Hardware Mechanism and Performance Evaluation of Hierarchical Page-Based Memory Bus Protection

Lifeng Su*, Albert Martinez*, Pierre Guillemin*, Sébastien Cerdan†, Renaud Pacalet†
*STMicroelectronics, France
Email: firstname.lastname@st.com
†Institut Télécom; Télécom ParisTech; CNRS LTCI, France
Email: firstname.lastname@telecom-paristech.fr

***Abstract*—SecBus project aims at building a two-level page-based memory bus protection platform. A trusted Operating System (OS) dynamically manages security contexts for memory pages. Via such contexts, an independent hardware module is driven to execute cryptographic protections. The fact that both processor and software tool chain are not modified strengthens platform realizability and market acceptability. This paper presents SecBus hardware mechanism. Performance improvement is sustained by such optimization measures as usage of multiple caches, incoherent-Hash-Tree management and speculative execution. Finally, performance evaluations are made for various configurations on virtual simulation platform.**

I. Introduction

Nowadays trusted computing is highlighted in computer society. Commercial commodities [1] whose security is supported by Trusted Platform Module (TPM) [2] appear in succession. However as external memory bus still carries critical data, TPM does not prevent board-level probing attacks. Their security must be still questioned. While building a trusted computing platform, memory bus security (confidentiality/integrity) must be considered. Concretely, an adversary could retrieve critical data from external memory bus or tamper with it. Confidentiality is compromised in the former case and integrity in the latter. Successful attacking cases have the microcontroller DS5002FP [3] and the Microsoft XBox [4].

Several research programs have addressed this issue during the past decade. XOM project [5] plans to build process protection environments in processor against both physical attacks and malware attacks. However, replay attacks are not considered at all and untrusted OS [6] has to be still customized. Its implementation is complex and heavy. CryptoPage project [7] is akin to XOM project. Its primary contribution lies in taking memory space permutation [8] as countermeasure against replay attacks. AEGIS project [9] addresses the building of secure execution environments at instruction flow level in processor. It is imperative to modify software tool chain. All three projects require significant modifications on both processor and OS. In other projects [10], bus protection is arranged only at hardware level and software aspect is not exploited for performance improvement.

Unlike most previous works, CPU is not modified at all in SecBus project and an independent hardware module is located at the frontend of memory controller. This constraint is required in many markets: for instance, while embedding legacy CPU sub-systems into consumer products, CPU modifications are usually believed as unrealistic. Figure 1 denotes its specific position.

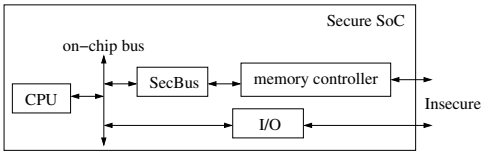

Fig. 1. SecBus module position

The key assumption in SecBus is that OS is secure and trusted. With this assumption, memory bus protection could be still granulated into memory pages without CPU modifications. SecBus implementation could be at reasonably low expense and have controlled performance impact. Modifying software tool chain is circumvented as well. More importantly, one new approach is revealed to consummate a genuine trusted computing platform where security tasks are more flexibly deployed at software level. This is the SecBus long-term goal. Although trusted OS is a strong hypothesis, some researchers have been attempting with persistence to effectuate this sort of work [11][12][13][14][15]. Such experiences anticipate that rigorous, complete, formal verification is practically achievable for OSes, at least such microkernels of small scale. Once it succeeds, SecBus secure platform could well prevent against software exploits, such as Denial-of-Service (DoS) attacks and memory exhaustion attacks. Some cases [16][17] demonstrate the importance of resistance to software attacks. In view of the compatibility with insecure large commodity OSes, the idea of virtualization technologies in [18][19] might be a profitable enrichment to SecBus platform. At present, SecBus project is dedicated to memory bus protection and this paper describes SecBus hardware mechanism and performance evaluation.

The rest of this paper is organized as follows: Section 2 recalls some basic concepts. Section 3 describes new cache-based mechanism for Hash Tree (HT) management. Section 4 presents SecBus hardware model. In Section 5, cycle-accurate performance evaluation is detailed. The last section states the future work.

II. Review of Basic Concepts

First of all, we briefly review the page-based protection scheme presented in [20]. As a whole, hierarchical memory pages are taken not only as the granularity of memory management in OS but also as the granularity of memory bus protection. Every page is associated to a security context which bundles related security parameters. Upon memory access, SecBus hardware module hierarchically searches specific security context by memory address. With assistance of the context, related cryptographic operations are undertaken. In this scheme the design of security contexts is crucial.

With respect to cryptographic protection, our primary concern is about the selection of cryptographic primitives. In SecBus platform a variety of primitives are selectively applied for performance issues. It is well known that every program page holds an execution attribute: Read-Write (RW) or Read-Only (RO). Regardless of this attribute's value, HT primitive is always appropriate to integrity protection and block cipher to confidentiality protection. But, as RO page is written only once and not sensitive to replay attacks, addressed Message Authentication Codes (MAC) are more preferable to its integrity protection than rather costly HT primitive [21][22]. Likewise its confidentiality could be well protected by stream cipher, One Time Pad, whose benefit consists in parallelizing mask generations with memory read accesses. In a word, this attribute is translated into the usage of cryptographic primitives.

To express this distinction, we define the structure, Security Policy (SP), which packages a confidentiality mode, an integrity mode and a secret key. As expected, the confidentiality mode may be set with any of three choices: *None*, *BlockCipher* and *StreamCipher*. And the integrity mode holds three similar options: *None*, *MAC* and *HashTree*. Since this execution attribute covers whole program linking segment in OS, various pages in a segment possibly share one common SP. In addition, for the purpose of integrity protection of program pages, some auxiliary pages, HT/MAC pages, are allocated in memory to store either HTs or MAC-sets[1]. Such pages other than program pages do not possess this execution attribute so that SP structure is unsuited for them. Their security contexts need arranging further.

Another structure, Page Security Parameters Entry (PSPE), is specific to every memory page. Its role is a bit similar to page table entry in Memory Management Unit (MMU). All PSPEs are disposed in a hierarchy of tables, as are page tables of MMU. As SecBus project takes two page size levels (4MBytes/4KBytes), PSPE search usually enforces two PSPE table walks at the basis of physical address. Regarding its concrete definition, two different formats, master PSPE and slave PSPE, are shown in Figure 2. Master PSPE is mainly applied for program pages. Its task is to map physical address to related SP index and related HT/MAC page. Slave PSPE works for HT/MAC pages. It is known that the integrity of HT page relies on root hash value and MAC page on MAC key. However, Figure 2 denotes that this format only preserves root hash values for HT pages. In reality no format is defined for MAC page in that MAC key which is also shareable could embezzle the enciphering key from relevant RO page's SP. The PSPE of MAC page could be of any value. In the end, three indicative bits are added to facilitate hierarchical PSPE table walks and differentiate both formats. In summary, the security context for RW page is composed of SP, master PSPE and slave PSPE (if integrity is protected), whereas the security context for RO page is represented by SP and master PSPE.

[1]A HT protects the integrity of a RW page; a MAC-set contains all MAC values on which the integrity of a RO page relies.

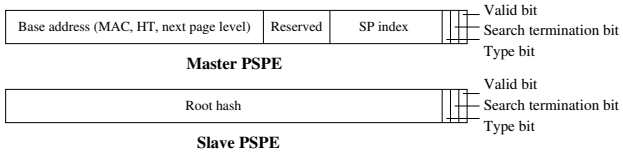

Fig. 2. PSPE definition

In SecBus, three memory blocks are particularly allocated to store all SPs, PSPEs as well as a Master Hash Tree (MHT). The MHT protects the integrity of all SPs and PSPEs. Such three blocks form a Master Block which is reserved in memory and excluded from traditional paging mechanisms (page allocation/recycling) in OS. The protection of Master Block is based on the special security context, Master Security Parameter Group (MSPG). As root of trust, the MSPG is permanently kept in SecBus hardware module. While regulating security contexts for memory pages, OS does not operate straightforward on Master Block. Instead, OS employs I/O interface commands to steer SecBus hardware module to configure them. Figure 3 illustrates global memory footprint.

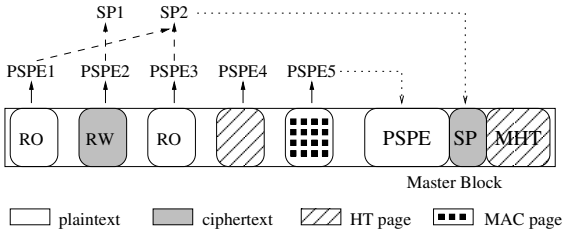

Fig. 3. Page-based Protection Framework

III. Incoherent-Hash-Tree Management

To alleviate performance degradation made by traditional HT management, we exert much effort on elaborating a cache-based incoherent-HT management mechanism.

HT primitive behaves as either HT check or HT update. Both operations are a sequence of hash computation steps and the terminal (root) ensures all nodes' integrity. To attain integrity protection, either of them traverses a whole path until the root. Assuming some intermediate node is trusted, HT check can be terminated in advance by this node. This is the As-Soon-As-Possible (ASAP) policy for HT check. If some node in HT update is saved in a trusted place, this updating

path can be also ended ahead of schedule. This is the As-Late-As-Possible (ALAP) policy for HT update. In essence only one step is walked for protection. Using conventional write-back cache, HT controller has only the ASAP policy work. To make both policies run together, such write-back cache must be personalized. Note that, when an updated node is cached, this node is tagged as dirty with the implication that the synchronization between cache and memory is not done and the residual path is not traversed. HT consistency is crashed in this case and its settlement converges upon two knotty issues.

The first is that the cache ought to realize specific context where requested node lies in. Every HT node holds four possible contexts: its children's checking/updating process and its siblings' checking/updating process. A dirty node in cache can be freely involved in its children's checking/updating process as well as in its siblings' updating process, whereas it is inutile to its siblings' checking process. Alternatively its past value has got to be fetched from memory because its parent node hinges on it yet. It denotes that cache read/write interface should be characterized with such four contexts and cache behavior must be customized. As such, HT controller is capable of emitting more precise commands to cache and the latter renders appropriate node values. Figure 4 depicts a 4-ary HT as example.

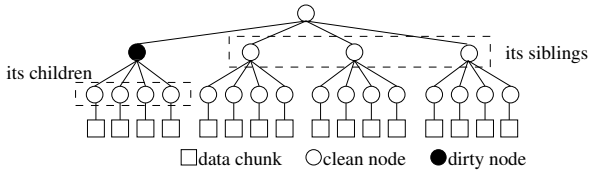

Fig. 4. a 4-ary hash tree

The second issue is how to evict dirty nodes. Upon cache write request, incoming node needs to supersede a node in cache. If this superseded node is dirty, the cache is obligated to advance its updating process before evicting it to memory. To do this HT update, the cache has to turn to HT controller but it is a deadlock between HT controller and cache. For this reason a peculiar replacement policy is proposed for cache: new entrants always substitute for clean nodes. Nevertheless, with the increase of cache writes, dirty nodes grow more and more and never decrease. In the end, when all cached nodes are dirty, two new problems are engendered: the performance of ASAP policy is drastically obliterated because memory nodes can not be cached and ALAP policy is void of sense because incoming dirty nodes can not be cached. It signifies that the cache must strictly control the number of dirty nodes. As soon as it occcurs too many cache writes, the cache solicits HT controller to advance the updating process of some dirty nodes.

To restrain the increment of dirty nodes in a regular manner, a new recursive mechanism is conceived. Upon cache write request, a clean node in cache is first vacated for incoming dirty node and then the cache checks whether this number exceeds preset threshold or not. If exceeded, the cache will select some dirty node under predefined policy (FIFO, LRU, etc) and send it back to HT controller. Accordingly, HT controller will proceed with the updating process of captured node. Thanks to the ALAP policy, only one updating step needs walking. In this step, all dirty siblings are involved together in their parent node update. Related dirty nodes are subsequently evicted to memory and their dirty bits are removed. Finally another new cache write request is launched to conserve this updated parent node (a dirty node is generated again). This is visibly a recursive process. Indeed when one or more dirty siblings participate in this updating step, the recursive process is able to be terminated and even the number of dirty nodes is diminished.

Up to now, both ASAP policy and ALAP policy can operate smoothly by the collaboration between HT controller and characteristic HASH cache. An important point must be made: dirty nodes received by HT controller are possibly from different HT pages. To proceed with their updating processes, HT controller need aggregate their siblings. In some cases, slave PSPEs (root hash of HT page) are necessitated for HT check. Bearing it in mind avails to understand SecBus architecture model in the next section.

IV. SecBus Hardware Module

This section illustrates the backbone of SecBus hardware mechanism. Figure 5 depicts its global functional model. Roughly, SecBus hardware is partitioned into 5 main func-

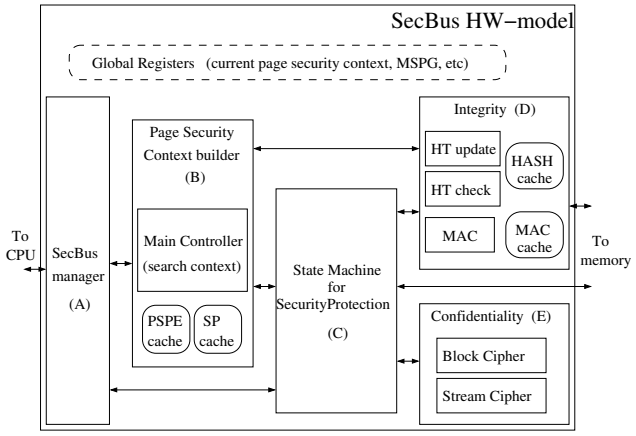

Fig. 5. SecBus hardware model

tional blocks and some registers:

1) Block A: regulate memory accesses from processor and request services from block B/C
2) Block B: search page security contexts
3) Block C: supervise security protection with scheduling cryptographic services (encipher, HT check, etc)
4) Block D: execute integrity protection
5) Block E: execute confidentiality protection

Upon memory access from processor, block A first informs block B to search page security context. After the seeking ceases, block A delivers the control to block C for security

protection. Under that security context, block C invokes cryptographic services from block D/E. This is the main execution track in SecBus hardware module. With the use of speculative execution, the communication between different blocks comes to be a bit complex. This point is intensively clarified below. We suppose that every block is attached with a flag holding two values, *idle* and *busy*, which testifies this block's usability to others. When a block is activated, its flag alters from *idle* to *busy*. When its task is accomplished, it goes back to *idle*. As the necessity of speculative execution, integrity protection must be enforced on ciphertexts, which makes it possible that the integrity check of read data is done in parallel to decryption. As decryption operation is normally much more rapid than integrity check, CPU execution is judiciously reinstated witnout the result of integrity check. In other words, other blocks have been *idle* already and yet block D might be *busy*. And the next memory access from processor could be forwarded directly to block C to expedite its handling. Upon write access, block D stays *busy* as well due to the long HT updating process. Here the special point is that, to evict dirty nodes from HASH cache, HT controller perhaps claims slave PSPEs from block B. Thereby block B/C are deemed to be *busy* along with block D and new memory accesses must be stopped at block A. As soon as HT update is completed, all of them meanwhile return to *idle*. This speculative execution has a prescient positive effect on system performance. In the following, three chief blocks A/B/C are elucidated in turn.

Block A, SecBus manager, is the memory access controller. Its function is outlined by two primary stages. At the first stage, block B is requested to search page security context and, at the second stage, block C is required for security protection. Each stage's prerequisite is that relevant block must be *idle* and otherwise this state machine is suspended. This functional description is simplified quite a lot. In practice, the fact of on-chip bus burst operation is also reflected in this controller. A tiny data buffer is imported to keep some related data in burst operations. This optimization enhances data throughput because the bit-width of data chunks for cryptographic primitives is several times bigger than memory bus width. Besides, a succession of memory accesses often belong to one same page. After the security context is sought for the first access and put into registers, it is not mandatory to make a search again for immediate accesses. This point effectively reduces the search number.

Block B is the controller which seeks security contexts for memory pages. Algorithm 1 describes the whole search process. This process includes two essential steps. One step is to do two-level PSPE search for physical address. The other is to search related SP with its index. As the security context for RW page perhaps contains two PSPEs (master PSPE and slave PSPE), such two-level PSPE search might be executed for two times. On the contrary, the security context for either RO page or HT page contains only one PSPE so that PSPE search is performed only for one time. After the seeking success, all security parameters are deposited in registers to bolster the execution of other parts in SecBus module.

Algorithm 1 Search page security contexts

Require: input: 32-bit memory address & MSPG
Ensure: output: page security context
1: Extract the highest 10 bits from memory address
2: Compute PSPE address at the 1st level
3: Read PSPE from cache
4: **if** the validity bit is 1 **then**
5: **if** the type bit indicates master PSPE **then**
6: Go to *line 20*
7: **else**
8: Go to *line 26*
9: **end if**
10: **else**
11: **if** this PSPE is at the 1st level **then**
12: Extract the middle 10 bits from memory address
13: Extract base address at the next level
14: Compute PSPE address at the 2nd level
15: Return to *line 3*
16: **else**
17: ERROR!
18: **end if**
19: **end if**
20: Extract SP index from master PSPE
21: Read SP from cache
22: **if** HashTree mode is valid **then**
23: Extract base address of HT from master PSPE
24: Return to *line 1*
25: **end if**
26: Put parameters into associated registers

In block B, both PSPE cache and SP cache are utilized to accelerate the formation of security contexts. The use of SP cache has no unusual points but PSPE cache is a bit exceptional. In most cases, write-back cache is more expected than write-through cache. But, owing to a deadlock caused by ALAP policy, PSPE cache must be implemented as write-through cache. During HT update, HT controller probably requests block B to seek slave PSPEs. Upon PSPE cache miss, PSPE cache controller reads it from memory and supersedes some cached PSPE. If the superseded PSPE is dirty, it will have to be evicted to memory first. During the evicting process, block C is requested and further block D is invoked again for a new HT update (MHT update). This is a deadlock. Fortunately, with the aid of ALAP policy, slave PSPEs are renewed much less frequently and this drawback is compensated to the great extent.

Block C acts as the state machine for the management of cryptographic operations. While scheduling related cryptographic services according to two security modes in SPs, synchronous issues ought to be soundly coordinated. This block normally serves for block A but, when SP/PSPE cache miss happens, its function is also postulated by block B. Algorithm 2 presents the frame of this state machine. Here two respects are accentuated to make it more comprehensible. The first is at *line 2*. When a data is written in memory, some adjacent ones are often wanted to assemble a whole data chunk aligned for cryptographic operations. It implies that this algorithm is recursive because memory write accesses trigger new memory read accesses. At *line 15*, mask generations in stream cipher are carried out in parallel to memory read accesses, which has been mentioned in Section 2.

Algorithm 2 Simplified State Machine

```
Require: input: memory access & security context
if it is write access then
    Collect whole data chunk {read access!}
    Request block E for encryption
    while block E is busy do
        Please wait...
    end while
    while block D is busy do
        Please wait...
    end while
    Request block D for HT update
else
    while block D is busy do
        Please wait...
    end while
    Read whole data chunk
    Request block D/E for decryption&integrity check {parallel}
    while block E is busy do
        Please wait...
    end while
end if
```

So far, the backbone of SecBus hardware mechanism is lucidly presented. Other elements which have little impact on the principal architecture are not unfolded in this paper. For instance, block cipher, MAC-checker, random number generator, initialization mechanism, OS interface controller, ... In the next section, the performance of this hardware model is evaluated.

V. Timing Performance Evaluation

SecBus module, located in the middle of critical path, has prompt impact on memory latency and program execution time. To evaluate timing performance degradation, we use the cycle accurate simulator, SoC designer [23], which is based on ARM instruction set. Table I summarizes critical parameters of this simulation platform. Various standalone EEMBC benchmarks [24] are respectively tested. Four caches, SP/PSPE/HASH/MAC, are separately introduced to ameliorate performance. In reality, according to experimental results, MAC cache has negligible effect because fetching original MAC values can be hidden by MAC computations. Thereby MAC cache will not be considered later. On the contrary, three others specified in Table II show remarkable effect. To more accurately assess them, we configure four different simulation schemes:

1) No caches (the worst case).
2) Use only two caches, SP/PSPE. PSPE cache is write-back cache.
3) Besides SP/PSPE caches, HASH cache, traditional write-back cache, is joined and only ASAP policy works.
4) Besides SP/PSPE caches, HASH cache, characteristic write-back cache, is applied for incoherent-HT management. PSPE cache must be write-through cache.

Figure 6 demonstrates general performance overhead in various schemes compared to the case where memory bus is unprotected. According to Chart (a), the scheme without any cache leads to huge degradation. In most cases, performance overheads exceed 3000%. In the worst case, it reaches 23753%. Thereby SecBus module without any cache is clearly unserviceable. Chart (b) denotes that both SP cache and PSPE cache greatly mitigate performance degradation. With the use of both policies (ASAP/ALAP), some overheads among the benchmarks are almost negligible and most are lower than 100%. Even the worst case has only 472% overhead. It must be pointed out that the benchmark *rgbtocmyk* which converts different color spaces generates plenty of memory accesses. It is definitely unreasonable to assign such strong cryptographic primitive as HT primitive to protect each memory access. A more practical way is to execute security protection only at the beginning/end of conversion process. Here we test it only to expose the significance of such caches.

Processor	ARM1176JZF
I-Cache/D-cache	16KB, 32B line, 4-way
write policy	write-back
Replacement policy	Random
Allocation policy	Write-No-Allocate
On-chip Bus	AMBA AXI (32 bits)
Bus clock ratio (Fcpu/Fbus)	1
Memory volume	256MB
Memory latency (precharge, active, CAS)	3 cycles
hash function (input/output, time)	256/64 bits, 16-20 cycles
MAC function (input/output, time)	288/64 bits, 20 cycles
block cipher (data chunk, time)	64 bits, 4 cycles

TABLE I
PLATFORM PARAMETERS

	SP cache	PSPE cache	HASH cache
Architecture	direct mapped	full associative	set associative
Write policy	write-back	write-through/back	write-back
Allocation policy	write-allocate	write-allocate	write-allocate
Way number	-	-	4
Line number	2	16	256
Line size	16B	8B	8B

TABLE II
CACHE PARAMETERS

Figures 7 and 8 present cache contributions normalized by the worst case. On the whole, the effect of SP/PSPE caches is particularly distinguished. On average, the performance is enhanced by 76.4% and the number of hash computations is reduced by 78.1%. Afterwards both policies (ASAP/ALAP) further contribute about 20% so that both indexes are finally up to 97.8% on average.

VI. Conclusions and Future Work

In this paper, we present SecBus hardware mechanism and experimental performance results. As a rule, performance degradation rests highly on specific applications. The majority of popular embedded applications acquire acceptable performance degradations. But current evaluation is only based on standalone applications and the management of security contexts in OS has not been implicated. Therefore this conclusion is not comprehensive. In the coming future, we will specify security management in OS, implement it in an OS and erect a system demonstrator combining software and hardware. Ultimately this customized OS (at least the part of security management) will be deeply verified using formal method.

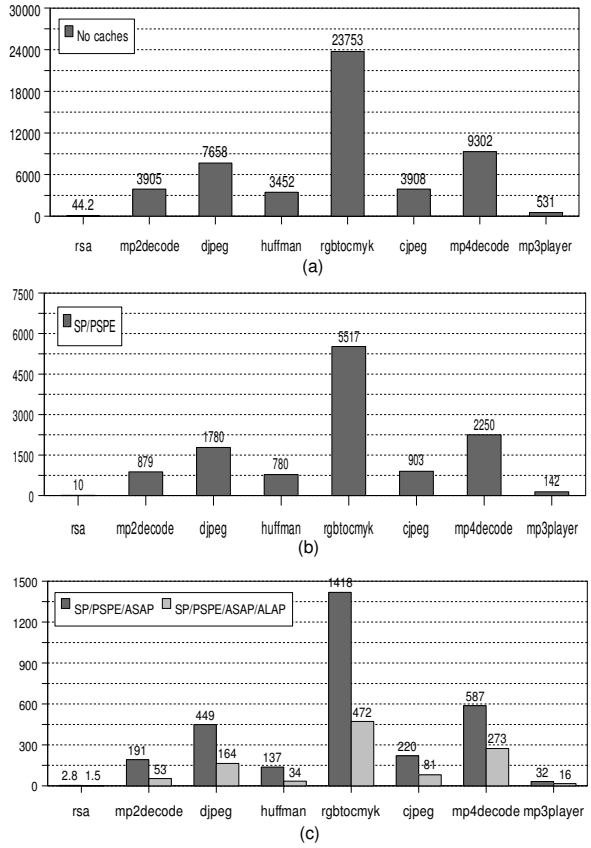

Fig. 6. Performance Overhead (%)

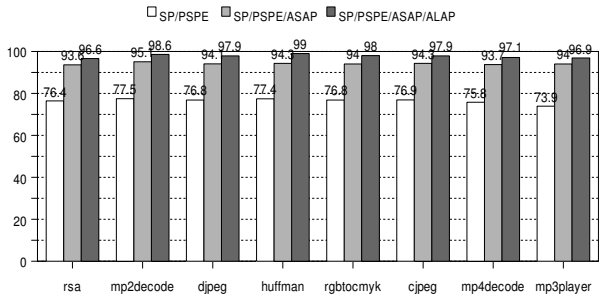

Fig. 7. Performance Overhead Reduction (%)

