TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense against Memory Replay Attacks

Reouven Elbaz[1], David Champagne[1], Ruby B. Lee[1], Lionel Torres[2], Gilles Sassatelli[2], and Pierre Guillemin[3]

[1] Department of Electrical Engineering, Princeton University,
Princeton, NJ, 08544, USA
{relbaz, dav, rblee}@princeton.edu
http://palms.ee.princeton.edu/

[2] LIRMM UMR University of Montpellier II/ CNRS C5506,
34392 Montpellier, France
{torres, sassatelli}@lirmm.fr
http://www.lirmm.fr/

[3] STMicroelectronics, Advanced System Technology,
13106 Rousset, France
pierre.guillemin@st.com
http://www.st.com/

Abstract. Replay attacks are often the most costly attacks to thwart when dealing with off-chip memory integrity. With a trusted System-on-Chip, the existing countermeasures against replay require a large amount of on-chip memory to provide tamper-proof storage for metadata such as hash values or nonces. Tree-based strategies can be deployed to reduce this unacceptable overhead; for example, the well-known Merkle tree technique decreases this overhead to a single hash value. However, it comes at the cost of performance-killing characteristics for embedded systems – e.g. non-parallelizable hash computations on tree updates. In this paper, we propose an alternative solution: the Tamper-Evident Counter Tree (TEC-Tree). It allows for tamper-evident off-chip storage of the nonces involved in a replay countermeasure; TEC-Tree parallelizes the computations involved in both the authentication and tree update processes. Moreover, because our tree relies on block encryption, it provides data confidentiality at no extra cost. TEC-Tree is a deployable solution for memory integrity, with low performance hit and hardware cost.

Keywords: Memory Integrity, Replay Attacks, Physical Attacks, Tamper Evidence, Merkle Trees, Parallelizability, Confidentiality.

1 Introduction

As the range of services provided by embedded systems grows, the amount of sensitive data they manipulate increases. As a result, there are currently significant incentives for attackers wishing to benefit from attacks on these systems [1]. Board-level attacks such as bus probing provide a point of entry to the system for adversaries, as shown by the well-known Xbox cracking exploit [2]. Attacks on buses allow retrieving bus data going to or from memory and thus raise the issue of data confidentiality. Typically, encryption schemes are implemented to ensure data confidentiality. However, the attack launched by Markus Kuhn [3] on a bus encryption system demonstrates encryption alone is not enough when an adversary is able to modify or manipulate data without detection. The issue of memory integrity is thus a prime concern in protecting a computing platform against active attackers.

In this paper, we focus on protecting the integrity of data transferred by a System-on-Chip (SoC) to or from a RAM (Random Access Memory) chip. Our objective is to deploy an efficient mechanism providing a tamper-evident environment to software executing on the SoC: data deletion, corruption or permutation within the system's memory space must be detected before the tampered data is sent to the Central Processing Unit (CPU) pipeline.

When dealing with memory integrity, one of the most costly attacks to thwart is replay – i.e. temporal permutation of a memory block at a given address. Existing techniques allow for easy prevention of replay attacks but are very expensive in terms of on-chip memory overhead: some techniques store on-chip a hash value computed over each memory block written off-chip, while others store on-chip nonces used in MAC (Message Authentication Code) computations or in a block-level AREA (Added Redundancy Explicit Authentication [4]) scheme [5, 6]. The well-known Merkle Tree technique [7, 8, 9] allows reducing the overhead of the countermeasure storing hashes on-chip to a single hash value. However, it comes at the cost of performance-killing characteristics for embedded systems – e.g. non-parallelizable hash computation on tree updates. PAT (Parallelizable Authentication Tree) [10, 11] is another tree technique – based on MACs – reducing the on-chip memory overhead, but allowing for parallel tree updates. However, PAT is patented and has not been implemented or evaluated in our application domain.

In this paper, we propose a fully-parallelizable tree on both read and write operations: the TEC-Tree (Tamper-Evident Counter Tree). Our tree has advantages over both PAT and Merkle Tree and also introduces new concepts for integrity trees. First, the computations involved in this tree scheme are based on block encryption, thus providing data confidentiality for free – i.e. at no additional cost (performance, hardware, memory overheads) than that already incurred by data integrity verification. Second, our TEC-Tree solution can detect splicing – spatial permutation of memory blocks – and spoofing – arbitrary corruption of memory blocks – immediately after the first level verification of the tree authentication process is completed. This allows for splicing- and spoofing-free speculative execution. Finally, to reduce the frequency of costly re-encryptions due to the overflow of nonces used to provide freshness, we propose using a per-block local counter rather than a global counter of the same size.

The rest of the paper is organized as follows. Section 2 presents our threat model. Section 3 describes existing replay attack countermeasures and the related tree techniques proposed in the literature. In section 4, we detail the TEC-Tree scheme, address SoC implementation issues and provide a security analysis. Section 5 presents an implementation example with estimates of area cost and memory overhead. Finally, Section 6 summarizes the new properties provided by TEC-Tree and gives an overview of our current and future work.

