Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines

Reouven Elbaz[1,2], David Champagne[2], Catherine Gebotys[1], Ruby B. Lee[2], Nachiketh Potlapally[3], and Lionel Torres[4]

[1] Department of Computer and Electrical Engineering, University of Waterloo
Waterloo, Canada
`{reouven,cgebotys}@uwaterloo.ca`

[2] Department of Electrical Engineering, Princeton University
Princeton, USA
`{relbaz,dav,rblee}@princeton.edu`

[3] Security Center of Excellence (SeCoE), Intel Corporation
Hillsboro, USA
`nachiketh.potlapally@intel.com`

[4] Department of Microelectronics, LIRMM, University of Montpellier
Montpellier, France
`torres@lirmm.fr`

Abstract. Trusted computing platforms aim to provide trust in computations performed by sensitive applications. Verifying the integrity of memory contents is a crucial security service that these platforms must provide since an adversary able to corrupt the memory space can affect the computations performed by the platform. After a description of the active attacks that threaten memory integrity, this paper surveys existing cryptographic techniques – namely integrity trees – allowing for memory authentication. The strategies proposed in the literature for implementing such trees on general-purpose computing platforms are presented, along with their complexity. This paper also discusses the effect of a potentially compromised Operating System (OS) on computing platforms requiring memory authentication and describes an architecture recently proposed to provide this security service despite an untrusted OS. Existing techniques for memory authentication that are not based on trees are described and their performance/security trade-off is discussed. While this paper focuses on memory authentication for uniprocessor platforms, we also discuss the security issues that arise when considering data authentication in symmetric multiprocessor (shared memory) systems.

Keywords: Security, Trusted Computing, Memory Authentication, Integrity Trees, Active attacks, Board level attacks.

1 Introduction

The increasing connectivity of computing devices coupled with rapid growth in number of services these devices offer, has resulted in more and more end users deploying computing platforms for a wide variety of tasks including those which handle sensitive data. Examples of sensitive data include bank account number, passwords, social security number, health information etc. A typical user logs into online accounts using secret passwords almost every day, and procuring services requires the user to create new accounts (for instance, subscribing to a cable service, and creating an online account to manage it), and consequently create and store additional passwords. Even asking for an online insurance quote might require the user to give out personal information such as his or her social security number. Thus, going by these trends, we can say that the amount of sensitive information processed by and stored on the computing devices is projected to further increase with time. In such online transactions, users expect their sensitive data to be properly stored and handled by trusted computer systems of the service provider at the receiving end. However, if this trust assumption proves to be wrong, then the breach in user confidence can severely undermine the reputation and profits of the service provider, as was the case in a recent scandal where a computer previously owned by a bank and containing information on several million bank customers was sold on eBay [20]. On first glance, this appears solely to be an issue of data confidentiality and access control. However, when an attacker gets control of a computing system on which data is encrypted, the attacker can still compromise security of user transactions by appropriately manipulating the encrypted data. This was illustrated in an attack carried out on a crypto-processor where an adversary could alter the execution flow of software by corrupting encrypted code and data in memory to reveal confidential information [5]. Thus, it is not sufficient to just enforce data confidentiality and access controls, but, it is imperative to also take data integrity into account. Data integrity refers to ability to detect any adversarial corruption or tampering of data.

Several recent research efforts in academia [1, 2, 3] and industry [4] aim to provide trust in the computations performed by sensitive applications—e.g., Digital Rights Management (DRM) client, personal banking application, distributed computing client—on general-purpose computing platforms. Protecting confidentiality of private data and detecting any tampering are important features of trusted computing. Since an adversary corrupting the memory space of an application (through software [21] or physical attacks [5]) can affect the outcome of its computations, these computing platforms must provide memory authentication for sensitive applications. Memory authentication is defined as the ability to verify that the data read from memory by the processor (or by a specific application) at a given address is the data it last wrote at this address.

In past work on memory authentication, the main assumption of the trust model is that only the processor chip is trusted i.e., the trust boundary includes the computing and storage elements within the processor. A naïve approach to memory authentication would be to compute a cryptographic digest of contents of the entire external memory, store it on the trusted area (i.e., the processor chip) and use that digest to check the integrity of every block read from off-chip memory. Since the digest is inaccessible to the adversary, any malicious modification of the memory can be easily detected. However, this solution requires fetching on-chip all external memory contents on every read operation to check the digest, and on every write operation, to update it; clearly, this generates an unacceptable overhead in memory bandwidth. Another simple solution (based on the same trust model mentioned above) would be to keep on-chip a cryptographic digest of every memory block (e.g., cache block) written to off-chip memory. Although this strategy greatly reduces memory bandwidth overhead compared to the previous solution, it imposes an unacceptable cost in terms of the amount of on-chip memory required for storing all the digests. To improve on these two extreme solutions (i.e., to reduce memory bandwidth overhead to a small number of metadata blocks and the on-chip memory cost to a few bytes), researchers [6, 7, 8] have proposed tree-based structures whose leaves are the memory blocks to be protected, and the value of the root is computed by successively applying an authentication primitive to tree nodes starting from the leaves. Thus, the root node captures the current state of the memory blocks, and it is made tamper-resistant by being stored on-chip. When the processor reads a datum from external memory, a dedicated on-chip memory authentication engine computes the root node hash value by using values stored in internal nodes lying on the path from the leaf (corresponding to the memory block read) to the root. If the memory block was not tampered, then the computed root value matches the one stored on-chip else they would differ. On a write, the authentication engine updates the root hash value to reflect the change in state of memory due to the newly written value.

This paper surveys tree-based memory authentication techniques. We discuss various implementation-based issues raised by the integration of these trees into general-purpose computing platforms, namely: How can the processor address tree nodes in memory? Can integrity tree nodes be cached? Do the security properties of the integrity tree hold under Operating System (OS) compromise? To the best of our knowledge, we are not aware of any other work which offers a similar comprehensive analysis of architectural issues and trade-offs arising from implementing existing tree-based memory authentication schemes. By connecting theory to implementation, our work provides insights which will prove useful in designing more efficient memory authentication schemes, and we consider this to be an important contribution of this paper. For the sake of completeness, we also describe techniques proposed in literature that employ non-tree based techniques for verifying memory integrity, and show how they achieve improved performance at the cost of decrease in system security. Finally we briefly discuss the issues raised by data authentication in multiprocessor systems by highlighting the difference with a uniprocessor platform.

The paper is organized as follows. Section 2 describes the active attacks threatening the integrity of the memory contents. Then Section 3 presents the existing techniques providing memory authentication, namely integrity trees; in particular, we show why all existing memory authentication solutions defending against the attacks in Section 2 are based on tree structures. Section 4 presents the architectural features proposed in the literature to efficiently integrate those integrity trees to general-purpose computing platforms. Section 5 discusses the security of the integrity tree under operating system compromise and describes an architecture proposed recently to efficiently deploy an integrity tree on a computing platform running an untrusted OS. Section 6 describes techniques proposed in the literature to verify memory integrity and not based on tree structures. While this paper focuses on memory integrity for uniprocessor platforms, we present in Section 7 the additional security issues to consider for data authentication in symmetric multiprocessor (shared memory) systems. Section 8 concludes the paper.

2 Threat Model

This section describes the attacks challenging the integrity of data stored in the off-chip memory of computer systems. Section 2.1 describes the model we consider for active attacks in the context of physical adversaries. Section 2.2 widens the discussion by including the cases where these attacks are carried out through a malicious operating system; we conclude the section by defining two threat models upon which existing solutions for memory integrity are built.

