Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 66 (2006) 1116–1128
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

M-TREE: A high efficiency security architecture for protecting integrity and privacy of software

Chenghuai Lu[a], Tao Zhang[a], Weidong Shi[a], Hsien-Hsin S. Lee[b,*]

[a]*College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA*
[b]*School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA*

Received 17 December 2005; received in revised form 31 March 2006; accepted 10 April 2006
Available online 30 June 2006

Abstract

Secure processor architectures enable new sets of applications such as commercial grid computing, software copy protection and secure mobile agents by providing secure computing environments that are immune to both physical and software attacks. Despite a number of secure processor designs have been proposed, they typically made trade-offs between security and efficiency. This article proposes a new secure processor architecture called *M-TREE*, which offers a significant performance gain while without compromising security. The M-TREE architecture uses a novel hierarchical Message Authentication Code Tree (MACTree) for protecting applications' integrity at a minimal performance overhead. M-TREE also introduces a new one-time-pad class encryption mechanism that accelerates security computation over the existing block cipher-based schemes with high security guarantee. Based on the results of our performance simulation, the performance overhead of the M-TREE integrity check mechanism is as small as 14% in the worst case, a substantial improvement over the 60% slowdown reported by previously proposed techniques. Meanwhile, the overhead of M-TREE encryption scheme is approximately 30%, compared to 50% of using block cipher encryption. In overall, our M-TREE architecture can provide a tamper-resistant and tamper-evident computing environment with low-performance impact, thereby offering a transparent and practical security computing platform.

Keywords: Security and privacy; Processor architecture; Data integrity and encryption; Trusted computing

0. Introduction

Recently, there is a growing interest in integrating security features into processor architecture [6,13,17] for providing digital copyright protection, software confidentiality, ability for anti-reverse engineering, or security guarantee for running trusted applications on remote computing devices, to name a few. These security architectures rely on new tamper-resistant processors and certain cryptographic hardware for supporting a tamper-resistant and tamper-evident (TE) computing environment. The security implication is that data inside the processor die are protected while every data bit outside the die, particularly when stored in memory or transferred over the external buses, are assumed unsafe and subject to attacks. An effective security processor system should guarantee applications robust protection and immunity so that privacy and secrets of computational information will not be compromised by either software attacks or hardware-based physical tampering.

Despite a few security architectures were proposed [13,17,24], severe performance degradation still remains a major issue to be resolved in order to make them practical. The challenge is that when running applications, the traffic between the processor and off-chip memory can be very frequent. When a data item crosses the protection boundary, it needs to be encrypted/decrypted and checking of its integrity must be performed. In many cases, the performance overhead due to cryptographic computation increases the critical path timing, leading to substantial performance degradation. Besides the performance impact, some proposed schemes also incur a significant capacity increase in system memory as well.

In practice, the performance degradation could be more severe for memory-bound applications where the demand for

* Corresponding author.
E-mail address: leehs@gatech.edu (H-H.S. Lee).

0743-7315/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2006.04.011

memory is high. There were other cost-effective cryptographic schemes proposed [24,18] with performance in mind as the design objective, however, these techniques typically achieve performance improvement at the risk of weakening security. Therefore, it remains crucial to explore new cryptographic schemes for performance improvement while maintaining a high security level at the same time.

In this article, we present a new secure processor architecture, M-TREE that offers a strong security protection with a minimal performance impact. The M-TREE architecture employs a hierarchical message authentication code Tree (MACTree) for guaranteeing the integrity of applications. The MACTree scheme uses a 32-bit message authentication code (MAC) value for each tree node. Our analysis also shows that the MACTree scheme is as strong as the prior proposed schemes (e.g. CHTree in [6]) with a substantially lower-performance impact. The M-TREE encryption scheme also employs a one-time-pad (OTP) cipher [1,15] for protecting the confidentiality of applications, similar to what were proposed independently by [24,18], while our scheme addresses the security flaws found in these proposals. The M-TREE integrity protection scheme and the encryption scheme can be applied independently to provide a TE or a private computing environment. On the other hand, they can be combined to provide a complete tamper-resistant and private environment.

The rest of this article is organized as follows. Section 1 discusses the related work of secure processor architectures and our motivation. In Sections 2 and 3, we present our MACTree-based integrity checking scheme and the encryption scheme and propose the M-TREE architecture. The security and performance of the M-TREE architecture are evaluated against other existing techniques in Section 4. Finally, Section 5 concludes this work.

1. Related work and motivation

To prevent adversaries or unauthorized parties from compromising trusted applications is becoming a major challenge not only to software developers but also to processor architects and system designers. To combat these malicious exploits, alliance such as Trusted Computing Group (TCG) [20] was formed across a variety of industry segments to tackle the information security issues. An interesting trend is to define and incorporate security enhancements into the processor hardware directly such as Intel's LaGrande Technology [9]. A few other thrusts in academia also attempt the issues in a similar approach.

XOM [13] pioneered the design of a secure processor architecture to protect trusted applications from physical attacks. The security of the XOM architecture is achieved by a tamper-resistant processor design. Data within the processor die are assumed secure and stored in plaintext while those data stored in off-chip memory and peripherals are all protected via encryption as they are subject to software and physical attacks. Only when the data and instructions are brought into the tamper-resistant processor will they be decrypted and stored on-chip in plaintext. When executing applications, the architectures ensure that sensitive data and instructions of an application will not be revealed by untrusted parties at any given time. As the performance overhead of encryption/decryption is critical, XOM accelerates the process with a dedicated crypto-hardware. Suh et al. improved XOM's approach with the *AEGIS* secure processor architecture design [17] in which both privacy and integrity of applications are protected. They also described the implementation of a secure environment with a block cipher encryption and a CHTree integrity checking scheme [6].

The CHTree scheme in AEGIS was designed for protecting the integrity of an application by constructing a m-ary hash tree where m is the number of child nodes per parent node has and is equal to the size of the cache line divided by the size of hash values. The AEGIS settings use a 128-bit hash value and 512-bit, 1024-bit cache lines, i.e. m is 4 and 8, respectively. The integrity check of a cache line using the hash tree costs $\log_m(L)$ hashing computations, assuming there are L data cache lines in the application. The overhead can be large when m is small. Their simulation results showed that the CHTree scheme slows down the execution by 20% for most of the benchmark program with a worst case of 50%. In addition, the CHTree scheme consumes a large memory space for storing the hash tree. The memory overheads of the CHTree scheme are 33% and 14% for a 512-bit and a 1024-bit cache line, respectively. The block cipher encryption scheme is used for both AEGIS and XOM for protecting confidentiality of applications. When a cache line is brought in from memory, the line must be decrypted prior to use. The encryption increases the memory latency, thus resulting in a considerable performance loss. The original block cipher encryption scheme has an up to 25% performance degradation for confidentiality protection, with a 1 MB cache and a 512-bit cache line used in the study.

Several new cryptographic designs have been proposed since to improve the performance of the CHTree scheme and the block cipher encryption. For example, the LHash integrity check and a few OTP-class encryption schemes. Unfortunately, even though these techniques could improve performance with security enhancement, yet they achieve the goal at the cost of potentially reducing the level of security.

The Log Hash integrity checking scheme (LHash) [18] improves performance by logging memory operations and checking integrity only when a large number of memory operations are accumulated. The drawback lies in the delay of integrity check. Under certain circumstances, it is desirable to check the integrity on a per-instruction basis for detecting attacks such as when an adversary injects their instructions into a cache line. If the integrity check is delayed, the damage might be difficult to recover. As shown in [18], the LHash scheme outperforms the CHTree scheme only when the integrity check is performed for every 10^6 memory accesses or more. In other words, the LHash scheme leaves a long vulnerable instruction window for adversaries without any integrity check.

In [18], the performance of confidentiality protection was improved by replacing the block cipher encryption with an OTP-class encryption. The OTP-class encryption inserts a timestamp at the beginning of each cache line. When the cache line is read from memory, the processor will bring in its associated timestamp first. Once the timestamp arrives, the processor starts to compute the OTP, an encrypted value of the timestamp, the cache line virtual address and padding using AES. The computation can be performed in parallel with the fetching of the remaining cache line chunks. When all cache line chunks arrive and the OTP is computed, decryption will be performed by XORing the OTP and the encrypted cache line just fetched. The OTP approach improves the performance by hiding partial decryption delay behind the memory access. Yang et al. proposed a similar scheme in [24] for performance improvement for protecting confidentiality. Moreover, both proposed OTP schemes contain some flaws. More details are to be discussed in Section 2.2.2.