2 Threat Model

We assume the protected device is exposed to a hostile environment in which physical but non-invasive attacks are feasible. The main hypothesis of our threat model is that the System-on-Chip (SoC) is resistant to all physical attacks and is thus trusted. Side-channel and software attacks are not taken into account in this paper; we assume the operating system (OS) or at least the OS kernel is trusted. As a consequence, we consider that on-chip registers and memories cannot be observed or tampered with by an adversary.

In this work, we focus mainly on board-level attacks involving processor-memory (PM) bus probing or memory tampering. Such attacks allow observation of the memory contents and injection of arbitrary data on the PM bus or directly into the memory chip. We are particularly concerned with "Man-in-the-middle" attacks in which an attacker i) first monitors the PM communications to intercept data on the bus or directly reads data from memory (passive attack), ii) and then inserts chosen data on the processor-memory bus (active attack) and thus challenges data integrity. The objective of the attacker could be to take control of the system by injecting malicious code or to constrain the search space in a key or message recovery attack. There are three classes of active attacks – also feasible when data are encrypted – defined with respect to the attacker's possible ways to choose the inserted data:

1*) Spoofing attacks*: the adversary exchanges an existing memory block with an arbitrary fake one.

2) *Splicing or relocation attacks*: the attacker swaps a memory block at address A with a block at address B, where A≠B. Such an attack may be viewed as a spatial permutation of memory blocks.

3) *Replay attacks*: a memory block located at a given address is recorded and inserted at the same address at a later point in time; by doing so, the current block's value is replaced by an older one. Such an attack may be viewed as a temporal permutation of a memory block, for a specific memory location.

3 State of the Art

3.1 Memory Integrity Checking Techniques

In this paper, providing memory integrity for the SoC means ensuring the tamper-evidence of the data it stores in external memory, i.e. making sure we can detect when a datum read from memory was spliced, spoofed or replayed. We consider there are three distinct strategies for implementing a memory integrity protection mechanism on a computing platform: i) regular hashes, ii) Message Authentication Codes (MACs) and iii) AREA at the block level [5, 6].

Regular Hashes: A naïve solution for detecting all types of active attacks is to store on-chip a hash value for each memory block stored off-chip. This way, hashes are inaccessible to adversaries and any corruption in the loaded data is detected. Indeed, detection can be achieved by comparing the hash computed from the loaded block with the on-chip hash.

MACs: In the second approach, the SoC computes – for every data block in the protected memory space – a MAC instead of a hash. The key used by the SoC for MAC computation is securely stored on-chip such that only the SoC itself is able to compute valid MACs. As a result, the MACs can be stored in untrusted memory because the attacker is unable to compute a valid MAC over a corrupted data block. In addition to the data contained by the block, the pre-image of the MAC function contains the address of that block and a nonce. This allows protection against splicing and replay attacks. The address precludes an attacker from passing a data block at address A, along with the associated MAC, as a valid (data block, MAC) pair for address B, where A ≠ B. The nonce prevents the replay of a (data block, MAC) pair by distinguishing two pairs related to the same address, but written in memory at different points in time. To allow for re-computation of MACs during read operations and to protect the integrity of the nonces, the nonces can be stored on-chip.

Block-Level AREA: This principle – introduced with PE-ICE (Parallelized Encryption and Integrity Checking Engine) in [5] – leverages the diffusion property of block encryption to add the integrity checking capability to this type of encryption algorithm. This is achieved by applying the AREA (Added Redundancy Explicit Authentication [4]) technique at the block level: redundant data (a nonce) is added to each plaintext block before encryption and checked in the decrypted ciphertext block. Upon a memory write, the SoC appends an n-bit nonce to the data to be written to memory, encrypts the resulting plaintext block and then writes the ciphertext to memory. The encryption is performed using a key securely stored on the SoC. The SoC decrypts the block it fetches from memory on a read transaction and verifies that the last n bits of the resulting plaintext block are equal to the nonce that was inserted by the SoC upon encryption. To recover the nonce on a read operation and to protect its integrity, [5, 6] propose storing the nonce on-chip.

3.2 Tree Techniques

The techniques presented above prevent the active attacks described in our threat model. However, they incur an unacceptable on-chip memory overhead by requiring storage of references – i.e. hashes or nonces – on-chip to thwart replay attacks. To address this issue, several research efforts suggest applying the memory integrity protection methods recursively on the references so they can be stored off-chip. By doing so, a tree structure is formed and only the root of the tree needs to be stored on the SoC, the trusted area.

Merkle Hash Trees: In [8, 9], cryptographic hashes are computed over each data block composing the memory space being protected. We call these hashes *Level 1* hashes. One *Level 2* hash is computed for every group of *A Level 1* hashes. At this stage, *Level 1* hashes can be sent off-chip since *Level 2* hashes – stored on-chip – protect their integrity. This scheme is applied recursively until an *A*-ary hash tree with *L* levels is obtained, at which point a single *Level L* hash – called the root hash – is kept on-chip. This root hash reflects the current valid state of the external memory. The number of hash verifications on a read operation is *L*: each level of hashes must be verified to validate the integrity of the data read from memory. On read operations, the hash computations are parallelizable during the integrity checking process. On write operations however, updates to the tree are a sequential – and thus high-latency – process: the computation of a new hash value in the tree must be completed before the update of the related hash in the upper level can start.