2.1 Hardware Attacks

The common threat model considered for memory authentication assumes the protected system is exposed to a hostile environment in which physical attacks are feasible. The main assumption of this threat model is that the processor chip is resistant to all physical attacks, including invasive ones, and is thus trusted. Side-channel attacks are not considered.

The common objective of memory authentication techniques is to thwart active attackers tampering with memory contents. In an active attack, the adversary corrupts the data residing in memory or transiting over the bus; this corruption may be seen as data injection since a new value is created. Figure 1 depicts the example of a device under attack, where an adversary connects its own (malicious) memory to the targeted platform via the off-chip bus. We distinguish between three classes of active attacks, defined with respect to how the adversary chooses the inserted data. Figure 2 depicts the three active attacks; below, we provide a detailed description of each one by relying on the attack framework in Figure 1:

1) *Spoofing* attacks: the adversary exchanges an existing memory block with an arbitrary fake one (Figure 2-a, the block defined by the adversary is stored in the malicious memory, the adversary activates the switch command when he wants to force the processor chip to use the spoofed memory block).
2) *Splicing* or *relocation* attacks: the attacker replaces 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 (Figure 2-b: the adversary stores at address 5 in the malicious memory the content of the block at address 1 from the genuine memory. When the processor requests the data at address 5, the adversary activates the switch command so the processor reads the malicious memory. As a result, the processor reads the data at address 1).
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 (Figure 2-c: at time t1, the adversary stores at address 6 in the malicious memory the content of the block at address 6 from the genuine memory. At time t2, the memory location at address 6 has been updated in the genuine memory but the adversary does not perform this update in the malicious memory. The adversary activates the malicious memory when the processor requests the data at address 6, thus forcing it to read the old value stored at address 6).

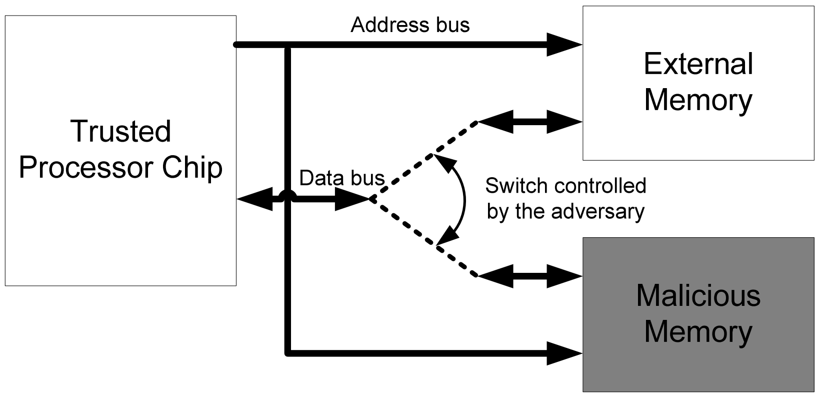

Fig. 1. An Example of Framework of Attack Targeting the External Memory of a Computing Platform

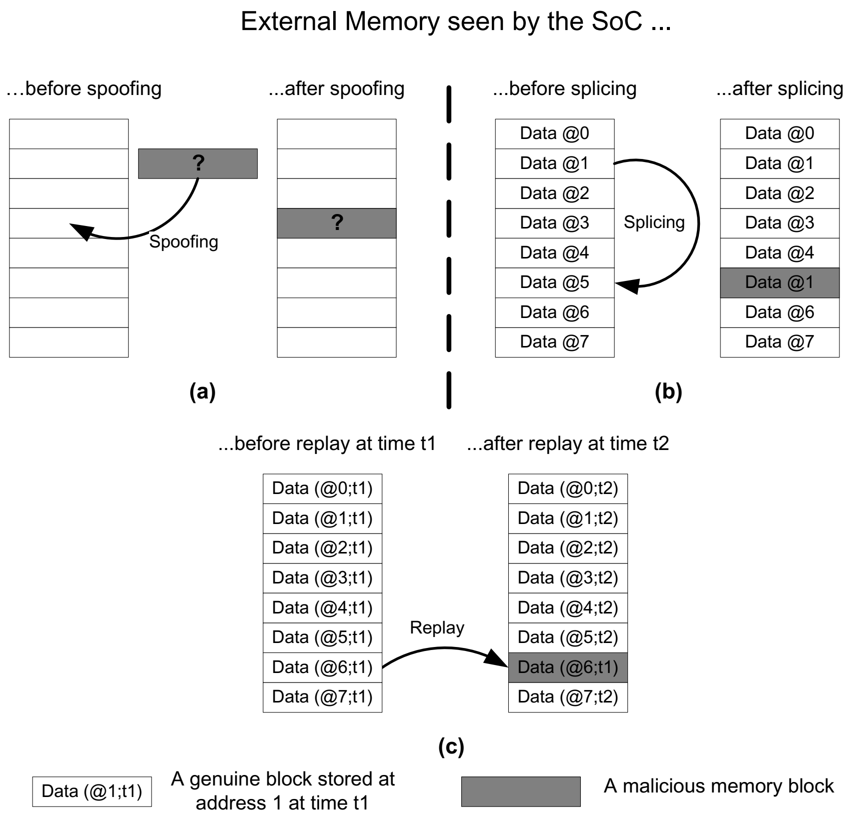

Fig. 2. Three Kinds of Active Attacks: (a) Spoofing, (b) Splicing and (c) Replay

2.2 Software Attacks

In a software attack, a compromised (or outright malicious) operating system or application tries to corrupt the memory space of a sensitive application. To model these attacks, we subsume all possible attack vectors into a single threat: a malicious, all-powerful operating system. Such an OS can directly read and write any memory location belonging to a sensitive application and can thus carry out any of the splicing, spoofing and replay attacks presented in the previous section.

In existing work on memory authentication, the threat model either excludes software attacks [2, 8, 9, 10, 11, 12, 13, 14] (referred in the following as *threat model 1*) or includes software attacks [2, 21] (referred in the following as *threat model 2*). In threat model 1, the hardware architecture must protect sensitive applications against attacks from software; in threat model 2, the hardware does not provide such protection. When software attacks are not considered (threat model 1), the operating system (OS) or at least the OS kernel must thus be trusted to isolate sensitive applications from malicious software. In the other case, the OS can contain untrusted code since the hardware protects sensitive applications against malicious software.

Conceptually, integrity trees are built and maintained in the same way regardless of the considered threat model. Section 3 describes existing integrity trees without specifying the threat model. Section 4 presents the strategies allowing efficient integration to computing platforms when threat model 1 is considered. Section 5 shows that threat model 2 requires trees built over the virtual (rather than physical) address space or, as recently proposed in [21], over a compact version of it.

3 Integrity Trees: Cryptographic Schemes for Memory Authentication

We consider there are three distinct strategies to thwart the active attacks described in our threat model. Each strategy is based on different authentication primitives, namely cryptographic hash function, Message Authentication Code (MAC) function and block-level Added Redundancy Explicit Authentication (AREA). In this section, we first describe how those primitives allow for memory authentication and how they must be integrated into tree structures in order to avoid excessive overheads in on-chip memory.

3.1 Authentication Primitives for Memory Authentication

Hash Functions. The first strategy (Figure 3-a) allowing to perform memory authentication consists in storing on-chip a hash value for each memory block stored off-chip (*write operations*). The integrity checking is done on *read operations* by re-computing a hash over the loaded block and by then comparing the resulting hash with the on-chip hash fingerprinting the off-chip memory location. The on-chip hash is stored on the tamper-resistant area, i.e., the processor chip and is thus inaccessible to adversaries. Therefore, spoofing, splicing and replay are detected if a mismatch occurs in the hash comparison. However, this solution has an unaffordable on-chip memory cost: by considering the common strategy [2, 3, 9, 13] of computing a fingerprint per cache line and assuming 128-bit hashes and 512-bit cache lines, the overhead is of 25% of the memory space to protect.

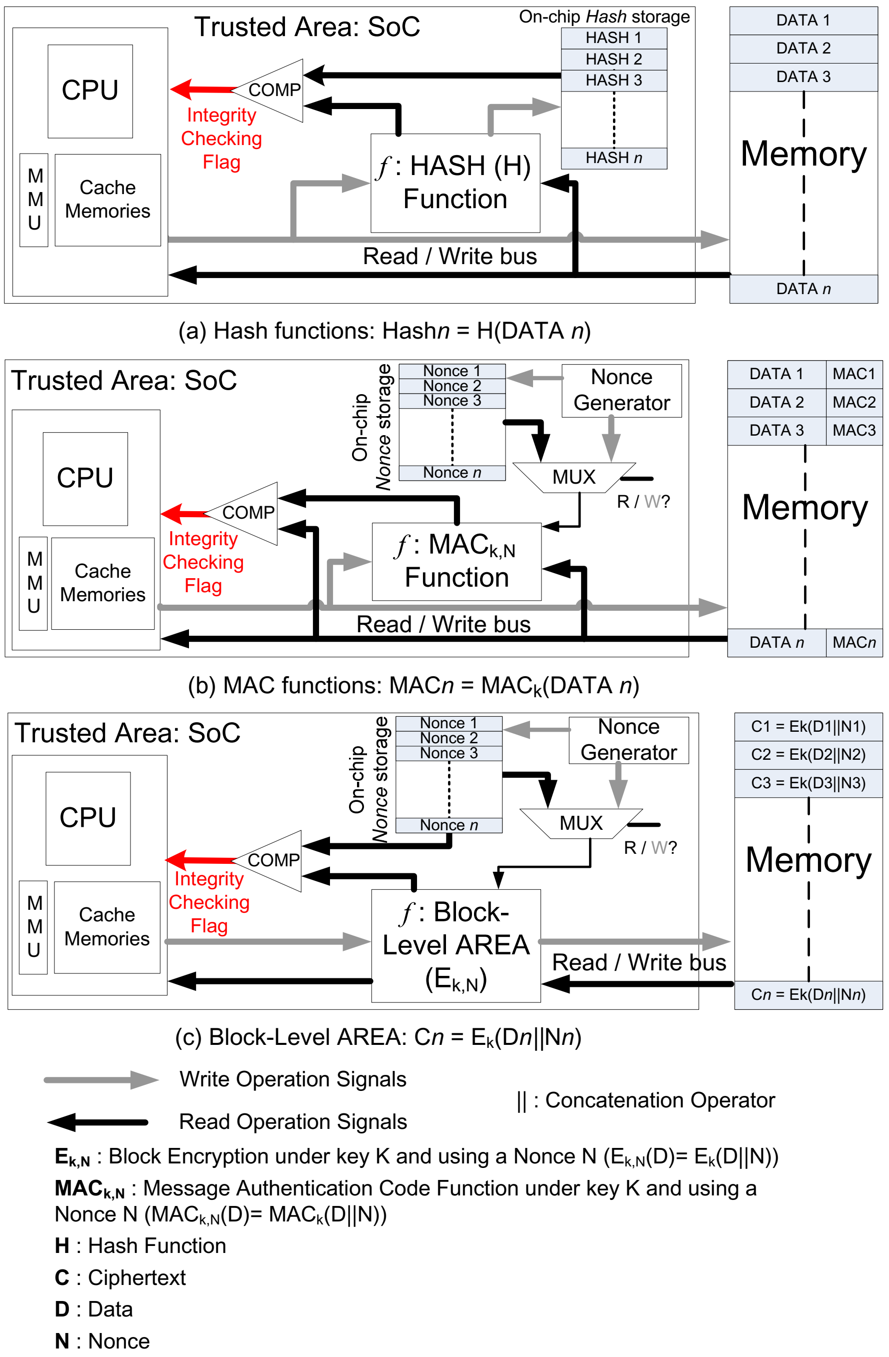

Fig. 3. Authentication Primitives for Memory Integrity Checking

MAC Functions: In the second approach (Figure 3-b), the authentication engine embedded on-chip computes a MAC for every data block it *writes* in the physical memory. The key used in the MAC computation is securely stored on the trusted processor chip such that only the on-chip authentication engine 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 a nonce. This allows protection against splicing and replay attacks. The nonce 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. It also 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. On *read operations*, the processor loads the data to read and its corresponding MAC from physical memory. It checks the integrity of the loaded block by first re-computing a MAC over this block and a copy of the nonce used upon writing, and then it compares the result with the fetched MAC. To ensure the resistance to replay and splicing, the nonce used for MAC re-computation must be genuine. A naïve solution to meet this requirement is to store the nonces on the trusted and tamper-evident area, the processor chip. The related on-chip memory overhead is 12.5% if we consider computing a MAC per 512-bit cache line and that we use 64-bit nonces.

Block-Level AREA: The last strategy [13, 14] (Figure 3-c) leverages the diffusion property of block encryption to add the integrity-checking capability to this type of encryption algorithm. To do so, the AREA (Added Redundancy Explicit Authentication [22]) technique is applied at the block level:

i) Redundant data (a *n*-bit nonce N) is concatenated to the data D to authenticate in order to form a plaintext block P (where P=D||N); ECB (Electronic CodeBook) encryption is performed over P to generate ciphertext C.
ii) Integrity verification is done by the receiver who decrypts the ciphertext block C' to generate plaintext block P', and checks the *n*-bit redundancy in P', i.e., assuming P'=(D'||N'), verifies whether N=N'.

Thus, upon a *memory write*, the on-chip authentication engine 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 processor chip. On *read operations*, the authentication engine decrypts the block it fetches from memory and checks its integrity by verifying that the last *n* bits of the resulting plaintext block are equal to the nonce that was inserted upon encryption (on the write of the corresponding data). [13, 14] propose a System-on-Chip (SoC) implementation of this technique for embedded systems. They show that this engine is efficient to protect the Read-Only (RO) data of an application (e.g., its code) because RO data are not sensitive to replay attacks; therefore the address of each memory block can be efficiently used as a nonce[1]. However, for Read/Write (RW) data (e.g., stack data), the address is not sufficient to distinguish two data writes at the same address carried out at two different points in time: the nonce must change on each write. To recover such a changing nonce on a read operation while ensuring its integrity, [13, 14] propose

[1] Note that the choice of the data address as nonce also prevent spoofing and splicing attacks of RO data when MAC functions are used as authentication primitives.

storing the nonce on-chip. They evaluate the corresponding overhead between 25% and 50% depending on the block encryption algorithm implemented.

3.2 Integrity Trees

The previous section presented three authentication primitives preventing the active attacks described in our threat model. Those primitives require the storage of reference values – i.e., hashes or nonces – on-chip to thwart replay attacks. They do provide memory authentication but only at a high cost in terms of on-chip memory. If we consider a realistic case of 1GB of RAM memory, the hash, MAC (with nonce) and the block-level AREA solutions require respectively at least 256MB, 128MB and 256 MB of on-chip memory. Those on-chip memory requirements clearly are unaffordable, even for high-end processors. It is thus necessary to "securely" store these reference values off-chip. By securely, we mean that we must be able to ensure their integrity to preclude attacks on the reference values themselves.

Several research efforts suggest applying the authentication primitives recursively on the references. By doing so, a tree structure is formed and only the root of the tree—the reference value obtained in the last iteration of the recursion—needs to be stored on the processor chip, the trusted area. There are three existing tree techniques:

i) Merkle Tree [6] uses hash functions,
ii) PAT (Parallelizable Authentication Tree) [7] uses MAC functions with nonces,
iii) TEC-Tree (Tamper-Evident Counter Tree) [8] uses the block-level AREA primitive.

In this section, we first present a generic model for the integrity tree, then we describe the specific characteristics of each existing integrity tree; we finally compare their intrinsic properties.

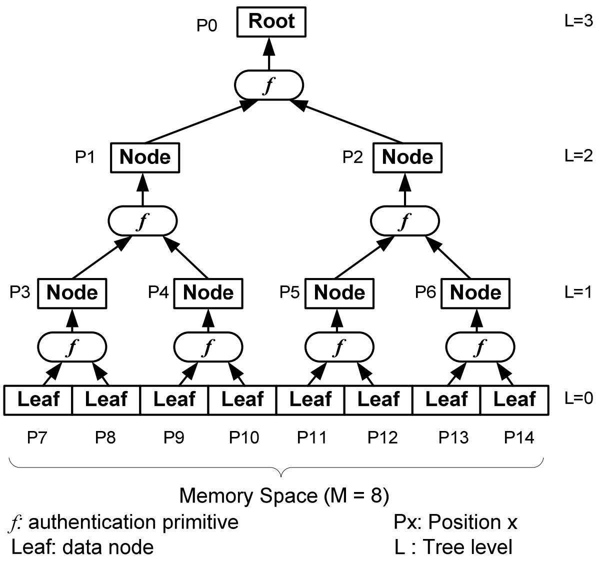

Fig. 4. General Model of 2-ary Integrity Tree

General Model of Integrity Tree. The common philosophy behind integrity trees is to split the memory space to protect into M equal size blocks which are the leaf nodes of the balanced *A*-ary integrity tree (Figure 4). The remaining tree levels are created by recursively applying the authentication primitive *f* over *A*-sized groups of memory blocks, until the procedure yields a single node called the root of the tree. The arity of the constructed tree is thus defined by the number of children *A* a tree node has. The root reflects the current state of the entire memory space; making the root tamper-resistant thus ensures tampering with the memory space can be detected. How the root is made tamper-resistant depends on the nature of *f* and is detailed next. Note that the number of checks required to verify the integrity of a leaf node depends on the number of iterations of *f* and thus on the number of blocks M in the memory space. The number of check corresponds to the number of tree levels N_L defined by: $N_L = \log_A(M)$.

Tree Authentication Procedure. For each memory block B (i.e., leaf node), there exists a branch[2] – starting at B and ending at the root – composed of the tree nodes obtained by recursive applications of *f* on B. For instance in Figure 4 for the leaf node at position P8, the branch is composed of the nodes at positions P3, P1 and the root P0. Thus, when B is fetched from untrusted memory, its integrity is verified by re-computing the tree root using the fetched B and the nodes – obtained from external memory – along the branch from B to the root (i.e., the branch nodes and their siblings; so for the leaf node at position P8, the nodes that need to be fetched are at position P7, P3, P4, P1 and P2). We confirm B has not been tampered with during the last step of the authentication process when the re-computed root is identical to the root (which has been made tamper-resistant).

Tree Update Procedure. When a legitimate modification is carried out over a memory block B, the corresponding branch – including the tree root – is updated to reflect the new value of B. This is done by first authenticating the branch B belongs to by applying the previous tree authentication procedure, then by computing on-chip the new values for the branch nodes, and finally by storing the updated branch off-chip – except for the on-chip component of the root.

Merkle Tree is historically the first integrity tree. It has been originally introduced by Merkle [6] for efficient computations in public key cryptosystems and adapted for integrity checking of memory content by Blum et al. [15]. In a Merkle Tree (Figure 5-a), *f* is a cryptographic hash function H(); the nodes of the tree are thus simple hash values. The generic verification and update procedures described above are applied in a straightforward manner. The root of this tree reflects the current state of the memory space since the collision resistance property of the cryptographic hash function ensures that in practice, the root hashes for any two memory spaces differing by at least one bit will not be the same. With Merkle Tree, the root is made tamper-resistant by storing it entirely on the trusted processor chip. The Merkle Tree authentication procedure is fully parallelizable because all the inputs required for this process can be made available before the start of this procedure; however, the update procedure is sequential because the computation of a new hash node in a branch must be completed before the update to the next branch node can start. By assuming that all tree nodes have the same size, the memory overhead MO_{MT} of a Merkle Tree [9] is of:

[2] This branch is also called in the following the authentication branch.

$$MO_{MT} = \frac{1}{(A-1)}.$$

The Parallelizable Authentication Tree (PAT) [7] overcomes the issue of non-parallelizability of the tree update procedure by using a MAC function (with nonces N) $M_{K,N}()$ as authentication primitive f where K and N are the key and nonce, respectively (Figure 5-b). The memory space is first divided into M memory blocks (in Figure 5-b, M=4). We begin by applying the MAC function to A memory blocks (in Figure 5-b, we have A=2) using an on-chip key K and a freshly generated nonce N, i.e., $MAC_{K,N}(d_1 || \ldots || d_A)$ where d_i is a memory block. Next, the MAC is recursively applied to A-sized groups formed with the nonces generated during the last iteration of the recursion. For every step of the recursion but the last, both the nonce and the MAC values are sent to external memory. The last iteration generates a MAC and a nonce that form the root of the tree. The root MAC is sent to external memory but the nonce N is stored on-chip; this way the tree root is made tamper-resistant since an adversary cannot generate a new MAC without the secret key K stored on-chip or replay an old MAC since it will not have been generated with the current root nonce. Verifying a memory block D in a PAT requires recomputing D's branch on-chip and verifying that the top-level MAC can indeed be obtained using the on-chip nonce.

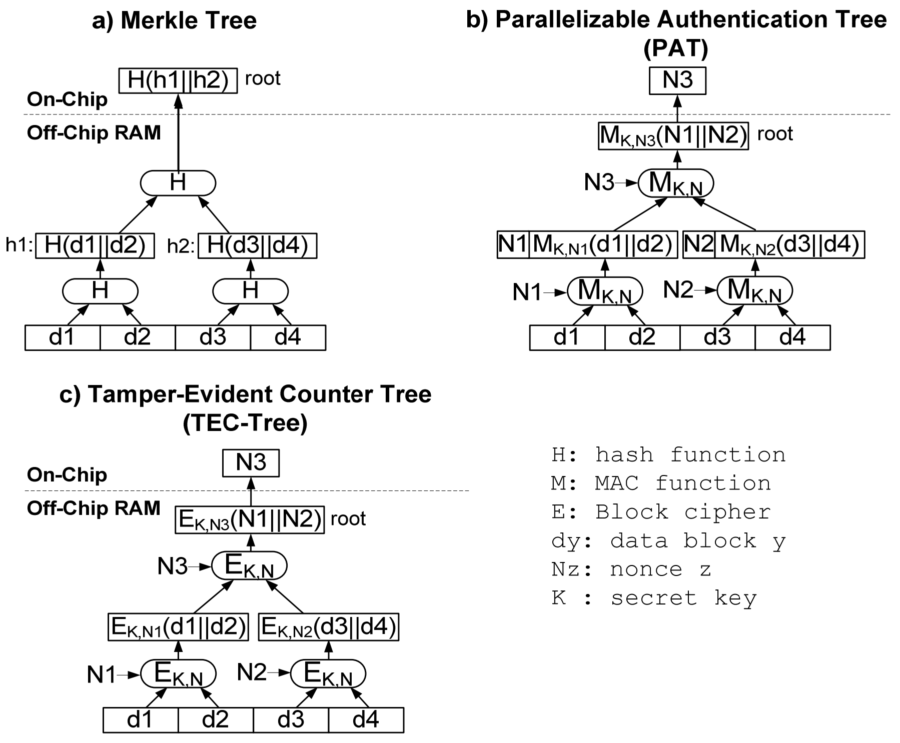

Fig. 5. Existing Integrity Trees

Whenever a block D is legitimately modified, the CPU re-computes D's branch using fresh nonces. The tree authentication procedure is parallelizable since all inputs (data and nonces) are available for all branch node verifications. The tree update procedure is also parallelizable because each branch tree node is computed from independently generated inputs: the nonces. In [7] the authors highlight that the birthday paradox implies the nonce does not need to be longer that *h*/2, with *h* the MAC-size. Thus, the memory overhead MO_{PAT} of PAT is of:

$$MO_{PAT} = \frac{1}{A-1} + \frac{1}{2}\frac{1}{A-1} = \frac{3}{2(A-1)}$$

The Tamper-Evident Counter Tree (TEC-Tree). In TEC-Tree [8], the authentication primitive *f* is the Block-level AREA. Thus, such authentication primitive tags its input (dy in Figure 5-c) with a nonce N before ciphering it with a block encryption algorithm in ECB mode and a secret key K kept on-chip. The block-level AREA is first applied to the memory blocks to be stored off-chip, and then recursively over A-sized groups of nonces used in the last iteration of the recursion. The resulting ciphered blocks are stored in external memory and the nonce used in the ciphering of the last block created – i.e., the root of the TEC-Tree – is kept on-chip making the root tamper-resistant. Indeed, an adversary without the key cannot create a tree node and without the on-chip root nonce he cannot replay the tree root. During verification of a data block D, D's branch is brought on-chip and decrypted. The integrity of D is validated if:

i) each decrypted node bears a tag equal to the nonce found in the payload of the node in the tree level immediately above;
ii) the nonce obtained by decrypting the highest level node matches the on-chip nonce.

The tree update procedure consists in:

i) loading D's branch decrypting nodes,
ii) updating nonces
iii) re-encrypting nodes.

TEC-Tree authentication and update procedures are both parallelizable because *f* operates on independently generated inputs: the nonces. The distinctive characteristic of TEC-Tree is that it allows for data confidentiality. Indeed, its authentication primitive being based on a block encryption function, the application of this primitive on the leaf nodes (data) encrypts them. The memory overhead[3] MO_{TEC} of TEC-Tree [8] is of :

$$MO_{TEC} = \frac{2}{(A-1)}$$

[3] [8] gives a different formula for their memory overhead because they consider ways to optimize it (e.g. the use of the address in the construction of the nonce). For the sake of clarity, we give a simplified formula of the TEC-Tree memory overhead by considering that the nonce consists only in a counter value.

Comparison. Table 1 sums up the properties of the existing integrity trees. PAT and TEC-Tree are both parallelizable for the tree authentication and update procedure while preventing all the attacks described in the state of the art. Parallelizability of the tree update process is an important feature when the write buffer is small (e.g., in embedded systems) to prevent bus contention due to write operation requests that may pile up. TEC-Tree additionally provides data confidentiality. However, TEC-Tree and PAT also have a higher off-chip memory overhead when compared to Merkle Tree, in particular because they require storage for additional metadata, the nonces.

Table 1. Summary of Existing Integrity Trees Properties

	Merkle Tree	**PAT** (Parallelizable Authentication Tree)	**TEC-Tree** (Tamper-Evident Counter Tree)
Splicing, Spoofing, Replay resistance	Yes	Yes	Yes
Parallelizability	Tree Authentication only	Tree Authentication **and** Update	Tree Authentication **and** Update
Data Confidentiality	No	No	Yes
Memory Overhead	1/(A-1)	3/2(A-1)	2/(A-1)

4 Integration of Integrity Trees in Computing Platforms

In this section we survey the implementation strategies proposed in the literature to efficiently integrate integrity trees in computing platforms when threat model 1 is considered (i.e., active physical attacks and trusted OS kernel). We first describe the tree traversal technique allowing for node addressing in memory. Then, a scheme leveraging caching to improve tree performance is detailed. Finally, the Bonsai Merkle Tree concept is described.

4.1 Tree Traversal Technique

One of the issues arising when integrating an integrity tree into a computing platform is making it possible for the processor to retrieve tree nodes in memory. [9] proposes a method for doing so with Merkle trees, while [8] adapts it for TEC-Tree. The principle of this method is to first associate a numerical position (P_x in Figure 3) to each tree node, starting at 0 for the root and incrementally up to the leaves. The position of a parent node P^l (at level l in the tree) can be easily found by subtracting one from its child at position P^{l-1} (on level l-1), by dividing the result by the tree arity A and by rounding down:

$$P^l = \left\lfloor \frac{P^{l-1} - 1}{A} \right\rfloor \qquad \text{(eq. 1)}$$

Now that we know how to find a parent node position from a child position, the issue is to retrieve a position number from the address of a child or parent node. To solve this issue, [9] proposes a simplified layout of the memory region to authenticate: the tree nodes are stored starting from the top of the tree (P1 in Figure 4) down to the leaves, with respect to the order given by positions. By having all tree nodes be the same size, the translation from position to address or from address to position can be easily done by respectively multiplying or dividing the quantity to be translated by the node size.

This method imposes that the arity be a power of 2 to efficiently implement the division of eq.1 in hardware. Moreover, all data to authenticate must be contained in a contiguous memory region to allow for the children to parent position and address retrieval scheme to work.

4.2 Cached Trees

The direct implementation of integrity trees can generate a high overhead in terms of execution time due to the $\log_A(M)$ checks required on each load from the external memory. In [9], Gassend et al. show that the performance slowdown can reach a factor of 10 with a Merkle Tree. To decrease this overhead, they propose to cache tree nodes. When a hash is requested in the tree authentication procedure, it is brought on-chip with its siblings in the tree that belong to the same cache block. This way, those siblings usually required for the next check in the authentication procedure are already loaded. However, the main improvement comes from the fact that once checked and stored in the on-chip cache, a tree node is trusted and can be considered as a local tree root. As a result, the tree authentication and update procedures are terminated as soon as a cached hash (or the root) is encountered. With this cached tree solution, Gassend et al. decreased the performance overhead of a Merkle Tree to less than 25%. By changing the hash function – from SHA-1[16] to the GCM[17] – [11] even claims to keep the performance overhead under 5%.

4.3 The Bonsai Merkle Tree

Memory authentication engines based on integrity trees should be designed for efficient integration into computing platforms. The last engine proposed toward this objective has been presented in [12] and is based on a concept called Bonsai Merkle Tree.

The idea behind the Bonsai Merkle Tree (BMT) is to reduce the amount of data to authenticate with a Merkle Tree in order to decrease its height, i.e., reduce the number of tree levels to obtain a smaller tree that can be quickly traversed (Figure 6). To do so, [12] proposes to compute a MAC *M* over every memory block *C* (i.e., every cache block) with a nonce, i.e., a counter *ctr* concatenated with the data address, as extra input of the MAC function $\mathrm{MAC_K}$: $M = \mathrm{MAC_K}(C, addr, ctr)$[4]. The counter *ctr* consists of a local counter *Lctr* concatenated with a global counter *Gctr*. Each memory block is associated with a local counter while each memory page is associated with a global counter. Each time a given memory block is updated off-chip, the corresponding *Lctr*

[4] The authentication primitive used in [12] is basically the MAC function with nonces presented in section 3.

is incremented. When *Lctr* rolls over, *Gctr* is incremented. In [12], the authors proposed the use of a 7-bit long *Lctr* and of a 64-bit long *Gctr*; this way *ctr* never rolls over in practice. *ctr* counter values are made tamper-evident while stored off-chip using a Merkle tree, thus making the memory space protected by the MAC-with-nonce scheme also tamper-evident.

