_upDC - time to update a Data chunk; t_up_CC - time to update the Counter Chunk; t_d - constant time needed for decryption using AES algorithm; t_e - constant time needed for encryption using AES algorithm.

$$Weight = ReadNCheck + WriteNUpdate \tag{1}$$

$$ReadNCheck = (L) * (2 * t\_d) \tag{2}$$

$$WriteNUpdate = ReadNCheck + t\_up\_DC + t\_up\_CC \tag{3}$$

$$t\_up\_DC = t\_d + t\_e \tag{4}$$

$$t\_up\_CC = (L-1) * (2 * t\_d + t\_e) \tag{5}$$

C. Tree Formation

Once our weights have been calculated we need to place the nodes on different levels of the tree so as to maximize the gain. For this purpose, we adopt the package merge algorithm [15] to construct our skewed memory tree and place the nodes in memory prior to running the main application code. The time complexity of this algorithm is $O(nL)$, where n is the number of nodes and L is the number of levels of the tree. We initially sort the weight in an increasing order and then the problem can be solved in linear time. The pseudo code used to perform the calculations is shown in Algorithm 1.

Algorithm 1: PACKAGE MERGE ALGORITHM (I, X)

```
begin
    $S \leftarrow \phi$
    for all $I$ ,$L_l$ $\leftarrow$ list of items of width $2^{-l}$, sorted by weight
    while $(X \geq 1)$ do
        $minwidth$= smallest item in the diadic expression of X
        if $I$ =$\phi$ then
            return "No Solution"
        end
        else
            $l \leftarrow$ the minimum such that $L_l$ is not empty
            $r \leftarrow 2^l$
            if $(r \geq minwidth)$ then
                return "No Solution"
            end
            else
                if $r = minwidth$ then
                    Delete the minimum weight item from $L_l$ and insert it
                      into $S$
                    $X \leftarrow X - S$
                end
            end
            $P_{l+1} \leftarrow$ PACKAGE($L_l$)
            discard $L_l$
            $L_{l+1} \leftarrow$ MERGE($P_{l+1}, L_{l+1}$)
        end
    end
    return S is the optimal Solution
end
```

A nodeset I is described as a set of ordered pairs (i,l) where $1 \leq i \leq n$ and $1 \leq l \leq L$. i being the node number and l representing the level of the node on the tree. We will use the notation l_i to denote the level of node i. Width is defined as 2^{-l} and weight of each node is a non negative number which in our case describes the frequency of data access. X is a non negative number denoting the $totalwidth$ i.e. $\sum_{i=1}^{n} 2^{-l_i}$. We should note here that $X \leq 1$ will give us a prefix free code as our solution is S. Given a prefix free code it is straight forward to create process a binary tree.

D. Implementation Example

We design a tree for the following data consisting of 8 data elements (i, j) where i is the data and j is the weight associated with its memory location. The weights have been calculated using Eq (1) to (3) as described in Section V-B as: {(3,1);(0,1);(1,2);(1,2);(1,3);(1,3);(7,4);(10,4)}. The height of the tree is 4. The package merge algorithm is performed on the weights {1,1,2,2,3,3,4,4} as illustrated in Fig. 4.

The step PACKAGE is performed to form a list P_{l+1} from L_l by combining items in consecutive pairs, starting from the lightest. Fig. 4 explains this step visually. The dotted pairs are the packages that are formed. Thus the package P_2 is formed from list L_1 on level 1 by combining the values of the dotted pairs. Combining is done by simply adding up the weights of the elements of each pair. Thus P_2 is the sum of the elements of all the dotted pairs on L_1 i.e.{2,4,6,8}. The MERGE step is the usual merging of two sorted lists. The successive lists in Fig. 4 are formed by performing the MERGE step. In general, the list L_l is created by forming package list, P_l, from list L_{l-1} and merging it with a copy of the first list developed. As shown in Fig. 4, L_2 is formed by merging P_2 (i.e. {2,4,6,8}) and the original elements i.e. {1,1,2,2,3,3,4,4}. The dotted line in Fig. 4 points to the sum of each package. The number of times the PACKAGE and MERGE steps are performed depends on the height, L, of the tree.

To get our desired solution we define a term $active$ leaves as the first $2 * n - 2$ items on the last list. This represents the number of times the while loop in Algorithm 1 will run. The rationale behind this is to make sure we get $X \leq 1$. We process the active leaves to get our solution. The solution is stored in the list S=$[s_i]$ $i \in 1...n$ and $1 \leq s_i \leq L$. s_i is set to the number of active leaves corresponding to each original element. The active leaves for our example are the first 14 elements on list 5. They have been shaded in Fig. 4 and corresponding to each of them s_i has been counted. The solution is {4,4,4,4,3,3,2,2}.