Parallelizable Authentication Tree (PAT): To reduce the on-chip memory overhead of a replay defense mechanism based on MACs, [10] computes a tree of (MAC, nonce) pairs and only keeps the root of that tree on-chip. To allow for parallelizable updates to the tree, [10] does not directly compute higher level MACs using lower level MACs as a pre-image. Instead, the pre-image of each MAC in the tree consists of the nonce associated to this MAC and the nonce in the upper level. With this approach, the root of the MAC tree is a simple nonce stored on-chip.

Our Approach: TEC-Tree, the tree scheme proposed in this paper, solves the issue of on-chip nonce storage that was problematic in [5, 6]. Similarly to [10], our tree is parallelizable on both read and write operations: all computation required for checking or updating a tree branch involves data which are generated independently from a tree level to another. However, our tree has advantages over [10] and [7]. Our parallelizable tree can detect splicing and spoofing attacks as soon as the first check is completed. Even if tree computations are parallelizable, memory bus transactions remain a sequential process allowing only a certain number of tree levels to be checked or updated concurrently. Quick detection of spoofing and splicing is important in reducing the risks involved with speculative execution [8, 12] – a strategy in which data and instructions are sent to the processor pipeline before tree authentication is completed. Moreover, since we rely on block encryption to construct our tree, TEC-Tree provides data confidentiality at no extra cost: without any additional impact on latencies, using the hardware already in place for integrity protection and by sharing the integrity metadata (off-chip memory overhead).

4 TEC-Tree – Tamper-Evident Counter Tree

TEC-Tree is a practical and low-cost solution to efficiently prevent active attacks challenging memory integrity – including replay attacks – for SoC embedded systems. We first describe how TEC-Tree protects the integrity of individual data blocks from the memory space using nonces. Then we explain how the protection method is applied recursively to those nonces in order to form a hardware-rooted tree.

4.1 Protecting Data

As in [5, 6], the protection of data integrity in our TEC-Tree scheme relies on the diffusion property identified by Shannon [13] for block ciphers to be considered as secure. Theoretically, a block cipher must be indistinguishable – from the point of view of an adversary without the key – from a random permutation with equiprobable outputs, meaning that the redundancy in the statistics of the plaintext has to be dissipated in the statistics of the ciphertext. Once a block encryption is performed, the resulting position and value of each bit in a ciphertext block C are a function of all bits of the corresponding plaintext block P and of the key. In this paper, the block cipher used for TEC-Tree processes *b*-bit blocks under a secret key K. E_K and D_K refer respectively to the encryption and decryption functions under key K.

In TEC-Tree, P is in fact composed of an l_p-bit data block D – hereafter called a payload – and an *n*-bit nonce N (P = D||N), also called tag; after encryption with a block cipher, it is impossible to identify the ciphered versions of D and N within the ciphertext block C = E_K(P). Moreover, if a C' is derived by flipping a single bit in C, there is a large probability that the last *n* bits of the plaintext P' = D_K(C') will be different from N. This probability depends on the nonce size *n*. The number of possible plaintext blocks with the same N resulting from the decryption of a tampered C is equal to 2^{b-n}. Hence the probability that N remains the same after decryption is $1/2^n$ (= $2^{b-n} / 2^b$). As we explain in the security analysis later in this paper, with a reasonable nonce size, this probability becomes sufficiently low to resist spoofing attacks. In the rest of this paper, we use the term data chunk (DC) to refer to an atomic block loaded from memory for authentication; the size of a DC is *b* bits, the size of blocks processed by the underlying block cipher. We call this authentication technique *block-level AREA*.

Reading and Writing DCs. Similarly to [5, 6], the previous property of ciphers is used as follows in TEC-Tree to add the integrity checking capability to the block encryption of data chunks:

On *write operations*, the data D provided by the processor – or coming from the last level of on-chip cache on a cache line eviction – is concatenated with the *n*-bit nonce N to produce a plaintext block P to be processed by a block cipher in ECB (Electronic Code Book) mode. The encryption provides confidentiality since it is performed by the SoC using a key it securely stores in an on-chip register. We ensure that N is a "Number used ONCE" by making it equal to the address of the block concatenated with an *r*-bit counter. Initially zero, a counter is incremented by one every time a block is written by the SoC. Using a counter to generate nonces requires changing the encryption key whenever the counter reaches its limit; replacing the encryption key requires a re-encryption of the entire memory space. Since such massive re-encryption is expensive, and since we want to keep counter width reasonably small, we cannot use a single *r*-bit global counter to generate all nonces – i.e. a counter which is incremented on every write to memory. Instead, we use an *r*-bit counter per chunk to ensure the occurrence of counter overflows – and thus the re-encryption of the memory space – remains a rare event. We characterize such a counter as "local" in opposition to "global" because it is dedicated to a specific chunk and because it is incremented only when this chunk is updated. By doing so, we extend the lifetime of the encryption key with respect to the implementation of an *r*-bit global counter[1]. Note that even when local counters have the same value, the address contained in tags ensures that those tags remain nonces. After the generation of N, the SoC encrypts this unique (D, N) pair and writes the resulting *b*-bit ciphertext block to memory. A plaintext data chunk is depicted in Figure 1. Note that the entire data chunk: data, address and counter, is encrypted as a single block before being stored in external memory.