2. The M-TREE integrity protection and encryption scheme

Assume that an application under protection is divided into data blocks of equal size, each equivalent to a 256-bit cache line. The protected application owns a set of secret tokens that contains the secret key (or *Sec Key*) and other crucial information such as an initial counter value ($ICnt$), encrypted using the processor's *public key*. When a program starts execution in a secure environment, it first passes the secret tokens to the processor for decrypting the application's secret key which will then be used to protect the privacy and the integrity of the application for data outside the protection boundary.

2.1. M-TREE integrity protection scheme—MACTree

For guaranteeing a TE computing environment, any unauthorized modification attempts to applications running on the system must be detected via a robust integrity checking design. In addition to its effectiveness, the design goal of such an integrity protection scheme is high performance without compromising security. In other words, the hardware support for a TE computing environment needs to be transparent to the users. Toward this goal, we propose a novel scheme called *MACTree integrity protection*. We describe the integrity tree construction and analyze its security and efficiency in the following sections.

2.1.1. MACTree construction

Our proposed integrity checking scheme, *MACTree*, uses 32-bit MAC nodes to construct an m-ary MAC tree. Note that a conventional MAC requires at least 128-bit to ensure a sufficient security level. However, as shown in prior study discussed in Section 1, using 128-bit MAC nodes suffers a substantial performance loss due to a large number of hash computations and storage overhead. Therefore, more deliberate support must be developed in the MACTree construction to achieve desired security.

A MACTree is constructed by the following steps:

- A 32-bit MAC value is generated for each cache line. First of all, an initial 256-bit MAC value is generated using the SHA-256 hash function [5] by concatenating the cache line data, its virtual address, and the secret key of the application as inputs. A new 32-bit MAC is then computed by XORing the eight evenly divided chunks.

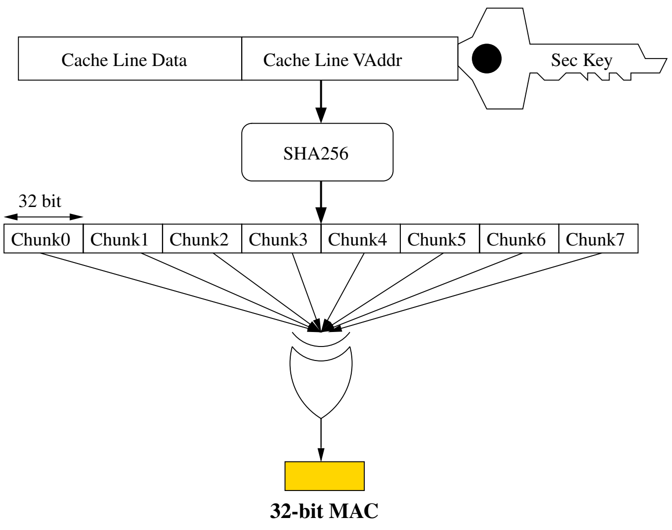

Fig. 1. The 32-bit MAC generation for a cache line.

- All 32-bit MACs form one layer of nodes in the MACTree and are stored linearly as shown in the left-hand side of Fig. 2. For this layer, a new MAC line is made by concatenating seven consecutive MACs together and padded with a 32-bit *random initial value* (RIV) which is generated by a random number generator using thermal noise in the core [10].
- Similar to the method described in Fig. 1, a new 32-bit MAC value for the next level in the MACTree is computed by concatenating the new RIV/MAC line and the secret key of the application as the inputs to the SHA-256 function.
- Repeat the last two steps until a root MAC is generated.

Whenever a MAC line is to be cast out of protection boundary, the RIV/MAC line is encrypted by the AES block cipher as illustrated in the dashed box of Fig. 2. It needs to mention that each MAC line in the MACTree is encrypted by a different key constructed using the following method. The key is computed by XORing the security key (or $Sec\ Key$) with a 128-bit value (*NodeIndex*, *NodeIndex* + 1, *NodeIndex* + 2, *NodeIndex* + 3), where *NodeIndex* is the 4-byte index of the MAC line in the MACTree. The index of the MAC line at ith level and jth position starting from left is $(7^i - 1)/6 + j$. It is easy to verify that the method allows us to generate different keys for encrypting different MAC lines in the MACTree.

Note that the *root MAC* is always kept inside the processor once the program enters the TE environment to avoid any potential tampering.

Our proposed MACTree scheme differs from the CHTree scheme in three aspects—first, 32-bit MAC values are used instead of 128-bit hash values; secondly, the MACtree is encrypted using the secret key ($Sec\ Key$) of a given application; thirdly, each MAC line is computed from the 32-bit MACs of its seven children lines with a RIV value generated randomly by circuits.

In MACTree integrity protection scheme, we achieve the performance advantage by employing new schemes that is uncommon to existing cryptographic scheme designs. The biggest difference of our design and common cryptographic designs is the

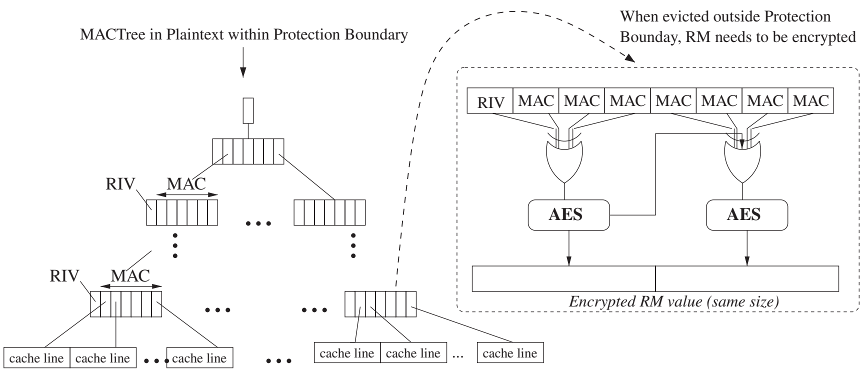
MACTree in Plaintext within Protection Boundary
When evicted outside Protection Bounday, RM needs to be encrypted
RIV
MAC
cache line
RIV MAC MAC MAC MAC MAC MAC MAC
AES
AES
Encrypted RM value (same size)

Fig. 2. Structure of the MACTree in plaintext within protection boundary. Note that a hash value needs to be encrypted when being evicted out of the protection domain.

use of 32-bit MAC values for cache lines obtained by XORing 32-bit chunks in 256-bit MAC values. This change, although very uncommon, can still ensure high guarantee of security in our system. And the reasons are explained below.

The most common attack to hash function is to find collisions [8], in other words, to find two distinct pieces of data that have the same hash value. If an adversary can find a new data that has the same hash as the data from a trusted source, he can use the new data to replace the data from a trusted source without his modification being detected. Assume an n-bit hash is used, an adversary can find a new data that has the same hash as a given data with 2^n computations, and an adversary can find two distinct pieces of data with the same hash value with $2^{n/2}$ computations (birthday attack) [8]. The computations of hash values can be performed at any computer. An adversary can even construct special hardware to expedite the computations [8]. Usually, the bit length of hash is at least 128. Therefore, an adversary needs about 2^{64} computations to find collisions which is computationally infeasible. But, if the bit length of a hash is reduced to 32, an adversary would only need about 2^{16} computations to find collisions which can be done very easily. However, the attack just described cannot be applied to our scheme since we use MACs instead of hashes. The key difference between MACs and hashes lies in that MACs are computed using a secret key which, in our case, is only known to the trusted CPU core. Therefore, an adversary cannot use untrusted computers or build specialized hardware to attack the MACTree scheme by computing MACs of cache lines with random content.

Although using MACs can greatly limit the capability of an adversary in launching collision attacks, itself is not sufficient to prevent all the possible attacks. Adversaries can possibly attack the MACTree scheme by running the trusted application they are attacking on the trusted CPU core. The adversaries must run the trusted application they are attacking on the trusted CPU core because this is the only way for them to generate MACs as they needs. Then, they can manipulate cache lines in the application's memory and observe the changes of MACTree nodes to find out collisions. Since we are using 32-bit MACs in our scheme, there is a high probability of collision, or finding two cache lines of data with the same MAC value. If an adversary collects 2^{16} distinct cache lines of data and the MAC lines of MACTree were not encrypted, the adversary can compare the cache lines of data and their MACs. He will likely find out two distinct cache lines of data with the same MAC value and attack the integrity checking scheme by swapping two cache lines. as shown in Fig. 3. Our MACTree scheme prevents this attack by encrypting MAC lines in the MACTree. Now, although collision does exist in cache lines of data, an adversary will not be able to figure it out since he will not be able to look into corresponding MACs and compare the values which are encrypted. For example, assume the two cache lines of data collected by the adversary are distinct and have the same MAC value. If the corresponding MAC lines in the MACTree are not encrypted, an adversary will find out the collision by comparing the cache lines and the MAC lines of the MACTree. If MAC lines of the MACTree nodes are encrypted, then the adversary will not be able to figure out the collision because he cannot compare the encrypted values of MAC lines as shown in Fig. 3.