The final solution as depicted in Fig. 3 obtained using this calculation is the level each item should be placed for a 4-level 2-ary tree. Items with weight {1,1,2,2} (i.e. Node 20 - Node 23) should be placed at level 4 and their parent are thus placed at level 3. Items with weight {3,3} (i.e. Node 18 and Node 19) should be placed at level 3 and thus the nodes on level 2 are their parents. Items with the maximum weight (i.e. Node 16 and Node 17) should be at level 2 (closest to the root) and hence their parent are placed on level 1, making the highest jump, skipping two levels.

We should note here that to run the package merge algorithm we must know the optimal height of our tree. Skewing the tree will in most cases end up increasing the height. For every level that is increased some paths get shorter and some get longer. Thus skewing a tree beyond a certain height can degrade performance. To calculate the most optimal height we make use of a theorem stated in [16], i.e. the upper bound for the height of an optimal weighted path length is $2 + H$ for a binary tree. H is the entropy of the frequency distribution of the data nodes. We use this theorem to determine the height of our skewed tree. Running the design on the upper bound is a safe option and gives us a fair estimate to evaluate the design. It is worth mentioning that this theorem is only applicable to binary trees i.e. trees with arity 2. If we extend this work on trees with higher arity we will need to adopt some other heuristic measure to find the most optimal height.

E. Custom Instructions

The NIOS II processor offers the capability of extending the basic instruction set using custom instructions which are realized as hardware accelerators that augment the ALU. In order to reduce the latency of encryption/decryption during memory integrity verification, we have implemented the 128-bit mix-column AES algorithm [17] as custom instructions. Since the NIOS II custom instruction interface are restricted to two 32-bit inputs and one 32-bit output port, the 128-bit AES input must be separated into 32-bit wide segments and passed to the custom instruction module in a sequential manner. An additional input signal is used to indicate which segment needs to be passed to the custom instructions. The Altera tool-chain automatically creates macros which can be used to call the custom instructions from the application code. We generated macros for the following functions of the AES algorithm: 1) Byte Substitution, 2) Skip Row, 3) Mix Columns, 4) Add Round Key, 5) Key Expansion and 6) Inverse Key Expansion.

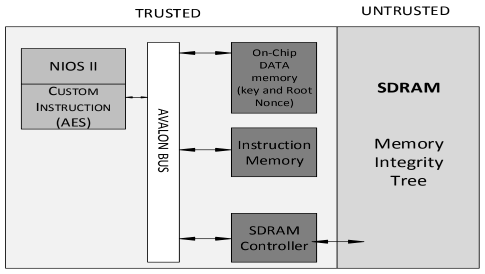
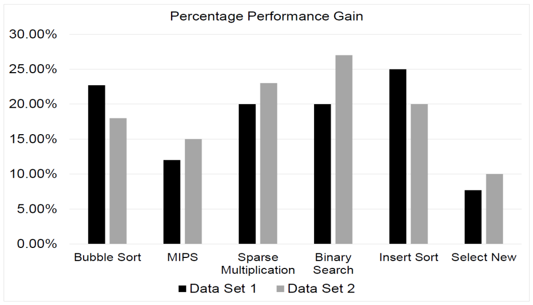


Fig. 5: System overview

VI. Performance Evaluation

We evaluated the performance benefits of the proposed method using the Altera DE2 board and the Qsys tool in



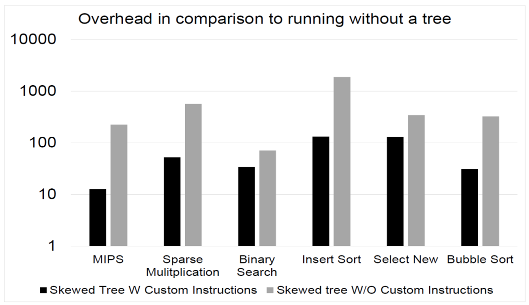


Fig. 6: (a) Performance gain in comparison to a balanced tree, and (b) Improvement with Custom Instructions

Altera Quartus II, v12.1. The system consisted of a NIOS II Processor operating at 50MHz, On-Chip Memory, JTAG-UART for connection between the system and the board, SDRAM Controller for using the SDRAM, Clock Series for DE-series Board Peripherals, and Performance Counter Unit for measuring the performance statistics such as time elapsed and number of clock cycles.

In our experiments, we ran six deterministic applications from the CHStone and SNU Real Time testbenches. The applications are tested: 1) using a balanced TEC tree, and 2) using the modified TEC tree as described in Section IV. We measured the time required to run the benchmarks. The tests were done using two different data sets to get a fair estimate of performance. Data set 1 was part of the original test-benches. Data Set 2 is synthetically generated using random values and have twice the size of Data set 1. Each data set was initially profiled offline for weight calculations. The results after testing are shown in Fig. 6(a). A significant performance improvement of 18% can be observed when the skewed memory integrity generated by our framework is used.

Note that the amount of gain will depend directly on the kind of application and its memory usage pattern. Thus we expect to see variation in the gain for different applications. Applications dealing with data sets with larger variance in the memory access frequencies are expected to result in larger performance benefits.

The run time improvements by using custom instructions for various applications using the proposed skewed memory integrity tree are shown in Fig. 6(b). It is worth noting here that the results have been plotted with a logarithmic y-axis. It is evident that using custom instructions for implementing the AES operation significantly reduces the overhead of employing memory integrity tree. In particular, there is a 10x improvement in the performance before and after the custom instructions were implemented. In addition when custom instructions are used in both the balanced tree and skewed tree, the proposed skewed memory integrity tree could still achieve 16% performance improvement.

VII. Conclusion

In this paper, we presented a framework for generating a customized skewed memory integrity tree based on the frequency of memory accesses of the application. We modified the TEC tree to allow for the more frequently accessed memory blocks to traverse a shorter verification path. This is achieved by including additional fields in the data chunk to indicate the parent of the leaf nodes. We formulated the weights to reflect the memory access overheads of read and write operations, and utilized them to calculate the weights of each memory block. The package merge algorithm was adapted to place the weighted memory blocks in the skewed tree so that memory blocks with larger weights will require less verification steps. By implementing AES as custom instructions we were able to further reduce the runtime overhead. The proposed framework can be extended for other types of memory integrity trees. Experimental results based on widely used benchmarks demonstrates the effectiveness of our approach. Our future work includes extending the framework to generate customized integrity trees with arbitrary arity, and examine the effects of cache on it.

References

[1] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet, and A. Martinez, "A parallelized way to provide data encryption and integrity checking on a processor-memory bus," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006.

[2] T. Unterluggauer and S. Mangard, "Exploiting the physical disparity: Side-channel attacks on memory encryption," 2016.

[3] D. C. Suresh, W. A. Najjar, and J. Yang, "Power efficient instruction caches for embedded systems," in *International Workshop on Embedded Computer Systems*. Springer, 2005.

[4] Arm trust zone. [Online]. Available: http://www.arm.com/products/processors/technologies/trustzone/index.ph

[5] Microsoft palladium: Next generation secure computing base. [Online]. Available: https://epic.org/privacy/consumer/microsoft/palladium.html

[6] S. Chhabra and Y. Solihin, "Green secure processors: towards power-efficient secure processor design," in *Transactions on computational science X*. Springer, 2010.

[7] G. E. Suh *et al.*, "Aegis: A single-chip secure processor," *Information Security Technical Report*, vol. 10, 2005.

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

[9] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres, "Hardware mechanisms for memory authentication: A survey of existing techniques and engines," in *Transactions on Computational Science IV*. Springer, 2009.

[10] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," 2016.

[11] T. S. Lehman *et al.*, "PoisonIvy: Safe speculation for secure memory," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, oct 2016.

[12] S. H. Kim *et al.*, "Fully Batch Processing Enabled Memory Integrity Verification Algorithm Based on Merkle Tree." Springer, Cham, 2016.

[13] J. Szefer and S. Biedermann, "Towards fast hardware memory integrity checking with skewed Merkle trees," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy - HASP '14*. New York, New York, USA: ACM Press, 2014.

[14] C. E. Shannon, "A mathematical theory of cryptography," *Memorandum MM*, vol. 45, 1945.

[15] L. L. Larmore and D. S. Hirschberg, "A fast algorithm for optimal length-limited huffman codes," *Journal of the ACM (JACM)*, vol. 37, no. 3, 1990.

[16] S. Nagaraj, "Optimal binary search trees," *Theoretical Computer Science*, vol. 188, 1997.

[17] H. Li and Z. Friggstad, "An efficient architecture for the aes mix columns operation," in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. IEEE, 2005.