On *read operations*, a *b*-bit ciphered data chunk C is loaded from memory and decrypted by the SoC. The last *n* bits of the resulting plaintext block – which we designate by N' – are compared to N, the nonce initially used by TEC-Tree to compute C and which was regenerated on-chip. If N' does not match N, it means that at least one bit of C has been modified during its transmission on the bus or while stored in off-chip memory. When this happens, our integrity verification mechanism raises a "memory integrity" exception to alert the processor of the memory corruption.

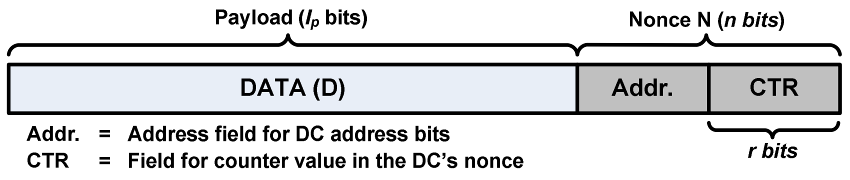

Payload (l_p bits)
Nonce N (*n bits*)
DATA (D)
Addr.
CTR
Addr. = Address field for DC address bits
CTR = Field for counter value in the DC's nonce
r bits

Fig. 1. Layout of a Data Chunk (DC) before encryption

The challenge lies in regenerating the correct nonces on read operations in order to perform the integrity checks. The approach proposed by [5, 6] is to store on-chip – in a dedicated memory – the counter part of the nonce values generated on write operations. These counter values are tamper-proof since they are stored on the SoC and thus within the trust boundary. However, this strategy incurs an unacceptable increase in on-chip memory capacity: [6] shows that protecting 1GB of RAM with a 32-bit counter requires 256MB of on-chip memory. Our approach, which we discuss next, reduces this overhead by securely storing counter values off-chip.

4.2 Storing Counter Values Off-Chip

To allow for off-chip storage of counter values while ensuring their tamper-evidence, we apply the block-level AREA scheme, discussed above, to those values, without requiring additional hardware. We concatenate counter values used to authenticate contiguous data blocks into a payload P_L to which we append a nonce N. We then encrypt the plaintext block $P = P_L || N$ to obtain the ciphertext C, which we send to external memory; we store on-chip the counter part of the nonce (CTR in Figure 2). In

[1] Note that this does not apply if the global counter is *n*-bit i.e. the size of the nonce. However, removing the address from the nonce precludes the splicing-free and spoofing-free speculative execution property which is highlighted in the security analysis.

the rest of this paper, we call such a plaintext block P a Counter Chunk (CC) [2]. Figure 2 shows a counter chunk CC. Note that the entire CC – payload (concatenated DC counters), and nonce (address and counter) – is encrypted as a single block before being stored in external memory.

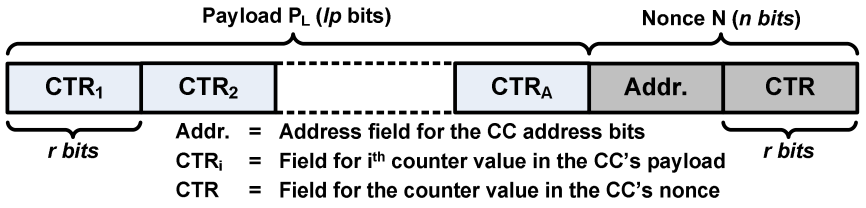

Payload P_L (*lp* bits)
Nonce N (*n bits*)
CTR_1
CTR_2
CTR_A
Addr.
CTR
r bits
r bits
Addr. = Address field for the CC address bits
CTR_i = Field for ith counter value in the CC's payload
CTR = Field for the counter value in the CC's nonce

Fig. 2. Layout of a Counter Chunk (CC) before encryption

The number *A* of counter values fitting in a CC payload depends on the payload size (l_p-bit) and on the counter width *r*. The formula for *A* is as follows:

$$A = \left\lfloor \frac{l_p}{r} \right\rfloor \qquad \text{Where } \lfloor X \rfloor \text{denotes } X \text{ rounding down.} \tag{1}$$

