SecBus: Operating System Controlled Hierarchical Page-Based Memory Bus Protection

Lifeng Su*, Stephan Courcambeck*, Pierre Guillemin*, Christian Schwarz*, Renaud Pacalet†
*STMicroelectronics, 13106 Rousset, France. Email: firstname.lastname@st.com
†Institut TELECOM ; TELECOM ParisTech ; CNRS LTCI, 06904 Sophia Antipolis, France.
Email: renaud.pacalet@telecom-paristech.fr

***Abstract*—This paper presents a new two-levels page-based memory bus protection scheme. A trusted Operating System drives a hardware cryptographic unit and manages security contexts for each protected memory page. The hardware unit is located between the internal system bus and the memory controller. It protects the integrity and confidentiality of selected memory pages. For better acceptability the processor (CPU) architecture and the software application level are unmodified. The impact of the security on cost and performance is optimized by several algorithmic and hardware techniques and by a differentiated handling of memory pages, depending on their characteristics.**

I. Introduction

Since the DS5002FP secure microprocessor [1] and the XBox by Microsoft [2] were successfully invaded in succession, external memory bus security has become an active research topic in both industry and academy. Board-level probing attacks being far less complex and expensive than on-chip probing, adversaries can acquire confidential data and even corrupt the execution of critical programs by much simpler means than what is required for silicon-level attacks. Passive probing violates the confidentiality of the program while active probing (injection) threatens its integrity and refines in three sub-classes: spoofing (substitution of regular memory content with forged data), splicing (space permutation in memory) and replay (time permutation).

In the past few years, several important results have been achieved in the field. Guilmont et al. [3] propose an improved Memory Management Unit (MMU) to guarantee memory confidentiality, but not integrity, on a memory page basis. The AEGIS project [4] aims at building a secure execution environment to protect software processes from memory bus attacks and from each other. eXecute Only Memory (XOM) [5], [6] proposes a secure process environment resistant against physical and software attacks. Integrity protection is based on addressed Message Authentication Code (MAC) but does not prevent replay attacks. CryptoPage [7] secures processes execution environments with hardware and software support. It even includes features from the HIDE project [8] to prevent information leakage through the address bus. All these projects require significant modifications of the processor and of the software tool chain. Ref. [9] is a notable exception as it presents an Operating System (OS) controlled bus enciphering technique without any hardware modification. However, it does not provide integrity protection. The architecture also relies on several strong hypotheses like, for instance, on-chip random generation, large embedded memories allowing on-chip execution of parts of the OS, secure startup phases, etc.

The typical target platform of the SecBus scheme is a medium-to-low cost mass market device, embedding legacy 32 bits CPUs. Our main goal is to provide a strong confidentiality and integrity protection of a processor's external memory bus, with acceptable cost and performance overhead, compatible with existing microprocessors and without any modifications of the CPU subsystem (core, MMU and caches) nor the software development tool chain. The acceptability of deep CPU modifications is low and frequently considered impractical. Similarly, the acceptability of modifications in the software design tool chain is low, software reuse without redesign being an important concern in many cases.

The cost and performance constraints lead to a careful selection of the cryptographic primitives but also to their selective and differentiated application, depending on the specific process security requirements and on the Read-Write (RW) or Read-Only (RO) nature of their memory pages. This, in turn, leads to the implication of the OS in the security policy management. The OS must therefore be trusted. Under this strong assumption, micro-kernels, hypervisors or isolated memory managers are preferred over large legacy OSes. A typical SecBus enhanced platform will run legacy OSes and other applications on top of a formally verified micro-kernel.

The rest of the paper is organized as follows: section 2 describes the main functional features of the SecBus scheme. Section 3 details the OS role in the security management. Section 4 presents some aspects of the hardware unit and discuss performance improvement.