The only possibility for an adversary to find out a collision is when they see two cache lines of data are distinct and the encrypted MAC lines are the same. Fig. 4 illustrates such example. This only happens when RIVs of the two MAC lines are the same and the 32-bit MACs for the corresponding seven child cache lines are the same. In order to obtain such collision, an adversary would have to collect about $2^{16} \times 2^{16\times7}$ distinct cache lines of data, assuming the cache lines of data are completely random. In Fig. 4, we use time $T^{(0)}$ and $T^{(1)}$ to represent the data of a MAC line at different times. In our analysis, we consider data at a MAC line with fixed index in the MACTree because MAC lines with different index are encrypted by different keys. So the encrypted data of MAC lines with different indices cannot be used to correlate each other.

We consider a rare case. We assume an adversary wants to replace a cache line of trusted data with a new data they fabricate. We assume that the adversary manages to keep the other six cache lines unmodified during the execution and only modify the data in the cache line. We assume that, after each modification, the adversary can trick the trusted CPU core into accepting the modification and recomputing the MACTree. The adversary then tricks the CPU into evicting the recomputed MAC lines out to the memory. These assumptions are extremely unlikely to happen in real world. In this case, the adversary would need

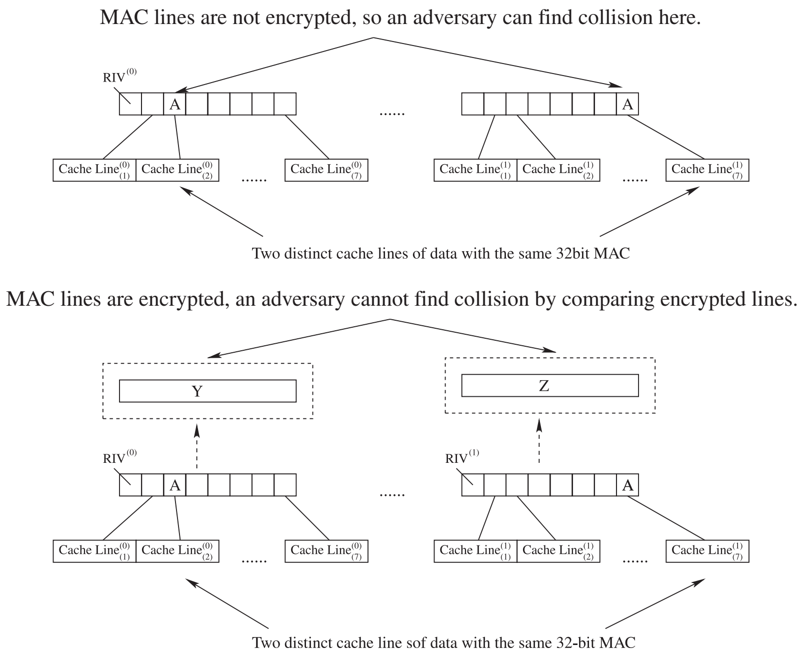
MAC lines are not encrypted, so an adversary can find collision here.
RIV(0)
A
A
Cache Line(0)(1)
Cache Line(0)(2)
Cache Line(0)(7)
Cache Line(1)(1)
Cache Line(1)(2)
Cache Line(1)(7)
Two distinct cache lines of data with the same 32bit MAC
MAC lines are encrypted, an adversary cannot find collision by comparing encrypted lines.
Y
Z
RIV(0)
RIV(1)
Two distinct cache line sof data with the same 32-bit MAC

Fig. 3. Attack the MACTree scheme if the nodes are not encrypted.

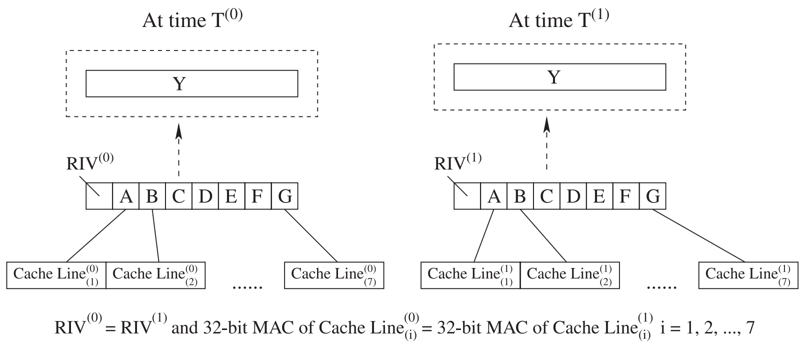

Fig. 4. Find two same encrypted MAC lines.

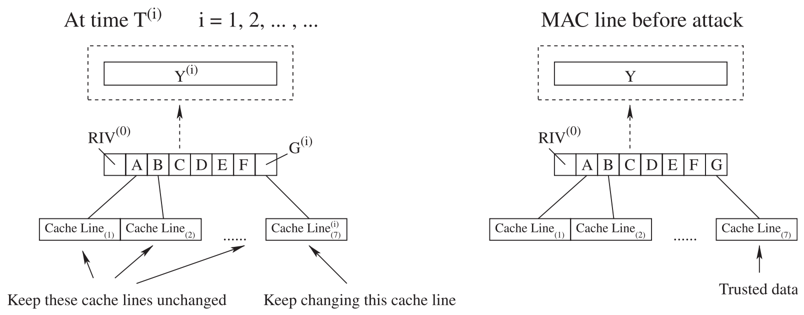

Fig. 5. A rare attack.

about 2^{64} computations, each of which includes cache line modification, cache line and MAC line read, MACTree recomputation and MAC line eviction, in order to find a fabricated data that has the same MAC value as the original data. Note that, although an adversary might be able to keep cache lines of data unmodified, he/she is unable to keep the RIV unchanged for each recomputation of MACTree. If an RIV were not associated with a MAC line, the adversary would need about 2^{32} computations, which they may possibly achieve. Fig. 5 illustrates the concept of the attack. An adversary modifies *cache line*$_{(7)}$ at time $T^{(i)}$, $i = 1, 2, \ldots$, and compute the encrypted MAC line $Y^{(i)}$ with Y. If he finds that $Y^{(k)} = Y$ at the time T^k, then he can replace the cache line *cache line*$_{(7)}$ with his fabricated data of *cache line*$^{(k)}_{(7)}$. But, in order for this to succeed, there will need about 2^{64} computations which is infeasible.

An adversary can possibly launch brutal force attack. In other words, the adversary directly modifies the data of cache line, hoping that the new data has the same MAC value as the original one. But, the chance for this to succeed without being detected is 2^{-32}. The processor in the M-TREE architecture will stop the execution of this application once an integrity verification failed, prohibiting the adversary from further attempted attacks.

We now compare the security of our MACTree scheme with that of the CHTree scheme that uses 128-bit hash values. Let us assume that there are 2^{15} computers, each of which can compute 2^{25} hashes per second. Since the attack on the CHTree scheme can be done offline on computers other than the targeted secure processor, the security of the CHTree scheme comes from the brute-force effort of finding hash collisions using the available computing power. Based on the assumption, we can obtain 2^{60} hash values by running computers for 2^{20} s. If a protected application consists of 4 billion cache lines, the possibility that any of its cache lines can be replaced is about 2^{-36} (assume hashes are 128-bit long), which is pretty close to 2^{-32} of the MACTree. Based on the above arguments, we conclude that both schemes have a comparable security level.

Finally, it is important to note that, although we use SHA-256 for discussion in the MACTree scheme, any secure hash functions can be used by the MACTree scheme to provide the same functionality. It is shown, in the past few years, that the advances of attack techniques on hash functions can be very fast [21,22]. So, the MACTree scheme can always choose more secure hash functions than SHA-256, if available.

2.1.2. *Efficiency analysis*

Here we analyze the advantage of the MACTree over the CHTree.[1] First of all, a 256-bit cache line can hold 7 nodes in the MACTree versus 2 nodes in the CHTree. Hence, the height of the tree is reduced to $\log_7 L$ nodes from $\log_2 L$, or a 180% reduction of the number of nodes needing to be examined. Nonetheless, the MACTree incurs decryption overheads caused by encryption. Assume that the decryption delay is T_d and the memory access delay is T_m. The overhead caused by memory access and decryption for integrity check is $\log_7 L \times (T_d + T_m)$ in the MACTree against $\log_2 L \times T_m$ in the CHTree scheme. Let $T_d = \frac{T_m}{2}$, similar to the assumption used in other secure processor architectures, the MACTree has 47% less overhead than the CHTree. Even when $T_d = T_m$, the MACTree has 35% less overhead. The MACTree scheme can be further improved by caching nodes within the protected boundary. With the same cache size, the MACTree can hold 3.5 times more nodes.

2.2. *M-TREE encryption scheme*

To ensure a private (PR) computing environment, all the instructions and data of a protected application must be kept confidential from any unauthorized accesses. We propose the M-TREE encryption scheme for protecting the privacy of applications in a secure processor architecture. Before detailing our scheme, the following notations are defined:

- *SecKey*: Secret key of the protected application.
- *ICnt*: Initial counter value of the application.
- RIV_i: 32-bit random initial value associated with the ith cache line.
- PRV_i: Pseudo random value for encrypting the ith plaintext cache line or decrypting the ith ciphertext cache line with XOR.
- MAC_i: 32-bit Message authentication code associated with the ith cache line. The MACs are generated in the same way as described in Section 2.1.1.

2.2.1. *PRV generation*

To encrypt the ith cache line of an application, the PRV_i block is first computed by Eq. (1). The PRV_i is used as the key similar to a OTP for encrypting the cache line via simple XORing. The decryption is vice versa

$$PRV_i = H(ICnt + i, RIV_i, MAC_i, SecKey). \tag{1}$$

In Eq. (1), the function H can be either a hash algorithm (e.g. SHA-256) or a block cipher (e.g. AES). Two alternatives for computing PRV blocks are shown in Fig. 6 in which the RIV/MAC is a concatenated value of the RIV_i with the MAC_i. The initial counter *ICnt* is used and incremented for each cache line i to prevent dictionary attack as in the AES counter mode encryption [14]. To decipher the plaintext of a cache line from off-chip memory, the processor first reads its corresponding RIV/MAC value. Then it computes the PRV block while fetching the encrypted line from memory in parallel. Once the encrypted line arrives, the decryption is done by XORing the encrypted line with the PRV block. Integrity checking of the line can be performed in parallel with the instruction execution, thus has no major impact to the performance. To evict a dirty cache line back to memory, the processor generates a new RIV/MAC value for the line. A new PRV block will then be computed for encryption. Finally, the line is evicted and its RIV/MAC value also needs to be updated at its corresponding memory location.

2.2.2. *Security analysis for M-TREE encryption*

Notwithstanding a better performance, the OTP-like scheme is less secure than a block cipher, a reality never explicitly analyzed in prior literature. Hence, it is particularly important to make a detailed security analysis on OTP-like schemes as processor architects must be very careful when determining what OTP schemes are more appropriate.

The OTP schemes make use of an additional variable or *timestamp* (also referred to as *sequence number* by other literature) for encryption. For a given virtual address, the PRV block generated solely depends on the associated timestamp. Therefore, the number of possible PRV block patterns is equal to the number of possible values for the timestamp. For example, if the timestamp is only 8-bit, there are only 2^8 possible PRV block patterns. If the timestamp is 64-bit, there is 2^{64} PRV block patterns. In generic OTP encryption schemes, the possible patterns of PRV block for a cache line of data are as large

[1] As discussed in Section 1, the LHash scheme provides much weaker integrity protection than the MACTree. Therefore, it is inappropriate to compare performance between the MACTree scheme and the LHash scheme.

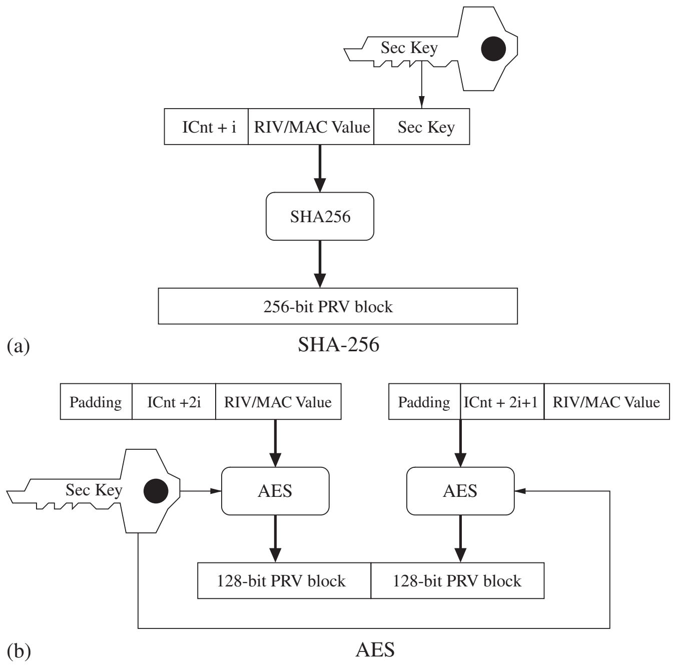

Fig. 6. Two options for PRV block generation.

as 2^{256}. The security of the OTP schemes depends on the size of the timestamp used as a larger timestamp offers exponentially more PRV patterns, making attacks more difficult from succeeding. Meanwhile, the size of the timestamp should not be infinitely big because it will lead to performance degradation. As a result, the size of the timestamp needs to be chosen meticulously for both performance and security consideration.

In [24], Yang et al. proposed to use a 16-bit timestamp, which is too small to provide sufficient security. If an adversary knows 1000 possible PRV patterns (one can obtain PRV patterns if some plaintext values at the virtual address are known), the adversary can decrypt almost $\frac{1}{65}$ of the cache line content, which is totally unacceptable. The scheme proposed by [18] uses a 32-bit timestamp. To prevent PRV blocks from being re-used, the timestamp is incremented, starting from 0, for each memory update. If the timestamp is exhausted, the whole program must be re-encrypted. However, this scheme has another potential vulnerability. In the scheme, when the processor dirty-writes back a cache line, it first increments the associated timestamp by one, then computes the OTP using the cache line's virtual address and the timestamp, finally the plaintext cache line is XORed with the OTP. The timestamps are not protected from modification, hence an adversary can attack the encryption scheme by manipulating the timestamps. Assume we have the following code whose data need to be kept confidential:

$$\textit{while}(1)\ \{\ldots, a = k; \ldots, \}.$$

The value a and k are stored in an encrypted cache line, however, since the adversary knows the PRV block of the cache line of a when the associated timestamp is 1. To obtain the secret value of k, an adversary can launch attack described as follows. When the execution reaches the location right before $a = k$, the adversary changes the timestamp of the cache line a to 0 and tricks the processor into loading the cache line. Therefore, the timestamp of the cache line is changed to 0 now. Right after the processor executes the instruction $a = k$, the adversary forces a dirty writeback. Then, the processor will encrypt the cache line a by XORing it with the PRV block generated using timestamp 1. As the adversary knows the PRV block with timestamp 1, he can reveal the encrypted value of k. Therefore, the OTP-based scheme must be used together with an integrity protection even for the sake of protecting program's confidentiality, which may incur additional overhead.

Moreover, both encryption schemes carry the vulnerability of weak integrity protection from OTP encryption. More specifically, if an adversary knows the plaintext, P, of an encrypted data chunk, C, he can very easily change it to another plaintext P' by replacing the original ciphertext C with $C' = P \bigoplus C \bigoplus P'$. Such attacks can be prevented by using an integrity check scheme along with the encryption scheme.

The design of the M-TREE encryption scheme is different from the previous schemes in two aspects. First, the RIV/MAC value used in M-TREE encryption is 64-bit which offers up to 2^{64} different PRV blocks for a given cache line address. Therefore, if an adversary knows n possible PRV blocks at that cache line address, the possibility that he can decrypt a given encrypted values at that cache line address is only $\frac{n}{2^{64}}$. Even for an n as large as one million, the possibility for the adversary to decrypt a cache line using the PRV blocks they have is still as small as 2^{-44}. Additionally, an RIV/MAC value consists of 32-bit MAC value which can be used to verify the integrity of the associated cache line. Because of this, an adversary is unable to modify a cache line or the associated RIV/MAC value as unauthorized modifications can be detected when the associated MACs of the cache lines are verified. Indeed, the M-TREE encryption scheme is the closest paradigm to the block cipher encryption among all OTP-based schemes. Note that a block cipher encryption will prevent an adversary from changing a cache line of data to whatever they favor. This feature, however, is only contained in the M-TREE encryption scheme among all other OTP-based schemes. Hence, the M-TREE encryption scheme provides much better security.

When using the M-TREE encryption scheme alone, people must be aware that it might be vulnerable under certain replay attacks, which are also applicable to block cipher encryption scheme. If an adversary replaces a cache line as well as its associated RIV/MAC value, the attack activity will not be detected by the M-TREE encryption scheme. In order to prevent replay attacks, the M-TREE encryption scheme must be used together with the MACTree.

3. The M-TREE architecture

The M-TREE security architecture is illustrated in Fig. 7 by integrating the M-TREE encryption mechanism with the MACTree integrity protection into a processor. Within the protection boundary, three new microarchitectural components are introduced including the *integrity verification unit* (*IVU*), the *RIV/MAC* and *MAC cache* (*RM/MAC cache*), and the encryption/decryption unit (EDU), for enabling the security support.

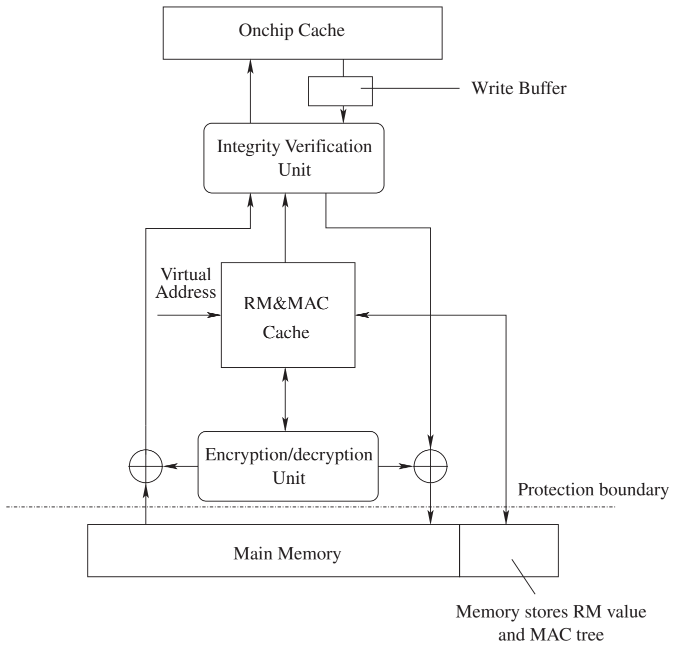
Onchip Cache
Write Buffer
Integrity Verification Unit
Virtual Address
RM&MAC Cache
Encryption/decryption Unit
Protection boundary
Main Memory
Memory stores RM value and MAC tree

Fig. 7. M-TREE architecture.

Additional memory space is consumed by storing the RM values and the MACTree. The IVU and EDU are both required for providing a TE and tamper-resistant system while the RM/MAC cache is primarily a performance feature. The RM portion accelerates data decryption by caching the RM values. The MAC portion keeps partial MACTree for speeding up integrity verification. Both caches operate in a similar manner as the victim cache does [11]. For cache lines that are being evicted from the protected domain, their RM and MAC values are deposited into the RM/MAC cache. During a cache miss, the processor first checks if the missed line's corresponding RM value hits the RM/MAC cache. Similar to a regular cache lookup, the RM/MAC caches are indexed by the virtual address of the missed cache line. If hit, the PRV block generation is processed simultaneously with the encrypted missed cache line fetching by using the RM value obtained from the RM/MAC cache. The overhead of the encryption scheme is equal to either the memory access delay or the decryption delay, whichever is bigger. If miss, the M-TREE will incur extra delay for fetching the RM value from memory. In this case, the MACTree encryption scheme can still outperform the block cipher as the analysis shows below. Under the M-TREE encryption scheme, the decryption ready time is

$$\begin{aligned}&max(RM\ value\ latency\ + AES\ latency,\\&\quad pipelining\ interval + data\ block\ latency) + XOR\ latency.\end{aligned}$$

That is smaller than the delay caused by the block cipher scheme which is

$$Data\ block\ latency + AES\ decryption\ latency.$$

The RM values are stored linearly at the end of the program starting at the virtual address RM_{Base}. Assume the cache block is 32 B, the RM value is 8 B. The RM value for the memory block starting at virtual address 0 is placed at RM_{Base}. The RM value for the next block starting at virtual address 32 is placed at $RM_{\text{Base}} + 8$, and so on. When a cache line is accessed, the virtual address of its RM value can be simply computed by

$$RMAddr = RM_{\text{Base}} + VAddr/4,$$

where *VAddr* is the current cache line's virtual address. MACTree nodes are laid out linearly in a similar way. The root of MACTree is placed at MAC_{Base}. The second level nodes follow, from the left most node to the right most one, and so on.

In addition to the secret key of the application, the other secret tokens of the application must also include the *ICnt*, RM_{Base} and the base addresses for each layer of the MACtree.

Speculative execution can be employed to further improve the performance for the M-TREE security processors. The results of unverified instructions can be speculatively consumed by dependent instructions without stalling the pipeline. However, all the unverified instructions and their dependent instructions must not be committed before the corresponding integrity checks are completed. The reorder buffer can be used to satisfy this need with a minor modification which adds the verification completion as a condition for retirement. In other words, in addition to branches, the machine states induced by instructions with pending integrity checks are also considered speculative.

4. Performance analysis

4.1. Memory overhead

Integrity checking schemes need additional memory space to store the integrity codes. In the MACTree scheme, the additional memory space needed is approximately $\frac{1}{(m-1)}$ of the data memory space with a *m*-ary balanced MACTree. As we use a 256-bit cache line and a 32-bit MAC value, the memory overhead is about 15%. In contrast, a 128-bit hash value used in the CHTree scheme will result in a 100% memory overhead. The M-TREE encryption scheme uses a 64-bit RM value for each cache line, leading to 25% memory overhead for encryption, larger than 12% in a direct block cipher that uses 32-bit random initial values, but can achieve a better performance.

4.2. Simulation framework

Our simulation work is based on SimpleScalar [2] running alpha binaries compiled with `-O3` option. Each benchmark is fast-forwarded according to SimPoint's suggestion [16] and then simulated for 100 million instructions in performance mode. During fast-forwarding, L1 and L2 caches were warmed up. Table 1 summaries the architectural and microarchitectural parameters. An 80 ns latency was used for both SHA-256 and AES-256 given they are custom designed. While both values are quite optimistic [4,12,19,23], an 80 ns AES decryption delay is more conservative than that used in [18] and an 80 ns hash computation delay is the same as used in [6]. The

Table 1
Processor model parameters

Parameters	Values	Parameters	Values
Frequency	1.0 GHz	Memory bus	200 MHz, 8 B wide
Fetch/decode width	8	Issue/Commit width	8
L1 I/D cache	DM, 8 KB, 32 B line	L1 latency	1 cycle
L2 cache	4 W, Unified, 32 B line,	L2 latency	
	256 KB, 0.5 MB, 1 MB, 2 MB	256 KB/512 KB/1 MB/2 MB	6/8/10/12 cycles
AES/SHA-256 latency	80 ns/80 ns	Memory latency	X-5-5-5 (cpu cycles)
			X depends on mem page status
MAC length	32 bits	RIV/MAC	64 bits
RM/MAC cache	4 W, 8 KB, 32 B line	RM/MAC cache latency	6 cycles

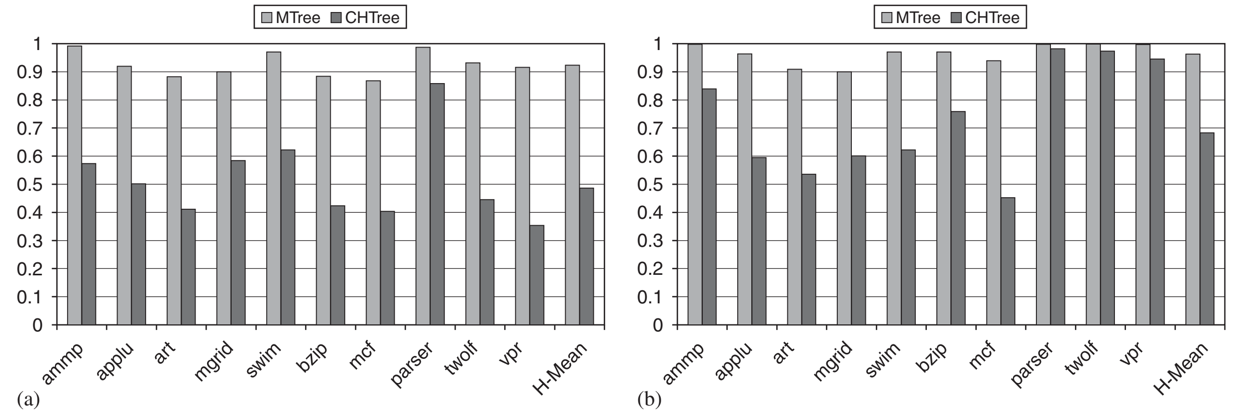

Fig. 8. Normalized IPC for MACTree and CHTree with 8 K MAC cache: (a) 256 KB L2 cache, (b) 2 MB L2 cache.

performance sensitivity of hash function delay will be further discussed in Section 4.3.3. We integrated a more accurate DRAM model [7] to improve the system memory modeling, in which bank conflicts, page miss, row miss are all modeled following the PC SDRAM specification. This memory model has a trailing-edge chunk latency of 5 core cycles (meaning a 1-to-5 frequency ratio to the processor) while the critical chunk latency is determined by various dynamic factors such as previous command, open page, same bank or not, etc. Both SPEC2000 INT and FP benchmark programs were used for our evaluation. We subset our simulations for those with high L2 misses.

4.3. MACTree integrity check evaluation

4.3.1. Performance comparison with the CHTree

Fig. 8 compares the performance of the MACTree and the CHTree for two different L2 cache sizes. The IPC numbers were normalized to the baseline.[2] The results clearly show the performance advantage of the MACTree scheme over the CHTree. The performance overhead over the baseline for the MACtree scheme is 8% on average and up to 14% in the worst case, while the CHTree scheme has 50% slowdown on average and as much as 60% overhead in the worst case with a 256 KB L2 cache. Even with a 2 MB L2, the performance degradation of the CHTree is reduced to 35% in the worst case and 11% on average, still less efficient than the MACTree scheme which shows an overhead of 5% on average and 10% in the worst case.

The performance advantage of the MACTree is twofold. As shown in Fig. 9(a), with an 8 KB MAC cache and a 256 KB L2 cache, the MACTree has fewer MAC cache accesses than the CHTree. It can be seen that the number of accesses to the MAC cache in the MACTree are reduced to less than half of those in the CHTree for all cases. On the other hand, since the same cache capacity will hold more nodes for the MACTree, the cache hit rate is also significantly higher as shown in Fig. 9(b).

We also simulate the case when an unified L2 cache is to be shared by the MAC values, a similar approach used in [6]. In this case, the MAC values or Hash values will compete cache space with data values, leading to more conflict misses and cache pollution. However, due to the similar reason aforementioned, the pollution caused by the MACTree is much smaller than the CHTree, which translates into a significant performance gain. The number of the L2 misses and its miss rate for both schemes are shown in Fig. 10. Fig. 11 shows the performance comparison for the MACTree scheme and the CHTree scheme using a shared MAC/L2 cache.

[2] Baseline has the same configuration with no security feature.

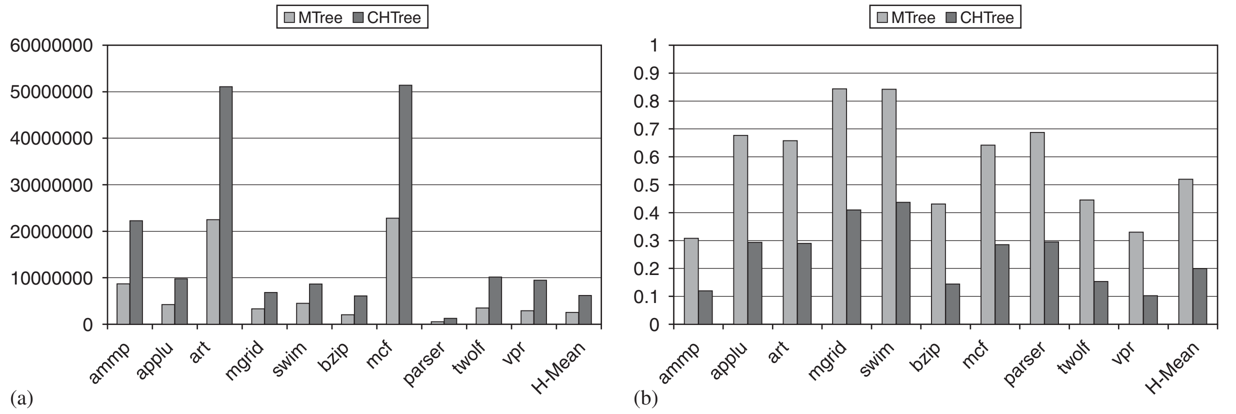

Fig. 9. MAC cache accesses/hit rates (256 KB L2): (a) 8 KB MAC cache, number of cache accesses, (b) 8 KB MAC cache, cache hit rates.

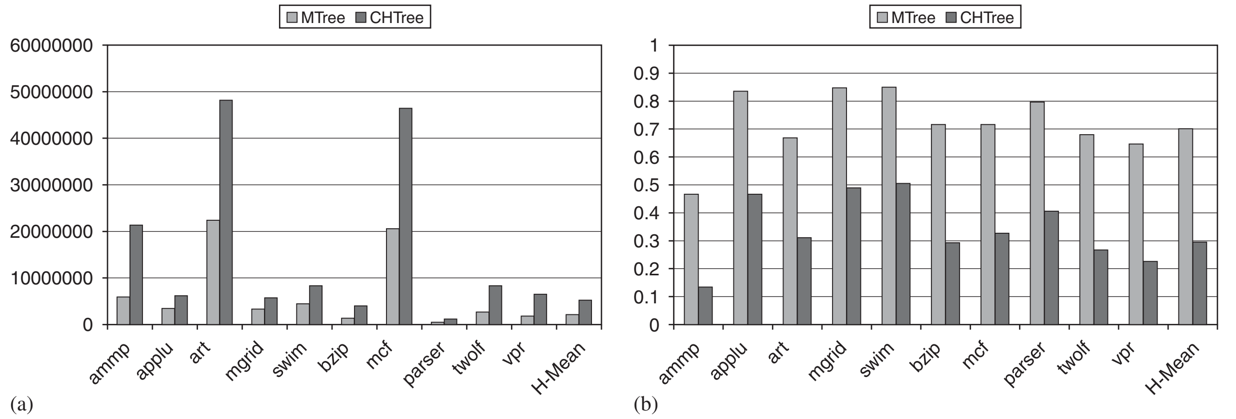

Fig. 10. Accesses/hit rates of an unified MAC/L2 cache (256 KB): (a) number of cache accesses, (b) cache hit rates.

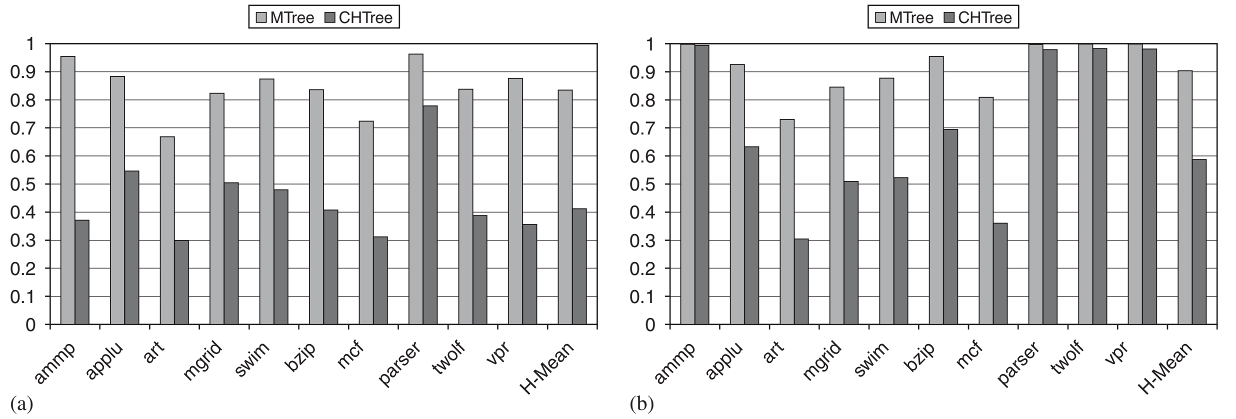

Fig. 11. Normalized IPC for MACTree and CHTree with an unified MAC/L2 cache: (a) 256 KB MAC/L2 cache, (b) 2 MB MAC/L2 cache.

4.3.2. Performance sensitivity of the MAC cache size

Now we investigate the performance sensitivity of using different MAC cache sizes for the MACTree. The normalized IPC results are shown in Fig. 12 for four different MAC caches including a perfect MAC cache, a transparent security support representing the baseline. As shown, even for a MAC cache as small as 8 KB, the performance with integrity check can approach closely to the baseline. The good performance of the MACTree scheme is attributed to the following reasons. First, the 8 KB MAC cache can hold a sufficient number of nodes for the MACTree in order to exploit temporal locality and thus achieve a high hit rate. Another reason is speculative execution. In other words, instructions are issued as soon as the data decryption is done so that the integrity check can be performed in

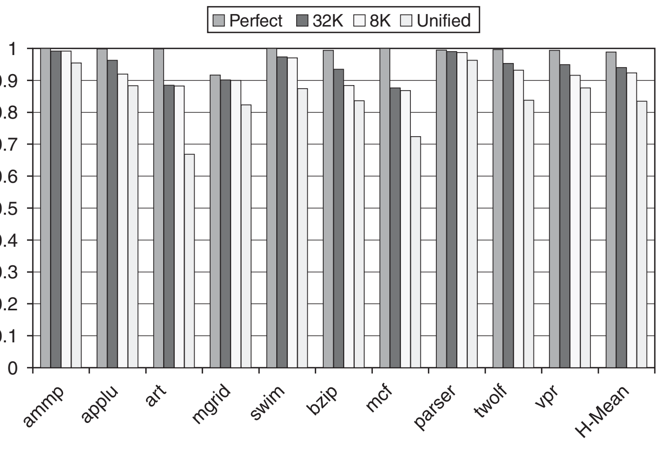

Fig. 12. Performance sensitivity of MAC cache sizes of MACTree (256 KB L2).

parallel with instruction execution. However, the instruction is not allowed to commit to the architectural states until integrity check is completed. As such, the latency of integrity check can be partially or even completely hidden. In the same figure, it also shows that when a MAC/L2 unified cache is employed, the performance is evidently degraded due to contention. Comparing with a separate 8 KB MAC cache, performance is reduced by 8% on average and 12% in the worst case, suggesting that a more efficient M-TREE implementation should consider the use of a separate MAC cache to store MAC values.

4.3.3. *Performance sensitivity of the hash latency*

The 80 ns hashing latency used by [6] might be optimistic according to either synthesized based hardware design or ASIC [4,12,19,23]. Thus, we also study the performance impact when the hashing latency is doubled to be 160 ns for both the MACTree and the CHTree scheme. The results are shown in Fig. 13, which shows that a higher hashing latency is not too sensitive to the overall performances regardless of the integrity check scheme used. In fact, the major performance bottleneck comes from the additional number of memory accesses induced by the integrity checking schemes. These additional memory accesses contend with regular memory accesses and thus degrade the overall performance. Meanwhile, the hash/MAC computations can be hidden by instruction executions, thus are not on the critical path. In summary, the effectiveness of the MACTree lies in the significant reduction in the memory bandwidth.

4.4. *M-TREE encryption evaluation*

4.4.1. *Performance comparison with the block cipher-based scheme*

Here we analyze the performance of M-TREE encryption scheme against the block cipher-based scheme under two different L2 configurations. The normalized IPCs for the M-TREE encryption scheme and the block cipher-based scheme are shown in Fig. 14. Apparently, the M-TREE encryption scheme outperforms the block cipher-based scheme for all benchmark programs. The slowdown of the M-TREE encryption scheme is about 30% in the worst case and 20% on average; while the block cipher-based scheme shows a 50% slowdown in the worst case and 30% on average.

With an 80 ns AES latency, there is still a considerable performance gap between the M-TREE encryption scheme with a perfect RIV/MAC cache and a realistic one. The reason is that the memory access latency cannot always hide the AES decryption latency entirely. Under our accurate memory modeling, the latency of the critical chunk is determined by many factors such as previous command, memory page hits, row misses, etc. In many cases, the memory lead-off latency turns out to be less than 80 cycles, not enough to cover the entire AES latency. This is manifested by reducing the AES latency to 40 ns. As shown in Fig. 14(b), performance under 40 ns AES latency is better than its counterpart under 80 ns AES shown in Fig. 14(a).

Also, for some benchmark programs, increasing the size of RM cache does not always lead to significant performance improvement. This is due to the RM cache hit rate does not improve by increasing the RM cache size for those applications, in particular, the SPEC2000FP programs. Furthermore, the RM values cached inside the RM cache show poor temporal locality, i.e. most of the cached RM values are not re-accessed prior to eviction.

4.4.2. *Performance sensitivity of the decryption delay*

As mentioned earlier, the performance impact of two different AES decryption latencies to the M-TREE encryption scheme is also examined. Fig. 14 shows the normalized IPCs with two AES latencies: 40 and 80 ns.[3] With an optimistic 40 ns AES decryption latency, the performance of M-TREE encryption scheme with a perfect RM cache now can approach the baseline. So long as the memory access latency is bigger than 40 cycles, the AES latency will be hidden by memory access latency, which is the common case. Under this circumstances, the performance overhead of the M-TREE encryption scheme is about 13% in the worst case and about 10% on average by using a 32 K RM cache. On the other hand, the performance overhead of the block cipher scheme is 25% in the worst case and 15% on average. For a more realistic AES latency 80 ns, the average degradation of the M-TREE is about 20% versus 30% of the block cipher.

4.5. *PTR processing*

Finally, we study the performance of the PTR processing by putting our proposed integrity checking and encryption together. We compare the performance of the M-TREE integrity checking and encryption scheme with that of the CHTree scheme and block cipher encryption scheme. Fig. 15 shows the IPC results normalized to the baseline. In general, the M-TREE architecture suffers much less performance penalty against the CHTree integrity checking scheme and block cipher encryption scheme. For a 256 KB L2, the M-TREE architecture can

[3] 40 ns latency for AES decryption was used in [17] which is fairly optimistic. Hence, we also use 80 ns for sensitivity evaluation.

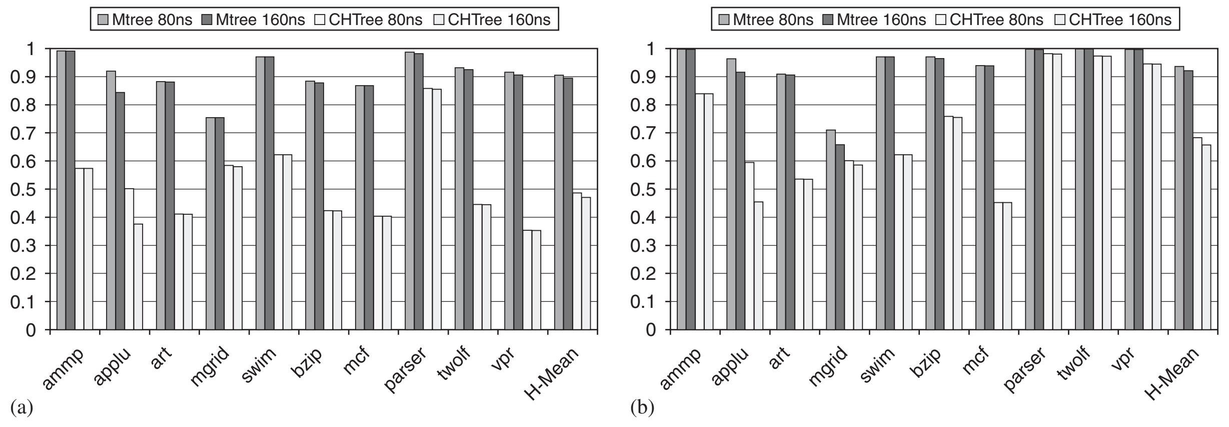

Fig. 13. Performance sensitivity on hash latency (8 KB MAC cache): (a) 256 K L2 cache, (b) 2 M L2 cache.

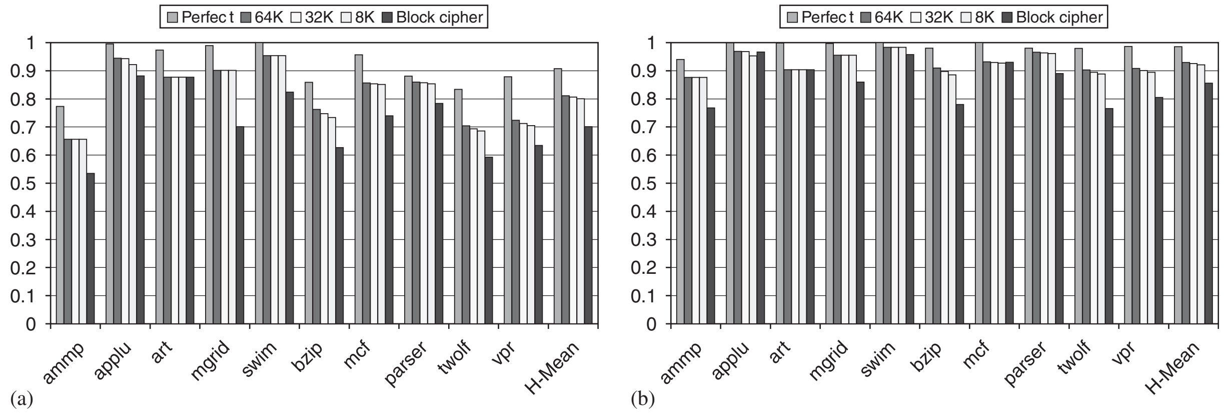

Fig. 14. Normalized IPC for the M-TREE encryption scheme (256 KB L2): (a) 80 ns decryption delay, (b) 40 ns decryption delay.