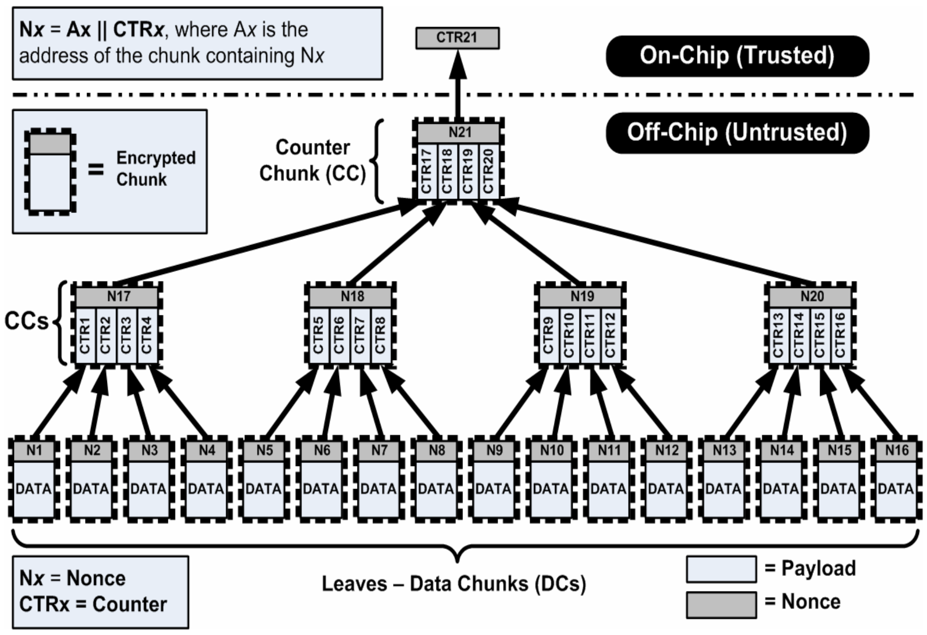
Off-chip storage of the counters allows for a reduction of the on-chip memory overhead by a factor *A*. Since this reduction is insufficient in most cases, we recursively apply the block-level AREA scheme to the on-chip counter values until only one counter value – called the trusted counter value – remains on-chip, thus obtaining an *A*-ary tree of tamper-evident counter values. The main objective of this tree we call TEC-Tree is to reduce the on-chip memory overhead to a single counter value while providing full memory integrity and faster detection of splicing and spoofing attacks.

4.3 TEC-Tree

A TEC-Tree is an *A*-ary tree[3] with data chunks (DC) as leaves and counter chunks (CC) as intermediate nodes. Each element of the TEC-Tree is *b*-bit long. The data to protect are contained in the DCs, which are the leaves of the tree. Each intermediate node or CC in the tree is used to authenticate, using the block-level AREA, the *A* leaves or *A* nodes below itself – its *A* children. The root chunk of the tree is authenticated using the trusted counter value kept on-chip. Each DC is composed of an l_p-bit data payload and an *n*-bit nonce; each CC is composed of *A* *r*-bit counter values and an *n*-bit nonce. Figure 3 depicts a 3-level, 4-ary TEC-Tree. Arrows in the figure go from a node to the counter part of the nonce which protects that node.

[2] It is important to note that there is no need to store the whole nonce of the contiguous DCs in the counter chunks (CC) since the address part of the nonce is generated by the processor whenever the chunk is accessed.

[3] In this section as in the rest of this paper, we consider that all the trees we work with are balanced – i.e. each node in the tree has the same number of children.



N*x* = A*x* || CTR*x*, where A*x* is the address of the chunk containing N*x*
CTR21
On-Chip (Trusted)
Off-Chip (Untrusted)
= Encrypted Chunk
Counter Chunk (CC)
N21
CTR17 CTR18 CTR19 CTR20
CCs
N17 CTR1 CTR2 CTR3 CTR4
N18 CTR5 CTR6 CTR7 CTR8
N19 CTR9 CTR10 CTR11 CTR12
N20 CTR13 CTR14 CTR15 CTR16
N1 N2 N3 N4 N5 N6 N7 N8 N9 N10 N11 N12 N13 N14 N15 N16
DATA
Leaves – Data Chunks (DCs)
N*x* = Nonce
CTRx = Counter
= Payload
= Nonce

Fig. 3. A 3-level, 4-ary TEC-Tree

TEC-Tree at Run-Time. The function `ReadAndCheck` and `WriteAndUpdate` of Figure 4 describe how TEC-Tree works at run-time when data is read or written from memory, respectively.

On *read operations*, the data chunk at address *addr0* containing the data to be read is loaded on-chip and decrypted. The parent chunk is also fetched and decrypted to retrieve the counter value required – in addition to *addr0* – to reconstitute the DC's nonce. Authentication of the DC is performed by comparing the reconstituted nonce to the last n bits of the plaintext resulting from the decryption of the DC. If the two values match, the data is forwarded to the processor for speculative execution while the rest of the chunks on the branch from the DC to the root chunk are authenticated. To do so, the procedure described previously is recursively applied to parent nodes until the root of the tree is authenticated. If a nonce mismatch is detected at any point during the verification of the branch, an integrity checking exception is raised to warn the processor. Once the chunks on the branch from the DC to the root chunk are loaded, all decryption operations can be performed in parallel since the chunks are deciphered using a block cipher in ECB mode. The number of nonce verifications V required to complete the authentication process – and thus the number of chunks to load and decrypt during that process – is $V = L+1$, where L is the number of levels of the TEC-Tree. L is easily computed from Nb_r – the number of data chunks – and from the tree's arity A: $L = \log_{A}(Nb_r)$. The unit increment to L is due to the check of the TEC-Tree's root chunk using the on-chip trusted counter value.

