SecBus, a software/hardware architecture for securing external memories

Jeremie Brunel
CNRS LTCI
Télécom Paristech
Institut Mines-Télécom
Sophia-Antipolis, France
brunel@telecom-paristech.fr

Salaheddine Ouaarab
CNRS LTCI
Télécom Paristech
Institut Mines-Télécom
Sophia-Antipolis, France
ouaarab@telecom-paristech.fr

Renaud Pacalet
CNRS LTCI
Télécom Paristech
Institut Mines-Télécom
Sophia-Antipolis, France
pacalet@telecom-paristech.fr

Guillaume Duc
CNRS LTCI
Télécom Paristech
Institut Mines-Télécom
Paris, France
duc@telecom-paristech.fr

***Abstract*—Embedded systems are ubiquitous nowadays. In many cases, they manipulate sensitive applications or data and may be the target of logical or physical attacks. On systems that contain a System-on-Chip connected to an external memory, which is the case of numerous medium to large-size embedded systems, the content of this memory is relatively easy to retrieve or modify. This attack can be performed by probing the memory bus, dumping the content of the memory using a memory analyzer or by exploiting flaws in DMA-capable devices. Thus, if the embedded system manipulates sensitive applications or data, the confidentiality and the integrity of data in memory shall be protected. SecBus is a combined hardware/software architecture that guarantees these two security properties. This paper describes the different software components that are in charge of the management of the SecBus platform, from the early initialization to their use by the sensitive applications.**

I. Introduction

Since the internal security mechanisms of the Xbox Microsoft game console has been compromised [1], the protection of the content of external memories has become an important research topic. The memory buses are vulnerable to probing attacks. These attacks, while sometimes challenging, are less expensive and complex than on-chip probing. They threaten both the confidentiality and the integrity of the data transmitted on the bus.

The XOM [2] project protects the program execution in the processor both in integrity and confidentiality. In XOM, the operating system and the external memory are not considered as trustworthy. The trust zone is limited to on-chip operations. New instructions are added to the processor to protect the runtime environment. Modifications within the processor and the software toolchains are required. The integrity protection is based on Message Authentication Codes (MAC) only; as a consequence, replay attacks[1] cannot be detected.

CryptoPage [3]–[5], a project similar to XOM, also adds new processor instructions to secure the program execution. It is robust against replay attacks and does not rely on a trusted OS. However the processor and the toolchain must be modify.

MESA [6] is based on a security architecture that protects software in confidentiality and integrity. MESA ties cryptographic properties and security attributes to memory instead of individual user processes. Various virtual memory segments are associated with different security protections.

The PE-ICE [7] hardware cryptographic engine is dedicated to the protection against board level probing attacks. The scheme consists in a monolithic protection of the whole memory. The software is not involved in the management of the security of the memory bus. The main drawbacks of PE-ICE are that it is hardly scalable to large external memory and the performance overhead applies even on non-critical data.

The AEGIS project [8], [9] aims at building a secure execution environment to protect software processes from memory bus attacks and from each other.

In this paper, we present a solution to ensure the security both in confidentiality and integrity of the memory bus (and so the memory itself). The goal is to guarantee a fully secure boot of the platform and to insure that no adversary has compromised the system during its loading. We propose a complete software architecture to ensure, after start-up, that the executed applications will be correctly protected and no adversary can tamper with their instructions or data.

The rest of this paper is organized as follows. Section II gives the main characteristics of the SecBus hardware module and the security model; section III defines the SecBus hardware architecture; in section IV, we describe the specific software architecture of SecBus; section V presents some results and analysis based on SystemC simulation.

II. SecBus Aims

In the SecBus project the approach is rather different than in the above-mentioned related works. Four strong requirements have been set:

- the SecBus architecture is scalable and the size of the protected external memory is not be limited by SecBus,
- the processor unit does not require modifications: the acceptability of solutions relying on such modifications is usually low; the software development kit does not require modification,
- the software applications that are not SecBus-aware run natively on the system: software which does not require security is left unmodified,

[1] A replay attack consists in substituting a data by an older, outdated one

- the SecBus granularity is thin enough to have time penalty only when security is required

A. Threat model

Adversaries can be very different in terms of equipment, money, skills and time, from absolute beginners to large criminal organisations or governments. The techniques used to reach the security objectives depend on the considered attackers. SecBus considers adversaries with complete physical access to the target embedded system, except the internals of the main SoC: SecBus does not deal with on-chip hardware attacks (side channels, fault attacks, on-silicon probing). On-silicon attacks are very difficult to defeat but they usually require complex and sophisticated equipments; it is thus unlikely that they are used against low to medium value secrets like the ones found in game consoles, set top boxes, Internet Services Provider's boxes... Moreover, a lot of research works address these threats with completely different and complementary counter-measures. SecBus follows a modular approach, where each problem is dealt with as independently as possible from the others.