On average, an 8-bit counter is required for 4KB memory pages and 64B cache blocks. The amount of memory to authenticate with the Bonsai Merkle tree is thus decreased of a ratio 1:64 when compared to a regular Merkle tree applied directly to the memory blocks. However, the shortcoming of this scheme is that a full page needs to be cryptographically processed every time a local counter rolls over. In other words, in this case the MACs of all memory blocks belonging to the page having *Gctr* updated must be recomputed as well as the tree branches corresponding to the tree leaves containing the updated *Gctr* and *Lctr*. Despite this, according to [12], this approach decreases the execution time overhead of integrity trees from 12.1% to 1.8% and reduces external memory overhead for node storage from 33.5% to 21.5%.

Duc et al. proposed a similar architecture in [25] except that instead of using a nonce in the MAC computation, they include a random number; therefore, their architecture, called CryptoPage, is sensitive to replay with a success probability proportional to the size of the random value.

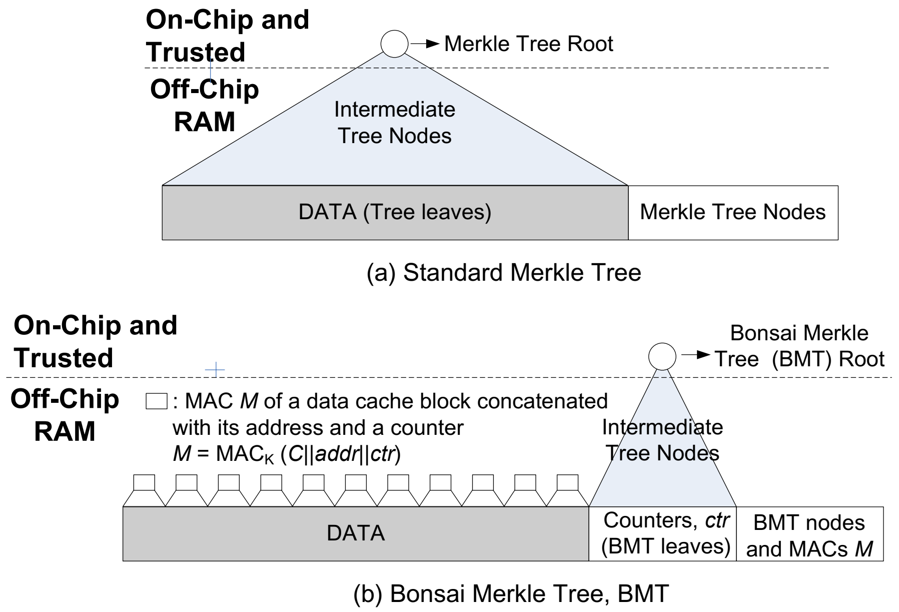

Fig. 6. Bonsai Merkle Tree Principle

5 Memory Authentication with an Untrusted Operating System

In many scenarios, the security policy of a computing platform must exclude the operating system from the trusted computing base. With modern commodity operating systems in particular, it is practically impossible to verify that no exploitable software vulnerability exists in such a large, complex and extendable software system. As a result, the OS cannot be trusted to isolate a sensitive application from malicious software, hence it needs to be considered as untrusted in the threat model (as in our threat model 2).

The first step towards protecting the memory of a sensitive application running on an untrusted OS is to build an integrity tree which covers only pages belonging to the application and which can only be updated when the application itself is running. This precludes the OS, with certain well-defined exceptions for communication, from effecting modifications to the application's memory space that cause the tree to be updated, i.e., any OS write to the application's memory is considered as corruption. Although this approach prevents the OS from carrying out splicing, spoofing and replay attacks through direct writes to the application's memory (because such corruptions would be detected by the integrity tree scheme), [21] shows that the OS can still perform splicing attacks indirectly, by corrupting the application's page table.

The Branch Splicing Attack. The page table is a data structure maintained by the OS, which maps a page's virtual address to its physical address. On a read operation, when a virtual-to-physical address translation is required by the processor, the page table is looked up[5] using the virtual address provided by the running process to obtain the corresponding physical address. The *branch splicing attack* presented in [21] corrupts the physical address corresponding to the virtual address of a given memory block. This causes the on-chip integrity verification engine not only to fetch the wrong block in physical memory, but also to use the wrong tree branch in verifying the integrity of the block fetched. [21] shows that with threat model 2, building an integrity tree over the Physical Address Space (PAS tree) is insecure because it is vulnerable to the branch splicing attack. In a PAS tree, the physical address determines the authentication branch to load to re-compute the root during verification. As a result, the OS can corrupt block A's virtual-to-physical address translation to trick the integrity checking engine into using block B's authentication branch to verify block B, hence substituting B for A (In Figure 7, substituting data at address @0 for data at address @4).

Building a Tree over the Virtual Address Space (VAS-Tree). To defend against this attack, the integrity tree can be built over the Virtual Address Space (VAS tree). In this case, the virtual address generated by the protected application is used to traverse the tree so page table corruption has no effect on the integrity verification process. The VAS tree, unlike a PAS tree, protects application pages that have been swapped out to disk by the OS paging mechanism since it can span all pages within the application's virtual memory space. PAS trees only span physical memory pages: they do not keep track of memory pages sent to the on-disk page file, hence they require a separate mechanism to protect swapped out pages [12]. Without such a mechanism, a PAS tree scheme cannot detect corruption of data that might occur during paging—i.e. between the time a page is moved from its physical page frame to the on-disk page file and the time that page is fetched again by the OS, from disk back into a physical page frame.

[5] In modern processor, the page table is cached on-chip in a Translation Lookaside Buffer. This does not affect the feasibility of the attack described here.

The VAS tree is not a panacea however, as it presents two major shortcomings: it must span a huge region of memory and it requires one full-blown tree for each application requiring protection, rather than a single tree protecting all software in physical memory. In addition, this solution requires extra tag storage for the virtual address [2] in the last level of cache (when implemented) which is usually physically tagged and indexed. Indeed, on cache evictions, the virtual address is required to traverse and update the integrity tree.

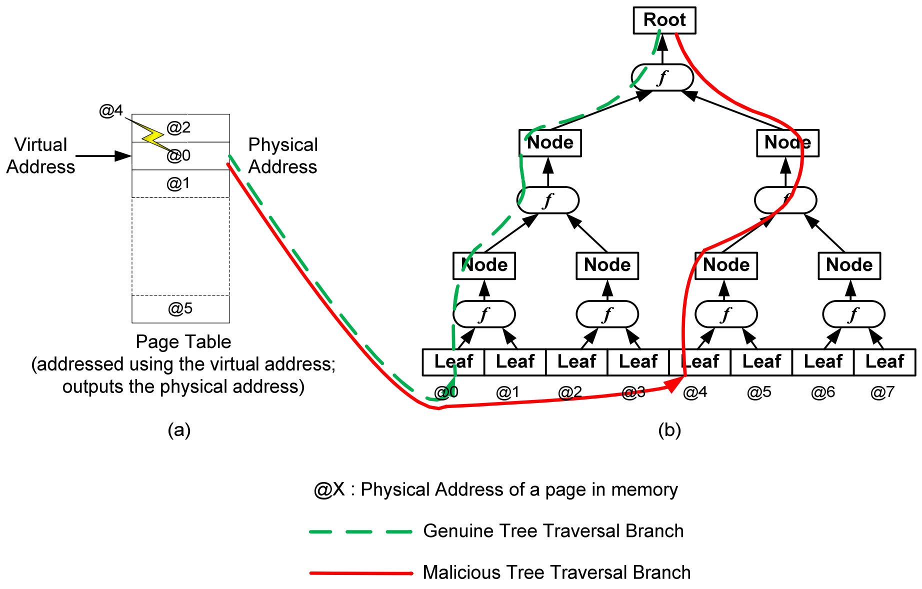
@4
@2
@0
@1
@5
Virtual Address
Physical Address
Page Table
(addressed using the virtual address; outputs the physical address)
(a)
Root
f
Node
Node
Leaf
@0
@1
@2
@3
@4
@5
@6
@7
(b)
@X : Physical Address of a page in memory
Genuine Tree Traversal Branch
Malicious Tree Traversal Branch

Fig. 7. The Branch Splicing Attack. (a) The OS tampers with the Page Table and changes the second entry from physical address @0 to @4. (b) data at @4 are verified instead of data at @0; however, since the physical address – which is corrupted – is used to retrieved the branch nodes required to verify the data read, the attack is undetected and data at @4 are considered genuine.

Impractical VAS-Tree Overheads. The extra security afforded by the VAS tree over the PAS tree comes at the cost of very large memory capacity and initialization overheads. Application code and data segments are usually laid out very far apart from one another in order to avoid having dynamically growing segments (e.g., the heap and stack) overwrite other application segments. The VAS tree must thus span a very large fraction of the virtual address space in order to cover both the lowest and highest virtual addresses that may be accessed by the application during its execution. The span of the tree is then several orders of magnitude larger than the cumulative sum of all code and data segments that require protection. In the case of a VAS tree protecting a 64-bit address space, the tree span can be so enormous as to make VAS tree impractical, i.e., VAS tree is not scalable. Indeed, it not only requires allocating physical page frames for the 2^{64} bytes of leaf nodes that are defined during initialization, but also requires allocating memory for the non-leaf tree nodes, which represent 20% to 100% of the leaf space size depending on the memory overhead of the underlying integrity tree [7, 8, 9]. The CPU time required to initialize such a tree is clearly unacceptable in practice.

The Reduced Address Space. To overcome these problems, [21] introduces a new processor hardware unit which builds and maintains an integrity tree over a *Reduced Address Space* (RAS). At any point in time, the RAS contains only those pages needed for the application's execution; it grows dynamically as this application memory footprint increases. Virtual pages are mapped into the RAS using an index that follows the page's translation. The new hardware builds an integrity tree over the application's RAS (a RAS tree), carries out the integrity verification and tree update procedures and expands the tree as the underlying RAS grows. Because the RAS contains the application's memory footprint in a contiguous address space segment, the RAS tree does not suffer from the overheads of the VAS tree, built over a sparse address space. The value of tree nodes along a block's verification path is tied to the block's virtual address so corruption of the RAS index or the physical address translation by the OS is detected.

With both the VAS and RAS trees, a full-blown integrity tree must be built and maintained for every protected application, as opposed to a single PAS tree for all software. Therefore, a scheme must be designed to manage tree roots for a Merkle Tree scheme, or the on-chip roots and secret keys for PAT and TEC-Tree. [2] presents an implementation where the processor hardware manages the roots of several Merkle Trees stored in dedicated processor registers. To provide some scalability despite a fixed amount of hardware resources, the authors suggest implementing spill and fill mechanisms between these registers and a memory region protected by a master integrity tree. Minimizing the hardware resources required to operate multiple integrity trees is an open research area.

6 Memory Authentication without a Tree Structure

Several engines that are not based on tree schemes have been proposed in the literature to ensure memory authentication. In most cases however, these techniques intentionally decrease the security of the computing platforms to cope with the overhead they generate. This section surveys these techniques.

Lhash. The researchers who proposed the cached tree principle also designed a memory authentication technique not based on a tree and called Log hash (Lhash [23]). Lhash has been designed for applications requiring integrity checking after a sequence of memory operations (as opposed to checking off-chip operations on every memory access as in tree schemes). Lhash relies on an incremental multiset hash function (called Mset-Add-Hash) described by the authors in [24]). This family of hash takes as an input a message of an arbitrary size and outputs a fixed size hash as a regular hash function, but the ordering of the chunks constituting the message is not important. The Lhash scheme has been named after the fact that a write and a read log of the memory locations to be checked are maintained at runtime using an incremental multiset hash function and stored on-chip. The write and read logs are respectively called ReadHash and WriteHash.

The Lhash scheme works as follows for a given sequence of operations. At initialization WriteHash is computed over the memory chunks belonging to the memory region that needs to be authenticated; WriteHash is then updated at runtime when an off-chip write is performed or when a dirty cache block is evicted from cache. This way, WriteHash reflects the off-chip memory state (and content) at anytime. ReadHash is computed the first time a chunk is brought in cache and updated on each subsequent off-chip read operations. When checking the integrity of the sequence of operations, all the blocks belonging to the memory region to authenticate and not present in cache are read to make even the number of chunks added to the read log and those added to the write log. If ReadHash happens to be different from WriteHash, it means that the memory has been tampered with at runtime. Re-ordering attacks are prevented by including a nonce in each hash computation. Authors performed a comparison of Lhash with a cached hash tree (4-ary). They showed that overhead can be decreased from 33% to 6.25% in term of off-chip memory and from 20-30% to 5-15% at runtime. However, the scheme does not suit threat models such as the ones defined in this paper since an adversary is able to perform an active attack before the integrity checking takes place.

PE-ICE. A Parallelized Encryption and Integrity Checking Engine, PE-ICE, based on the block-level AREA technique was proposed in [13] and [14] to encrypt and authenticate off-chip memory. However, to avoid re-encryption of the whole memory when the nonce reaches its limit (e.g., a counter that rolls over), the authors propose to replace it with the chunk address concatenated with a random number. For each memory block processed by PE-ICE, a copy of the random value enrolled is kept on-chip to make it tamper-resistant and secret. The drawback of using a random number is that a replay attack can be performed with a probability p of success inversely proportional to the bit length r of the random number ($p = 1/2^r$). Since a small random number is advised [14] to keep the on-chip memory overhead reasonable, the scheme is likely to be insecure with respect to replay attacks.[6]

7 Data Authentication in Symmetric Multi-Processors (SMP)

High-end computing platforms are more and more based on multi-processor technology. In this section, we highlight the new security challenges related to runtime data authentication that arise on such shared memory platforms.

We have defined memory authentication as "verifying that data read from memory by the processor at a given address is the data it last wrote at this address". However, in a multiprocessor system with shared memory, the definition of data authentication cannot be restricted to that of memory authentication: additional security issues must be considered. First, every processor core of a SMP platform may legitimately update data in memory; thus each processor must be able to differentiate a legitimate write

[6] Note that in [14], the author proposes to build a tree – called PRV-Tree, for PE-ICE protected Random Value Tree – similar to TEC-Tree except that it uses random numbers instead of nonces. The purpose of PRV-Tree is to decrease the probability of an adversary succeeding with a replay attack by increasing the length of the random number, while limiting the on-chip memory overhead to the storage of a single random number (the root of PRV-Tree).

operation done by a core of the system from malicious data corruption. Moreover, in addition to the traditional memory authentication, the data authentication issue in a SMP platform must also consider bus transaction authentication on cache-to-cache transfers required in cache coherency protocols (Figure 8). [18] notes that to take into consideration bus transaction authentication, an additional active attack must be considered: message dropping. In SMP platforms, when a processor sends a data to another core, it broadcasts the same data to all other SMP's cores to maintain cache coherency. Thus, message dropping takes place upon those broadcasting cache-to-cache communications and consists in blocking a message destined to one of the processor cores (In Figure 8, CPU2 is temporarily disconnected from the bus to perform a message dropping).

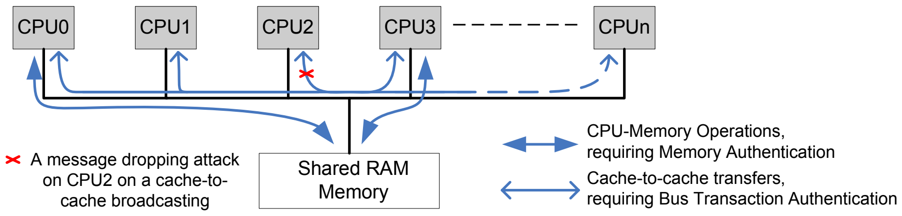
CPU0
CPU1
CPU2
CPU3
CPUn
A message dropping attack on CPU2 on a cache-to-cache broadcasting
Shared RAM Memory
CPU-Memory Operations, requiring Memory Authentication
Cache-to-cache transfers, requiring Bus Transaction Authentication

Fig. 8. Data Authentication Requirements in a Symmetric Multi-Processors (SMP) Platform

[18] and [19] propose solutions for data authentication in SMP platforms. However, as highlighted in [18], [19] is sensitive to message dropping. Moreover, [18] and [19] focus on SMP systems with a small number of processors and do not evaluate the scalability of their solutions. Thus, data authentication at runtime in SMP platforms is still an open research topic.

Active research efforts [26, 27, 28] are also being carried out in the field of data authentication in multiprocessor systems with Distributed Shared Memory (DSM). Data authentication issues in these systems are similar to those mentioned for SMP platforms, except that designers of security solutions have to deal with additional difficulties due to the intrinsic characteristics of the DSM processor. First, a typical DSM system does not include a shared bus that could help synchronization of metadata (e.g., counter values) between the processors participating in a given memory authentication scheme. Also, as mention in [28], the interconnection network that enables processor-to-memory communications is usually exposed at the back of server racks, providing an adversary with an easier way to physically connect to the targeted system compared to a motherboard in an SMP platform

8 Conclusion

In this paper we described the hardware mechanisms to provide memory authentication, namely integrity trees. We presented a generic integrity tree model, the intrinsic properties of each existing integrity tree (Merkle Tree, PAT and TEC-Tree) and the architectural features proposed in the literature to efficiently implement those trees in computing platforms. We also discussed the impact of operating system compromise on hardware integrity verification engine and presented an existing solution for secure and efficient application memory authentication despite an untrusted OS. Finally, we showed the additional security issues to consider for data authentication at runtime in symmetric multi-processors platforms and how they differ from memory authentication in uniprocessor systems.

References

1. Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M.: Architectural Support for Copy and Tamper Resistant Software. In: Int'l. Conf. on Architectural Support for Programming Languages and OS (ASPLOS-IX), pp. 168–177 (2000)
2. Suh, G.E., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In: Proc. of the 17th Int'l. Conf. on Supercomputing (ICS) (2003)
3. Lee, R.B., Kwan, P.C.S., McGregor, J.P., Dwoskin, J., Wang, Z.: Architecture for Protecting Critical Secrets in Microprocessors. In: Int'l. Symp. on Computer Architecture (ISCA-32), pp. 2–13 (June 2005)
4. IBM Extends Enhanced Data Security to Consumer Electronics Products, IBM (April 2006), `http://www-03.ibm.com/press/us/en/pressrelease/19527.wss`
5. Kuhn, M.G.: Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP. IEEE Trans. Comput. 47, 1153–1157 (1998)
6. Merkle, R.C.: Protocols for Public Key Cryptography. In: IEEE Symp. on Security and Privacy, pp. 122–134 (1980)
7. Hall, W.E., Jutla, C.S.: Parallelizable authentication trees. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 95–109. Springer, Heidelberg (2006)
8. Elbaz, R., Champagne, D., Lee, R.B., Torres, L., Sassatelli, G., Guillemin, P.: EC-Tree: A Low Cost and Parallelizable Tree for Efficient Defense against Memory Replay Attacks. In: Cryptographic Hardware and embedded systems (CHES), pp. 289–302 (2007)
9. Gassend, B., Suh, G.E., Clarke, D., van Dijk, M., Devadas, S.: Caches and Merkle Trees for Efficient Memory Integrity Verification. In: Proceedings of Ninth International Symposium on High Performance Computer Architecture (February 2003)
10. Suh, G.E.: AEGIS: A Single-Chip Secure Processor, PhD thesis, Massachusetts Institute of Technology (September 2005)
11. Yan, C., Rogers, B., Englender, D., Solihin, Y., Prvulovic, M.: Improving Cost, Performance, and Security of Memory Encryption and Authentication. In: Proc. of the International Symposium on Computer Architecture (2006)
12. Rogers, B., Chhabra, S., Solihin, Y., Prvulovic, M.: Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS– and Performance–Friendly. In: Proc. of the 40th IEEE/ACM Symposium on Microarchitecture (MICRO) (December 2007)
13. Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M., Martinez, A.: A Parallelized Way to Provide Data Encryption and Integrity Checking on a Processor-Memory Bus. In: Proceedings of the 43rd Design Automation Conference DAC (July 2006)
14. Elbaz, R.: Hardware Mechanisms for Secured Processor Memory Transactions in Embedded Systems, PhD Thesis, University of Montpellier (December 2006)
15. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: Proc. 32nd IEEE Symposium on Foundations of Computer Science, pp. 90–99 (1991)

16. National Institute of Science and Technology (NIST), FIPS PUB 180-2: Secure Hash Standard (August 2002)
17. NIST Special Publication SP800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC (November 2007)
18. Zhang, Y., Gao, L., Yang, J., Zhang, X., Gupta, R.: SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors. In: Proc. of the 11th International Symposium on High-Performance Computer Architecture (2005)
19. Shi, W., Lee, H.-H., Ghosh, M., Lu, C.: Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems. In: Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques (2004)
20. http://news.bbc.co.uk/2/hi/uk_news/7581540.stm
21. Champagne, D., Elbaz, R., Lee, R.B.: The Reduced Address Space (RAS) for Application Memory Authentication. In: Wu, T.-C., Lei, C.-L., Rijmen, V., Lee, D.-T. (eds.) ISC 2008. LNCS, vol. 5222, pp. 47–63. Springer, Heidelberg (2008)
22. Fruhwirth, C.: New Methods in Hard Disk Encryption, Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology (2005)
23. Suh, G.E., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: Efficient Memory Integrity Verification and Encryption for Secure Processors. In: Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO 36), San Diego, CA, pp. 339–350 (December 2003)
24. Clarke, D., Devadas, S., van Dijk, M., Gassend, B., Suh, G.E.: Incremental multiset hash functions and their application to memory integrity checking. In: Laih, C.-S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 188–207. Springer, Heidelberg (2003)
25. Duc, G., Keryell, R.: CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection. In: Jesshope, C., Egan, C. (eds.) ACSAC 2006. LNCS, vol. 4186, pp. 483–492. Springer, Heidelberg (2006)
26. Rogers, B., Prvulovic, M., Solihin, Y.: Effective Data Protection for Distributed Shared Memory Multiprocessors. In: Proc. of International Conference of Parallel Architecture and Compilation Techniques (PACT) (September 2006)
27. Lee, M., Ahn, M., Kim, E.J.: I2SEMS: Interconnects-Independent Security Enhanced Shared Memory Multiprocessor Systems. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 94–103. Springer, Heidelberg (2007)
28. Rogers, B., Yan, C., Chhabra, S., Prvulovic, M., Solihin, Y.: Single-Level Integrity and Confidentiality Protection for Distributed Shared Memory Multiprocessors. In: Proc. of the 14th International Symposium on High Performance Computer Architecture (HPCA) (2008)