— *addr* is the CPU address of the data to read or write
— $addr_i$ is the physical address of a chunk at level *i*
— *L* is the number of levels in the TEC-Tree.
We define the following functions:
— `ParentFromChild(`$addr_i$`)` is a function which – given as an input the address $addr_i$ of a chunk X at level *i* - returns the address $addr_{i+1}$ of X's parent chunk. If $addr_i$ is the address of the root, the function returns -1.
— `DC_ADDR`(*addr*) is a function computing a DC address from *addr*.
— `REF_NONCE`(*addr*, P) returns *addr* || CTR, where CTR is retrieved from P. When P = -1, the function uses the on-chip counter as CTR.
— `ADDR`(P) returns the address field of plaintext chunk P's nonce
— `COUNTER`(P) returns the counter field of plaintext chunk P's nonce
— `UpdateCCPayload`(*addr*, P) returns P updated with X's counter incremented, where X is P's child chunk at address *addr*.

```
ReadAndCheck(addr)
    $addr_0$ = DC_ADDR(addr)
    FOR ($i$ = 0, $i$ = $L$, $i$++)
      IF $i$ = 0 — first level of the tree ⇔ leaves
         1. Load $C_0$, the DC at $addr_0$ from memory
         2. Load $C_1$, the parent CC at $addr_1$ := ParentFromChild($addr_0$)
         3. Compute $P_0 := D_K(C_0)$ and $P_1 := D_K(C_1)$
         4. IF $addr_0 \neq$ ADDR($P_0$) THEN ERROR — first check against splicing and spoofing
          ELSE send $P_0$ to CPU — speculative execution
         5. IF NONCE($P_0$) $\neq$ REF_NONCE($addr_0$, $P_1$) THEN ERROR
      ELSE IF i $\neq L$ — authentication of intermediate RC in the tree
         1. Load $C_{i+1}$, the parent CC at $addr_{i+1}$ := ParentFromChild($addr_i$)
         2. Compute $P_{i+1} := D_K(C_{i+1})$
         3. IF $addr_i \neq$ ADDR($P_i$) THEN ERROR
         4. IF NONCE($P_i$) $\neq$ REF_NONCE($addr_i$, $P_{i+1}$) THEN ERROR
      ELSE — $i = L$: the root of the tree is reached
         1. IF $addr_i \neq$ ADDR($P_i$) THEN ERROR
         2. IF (NONCE($P_i$) $\neq$ REF_NONCE($addr_i$, -1)) THEN ERROR

WriteAndUpdate(addr, new_data)
     ReadAndCheck(addr) — Load, decrypt and authenticate all chunks
     For ($i$ = 0, $i$ = $L$, $i$++)
      IF $i$ = 0 — update data in DC
           1. Compute $U_0 := E_K(new\_data \,||\, addr_0 \,||\,$ COUNTER($P_0$) + 1)
           2. Store $U_0$ to memory at $addr_0$
      ELSE IF i $\neq L$ — update counters in CCs
           1. Compute $U_i := E_K($UpdateCCPayload($addr_{i-1}$, $P_i$) || $addr_i$ || COUNTER($P_i$) + 1)
           2. Store $U_i$ to memory at $addr_i$
      ELSE — $i = L$: the root of the tree is reached
           1. On-Chip Counter := COUNTER($P_i$) + 1
```

Fig. 4. Pseudo-Code for TEC-Tree Checks and Updates

On *write operations*, the Data Chunk (DC) to be written is loaded, decrypted and authenticated using the parent Counter Chunk (CC), loaded immediately after the DC. Then, the DC's payload is updated, concatenated with a new nonce – the chunk's address concatenated to the incremented counter value CTR –, encrypted and written off-chip. The payload of the parent CC must then be updated with the new value of CTR. This process is done for all parents to update the trusted on-chip value. Note that all loaded chunks must be authenticated before being updated. This is done to preclude adversaries from injecting a fake chunk before the write, with the aim of corrupting parts of the chunk which are not affected by the write. As on read operations, the number of chunks to update is equal to *V*. Assuming that all chunks required to update the tree are on-chip, the computations involved in the update process are parallelizable: ECB decryptions can be done in parallel and so can the re-encryptions.

Security Analysis. TEC-Tree implements ECB mode to encrypt chunks. The weak point of this encryption mode is that a given plaintext block always yields the same ciphertext block upon encryption with a given key. This could leak some information to an adversary regarding the chunk's contents. In our scheme however, each encrypted chunk contains a nonce, so the same ciphertext block never occurs twice.

Moreover, our threat model considers an adversary can only access off-chip data. Thus, for spoofing attacks the adversary can only modify ciphertext blocks. As mentioned in Section 4.1, attacks consisting in the insertion of random ciphertext or the tampering with certain ciphertext bits succeed with probability $1/2^n$, where n is the bit width of the nonce. An adversary able to predict – without knowledge of the key – the effect of ciphertext bit manipulations on chosen bits in the plaintext (e.g. nonce bits) implies the underlying block cipher is broken.