II. The SecBus scheme

Our main concerns are performance and cost. Using different cryptographic primitives to protect RW and RO pages is a way to optimize both. As is now well known, integrity protection schemes against replay attacks are the most expensive (see, for instance, [10] for a comprehensive survey on memory integrity). They rely either on Merkle hash trees [11], [12] or on data permutations between two consecutive write [8], [7]. Both are expensive in terms of extra memory accesses, increased latency (performance) and memory usage (cost). We apply a Merkle hash tree based integrity protection to RW memory pages only. RO pages are written once and

978-3-9810801-5-5/DATE09 © 2009 EDAA

read many times, so they are not sensitive to replay attacks and their integrity protection relies on addressed Message Authentication Codes (MACs)[1].

We use the same mechanism also for the memory page confidentiality. A strong symmetric block cipher is needed to protect RW pages. This increases the memory latency both on reads and writes and impacts the performance. We use block ciphers to protect the confidentiality of RW memory pages only. The chosen mode is Electronic Code Book (ECB) because it avoids inter-blocks dependency. RO pages are protected with a block cipher in counter mode used as One Time Pad (OTP). The read and OTP computation can be parallelized, thus the performance overhead is reduced and even completely avoided as soon as the memory latency exceeds the OTP computation time. Code pages, most critical to a typical system's performance, are RO pages and thus protected by the lightest techniques and thus SecBus does not have a strong negative impact.

The second mean by which the performance and cost overheads are limited as much as possible is the selective application of security policies. Processes requiring no protection at all have their memory pages unprotected. Security requirements are defined in a very flexible way, on a process and even program segment basis. Confidentiality and integrity are handled separately. Requirements can be expressed by applications if they are SecBus aware or by the OS. They are finally translated into 3 confidentiality modes (None, Block Cipher, One Time Pad) and 3 integrity modes (None, MAC, Hash Tree) per physical memory page. Together with the secret keys, the 2 modes form a Security Policy (SP). Different sections of different processes are assigned different SPs with different secret keys to avoid several weaknesses such as OTP re-use, known plain text attacks, ...

The cryptographic operations are performed by a dedicated hardware module, *SecBus*, located between the on-chip bus and the memory controller, as depicted in Fig. 1. In the following SecBus refers to this hardware module or to the global scheme, depending on the context.

SecBus operates on physical addresses and has no knowledge of the virtual address space. It is configured by the OS through a set of memory mapped control and interface registers. Note that the OS is trusted and so fully responsible for processes isolation. The goal of SecBus is limited to protecting the memory bus against probing attacks; inter-process isolation must be guaranteed by the OS. The next section presents the OS role in this security chain.

III. Memory organization and OS support

Note: keeping in mind the way an OS controls a MMU and its page tables will help understanding the following: there are many commonalities with the control of SecBus. The discussion is kept as generic as possible but, when concrete examples are given, they are based on a 32 bits CPU with two

[1]Addressed MACs protect against splicing attacks by blending the memory address with the data

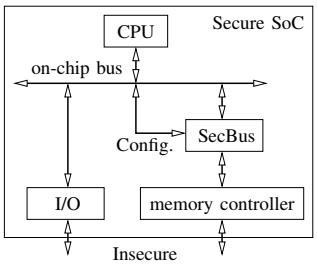

Fig. 1. System Architecture

physical page sizes (4 KBytes and 4 MBytes), 4-ary MACs and hash trees; MACs and hashes are 64 bits and protect four 64 bits double words.

The physical memory page is chosen as the finest protection grain. This eases the implementation, brings security management closer to regular memory management and offers simple solutions to shared memory issues that would otherwise be very complex to deal with. Thanks to this, sharing memory pages between processes usually requires no special processing, unless the process that allocated the page has no or lower security requirements than the sharing one. In this situation, the two SPs are combined to create a new one with sufficient security modes for both processes and the shared page is mutated accordingly.