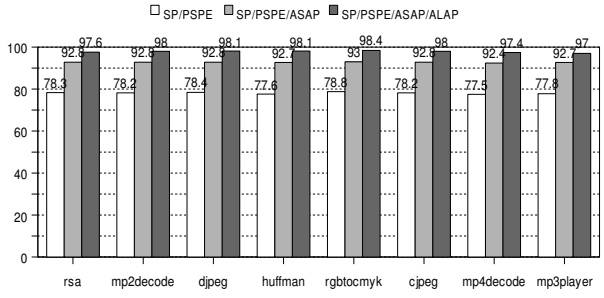

Fig. 8. Hash Computation Reduction (%)

ACKNOWLEDGMENT

This work was supported by the French Conseil Régional Provence-Alpes-Côte d'Azur and by the French Association Nationale de la Recherche Technique.

REFERENCES

[1] Intel Corporation, "Intel®Trusted Execution Technology," http://www.intel.com/technology/security/.
[2] Trusted Computing Group, "Trusted Platform Module (TPM) Main Specification," http://www.trustedcomputinggroup.org/specs/TPM/, 2006.
[3] M. Kuhn, "Cipher Instruction Search Attack on The Bus-Encryption Security Microcontroller DS5002FP," *IEEE Transactions on Computers*, 1998.
[4] A.B. Huang, "Keeping Secrets in Hardware: The Microsoft XBox Case Study," *MIT, AI Memo*, 2002.
[5] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz, "Architecture Support for Copy and Tamper Resistant Software," in *ASPLOS*, 2000.
[6] D. Lie, C. Thekkath and M. Horowitz, "Implementing an Untrusted Operating System on Trusted Hardware," in *SOSP*, 2003.
[7] G. Duc and R. Keryell, "CRYPTOPAGE: an Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection," in *ACSAC*, 2006.
[8] X. Zhuang, T.Zhang and S.Pande, "HIDE: an infrastructure for efficiently protecting information leakage on the address bus," in *ASPLOS*, 2004.
[9] G.E. Suh, D. Clarke, B. Gassent, M. van Dijk and S. Devadas, "AEGIS: Architecture For Tamper-Evident And Tamper-Resistant Processing," in *ICS*, 2003.
[10] R. Elbaz, D. Champagne, R.B. Lee, L. Torres, G. Sassatelli and P. Guillemin, "TEC-Tree: a Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks," in *CHES*, 2007.
[11] J. Rushby, "Design and verification of secure systems," in *SOSP*, 1981.
[12] W. Bevier, "Kit: A study in operating system verification," *IEEE Transactions on Software Engineering*, 1989.
[13] W.B. Martin, P. White, A. Goldberg and F.S. Taylor, "Formal construction of the mathematically analyzed separation kernel," in *ASE'00, IEEE Computer Society*, 2000.
[14] E. Alkassar, N. Schirmer and A.Starostin, "Formal pervasive verification of a paging mechanism," in *TACAS*, 2008.
[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin and others, "seL4: Formal Verification of an OS Kernel," in *SOSP*, 2009.
[16] A. Seshadri, M. Luk, N. Qu and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *SOSP*, 2007.
[17] L. Litty, H. Andrés Lagar-Cavilla and D. Lie, "Hypervisor Support for Identifying Covertly Executing Binaries," in *USENIX Security Symposium*, 2008.
[18] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," in *SOSP*, 2003.
[19] J. Criswell, A. Lenharth, D. Dhurjati and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *SOSP*, 2007.
[20] L. Su, S. Courcambeck, P. Guillemin, C. Schwarz and R. Pacalet, "SecBus: Operating System Controlled Hierarchical Page-Based Memory Bus Protection," in *DATE*, 2009.
[21] B. Gassend, G.E. Suh, D. Clarke, M. van Dijk and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," in *HPCA*, 2003.
[22] D. Champagne, R. Elbaz and R.B. Lee, "The Reduced Address Space (RAS) for Application Memory Authentication," in *ISC*, 2008.
[23] "SoC Designer," http://carbondesignsystems.com/products_socd.shtml.
[24] "EEMBC consumerbench 1.1," http://www.eembc.org/about.