Splicing attacks are detected as soon as a DC decryption is done by checking the address bits included in the nonce: when a chunk is spliced, the address the SoC uses to fetch the chunk does not match the bits extracted from the address field in the decrypted spliced chunk. Similarly to spoofing attacks, an adversary able to predict – without knowledge of the key – the effect of ciphertext bit manipulations on the address field of the plaintext implies the underlying block cipher is broken.

Checking address bits upon decryption of a DC thus allows for detection of splicing attacks before the DC's data payload is forwarded to the CPU pipeline. Spoofing attacks are also detected just after a DC decryption by applying the block-level AREA scheme with the address bits as redundancy. Since our scheme does not forward spliced or spoofed data to the pipeline, we say it provides splicing- and spoofing-free speculative execution. This way when the pipeline executes speculatively over data for which the tree authentication process has not terminated, we reduce the risk of such a speculative execution to the occurrence of a replay attack. This is not true for a hash tree since an adversary is able to compute intermediate nodes for spliced and spoofed data chunks.

For replay attacks, the intrinsic property of the nonce – i.e. uniqueness during the lifetime of an encryption key – ensures that the first non-replayed chunk in the authentication branch in the tree allows detection of the attack; if all chunks of this branch are replayed; the attack is detected by the last verification, which involves the trusted counter value. This value being stored on-chip it cannot be tampered with or replayed.

Memory cost. The off-chip memory overhead of the TEC-Tree scheme corresponds to the amount of memory required to store the counter chunks and the nonces for the encryption of data chunks. The overhead incurred by the nonces in the DCs is defined by the ratio n/l_p, the bit widths of nonces and payloads respectively. The overhead incurred by the construction of an A-ary tree over a set of leaves is $1/(A-1)$ [8]. In our case, this overhead is applied to DC leaves. Therefore, the memory overhead O_{Tree} of TEC-Tree is of:

$$O_{\text{Tree}} = \frac{1}{A-1}\left(1+\frac{n}{l_p}\right)+\frac{n}{l_p} = \frac{l_p + nA}{l_p(A-1)} \qquad (\mathbf{2})$$

Address of the parent chunk. For an efficient implementation of TEC-Tree, the address calculation for data chunks and counter chunk parents along the authentication branch must be straightforward. Because we protect the physical address space, the address computation may not be handled by the CPU core itself. The addressing shift due to the nonce bits contained in DCs and the CC parent addresses are computed on-the-fly by TEC-Tree hardware logic. For the sake of clarity, this paper considers that the entire physical memory space is protected by our scheme and that the memory stores the CCs starting at address 0, followed by the DCs at address A_S. We assume the operating system ensures these addresses from 0 to A_S are not accessed by software.

To address data and counter chunks, we adapt the method presented in [8]: all elements of a balanced tree are numbered consecutively starting from 0 at the root up to the leaves, as depicted in Figure 5 for a 4-ary tree. The position (i.e. number) of a parent node can easily be found by subtracting one from the position of one of its children, dividing the result by the arity and then rounding down.

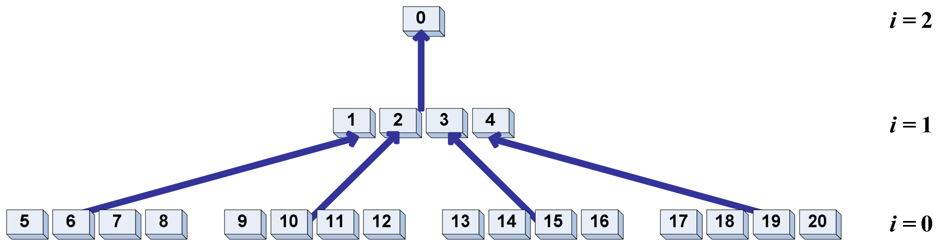


Fig. 5. Numbering of payloads

We now introduce the payload tree concept. A payload tree is formed by stripping the data and counter chunks of a TEC-Tree from their nonces, while keeping the same relationships between the nodes. The white parts of the TEC-Tree nodes in Figure 3 – along with the existing arrows – form a payload tree.

Let A_{CPU} be the address provided by the CPU; the position P_0 of a payload leaf in the payload tree is:

$$P_0 = \frac{(A_S + A_{CPU})}{l_{pb}} \quad \text{where } l_{pb} \text{ is } l_p \text{ expressed in bytes.} \qquad (\mathbf{3})$$

Applying the addressing scheme from [8] to the payload tree, the position of a parent payload P_i in the ith level of the payload tree is recursively computed from the position of its child P_{i-1} or from the position P_0 of the payload leaf to authenticate:

$$P_i = \left\lfloor \frac{P_{i-1} - 1}{A} \right\rfloor \quad \text{or} \quad P_i = \left\lfloor \frac{P_0 - i}{A^i} \right\rfloor \qquad (\mathbf{4})$$

When P_i becomes negative, this means the TEC-Tree root has been reached and the authentication must be done using the trusted counter value.