When a program is loaded by the OS or when a process is created, the OS creates SPs for its different sections. A random generator in SecBus is requested to generate the secret keys. All the memory pages that form a section share the same SP. SPs are stationary and not modified during process execution. An example of SP definition is given below:

```
typedef enum {None, BC, OTP} confidentialityMode;
typedef enum {None, MAC, HT} integrityMode;
typedef struct {
  confidentialityMode CM; // 2 bits
  integrityMode IM;       // 2 bits
  secretKey SK;           // 124 bits
} securityPolicy;
```

When the OS allocates physical memory pages it also associates them a second structure: the Page Security Parameter Entry (PSPE). The primary goal of the PSPE is to map physical addresses to a given SP index. PSPEs are arranged in a hierarchy of tables, as are the page tables of a MMU. As the page tables entries are used by a MMU to translate virtual addresses, PSPEs are used by SecBus to translate physical addresses to SP indexes.

If integrity protection is required for the allocated memory page, another memory page is also allocated to hold the corresponding MAC set or hash tree, unless there is already a free memory area in an existing MAC or hash tree page: with a 4-ary MAC and hash tree structure, a memory page can store 4 different MAC sets or 3 different hash trees; new pages are allocated only when needed. The memory space thus contains 4 different types of pages: regular RO or RW pages, MAC pages and Hash Tree (HT) pages. Each MAC page contains up to 4 MAC sets protecting one RO page each. Each HT page

contains up to 3 hash trees protecting one RW page each.

The second field of the PSPE is the base address of the MAC set, of the hash tree or of the next level PSPE table.

The 3 last fields are one bit flags. The *valid* flag indicates whether the PSPE is valid or not; during the tables walk, if SecBus encounters an invalid PSPE, it raises an interrupt to indicate that a proper SP was not defined for the current physical address and the OS takes the appropriate actions. The second flag indicates the table search termination and allows hierarchical page tables corresponding to different page sizes. The third flag is a type indicator and is used to identify a second type of PSPE: the slave PSPE. Regular (or master) PSPEs are associated regular RO and RW pages while slave PSPEs are associated HT pages and carry the 3 flags plus the root hash value of the HT page (that is, of the 3 hash trees it contains). Fig. 2 depicts the two PSPE structures (in this example the root hash is truncated to 61 bits to fit in a 64 bits double word).

The physical addresses issued by the CPU are used by SecBus to search a valid PSPE in the page tables hierarchy. HT pages are not supposed to be accessed directly by the CPU so an interrupt is raised when a slave PSPE is finally encountered. Else, the found master PSPE is used to fetch and apply the right SP. MAC and HT pages are allocated by the OS but accessed only by SecBus when checking integrity and updating MAC sets or hash trees. When accessing HT pages, SecBus uses the physical addresses to select the slave PSPE that holds the root hash value of the page.

The SPs, the slave and master PSPEs are stored in a dedicated memory area, the Master Block, together with a main hash tree that protects their integrity. In our concrete example, assuming a 4 GBytes physical memory space is protected and only 4 KBytes pages are used, 2^{20} PSPEs at most will be needed, one per physical page. The ratio of master/slave PSPEs among them will depend on the amount of physical memory protected by hash trees. In this example, a PSPE is 8 bytes (64 bits) long, so 8 MBytes are reserved for the PSPE memory area. Adding 4 MBytes for the SPs offers a maximum number of simultaneously active security policies of $4 \times 2^{20}/16 = 2^{18}$ which is far more than actually needed by a real 32 bits system. The main hash tree is thus $(8+4)/3 = 4$ MBytes. All in all, a 16 MBytes Master Block is sufficient to store the SPs, PSPEs and main hash Tree of a 4 GBytes system. The remaining available memory contains fully usable RO and RW pages, plus some MAC pages and HT pages whose number depends on the specific security requirements of the currently running processes[2]. The confidentiality of the 4 MBytes SP area is protected by block cipher because it contains sensitive secret keys while the confidentiality of the 8 MBytes PSPE can be left unprotected. Note: this structure introduces a two-levels hash trees organization: the main hash tree protects slave PSPEs which contain root hash values of HT pages (local hash tree). And the HT pages in turn protect regular RW pages.