On the software side SecBus considers embedded systems on which arbitrary applications can be loaded and launched, even by adversaries.

With a complete physical access to the hardware and software components the adversary is able to spy and control the system (data injection and modification, communication interception, applications execution, etc.) by using either purely software exploits or hardware attacks or a combination of the two. A typical example of combined attack could be the attacker populating as much external memory as allowed with custom code and probing the memory bus to flip an address bit while the embedded system runs in the highest privileged mode, forcing the embedded system to run the custom code in privileged mode and leading to a privilege escalation.

SecBus is a countermeasure against:

- on-board probing of the external memory bus (logic analyzer...)
- physical attacks on the memory components (cold boot [10], ...)
- software exploits of DMA capable peripherals (NIC, HDD, DVD, CDrom, USB, PCI_e, GPU...)

B. Requirements

The SecBus architecture shall protect the communication between the processor unit and the external memory. That is why we need an hardware support: we need to add a hardware module intercepting the read/write operations on the bus between the processor and the memory controller. This module must be as autonomous and transparent as possible. The Hardware Security Manager (HSM) shall embed cryptographic engines to encipher and decipher the written and read data, to check the integrity of read data and, on write accesses, to compute the digests against which the integrity is checked. Fig. 1 represents a SoC with its HSM. The HSM is driven by its software counterpart: the Software Security Manager (SSM).

A SecBus-equipped SoC

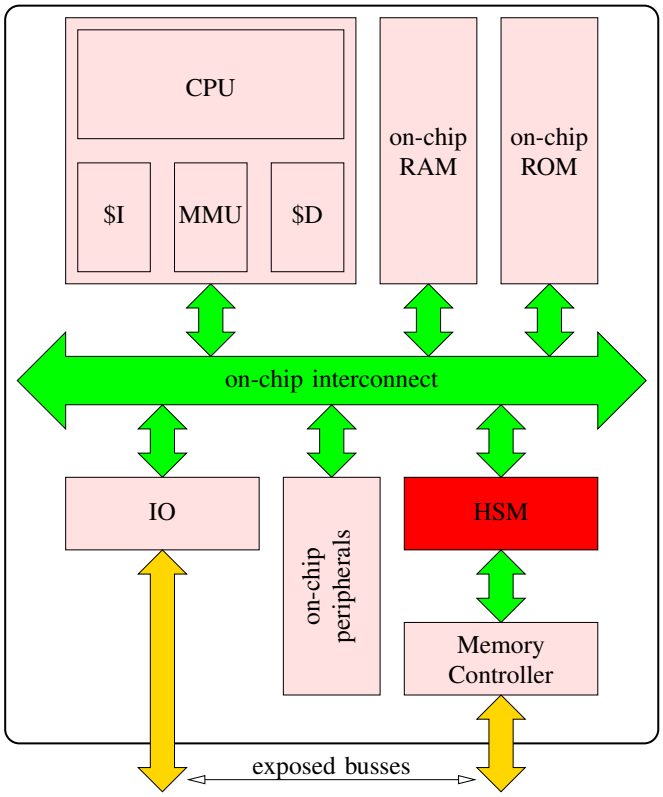

Figure 1: The HSM in its host SoC

III. SecBus Hardware architecture

A. Principles, memory organization

The model of cooperation between the SSM and the HSM is similar to that between a OS memory manager and a Memory Management Unit (MMU): depending on the security requirements of the different software components, the SSM dynamically manages Security Policies (SP) defining confidentiality and integrity protection levels and tables of Page Security Parameter Entries (PSPE) that bind every physical memory page to a Security Policy (SP). These SPs and PSPEs are stored in the Master Block (MB), an external memory area. Upon memory accesses the HSM performs a PSPE table walk, fetches the corresponding SP and runs the specified security algorithms (encryption, decryption and/or integrity protection and checks). This clean and simple organization eases the design of the SSM and of the HSM. It also allows to protect only sensitive memory pages, with the best available cryptographic primitives for the defined security policy.

In order to optimize performance and cost Read-Only (RO) and Read-Write (RW) memory pages are handled differently. When confidentiality is required, RO pages are enciphered with a block cipher in Counter (CTR) mode, because it allows the parallelization of the cryptographic computations

and the memory latency on read accesses, while RW pages are protected in Cipher Block Chaining (CBC) mode because the CTR mode is not applicable to modifiable data. Similarly, integrity is checked against CBC-MACs for RO pages while MAC trees are used for RW pages. The SP bound to each memory page packages a confidentiality mode (None, CTR, CBC), an integrity mode (None, CBC-MAC, MAC tree) and secret keys.

In addition to the Master Block, when integrity protection is required on a memory page, extra security-related information is stored in external memory: a set of CBC-MACs for RO pages or a Merkle MAC tree for RW pages. They are stored in dedicated memory pages, allocated by the SSM. The memory space thus contains four different types of pages: regular RO or RW pages, pages of CBC-MAC sets and pages of Merkle MAC trees. Finally, the SP and PSPE tables stored in the Master Block are integrity-protected by a dedicated Merkle MAC tree, the Master MAC Tree (MMT) that is a part of the Master Block. Its root is stored inside the SoC and never leaves it.

B. Internal Architecture

The internal architecture of the HSM is composed of five blocks and several registers. Figure 2 illustrates these different components.

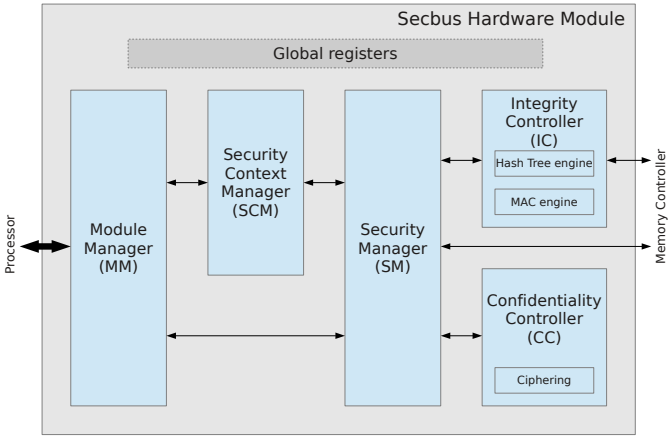

Figure 2: Hardware Security Module architecture

- The Module Manager (MM): regulates the memory accesses and requests the Security Context Manager and the Security Manager.
- The Security Context Manager (SCM): searches the security context bound to the requested memory page. It is able to browse the different PSPE levels to retrieve the right SP corresponding to the physical address.
- The Security Manager (SM): supervises the security protection using different cryptographic operations. t is the central block of the HSM, and it manages the different read and write accesses to and from the MB and the useful memory pages.
- The Integrity Controller (IC): applies the integrity protection, MAC and MAC trees.
- The Confidentiality Controller (CC): applies the confidentiality protection, CTR and CBC.

The HSM works as follows: when a memory access is issued by the processor, MM requests SCM to retrieve the SP for the target memory page. After a PSPE tables walk, the SCM fetches the SP and stores it in internal registers. Afterwards, MM passes the memory address requested by the processor to SM. According to the security parameters (SP), SM uses IC (if integrity is required) and/or CC (if confidentiality is required). SM passes back the result to MM which responds the processor.

C. Data structure

The SecBus specific data structures (SP and PSPE tables) are initialized and updated under supervision of the SSM, with the help of a small set of HSM operations. Like an MMU embeds a Translation Lookaside Buffer (TLB) to cache a small number of recently used page table entries, the HSM embeds small caches to store recently used PSPEs and SPs. The HSM is capable of walking through the SP and PSPE tables, without assistance from the SSM, when the information required to process a regular memory access is not available in its internal caches.

1) Security Policies: A SP contains:

- A `cnfkey` confidentiality secret key used for enciphering and deciphering in counter mode when $SP.cnfmode = ctr$ or in CBC mode when $SP.cnfmode = cbc$.
- A `intkey` integrity secret key used for CBC-MAC computations when $SP.intmode = mac$ or $SP.intmode = mactree$.
- A `cnfmode` confidentiality mode indicator ($cnfmode \in \{none, ctr, cbc\}$).
- A `intmode` integrity mode indicator ($intmode \in \{none, mac, mactree\}$).
- A `valid` boolean flag ($valid \in \{false, true\}$) indicating whether the SP is usable or not. The HSM raises an interrupt when a memory access is performed in a page bound to a SP with $SP.valid = false$.

The SPs are organized as an array and stored in a contiguous memory region protected in confidentiality and integrity by the MMT. Their number is implementation dependent. In other data structures they are referred to by their index. The SP stored at the lowest address in memory has index 0. Their bit-width depends on the key lengths, that is, on the selected block cipher. If the resulting bit-width is not a multiple of a CBC chain, it is extended to the next multiple by zero padding. The listing 1 gives an example of SP data structure for a SecBus architecture using the DES-X block cipher and CBC chains of four blocks (256 bits). It is zero-padded to $2 \times 256 = 512$ bits.

```
1 typedef struct {
2   uint139_t dummy = z139;
3   uint64_t  cnfk1;    // Conf. DES-X key #1
4   uint56_t  cnfk2;    // Conf. DES-X key #2
```

```
  uint64_t  cnfk3;    // Conf. DES-X key #3
  uint64_t  intk1;    // Int. DES-X key #1
  uint56_t  intk2;    // Int. DES-X key #2
  uint64_t  intk3;    // Int. DES-X key #3
  uint2_t   cnfmode;  // Conf. mode (NONE/CTR/CBC)
  uint2_t   intmode;  // Int. mode (NONE/MAC/MACTREE)
  uint1_t   valid;    // Valid flag (TRUE/FALSE)
} SP512;
```

Listing 1: The SecBus SP data structure

2) Page Security Parameters Entry: The PSPEs are not exactly organized in a hierarchy of tables as MMU page table entries. Instead they are all present, in a frozen, natural order, and they embed a size indicator that is used to decide which page size they are associated with. In a system with two page sizes, 4kB and 4MB, for instance, a PSPE corresponding to a page aligned on a 4MB boundary can be that of a 4kB page (the 4MB page is broken in 1024 4kB pages) or of the 4MB page (the 4MB page exists). The size indicator disambiguates this. All other PSPEs can only be 4kB pages PSPEs because they correspond to non 4MB-aligned memory pages. PSPEs also carry a `valid` flag indicating whether they are valid or not. Note that, in order to be valid, a PSPE must not be in the scope of a PSPE of a large page that is also valid and that would take precedence.

There are two different PSPE formats: master and slave. Master PSPEs bind regular memory pages to SPs. When integrity is required they point to a MAC set or MAC tree page (and to a specific MAC set or MAC tree in the page). Slave PSPEs are associated to the MAC set or MAC tree pages. For MAC tree pages they contain the root of the MAC trees. The listing 2 presents an example of master and slave PSPE data structures.

```
typedef union {
  MASTER_PSPE64 m;
  SLAVE_PSPE64  s;
} PSPE64; // 64 bits

typedef struct {
  uint20_t msmtadd; // MAC set or tree page
  uint2_t  msmtidx; // MAC set or tree index in page
  uint20_t ivadd;   // IV set page
  uint2_t  ividx;   // IV set index in page
  uint16_t spidx;   // Security Policy index
  uint2_t  size;    // Page size
  uint1_t  prot;    // Protected flag (TRUE/FALSE)
  uint1_t  valid;   // Valid flag (TRUE/FALSE)
} MASTER_PSPE64;

typedef struct {
  MAC63_WITH_NULL mac;
  uint1_t valid; // Valid flag (TRUE/FALSE)
} SLAVE_PSPE64;
```

Listing 2: The SecBus PSPE data structures

Walking in the PSPE tables consists in fetching first the PSPE of the largest possible page size in which the requested address could fall, and checking its size indicator and valid flag. If it is valid and its size indicator extends up to the requested address, the search is over. Else, fetch the PSPE of the next largest possible page size,... until a PSPE is valid and its size indicator extends up to the requested address.

IV. Software Architecture

The aim of this section is to detail the software part of the SecBus architecture: how the bootloader guarantees that the SSM is properly loaded and how applications interact with the SSM.

A. Bootloader

The bootloader is the first software component to be launched when the system is powered up. In the SecBus architecture, the bootloader initializes the HSM, loads, checks and starts the kernel (and the SSM). It is a critical element in the security of the platform because it is responsible for verifying that the right SSM is loaded. An adversary who would be capable of replacing the SSM with her own piece of software could, for instance, configure the HSM to apply no protection at all to the whole memory and thus bypass the security.

At this stage, the HSM is not configured yet, so it cannot protect the code of the bootloader and the data it manipulates against an adversary. The bootloader must thus be executed entirely on-chip where, by hypothesis, it can not be tampered with. The code of the bootloader is stored in a non-volatile memory inside the SoC (ROM or flash). As the external memory is not yet protected, the bootloader also uses a small internal RAM to store the data it manipulates.

For all these reasons this tiny software stack is critical and difficult or even impossible to replace; it must be flawless.

The first task of the bootloader is to initialize in external memory the data structure used by the HSM: the Master Block (MB). It marks all the Page Security Policy Entry[2] (PSPE) and SP invalid. It requests the HSM to build the root MAC tree that protects the integrity of the MB: the Master MAC Tree.

Next, the bootloader prepares the external memory pages in which the kernel is to be loaded and configures the HSM to protect them: it creates a SP with integrity protection and configures the PSPEs to bind it to the memory pages.

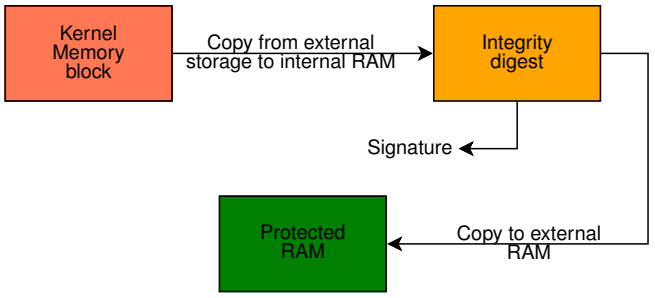

Figure 3: Kernel block's path during its initialization

The bootloader then loads the kernel from a non-volatile, external, mass storage (flash, ROM, hard drive, network, etc.) in the external memory (as shown in Fig. 3); it computes on the fly the kernel's signature. The computation of this signature is performed either in software (flexible) or using

[2]the data structure that binds a SP to a memory page

the cryptographic primitives exported by the HSM (faster but less flexible). This signature is used to verify that the kernel (and so the SSM) has not been tampered with by an adversary. An important point is that it must be possible to update the kernel (for instance to fix a flaw in the SSM) and it must be impossible for an adversary to use an old version of the kernel after an update (downgrading). Two options can be used to guarantee this boot to boot integrity:

- The signature can be compared with a reference signature stored in a non-volatile memory in the SoC. This reference signature is updated when the kernel is updated so it always contains the reference signature of the latest version of the kernel. This solution is the one currently implemented.
- The bootloader can communicate with an external trusted entity (smart-card, network server, etc.) to check whether the signature is correct or not. In this case the protocol used between the bootloader and the trusted entity must allow the bootloader to authenticate the trusted entity and must be protected against replay attacks. This solution increases the footprint of the bootloader because it must embed, for instance, a secure network stack. As the bootloader is stored and executed on chip this has an impact on the hardware cost.

Once the kernel is verified and loaded into a protected area in the external memory, the bootloader passes the control to the kernel.

The boot procedure ensures that an adversary cannot load a modified version of the SSM (due to the signature and verification mechanism) or tamper with it during the boot (the kernel and the SSM are stored in an area of the memory protected in integrity by the HSM).

B. Software Security Manager

The SSM manages the configuration of the HSM according to the security needs of the operating system and the applications. On the Fig. 4, we introduce the software architecture of SecBus. The left part of the scheme presents a regular software architecture, on the right part we show the new modules required for a fully SecBus compliant software system. The applications loader and the memory manager of the operating system need to be modified. Some security features are exported for applications with a new user API.

1) Application interface: When an application is not designed to use the SecBus architecture, all its pages are protected using a default Security Policy. This default policy is configured by the SSM and must be adapted to the specific system's needs. For instance, the default policy could be to protect everything in confidentiality and integrity (conservative) or nothing at all (performance).

The application's designer knows which Security Policy is required by the different parts of her application. On behalf of the application, the SSM binds the application's memory pages to the right Security Policies through a dedicated system call. When receiving this system call, the SSM configures the HSM to apply the given Security Policy to the newly allocated

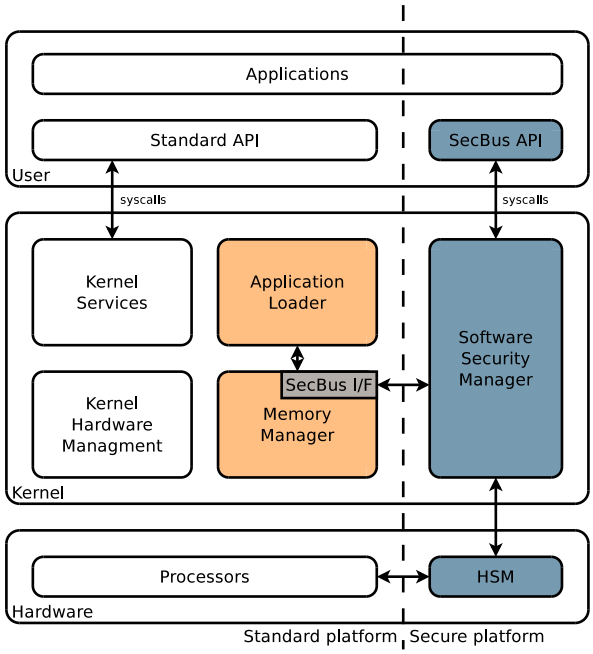

Figure 4: Global SecBus Software architecture

page(s). So the interface between the applications and the SSM is small and simple.

Some software mechanisms are more difficult to handle in the context of the SecBus architecture. One of these mechanisms is dynamically-linked libraries. Applications often use the same libraries (at least the standard C library for instance). This classical mechanism ensures that the code of a given library is loaded only once in memory and that all applications that need this library use this single instance thanks to virtual memory. In the SecBus architecture, when two applications need the same library but with two different security policies (for instance one application needs no security and the other needs integrity), several solutions are possible:

- The first solution is not to use dynamically-linked libraries in applications that have security needs, but only statically-linked libraries.
- The second solution is to load in memory several instances of the same library, each one with a different security policy. For example, one instance with no security and one instance with integrity protection. Applications that do not need security use the first instance while applications that need integrity protection use the second instance.
- A last solution is to load only one instance of the library in memory and to apply it the maximum security policy required by the different applications. For instance, if one application needs no security and one needs integrity, the security policy applied to the library would be integrity.

The choice between these three solutions depends on a trade-off between performance and the memory footprint.

The same holds with several Inter-Process Communication

(IPC) mechanisms such as shared memory.

2) Applications loader: When an application is loaded, its different memory segments are stored in the external memory. In our case, the application designer would like to add some indications about the security policy to apply on each segment.

We distinguish three kinds of applications in our point of view:

- SecBus unaware applications
- Applications with static security requirements: the provider knows which security policy to apply to which memory segments. The security requirements are added after design, as a set of meta-information.
- Fully SecBus-aware applications, with well defined, dynamic security requirements.

One of our requirements is that a regular application can be executed on a SecBus-equipped system without any transformation. We provide default security policies for SecBus unaware applications.

In order for the system to enforce the right security policy, the application loader as to read the policy from the application binary file.

The next subsection deals with the memory used during software execution.

3) Dynamic Memory: For some data, the security needs can be different. With the SecBus architecture software designer can select a different security policy for each allocated memory block.

During its execution, a program will use a certain amount of memory. In most cases it relies on a stack (e.g. used for function calls or when the amount of registers is exceeded) but also a heap, for dynamical memory allocations.

As the main stack is created before the application execution, it should be protected during the loading of the application. Each stack allocated for a thread creation can be protected with the security policy selected by the software designer.

The software designer can select a default policy to be applied on each memory block allocated during the software execution life. As each thread can manipulate different kind of data, it is also useful to apply a default security policy for each entire thread. For some data, the security needs can be different, the software designer should be able to select a different security policy for each allocated memory block.

V. Experiments and Results

Results were obtained from simulations with a SystemC [11] model. The complete hardware system has been built with SocLib [12]/SystemC. The initial SecBus model from [13], [14] has been deeply reworked to make it usable with our Software Security Module.

SocLib provides hardware bricks as processors (based on ISS[3] model), cache controller, on-chip bus, memories, IO controllers and so on. Those bricks are based on the CABA[4] modeling rules.

[3]Instruction Set Simulator
[4]Cycle Accurate Bit Accurate

The scheme of the hardware platform used for those simulations is given by Fig. 5. The processor used is a Mips32, so we assume that the platform is in 32 bits. The memory controller does not exist in SocLib, the RAM access are modeled using read/write on the VCI protocol with a given latency. The external memory flash contains the kernel. The external harddrive contains a filesystem with user applications.

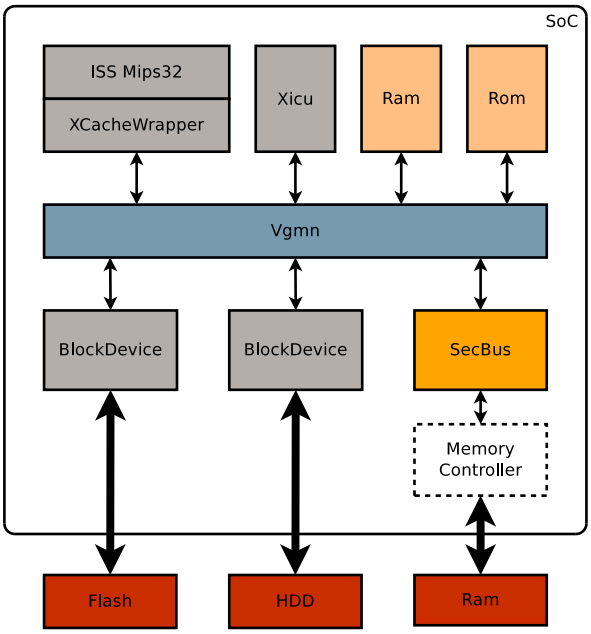

Figure 5: SystemC/SocLib platform with HSM

A. Secure Bootloader

Before the external memory is protected, every software operations has to be done on the internal RAM. So the first thing to do is to defined the stack pointer on the internal RAM. The next operation is to initialize the hardware drivers ; in our case, the drivers are:

- Block device driver - used for the kernel storage
- TTY device driver - used to print information/debug
- HSM driver

After this initialization we load the first kernel memory block in the internal RAM, we compute its signature.

In order to minimize the bootloader memory footprint, we did not used an ELF binary format. We wrote a short header format which describes the different memory sections of the kernel, four are necessary:

- instructions
- RO datas
- RW datas
- stack

This information is written in the first 256 bits memory block of the kernel binary. The security policy linked to those segments are defined in the bootloader, they can not be changed after writing the bootloader in the internal ROM.

After reading this security information, we properly protect the external RAM segments used to store the kernel. Then we read each memory block of the kernel stored in the ROM, we compute the kernel signature and we copy it on the protected external RAM.

For our tests, we used a maximum protection as the software security module can manipulate cryptographic tools (encryption keys, authentication keys), we assume that RO segments are encrypted with a OTP algorithm, protected in integrity by MAC set and RW segments are encrypted in CBC mode and protected in integrity by MAC Trees.

1) Memory footprint: We have simulated the SecBus platform and executed the bootloader with three different integrity systems, two with software (sha256 and DES-X) and one with the HSM coprocessor. On the Fig. 6, we evaluate the memory needed to store the bootloader on the ROM; on the Fig. 7, we evaluate the internal RAM usage.

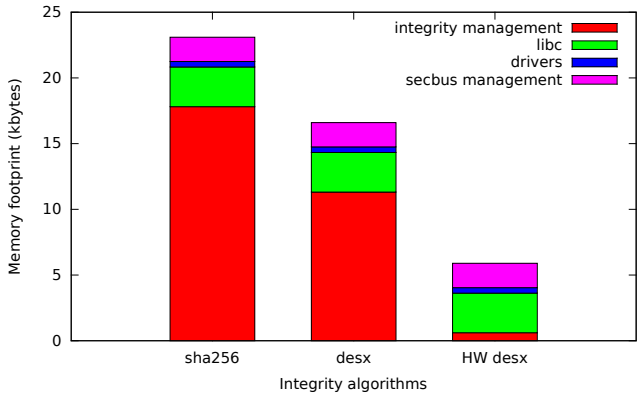

Figure 6: Bootloader memory footprint

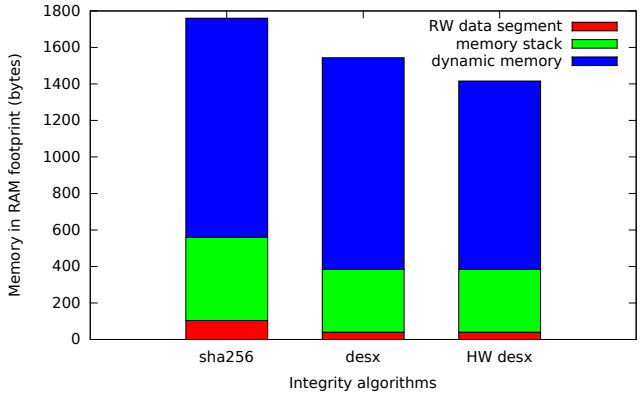

Figure 7: Bootloader memory footprint during execution on internal RAM

The solution with hardware integrity computation is less expensive in term of memory storage, no software algorithm is used (only calls to the HSM) and no data are required for integrity computation.

This bootloader fits in a ROM of 8kB when using the HW integrity checking otherwise it can fit in a ROM of 32kB (with the software integrity checking). The internal RAM needed to properly execute the bootloader is a 2kB RAM (the Xenon processor of the XBox 360 has 64kB of SRAM).

B. Software Security Module

The Software Security Module works with the operating system kernel. We selected Mutekh [15] kernel for several reason: it is a small, very modular kernel (called hexo kernel), it has the support for SocLib platform (CPU and implemented IPs in SocLib project).

1) Application loader: For applications with security requirements a meta-section is added to the ELF[5] file. The ELF format is flexible as it has been designed to store various kind of sections. Some of the sections are loaded in memory, other sections contains meta information like symbol names, additional sections can be freely added without disturbing tools and loaders.

In our case, a `.secbus` named section defines the security policies to apply to the other loadable sections. The SecBus aware application loader is responsible for the parsing of this additional meta-section.

This section contains a list of entries which associates a section name to a 4 bits security policy value. Bits 0 and 1 encode the confidentiality policy, possible values are: *none*, *OTP* or *CBC*. Bits 2 and 3 encode the integrity policy, possible values are: *none*, *MAC* and *MAC tree*.

2) User API: The user API exposed for the application development is defined by the listing 3.

```
1  enum secbus_integrity_mode_e
2  {
3    // no integrity protection
4    secbus_i_no,
5    // integrity by MAC (for RO datas)
6    secbus_i_mac,
7    // integrity by HashTree (for RW datas)
8    secbus_i_ht
9  };
10
11 enum secbus_confidentiality_mode_e
12 {
13   // no confidentiality
14   secbus_c_no,
15   // confidentiality with a One Time Pad scheme (for
         RO datas)
16   secbus_c_otp
17   // confidentiality with a Cipher Block Chaining
       scheme (for RW datas)
18   secbus_c_cbc,
19 };
20
21 enum secbus_type_request_e
22 {
23   // default policy apply for the entire process
24   secbus_default_policy,
25   // policy apply on the next creating thread stack
26   secbus_thread_stack_policy,
27   // policy apply on each allocated memory block in
       a thread
28   secbus_thread_policy,
```

[5]Executable and Linkable Format. This format is used in every Unix-like Operating System.

```
  // policy apply on the next allocated memory block
  secbus_next_policy
};

/** @secbus_create_policy generates a new secbus
    policy */
secbus_err_t secbus_create_policy(
  enum secbus_integrity_mode_e i,
  enum secbus_confidentiality_mode_e c,
  secbus_sp_t *sp
);

/** @secbus_policy_request adds a new security
    request for the current execution context */
secbus_err_t secbus_policy_request(
  secbus_sp_t sp,
  enum secbus_type_request_e type
);
```

Listing 3: The SecBus user API

This API is clear and simple and does not require deeply modifications on applications with security needs. For a single application without thread, you can create and apply, for example, one security policy by adding this routine:

```
secbus_err_t err;
secbus_sp_t my_policy;
if (err = secbus_create_policy(secbus_i_ht,
      secbus_c_no,
      &my_policy))
  return err;

if (err = secbus_policy_request(my_policy,
       secbus_default_policy))
  return err;
```

Listing 4: Short example for applying a security policy

VI. Conclusion

In this paper we presented a complete secure software architecture for embedded systems with external memory. The security is assured during the whole execution time: to the startup of the platform up to the user applications execution. This security is based on different modules. At the beginning, the bootloader insures the correct configuration of the hardware security module and verifies the software first level integrity. The kernel interacts with the Security Software Module: when it has a need of security for applications loading or for the memory manager. The user applications can also interact with the SSM via the user API and therefore can use the security properties of the HSM. The next works are: to formally prove that both the bootloader and the SSM are flawless, as for now we have only done simulations and we have validated our approach, we need an hardware implementation to perform real performance measurements. The mutekh kernel is small and easy to apprehend, the next step is to replace it with a Linux kernel to have a complete software stack and to perform performance measurements on a concrete system.

VII. Acknowledgment

This work was supported in part by the ENIAC Joint Undertaking, within the Trusted Computing for European Embedded Systems (TOISE) project, call ENIAC-2010-1, proposal n° 282557-2. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement TRESCCA n° 318036.

References

[1] A. Huang, "Keeping secrets in hardware: The microsoft xbox™ case study," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '02. London, UK, UK: Springer-Verlag, 2003, pp. 213–227. [Online]. Available: http://dl.acm.org/citation.cfm?id=648255.752707

[2] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, Oct. 2000, pp. 168–177.

[3] C. Lauradoux and R. Keryell, "CryptoPage-2 : un processeur sécurisé contre le rejeu," in *Symposium en Architecture et Adéquation Algorithme Architecture (SympAAA'2003)*, La Colle sur Loup, France, Oct. 2003, pp. 314–321.

[4] R. Keryell, "CryptoPage-1 : vers la fin du piratage informatique ?" in *Symposium d'Architecture (SympA'6)*, Besançon, Jun. 2000, pp. 35–44.

[5] G. Duc and R. Keryell, "CryptoPage: an efficient secure architecture with memory encryption, integrity and information leakage protection," in *Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC'06)*. IEEE Computer Society, Dec. 2006, pp. 483–492.

[6] W. Shi, C. Lu, and H.-H. S. Lee, "Transactions on high-performance embedded architectures and compilers i," P. Stenström, Ed. Berlin, Heidelberg: Springer-Verlag, 2007, ch. Memory-Centric Security Architecture, pp. 95–115. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-71528-3_7

[7] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet, and A. Martinez, "A parallelized way to provide data encryption and integrity checking on a processor-memory bus," in *Proceeding of the 43rd ACM/IEEE Design Automation Conference*, Jul. 2006, pp. 506–509.

[8] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th International Conference on Supercomputing (ICS'03)*, Jun. 2003, pp. 160–171.

[9] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and implementation of the Aegis single-chip secure processor using physical random functions," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE Computer Society, Jun. 2005, pp. 25–36.

[10] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Cal, A. J. Feldman, and E. W. Felten, "Least we remember: Cold boot attacks on encryption keys," in *In USENIX Security Symposium*, 2008.

[11] "Systemc website." [Online]. Available: http://www.accellera.org

[12] "Soclib website." [Online]. Available: http://www.soclib.fr

[13] L. Su, "Confidentialité et intégrité du bus mémoire," Ph.D. dissertation, Télécom ParisTech, Mar. 2010.

[14] L. Su, S. Courcambeck, P. Guillemin, C. Schwarz, and R. Pacalet, "Secbus: Operating system controlled hierarchical page-based memory bus protection," in *Design Automation & Test in Europe (DATE 2009)*, Apr. 2009, pp. 570–573.

[15] "Mutekh website." [Online]. Available: http://www.mutekh.org