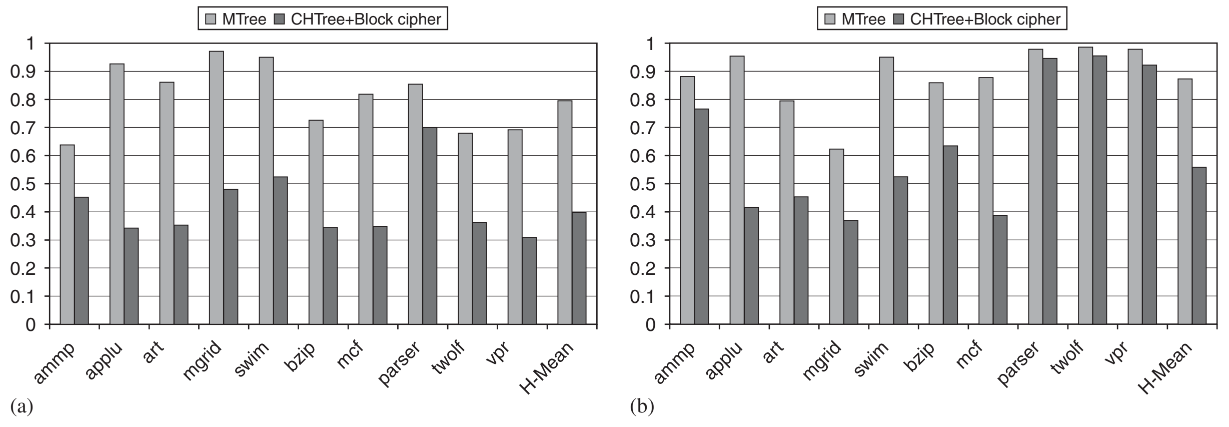

Fig. 15. Normalized IPC for PTR processing: (a) 256 KB L2 cache, (b) 2 MB L2 cache.

reduce performance overhead from 60% of the prior art down to 20% and from 45% to 12% for a 2 MB L2.

5. Conclusions

In this article, we propose the M-TREE processor architecture that implements novel integrity protection and encryption mechanisms for providing a tamper-resistant and tamper-evident computing environment. The architecture consists of two new cryptographic features: the MACTree integrity protection scheme and an OTP-class encryption scheme with novel and enhanced security mechanisms. We analyze the efficiency of the proposed integrity checking scheme and the encryption schemes against the existing techniques and show that the M-TREE architecture offers a significant performance improvement as well as security advantages over the prior art.

Based on our simulation results, it is shown that the MACTree integrity verification suffers the worst case overhead of

14%, which is substantially lower than the worst case overhead of 60% reported in published secure processing schemes. M-TREE encryption scheme improves the performance by more than 10% of that of the block cipher-based encryption scheme. Combined both integrity verification and encryption schemes of M-TREE, we show the performance is degraded within 20% versus the 60% in prior scheme using CHTree with a block cipher.

Acknowledgments

This work is supported in part by NSF grants CCF-0326396, CNS-0325536, and the Department of Energy Early CAREER Award.

References

[1] R.J. Anderson, Security Engineering: A Guide to Building Dependable Distributed Systems, Wiley, New York, 2001.

[2] T.M. Austin, Simplescalar 4.0 release note, ⟨http://www.simplescalar.com/⟩.

[4] A.J. Elbirt, W. Yip, B. Chetwynd, C. Paar, An FPGA-based performance evaluation of the AES block cipher candidate, IEEE Trans. VLSI Systems 9 (4) (2001).

[5] FIPS PUB 180-2: SHA256 hashing algorithm, National Institute of Science and Technology.

[6] B. Gassend, G.E. Suh, D. Clarke, M. van Dijk, S. Devadas, Caches and merkle trees for efficient memory integrity verification, in: Proceedings of the ninth International Symposium on High Performance Computer Architecture, 2003.

[7] M. Gries, A. Romer, Performance evaluation of recent DRAM architectures for embedded systems, TIK Report Nr. 82, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, November 1999.

[8] Hashes and message digests, ⟨http://www.cc.gatech.edu/classes/AY2004/cs6262_spring/hashes.ppt⟩, 2004.

[9] Intel lagrande technology (LT) for safer computing, ⟨http://www.intel.com/technology/security⟩.

[10] Intel 82802 firmware hub: random number generator, Intel Corporation, December 1999.

[11] N.P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully associative cache and prefecth buffers, in: Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990.

[12] N.S. Kim, T. Mudge, R. Brown, A 2.3 Gb/s fully integrated and synthesizable AES Rijndael core, in: Proceedings of the IEEE 2003 Custom Integrated Circuits Conference, 2003.

[13] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, Architectural support for copy and tamper resistant software, in: Proceedings of the ninth Symposium on Architectural Support for Programming Languages and Operating Systems, 2000.

[14] H. Lipmaa, P. Rogaway, D. Wagner, Comments to NIST concerning AES modes of operations: CTR-mode encryption, ⟨http://csrc.nist.gov/encryption/modes/workshop1/papers/lipmaa-ctr.pdf⟩, Technical Report, 2000.

[15] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, Handbook of Applied Cryptography, CRC, Boca Raton, FL, 1996.

[16] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, in: Proceedings of the 10th Symposium on Architectural Support for Programming Languages and Operating Systems, October 2002.

[17] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, S. Devadas, AEGIS: architecture for tamper-evident and tamper-resistant processing, in: Proceedings of the 2003 International Conference on Supercomputing, 2003.

[18] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, S. Devadas, Efficient memory integrity verification and encryption for secure processors, in: Proceedings of IEEE/ACM 36th International Symposium on Microarchitecture, December 2003.

[19] P. Tandon, High-performance advanced encryption standard (AES) security co-processor design, Master's Thesis, School of Electrical and Computer Engineering, Georgia Institute of Technology, December 2003.

[20] Trusted computing group, ⟨http://www.trustedcomputinggroup.org/⟩.

[21] X. Wang, Y.L. Yin, H. Yu, Finding collisions in the full SHA-1, in: CRYPTO 2005, 2005.

[22] X. Wang, H. Yu, How to break MD5 and other hash functions, in: EUROCRYPTO 2005, 2005.

[23] B. Weeks, M. Bran, T. Rozylowicz, C. Ficke, Hardware performance simulations of round 2 advanced encryption standard algorithms, ⟨http://csrc.nist.gov/CryptoToolkit/aes/round2/NSA-AESfinalreport.pdf⟩, 2000.

[24] J. Yang, Y. Zhang, L. Gao, Fast secure processor for inhibiting software piracy and tampering, in: Proceedings of IEEE/ACM 36th International Symposium on Microarchitecture, 2003.

Chenghuai Lu is a Ph.D. student of the College of Computing, Georgia Institute of Technology. He is pursuing his Ph. D. degree in information security. His research focus is focused on secure embedded system and architecture. Lu has a B.S. and M.S. degree on Mathematics from the Shanghai Jiaotong University.

Tao Zhang is a Ph.D. candidate in the College of Computing, Georgia Institute of Technology. His research interests include compiler/micro-architecture optimizations for embedded systems and compiler/micro-architecture support for software security. Zhang has a M.S. in Computer Science from the Georgia Institute of Technology and a B.S. in Computer Science from the Peking University.

Weidong Shi is a Ph.D. candidate in the College of Computing, Georgia Institute of Technology. His research interests include anti-reverse engineering IC and processor, hardware-assisted media digital right management, high-performance networked media processing, future entertainment computing platforms. Shi has a M.S. in Computer Science from the Georgia Institute of Technology.

Hsien-Hsin S. Lee received his Ph.D. in Computer Science and Engineering from the University of Michigan, Ann Arbor. He is an Assistant Professor of the School of Electrical and Computer Engineering at the Georgia Institute of Technology. His research interests include computer architecture, low-power VLSI, information security, and 3D ICs. Prior to joining academia, he was a Senior Computer Architect at the Intel Corporation and later the Architecture Manager of the StarCore DSP Technology Center at Agere Systems. Dr. Lee's doctoral thesis was awarded the Horace H. Rackham School Distinguished Dissertation Award at the University of Michigan. He has co-authored 3 articles that won the Best Paper Award from MICRO-33, CASES-04 and IBM Watson PAC2 conferences. More recently, Dr. Lee received the Department of Energy Early CAREER Award and was named the recipient of the 2006 ECE Outstanding Jr. Faculty Member Award at the Georgia Tech. Dr. Lee holds 4 US patents and is a Member of Tau Beta Pi, ACM and IEEE.