[2]Integrity protection requires one MAC page for 4 protected RO pages and one HT page for 3 protected RW pages

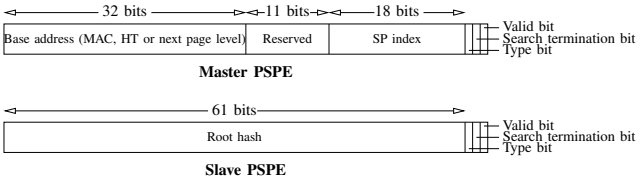

Fig. 2. PSPE definition

Accesses to the Master Block are controlled by a special set of parameters in the Master Security Parameters Group (MSPG). The MSPG is initialized at startup and is stored and locked in SecBus. It contains the root hash value of the main hash tree, the base addresses of the 3 areas and a randomly generated symmetric secret key used to encipher the SP area. The following code snippset gives an example of MSPG structure:

```
typedef struct {
  hashValue RHV;              // 64 bits
  baseAddress PSPE, SP, MHT; // 3x32=96 bits
  secretKey SK;               // 128 bits
} masterSecurityParametersGroup;
```

When applications or legacy OSes run on top of the trusted OS, they may benefit the memory bus protection either in a monolithic or in a self-managed way. Monolithic protection is defined globally for the considered application, on a program segment basis. The application is unmodified and protected by a cooperation between the trusted OS and the SecBus hardware accelerator. It is not protected against software exploits (which is not the purpose of the SecBus scheme) but its exchanges with the external memory are as safe as required by the security policies of its segments. Applications requiring memory bus protection but SecBus-unaware or potentially vulnerable to software exploits will typically be protected this way. The performance will be impacted because some pages will be protected with a more expensive scheme than what would be actually required, but this is the price to pay to protect the memory accesses of such applications.

In the self-managed mode, the application is SecBus-aware and uses the Application Programming Interface (API) exported by the trusted OS to manage the protection of its own memory pages. The calls to this API go through the trusted OS because allowing an untrusted application to directly access the SecBus hardware module would compromise the security of the whole platform. In this mode of operation, if an application is compromised by a software exploit, there is no guarantee that its memory pages remain confidential and that their integrity is preserved: the exploit could require protected pages to be mutated as unprotected and there would be no way for the trusted OS to detect this situation. As a consequence, only deeply verified applications should run in this mode.

IV. The SecBus hardware module

As explained in the previous section, the OS is responsible for the high level management of the security policies and

parameters for the allocated memory pages. All the cryptographic operations (enciphering, deciphering, MAC, hash, random generation, ...) are performed by the dedicated SecBus hardware accelerator. SecBus is responsible for the low level management of the master block, the HT pages and the MAC pages. Every access to these memory areas is performed by SecBus on behalf of the OS. The OS issues high level commands to SecBus which implements them as sequences of low level computations and memory accesses. For instance, when a new security policy is needed, the OS provides SecBus with a SP index, an integrity mode and a confidentiality mode and SecBus generates the key material and stores the new policy in the master block.

The main functionalities implemented by SecBus are confidentiality and integrity management but it also provides additional services like random number generation, initialization of hash trees and MAC sets, secure storage of the sensitive cryptographic material, plus some specific services dedicated to the startup phases. Despite all these capabilities, without any further optimization, the mechanisms we presented up to now would lead to significant performance degradation on every processor cache miss. In order to mitigate the performance degradation, 4 caches are embedded in SecBus to store recently used SPs, PSPEs, hashes and MACs. Thanks to these 4 caches, the security parameter search is accelerated and many external memory accesses and cryptographic computations are avoided. Their design largely relies on performance simulations. A simulation platform, based on a ARM Instruction Set Simulator, and running various applications, from standalone processes to applications on top of a Linux kernel, is currently designed and used to evaluate various cache architectures.

The MAC cache is a very classical one. Its main characteristics (size, architecture, write and replacement policies) must be adapted to the specificities of MAC accesses but it needs no unusual customization.

Hash caches architectures have already been studied in the literature ([12]). The main issues with hash caches are related to the relations between nodes of a tree. Write-through caches are rather straightforward but their efficiency is limited to shortening the hash tree check operations: the checking stops at the first cache hit. Write-back hash caches shorten the hash trees update too but they leave the cache in an inconsistent state: marking a hash dirty in the cache implies that all the other nodes depending on it must be considered outdated, in cache or in external memory. The design of a hash cache is thus a trade-off between hardware cost, complexity and efficiency. For performance reasons we are using a write-back hash cache despite its intrinsic complexity. The hash cache handles the internal dependencies between cached nodes.

PSPE and SP caches, like MAC cache are much simpler than hash caches but they are also much closer to the Translation Lookaside Buffer (TLB) of a MMU than to a regular cache. Their number of entries is thus limited. The optimal write policy depends on the frequency of cache writes: a write-through cache increases the rate of write accesses to the Master Block. Updating the Master Block is expensive, mainly because it forces a main hash tree update. If it is infrequent enough it could be negligible. But if cache writes are frequent events the implementation overhead of write-back caches could be worth the effort. System level simulations will help evaluating both options.

V. Conclusion and future work

We presented a page-based memory bus protection scheme. Its design targets a constrained context where the modifications of CPU and software tool chain are considered impractical. The impact of the security on the cost and performance is optimized by different algorithmic means and hardware caches. The trusted OS is largely involved in the security management. This scheme is a promising, cost effective solution. A simulator is being built to rigorously evaluate the performance/complexity trade-offs of the different caches. In the near future, simulation results will provide valuable inputs and will drive architecture choices. Meanwhile, a formal model of the security management unit will be designed and its correctness will be checked against security properties. In the long run, a hardware prototype will be designed, an OS will be selected and adapted and a complete demonstrator will be built to demonstrate that a true trusted computing platform is realizable at costs and with performance compatible with consumer markets.

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] M. Kuhn, "Cipher Instruction Search Attack on The Bus-Encryption Security Microcontroller DS5002FP," *IEEE Transactions on Computers*, 1998.

[2] A.B. Huang, "Keeping Secrets in Hardware: The Microsoft XBox Case Study," *MIT, AI Memo*, 2002.

[3] T. Gilmont and J.D. Legat and J.J. Quisquater, "Enhancing Security in the Memory Management Unit," in *EuroMicro*, 1999.

[4] G.E. Suh and D. Clarke and B. Gassent and M. van Dijk and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," in *ICS*, 2003.

[5] D. Lie and C. Thekkath and M. Mitchell and P. Lincoln and D. Boneh and J. Mitchell and M. Horowitz, "Architecture Support for Copy and Tamper Resistant Software," in *ASPLOS*, 2000.

[6] D. Lie and J. Mitchell and C. Thekkath and M. Horowitz, "Specifying and Verifying Hardware for Tamper-Resistant Software," in *IEEE Symposium on Security and Privacy*, 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 and T.Zhang and S.Pande, "HIDE: an Infrastructure for Efficiently Protecting Information Leakage on the Address Bus," in *ASPLOS*, 2004.

[9] X. Chen and R.P. Dick and C. Alok, "Operating System Controlled Processor-Memory Bus Encryption," in *DATE*, 2008.

[10] R. Elbaz and D. Champagne and R.B. Lee and L. Torres and G. Sassatelli and P. Guillemin, "TEC-Tree: a Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks," in *CHES*, 2007.

[11] R.C. Merkle, "Protocols for Public Key Cryptography," in *IEEE Symposium on Security and Privacy*, 1980.

[12] B. Gassend and G.E. Suh and D. Clarke and M. van Dijk and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," in *HPCA*, 2003.