From a position computed with the equation above, the physical address ADD_{DC} of a data chunk corresponds to the location of the $P_0{}^{th}$ block of l_{pb} + n_b bytes (chunk size) in memory, where n_b is the nonce width in bytes. Similarly, the physical address $ADD_{CC(i)}$ of a counter chunk corresponds to the location of the P_i^{th} block of l_{pb} + n_b bytes in memory.

$$ADD_{CC(i)} = P_i \times (l_{pb} + n_b) \quad \text{and} \quad ADD_{DC} = P_0 \times (l_{pb} + n_b) \qquad (\mathbf{5})$$

Note that l_{pb}, n_b and A must be powers of 2 to allow a straightforward hardware computation of addresses.

5 Implementation Example

The proposed implementation of TEC-Tree uses the Rijndael algorithm [14] processing 192-bit blocks. A chunk before encryption is composed of a 128-bit payload (l_p = 128) and a 64-bit nonce (n = 64). We concatenate a 32-bit address to a 32-bit counter (r = 32) to produce the nonce. The result is a 4-ary TEC-Tree.

With a 64-bit nonce, the probability for an adversary to wage a successful spoofing attack is only $1/2^{64}$. The address bits in the tag ensure that the whole physical address space is protected against splicing when considering a 32-bit address space. Moreover, the use of a nonce in this TEC-Tree scheme prevents replay attacks while requiring only 32 bits of secure on-chip storage.

[6] shows that block-level AREA with the maximum throughput implemented with the Rijndael algorithm (192-bit blocks) consumes 80K gates on a platform with a 2:1 CPU to memory bus frequency ratio. Since the hardware required to implement the TEC-Tree scheme is the same as the one required to implement the standalone block-level AREA technique, a TEC-Tree also consumes 80 Kgates.

The main overhead of TEC-Tree is the 2x increase in off-chip memory. PAT – which also offers parallelizability on write operations – has the same 2x off-chip memory overhead but does not provide data confidentiality.

6 Conclusion

This paper presents TEC-Tree, a novel technique providing memory integrity with insignificant on-chip memory requirements – i.e. a single counter value. As opposed to the schemes based on Merkle Trees, TEC-Tree is fully parallelizable on both read and write operations. Moreover, a benefit of our approach over existing integrity tree techniques is that the TEC-Tree provides data confidentiality at no additional hardware, performance and memory costs – and without the information leakage that typically results from ECB encryption. Our solution also provides a new security property for implementations using speculative execution: the TEC-Tree ensures detection of splicing and spoofing attacks before sending read data to the processor pipeline. Considering these benefits, implementing TEC-Tree for memory integrity in commercial embedded devices is a more viable solution than Merkle Trees or PAT.

Ongoing work includes the deployment of TEC-Tree to provide memory integrity with a secure processor architected with our threat model in mind.

Acknowledgments. This work was supported in part by DARPA and NSF Cybertrust 0430487 and 0636808, DoD and NSF ITR.0326372.

References

1. P. Kocher, R. B. Lee, G. McGraw, A. Raghunathan, and S. Ravi, “Security as a New Dimension in Embedded System Design”, Proceedings of the Design Automation Conference (DAC), pp. 753-760, June 2004
2. A. Huang, “Keeping Secrets in Hardware the Microsoft Xbox Case Study”, MIT AI Memo, 2002.
3. M. G. Kuhn, “Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP”, IEEE Trans. Comput., vol. 47, pp. 1153–1157, October 1998.
4. C. Fruhwirth, “New Methods in Hard Disk Encryption”, Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology, 2005.
5. 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”, Proceedings of the 43rd Design Automation Conference DAC, July 2006.
6. R. Elbaz – “Hardware Mechanisms for Secured Processor Memory Transactions in Embedded Systems”, PhD Thesis, University of Montpellier, December 2006.
7. R. C. Merkle, “Protocols for Public Key Cryptography”, IEEE Symp. on Security and Privacy, pages 122–134, 1980.
8. B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, “Caches and Merkle Trees for Efficient Memory Integrity Verification”, Proceedings of Ninth International Symposium on High Performance Computer Architecture, February 2003.
9. C. Yan, B. Rogers, D. Englender, Y. Solihin, and M. Prvulovic, “Improving Cost, Performance, and Security of Memory Encryption and Authentication”, Proc. of the International Symposium on Computer Architecture, 2006.
10. W. E. Hall and C. S. Jutla. Parallelizable authentication trees. In Cryptology ePrint Archive, December 2002.

11. W. E. Hall, C. S. Jutla, “Parallelizable Authentication Tree for Random Access Storage,” U.S. Patent No. 2004/0107341 A1, June 2004.
12. G. E. Suh, “AEGIS: A Single-Chip Secure Processor”, PhD thesis, Massachusetts Institute of Technology, September 2005.
13. C. Shannon, “Communication theory of secrecy systems”, Bell System Technical Journal, 28, 1949.
14. J. Daemen, V. Rijmen, “AES Proposal: Rijndael”, March 1999, available at: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf.