The $\mathcal{S}ec\mathcal{B}us$ architecture

J. Brunel and G. Duc and S. Ouaarab and R. Pacalet

October 22, 2014

Contents

1 Introduction **1**
- 1.1 Notations 1
- 1.2 Threat model 2
- 1.3 Security objective 2
- 1.4 Architecture design 3

2 Cryptographic primitives **7**
- 2.1 Notations 7
- 2.2 Cryptographic primitives 7
  - 2.2.1 Confidentiality 7
  - 2.2.2 Integrity 10
- 2.3 The cryptographic primitives of $\mathcal{S}ec\mathcal{B}us$ 14
  - 2.3.1 Enciphering and deciphering in counter mode 14
  - 2.3.2 Enciphering and deciphering in CBC mode 15
  - 2.3.3 Integrity checking against MACs 15
  - 2.3.4 Integrity checking against Merkle trees 15

3 Data structures **17**
- 3.1 Notations 17
- 3.2 The Master Block (MB) 17
  - 3.2.1 The Security Policy (SP) 18
  - 3.2.2 The Page Security Parameters Entry (PSPE) 18
  - 3.2.3 The Master MAC Tree (MMT) 21
- 3.3 The layout of external memory 21
  - 3.3.1 The memory pages dedicated to integrity protection 21
  - 3.3.2 The Initialization Vectors (IV) memory pages 22
  - 3.3.3 The global picture 22

4 HSM interface registers **23**
- 4.1 The configuration register 23
- 4.2 The status register 24
- 4.3 The master integrity key register 24
- 4.4 The master confidentiality key register 26
- 4.5 The group (or block) atomic read-write operations 27
- 4.6 The atomic group read-write address register 27
- 4.7 The atomic group read-write data register 28
- 4.8 The atomic group read-write command register 29

List of Tables

3.1 Security Policy fields 18
3.2 Page Security Parameters Entry fields 20

4.1 Configuration register fields 23
4.2 Status fields 24
4.3 Master integrity key fields 25
4.4 confidentiality key confidentiality key fields 26
4.5 Atomic group read-write address fields 27
4.6 Atomic group read-write data fields 28
4.7 Atomic group read-write command fields 29

List of Figures

1.1 The HSM in its host SoC 4

2.1 Stream ciphers 8
2.2 Block cipher in counter mode 8
2.3 CBC encryption and decryption 9
2.4 Hash functions 10
2.5 The computation of CBC-MACs 11
2.6 The computation of CBC-MACs with address 12
2.7 A Merkle tree protecting the integrity of a data set 13

3.1 Security Policy layout 19
3.2 Page Security Parameters Entry layout 20
3.3 Memory layout with integrity pages 22

4.1 Configuration register layout 23
4.2 Status register layout 24
4.3 Master integrity key register layout 25
4.4 Master confidentiality key register layout 26
4.5 Atomic group read-write address register layout 27
4.6 Atomic group read-write data register layout 28
4.7 Atomic group read-write command register layout 29

Chapter 1

Introduction

Embedded systems can be attacked by many different kinds of attackers, with various goals and means. $\mathcal{S}ec\mathcal{B}us$ is a hardware and software solution for the protection of micro-processor-based Systems-on-a-Chip (SoC) against the tampering with external memories, that is, attacks that aim at extracting or modifying memory contents to access confidential information or to alter the nominal behaviour of the system. A $\mathcal{S}ec\mathcal{B}us$-equipped platform is able to guarantee the confidentiality and the integrity of critical software applications even in the presence of an adversary who has a physical access to every components of the system, except the internals of the main SoC. The $\mathcal{S}ec\mathcal{B}us$ architecture is also designed to counter a large family of pure software attacks based on the exploit of Direct Memory Access (DMA) capable peripherals like, for instance, Graphics Processing Units (GPUs), Hard Disk Drive (HDD) controllers or network cards.

This document is a general introduction to the $\mathcal{S}ec\mathcal{B}us$ architecture. It presents the context of the $\mathcal{S}ec\mathcal{B}us$ project, its threat model and describes $\mathcal{S}ec\mathcal{B}us$ on a functional point of view. $\mathcal{S}ec\mathcal{B}us$ is a generic architecture, as agnostic as possible about the target computer system. The descriptions of the $\mathcal{S}ec\mathcal{B}us$ principles are thus kept as generic as possible. Whenever concrete examples of use are discussed, the considered system is a typical SoC with 32 bits ARM CPUs, with up to 4GB of external memory to protect.

1.1 Notations

This chapter uses the following conventions and notations.

- $\mathtt{w}n$ is the type of n bits words.
- $\mathtt{z}n$ is the all-zeroes $\mathtt{w}n$ constant.
- $A||B$ is the concatenation of the A and B words.
- $|A|$ is the bit-length of the word A.
- Bits in a n-bits word A are numbered from $n-1$ down to 0, left to right and denoted $A[n-1], ..., A[0]$.
- If A is a n-bits word and if $0 \leq q \leq p < n$, $A[p...q]$ is the $p-q+1$-bits word formed by bits p down to q of A.

- For any bijection F, F^{-1} is the inverse of F: $y = F(x) \Leftrightarrow x = F^{-1}(y)$.
- $\oplus$ is the exclusive-or operator.

1.2 Threat model

$\mathcal{S}ec\mathcal{B}us$ assumes that the adversary has a complete physical access to the target embedded system, except the internals of the main SoC. $\mathcal{S}ec\mathcal{B}us$ does not deal with on-silicon attacks, fault attacks or side channel attacks. A lot of research works deal with them and many countermeasure have been proposed and implemented. $\mathcal{S}ec\mathcal{B}us$ considers that such countermeasures are used every time it is required to protect the internals of the SoC. As usual in the security field, $\mathcal{S}ec\mathcal{B}us$ follows a modular approach, where each problem is dealt with as independently as possible from the others.

On the software side $\mathcal{S}ec\mathcal{B}us$ considers embedded systems on which arbitrary applications can be loaded and launched, even by adversaries.

With a complete physical access to the hardware (except the internals of the main SoC), the adversary can spy at every communication between the computer system and the outside world (network, sensors, actuators...) She can also spy at the communications between the hardware discrete components that compose the system itself like, for instance, the exchanges between the main SoC and its external memories. And last but not least, she is not limited to passive probing : she can also tamper with these communications and inject her own data on the communication channels.

The attacks can be either software attacks, hardware attacks, or a combination of the two.

A pure software attack can be the exploit of a software flaw in the embedded software that controls a DMA capable peripheral. Such attacks have been successfully mounted against game consoles, through the exploit of their DVD drives. Similar attacks have been reported on HDD, GPU, etc.

A pure hardware attack can consist in extracting the content of external memory chips (flash or DDR), as has been demonstrated with the famous cold boot attack (see for instance http://citp.princeton.edu/memory/). Memory bus probing and injection is another example that has been used against game consoles (Andrew "Bunnie" Huang against Xbox) or crypto processors (Markus G. Kuhn against the DS5002FP microcontroller).

A typical example of combined attack can 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 privileged mode, forcing the embedded system to run the custom code in privileged mode and leading to a privilege escalation.

To make it short: the main SoC cannot trust its environment. Anything that it sends to the outside can be spied at. And anything that comes in can be under control of an attacker, even a piece of data that the SoC has written in its external memory and that it reads back can have been modified.

1.3 Security objective

The main security objective of the $\mathcal{S}ec\mathcal{B}us$ hardware and software architecture is to allow the design of a trustworthy computing platform. The platform shall remain trustworthy even when operated by adversaries. On top of $\mathcal{S}ec\mathcal{B}us$ it is possible to design a software stack that can be attested remotely: a remote operator or user of the platform

can get the guarantee that the integrity of the platform has not been compromised and that it behaves as expected.

In order to guarantee that the software stack currently running on an embedded system is really what it should be and that the sensitive information it processes remain confidential, it is necessary to protect all physical external interfaces of the main SoC. The protection shall cover both the confidentiality and the integrity of the external communications. Most of these communications can be protected by software-implemented classical cryptography. Enciphering and deciphering can be used to guarantee the confidentiality. Integrity checking against hash digests or Message Authentication Codes (MAC) can be used to protect the integrity. A network interface, for instance, can be protected when needed by encryption of the payload of the data packets and by messages signature. The same software-implemented cryptographic primitives can be used to protect the communication with mass storage (HDDs or non volatile memories). These techniques are well known and massively used: there are many secure network protocols or HDD encryption solutions available. There is one important exception to the list of the communication channels between the SoC and the external world that these software-implemented techniques can protect: the channel between the SoC and its external memories. Indeed, if an attacker tampers with the content of the external memories, there is no way any more to guarantee that the running software and the cryptographic primitives it is supposed to implement is really what it is. Flipping one single bit of the address or data bus during a read or write transaction can be sufficient to execute arbitrary code in privileged mode.

Countermeasures based on Trusted Platform Modules (TPM) and attestation of the installed software stack can not defeat these attacks because their integrity checks are performed when software components are loaded, not at run time.

The $\mathcal{S}ec\mathcal{B}us$ architecture has been designed in order to defeat the above-mentioned attacks in the context of the discussed threat model. The next sections lists several important requirements that have been considered during the design of the $\mathcal{S}ec\mathcal{B}us$ architecture and draw several important conclusions:

- Performance
- Transparency
- Software assistance

1.4 Architecture design

The $\mathcal{S}ec\mathcal{B}us$ architecture protects the confidentiality and integrity of the communications between the on-chip CPUs and their off-chip external memories. This protection guarantees that sensitive information do not leave the SoC in clear form and can not be tampered with without this being trapped by an integrity check. This protection is applied on-chip, as transparently as possible. In order to limit the performance overhead, $\mathcal{S}ec\mathcal{B}us$ relies on hardware acceleration: a Hardware Security Module (HSM) is added to the system to take the cryptographic operations in charge.

The HSM embeds 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 or MACs against which the integrity is checked. Hardware modifications of well established architectures have a rather low acceptance level. This is especially true for CPUs: off-the-shelve CPUs, like ARM cores, for instance, are very complex

Industrial Property (IP) blocks. Their validation is an enormous workload. Modifying them would require access to confidential information about their internals and would also require to re-validate the modified architecture. The same applies to less complex, still sophisticated, blocks like memory controllers.

Conclusion: A very important design choice characterizes the $\mathcal{S}ec\mathcal{B}us$ architecture and makes it very different from other proposals with similar goals: it tries to be as transparent as possible. The HSM is added to the SoC with as few hardware modification as possible, the CPUs and other components of a classical SoC are left unmodified.

In order to be protected, all read - write accesses to the external memories must flow through the HSM so that it is given a chance to apply cryptographic primitives if needed.

Conclusion: The best possible location for the HSM is thus between the on-chip interconnect and the on-chip memory controller. This is where the $\mathcal{S}ec\mathcal{B}us$ architecture puts the HSM, as shown on figure 1.1.

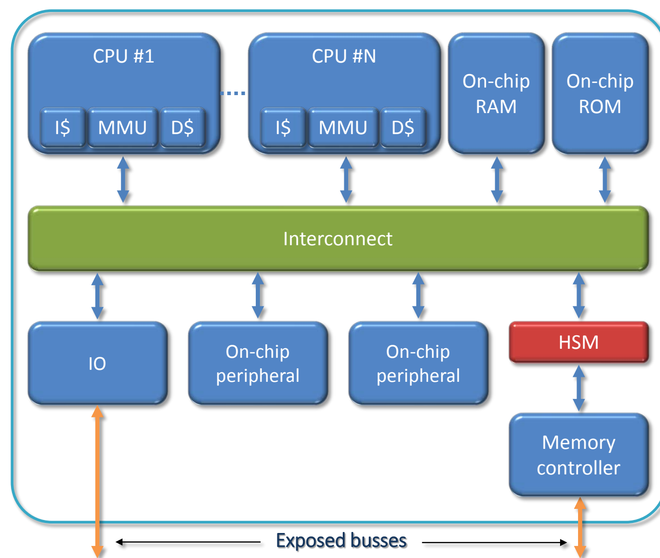


Figure 1.1: The HSM in its host SoC

As the HSM performs cryptographic operations and as these operations take time, the HSM possibly increases the latency of external memory accesses. When fetching an encrypted data, for instance, the data must first be decrypted before being sent to the CPU. One of the major challenges $\mathcal{S}ec\mathcal{B}us$ deals with is this performance overhead. It is as limited as possible and applies only on protected memory areas; when non-sensitive data are stored to and loaded from the external memory there is no penalty at all or a negligible one.

Conclusion: This performance constraints lead to an important conclusion: the HSM alone can not decide what memory area stores sensitive data and what other stores regular data; it must thus be driven by the software stack. This software assis-

tance consists in deciding what security policy to apply to what memory area, a bit like a Memory Management Unit (MMU) is driven by the memory manager of the Operating System (OS) to decide what translation to apply between virtual and physical addresses. In the $\mathcal{S}ec\mathcal{B}us$ architecture the HSM is driven by the Software Security Manager (SSM), a software component responsible for the definition of Security Policies (SP) and the binding between memory pages and SPs.

The definition of SPs can be very simple as, for instance, splitting the address space in a small number of fixed size contiguous regions and assigning each a SP. One region could be defined as unprotected, another as protected in integrity only, another in confidentiality only, etc. These simple definitions could be set during the boot sequence by configuring a few interface registers of the HSM. If required, the boundaries between the different regions could be adjusted during operation to reflect the current memory consumption of each kind. The main advantage of such solutions is their simplicity. Allocating a new memory page with a given SP simply consists in allocating it in the right memory region. Their main drawbacks are their lack of flexibility and, as will be shown later, the difficulty to mitigate the performance degradation with the use of different cryptographic primitives for Read-Only (RO) and Read-Write (RW) memory pages.

Conclusion: For all these reasons the $\mathcal{S}ec\mathcal{B}us$ architecture is as flexible as possible and allows the definition of SPs at the granularity level of individual memory pages. This is another major difference with related alternate proposals.

The model of the cooperation between the memory manager and the MMU is a very good image to illustrate the cooperation between the SSM and the HSM. The granularity of the memory bus protection is the physical memory pages, with as many different pages sizes as for the MMU. For each physical memory page the SSM assigns a SP, both in terms of confidentiality and integrity, and the HSM applies this policy on all memory accesses falling in the page. 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 SP. In the $\mathcal{S}ec\mathcal{B}us$ terminology, the data structure that binds a SP to a memory page is the Page Security Parameters Entry (PSPE). Each memory page has a corresponding PSPE and this PSPE points to a SP. When the external memories are accessed, the HSM fetches the PSPE of the memory page in which the access falls, uses it to retrieve the SP to apply and performs the required cryptographic operations.

Chapter 2

Cryptographic primitives

2.1 Notations

- n is the bit-width of the block cipher
- b is the number of blocks chained together in CBC mode and the arity of the Merkle trees
- b' is the number of blocks in the narrowest level of a Merkle tree
- $C = E_K(P)$ is the ciphertext block produced by the block cipher E when enciphering the plaintext block P with secret key K
- $P = E_K^{-1}(C)$ is the plaintext block produced by the block cipher E when deciphering the ciphertext block C with secret key K
- The consecutive plaintext data blocks in CBC mode are denoted $P_0, P_1, ...P_{b-1}$
- The consecutive ciphertext data blocks in CBC mode are denoted $C_0, C_1, ...C_{b-1}$
- $@(X)$ is the byte address in memory of data X
- $IV(@(X))$ is the initialization vector used for encryption and decryption of data X stored at memory address $@(X)$

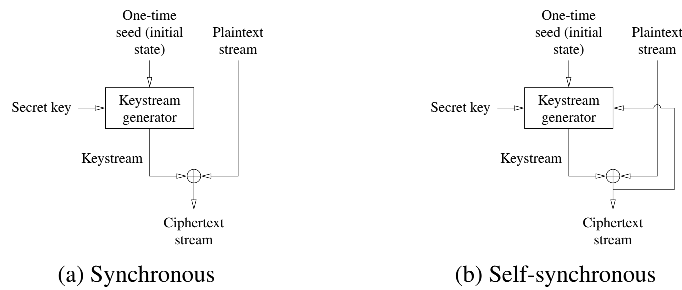
2.2 Cryptographic primitives

Before entering the details of the $\mathcal{S}ec\mathcal{B}us$ architecture, it is essential to present the cryptographic primitives used by the HSM and the reasons why they have been selected.

2.2.1 Confidentiality

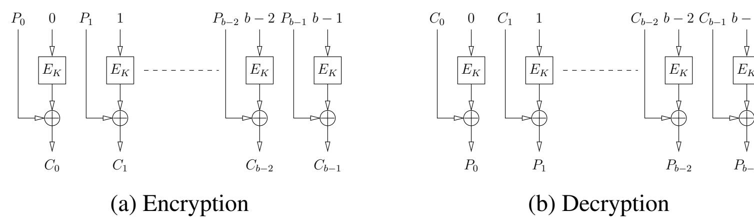
Confidentiality is guaranteed by encryption and decryption of the confidential data. Two main classes of primitives are used for encryption and decryption:

- public key cryptography,
- symmetric cryptography.



(a) Synchronous (b) Self-synchronous

Figure 2.1: Stream ciphers



(a) Encryption (b) Decryption

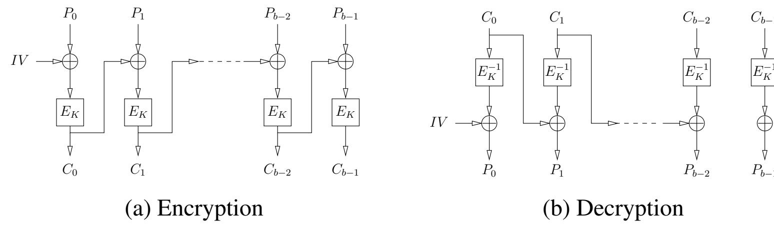
Figure 2.2: Block cipher in counter mode

Public key cryptography does not apply here because it is dedicated to protect the exchanges between different correspondents, without requiring that they first agree on a shared secret, thanks to a trusted certification third party and twofold keys with a public part and a private part. In the $\mathcal{S}ec\mathcal{B}us$ context there is only one correspondent: the $\mathcal{S}ec\mathcal{B}us$-equipped platform and it can rely on on-chip secrets for encryption. Moreover, public key cryptography is orders of magnitude more computation intensive than symmetric cryptography.

Symmetric cryptography uses one single secret key for encryption and decryption. It comprises stream ciphering (RC4) and block ciphering (DES, AES). Stream ciphers use the secret key only (synchronous stream ciphers) or the key and a sliding window on the ciphertext (self-synchronizing stream ciphers) to generate a key stream. The enciphering is done by exclusively oring (xoring) the key stream and the message (figure 2.1). Stream ciphers are an approximation of the theoretically perfect but practically unusable, one-time pad (one-time meaning that the key stream, the pad, is used only once). They are usually faster than block ciphers.

Block ciphers operate on messages by splitting them in blocks of constant length (64 bits for DES, 128 for AES). Each block is transformed by the block cipher in its equal length encrypted form. The transform is a bijection that depends on the secret key. In most cases the encryption of the consecutive blocks of a message are not done independently. They are chained in one form or another (CBC, OFB, CFB, ...) A block cipher can be turned into a stream cipher by using what is called a counter mode: a counter is used to generate consecutive blocks that are encrypted by the block cipher. The result is the key stream that is xored with the message (figure 2.2).

In the context of $\mathcal{S}ec\mathcal{B}us$, synchronous stream ciphers and block ciphers in counter mode are extremely interesting because the encryption and decryption only depend on the data during the final xor operation. When a read access of an encrypted data is issued by a CPU, the computation of the key stream needed for decryption can start immediately, before the loaded data actually returns from the external memory (with a



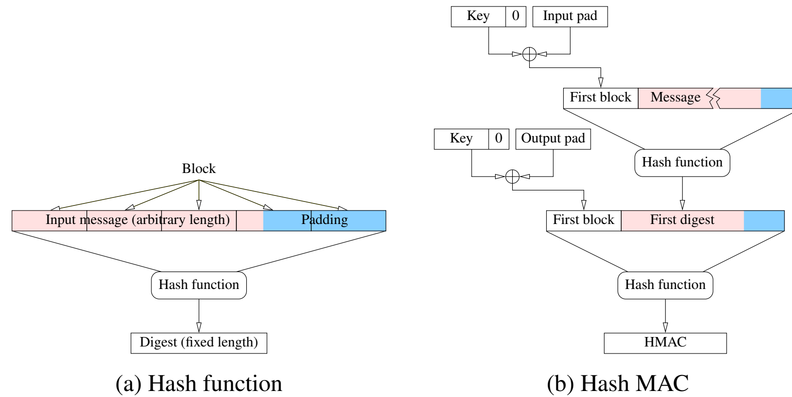
(a) Encryption (b) Decryption

Figure 2.3: CBC encryption and decryption

block cipher or with a self-synchronizing stream cipher, the cryptographic operations can start only when the data returns). The deciphering latency can thus be partially masked by the memory latency. In case the memory latency is larger than the computation time of the key stream, the deciphering latency is completely masked and there is no penalty at all on read accesses. Of course, write accesses cannot benefit from this interesting property. They must be delayed until the key stream is computed. As read latency, especially on instruction fetches, is the most critical parameter for the performance of the system, synchronous stream ciphers or block ciphers in counter mode are used, when applicable. As they are approximations of the one-time pad they can not be used for memory locations which content is modified. If a memory location protected this way is modified, the key stream must be changed. Read-Only (RO) memory pages containing RO data or code are good candidates for this mode of confidentiality protection. They are written (and enciphered) once, when loaded in memory, and read (and deciphered) many times. The one-time property is thus guaranteed. Of course, when a RO memory page is freed, it is never re-allocated and protected with the same one-time pad.

For Read-Write (RW) memory pages, the confidentiality can only be guaranteed by self-synchronizing stream ciphers or block ciphers. Block ciphers in Cipher Block Chaining (CBC) mode have a very interesting property: while the enciphering must be sequential, the deciphering can start with any block of the chain, provided the ciphertext of the previous block is also available. Figure 2.3 illustrates the encryption and decryption in CBC mode of a b-blocks chain, using a E_K block cipher and an Initialization Vector (IV). Typical CBC chain lengths to be considered are cache lines because they are frequently completely refilled on cache misses even on a single word load. The use of IVs is necessary for semantic security, that is, to prevent an adversary from getting any information from the comparison of ciphertexts (without IVs or with a constant IV the enciphering of a given data always produces the same ciphertext and an adversary can thus easily decide whether two plaintext data are equal or not). On the security point of view, they are most efficient when they are random and unpredictable by the adversary. Finally, to render dictionary attacks more difficult, the enciphering function must be unique for a given memory location. Blending the memory addresses during the enciphering / deciphering is a way to achieve this.

Conclusion: In the $\mathcal{S}ec\mathcal{B}us$ architecture a memory page can have its confidentiality unprotected, protected with a block cipher in counter mode (pseudo one-time pad) or with a block cipher in CBC mode. In CBC mode, random, unpredictable IVs can optionally be used. In counter mode and in CBC mode, the memory address is involved in such a way that enciphering the same data at two different locations leads to completely different results. The length of CBC chains is derived from the length of the cache lines of the host system.



(a) Hash function (b) Hash MAC

Figure 2.4: Hash functions

2.2.2 Integrity

Integrity is checked by hash functions or MACs. Hash functions and MACs compute a fixed-length digest from an any length input message (hash functions) and a secret key (MACs). Forging a message with a given digest (pre-image) or finding two different messages with same digest (collision) are two attacks against these cryptographic primitives. An essential characteristic of a hash function or a MAC is its resistance against these attacks. Most hash functions first pad the message with extra information like, for instance, its bit-length and stuffing bits, split the result in blocks of a fixed length and process them sequentially to update an internal state which final value is the digest (figure 2.4a). As MACs are using a secret key they can not be computed by an attacker who would not know the secret key. Finding collisions by brute force search is thus impractical with MACs while it may be with hash functions.

Hash functions can be turned into MACs by hashing a blend of the input message and a secret key (HMAC, figure 2.4b). According RFC 2104 this can be done by zero-padding the secret key to the block-length, xoring the result with a public, constant input pad block and prepending the result to the message. A first digest is then computed and, again, the zero-padded secret key is xored with a public, constant output pad block, prepended to the digest and a second digest is computed which gives the HMAC of the message.

MACs are frequently based on block ciphers and are thus interesting for $\mathcal{S}ec\mathcal{B}us$ because the same hardware cryptographic engine can be used for several purposes: enciphering / deciphering in counter mode for RO pages, in CBC mode for RW pages, and integrity protection by MACs. MACs based on block ciphers frequently use the CBC mode (CBC-MAC, figure 2.5) if the groups of data to protect have a fixed length (which is the case of cache lines); the last block of the CBC-encrypted group is used as the MAC of the group[1].

Moreover, as hash functions are public, an adversary can easily compute the digest of any data of her choice. Protecting the integrity of the memory content with hash functions thus requires that the digests are stored on-chip; if they were stored in the external memory, with the data they are supposed to protect, it would be trivial to alter

[1] It is thus very tempting to use the same CBC encryption as confidentiality and integrity protection. This is a well known flaw. If CBC-MAC and CBC-encryption are used on the same group of data, it is with different secret keys and double computation overhead.

both the data and the digests. Storing on-chip the digests of every integrity-protected group of data is not scalable. It is applicable only in situations where the amount of external memory to protect is extremely limited. It is used, for instance, in the XBox 360 game console, by Microsoft, to protect the hypervisor, but the limited amount of internal memory does not allow to protect other sensitive memory locations. MACs are more interesting with respect to this digests storage problem: as they can not be computed by an adversary, they can be stored in the external memory with the groups of data they protect.

Conclusion: The $\mathcal{S}ec\mathcal{B}us$ architecture uses MACs for integrity protection. Its HSM embeds MAC engines, computes MACs of sensitive data, stores them in such a way that tampering with them is always detected, and uses them as a reference upon data loads. For performance reasons the MACs are computed on small groups of data: when reading a data word the whole group it pertains to must also be read, its MAC computed and compared with the reference MAC. Large groups would thus lead to a significant read overhead. As for confidentiality protection, and for the same reasons, the length of the groups is derived from the length of the cache lines of the host system.

The integrity protection problem can be sub-divided in 3 different kinds of attacks:

- Spoofing: an attacker, by probing the memory bus or by manipulating the memory itself, answers a read request by returning a forged data, different from the requested one.
- Splicing: the attacker, still on a read request, returns a data stored at a different address than the requested one.
- Replay: the attacker, still on a read request, returns the data that was stored at the requested address bus has since been replaced by a regular write access.

The three kinds are extremely dangerous and can all lead to dramatic privilege escalation. The two first can be obtained, for instance, by simply flipping one single bit of the data bus (spoofing) or the address bus (splicing). The third one can be obtained by preventing a regular write access, still with a few probes on some control signals. Injection using an exploited DMA-capable peripheral is an even more powerful way to attack the integrity of the memories. While equally dangerous the 3 kinds differ in the kind of cryptographic primitives that can be used to defeat them.

A spoofing attack is easily detected because without the secret key used to compute the MACs, the attacker can not modify both a group of data and its MAC. By reading the spoofed group of data, its MAC, re-computing the MAC on the group of data (with the internal secret key) and comparing the two MACs, the HSM would detect the tampering.

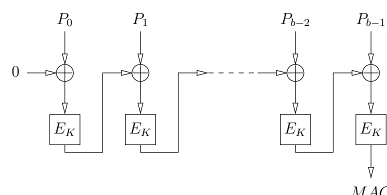


Figure 2.5: The computation of CBC-MACs

In scalable systems, for each memory page with protection against spoofing attacks, another memory page must be allocated to store the corresponding MACs. This memory overhead is the price to pay for scalability and integrity protection. Note that MACs being usually smaller than the data they protect (typically 64 or 128 bits for 256 or 512 bits), the same dedicated memory page can store the MACs of several protected regular pages.

MACs, as introduced above, can not protect against splicing attacks: an attacker capable of permuting in memory two groups of data and their MACs would remain undetected. A simple solution to this problem consists in blending the address of a group of data during MAC computations. With CBC-MAC this can be achieved by using the address as the first block to encrypt. There is a one-block enciphering overhead but splicing attacks are now detected because the MAC function is rendered unique for a given address. RO pages are not sensitive to replay attacks (replaying a constant data does not provide any advantage). As for confidentiality, the integrity of RO pages can thus be handled with relatively light cryptographic primitives and a limited performance and memory overhead. Figure 2.6 illustrates the computation of a CBC-MAC on a $b + 1$ CBC chain, with a right-padded address as first block, using a E_K block cipher. The resulting MAC can be truncated by discarding its rightmost bits in order to fit a target MAC length.

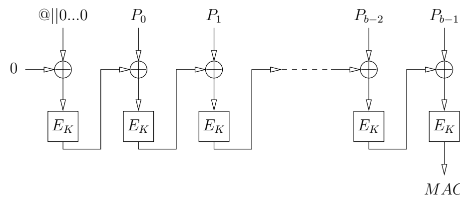


Figure 2.6: The computation of CBC-MACs with address

Conclusion: The $\mathcal{S}ec\mathcal{B}us$ architecture protects the integrity of sensitive RO pages with sets of MACs stored in external memory. Each protected RO page is associated a MAC set in another dedicated memory page. As a MAC is smaller than the data group it protects, a page of MAC sets contains several sets and thus protects several regular pages. The computation of the MAC of a data group involves the group's address in order to counter splicing attacks.

Even with the blending with the memory addresses, MACs can not protect against replay attacks because an adversary could replay a group of data and its MAC without being detected. Replay attacks are the most difficult to defeat because they necessarily rely on the on-chip storage of a reference MAC. In scalable systems, storing all MACs on-chip is not practical. Merkle trees are an option: they protect a whole set of groups of data with a tree-like, hierarchical, organization. Each group of data has an associated MAC in external memory, the level #1 MACs. The level #1 MACs, in turn, are grouped and the MACs of these level #1 groups are computed, giving level #2 MACs. As one MAC protects a whole group, the number of MACs per level decreases from one level to the next. This process is repeated until there is only one remaining MAC, the root of the tree, which, as usual with the trees of the computer science forest, is usually represented at the top while the leaves are at the bottom. Figure 2.7 represents a s-levels, a-ary Merkle tree protecting a set of a^s blocks of data stored in nodes $(0, j),\ 0 \leq j < a^s$. Each node $(i, j),\ 0 \leq i \leq s, 0 \leq j < a^{s-i}$ of the tree, including that

containing the data blocks, has the size of the blocks of the symmetric cipher used for the MAC computation.

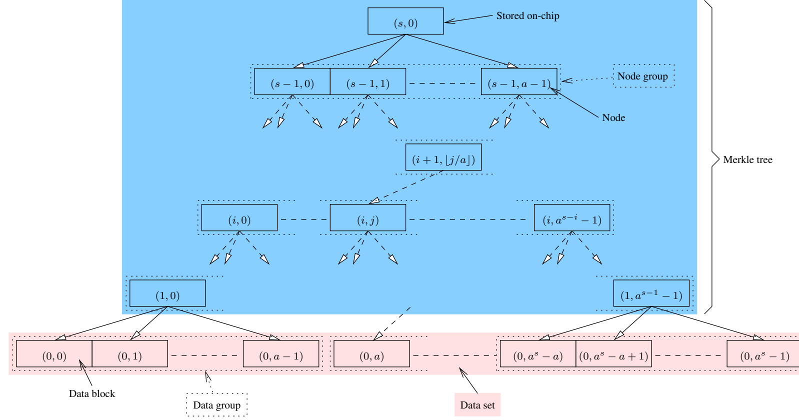


Figure 2.7: A Merkle tree protecting the integrity of a data set

The root MAC is stored on-chip, preventing replay attacks. When checking the integrity during a read operation, the HSM walks through the branch of the Merkle tree, starting from the data itself, fetches the sibling data to compute the MAC of the group, moves one level up and so on, until it can compute the root MAC and compare it with the on-chip reference. Algorithms 1 and 2 represent respectively a verified read and write of the value of a node (i, j) of the Merkle tree of figure 2.7. $Read(i, j)$ is a regular un-verified read of the value of node (i, j). $MAC_K(i, j, X_0, X_1, ..., X_{a-1})$ is the MAC computed with secret key K on the values $X_0, X_1, ..., X_{a-1}$ stored in the $(i, j), (i, j + 1), ..., (i, j + a - 1)$ nodes. $Write(i, j, V)$ is the regular un-verified write of value V in node (i, j). These algorithms are only examples illustrating the principles; several optimizations are possible to reduce the latency and the number of memory accesses.

Algorithm 1 $VerifiedRead(i, j)$: verified read of the value of node (i, j)

1: **if** $i = s$ **then**
2: $X_{i,j} \leftarrow Read(s, 0)$ ▷ On-chip root
3: **else**
4: $j_0 \leftarrow a \times \lfloor \frac{j}{a} \rfloor$ ▷ Index of first sibling
5: **for** $k \in \{j_0 ... j_0 + a - 1\}$ **do**
6: $X_{i,k} \leftarrow Read(i, k)$ ▷ Un-verified read of a siblings
7: **end for**
8: $M' \leftarrow MAC_K(i, j_0, X_{i,j_0}, X_{i,j_0+1}, ..., X_{i,j_0+a-1})$ ▷ Parent MAC computation
9: $M \leftarrow VerifiedRead(i + 1, j_0/a)$ ▷ Verified read of reference parent MAC
10: **if** $M \neq M'$ **then** ▷ If mismatch
11: **error**
12: **end if**
13: **end if**
14: **return** $X_{i,j}$ ▷ Return requested, verified, node's value

Algorithm 2 $VerifiedWrite(i, j, V)$: verified write of value V in node (i, j)

1: $X_{i,j} \leftarrow V$ ▷ The value to write
2: **if** $i \neq s$ **then**
3: $j_0 \leftarrow a \times \lfloor \frac{j}{a} \rfloor$ ▷ Index of first sibling
4: **for** $k \in \{j_0 ... j_0 + a - 1\},\ k \neq j$ **do**
5: $X_{i,k} \leftarrow VerifiedRead(i, k)$ ▷ Verified read of $a - 1$ siblings
6: **end for**
7: $M \leftarrow MAC_K(i, j, X_{i,j_0}, ..., X_{i,j}, ..., X_{i,j_0+a-1})$ ▷ New parent MAC
8: $VerifiedWrite(i + 1, j_0/a, M)$ ▷ Verified write of parent MAC
9: **end if**
10: $Write(i, j, X_{i,j})$ ▷ Regular write

Using this hierarchical organization it is possible to protect an any-size external memory with only one internal root MAC. As for the spoofing and the splicing, there is an associated memory overhead to store the Merkle trees, but the root of all.

Conclusion: The $\mathcal{S}ec\mathcal{B}us$ architecture protects the integrity of sensitive RW pages with Merkle trees of MACs stored in external memory. Each protected RW page is associated a MAC tree in another dedicated memory page. As with MAC sets, MAC trees are usually smaller than the pages they protect[2]. A page of MAC trees thus contains several trees and protects several regular pages. As with MAC sets, the computation of the MAC of a data group involves the group's address. This reduces the risk of exploitable accidental collisions: in order to be exploitable a collision must be happen at the same address.

The memory and performance overheads vary among the three kinds of integrity attacks that have been analysed, the worst of all being the replay attacks. The performance overhead can be mitigated by caches in the HSM, in which a small number of recently used MACs are stored. As these caches are on-chip, in our threat model, they are considered as trust-able. In the case of replay attacks, for instance, the verification process can stop as soon as one of the MACs along the branch is found in an internal cache. Similar cache-based optimizations are possible upon write accesses and the corresponding update of the MACs.

Conclusion: The HSM of the $\mathcal{S}ec\mathcal{B}us$ architecture implements these cache-based optimizations in order to offer a trade-off between the performance overheads and the cost of the solution.

2.3 The cryptographic primitives of $\mathcal{S}ec\mathcal{B}us$

The $\mathcal{S}ec\mathcal{B}us$ architecture is based on a single block cipher that is used for encryption-decryption and for integrity checking against Message Authentication Codes (MACs). The choice of the block cipher, its modes of operation and parameters is essential as it also freezes many other hardware and software aspects. This section presents the four cryptographic primitives used by the $\mathcal{S}ec\mathcal{B}us$ architecture and how they are built on top of the block cipher.

2.3.1 Enciphering and deciphering in counter mode

The confidentiality of RO memory pages is protected by a pseudo one-time pad cipher. Each block is xored with the enciphering of its byte address in memory, right-padded

[2] At least for trees with an arity larger than 2

with zeroes:

$$C = E_K(@(P)||z(n - |@(P)|)) \oplus P$$
$$P = E_K(@(P)||z(n - |@(P)|)) \oplus C$$

2.3.2 Enciphering and deciphering in CBC mode

The confidentiality of RW memory pages is protected in CBC mode with chains of $b+1$ blocks:

$$C_{-1} = E_K((@(P_0)||z(n - |@(P_0)|)) \oplus IV(@(P_0)))$$
$$\forall\, 0 \leq i \leq b-1 \; C_i = E_K(P_i \oplus C_{i-1})$$
$$\forall\, 0 \leq i \leq b-1 \; P_i = E_K^{-1}(C_i) \oplus C_{i-1}$$

$IV(@(P_0))$ is a n-bits initialization vector. The initialization vectors used to protect a memory page form an «IV-set» and are randomly generated upon every store access to the CBC chain. The IV-sets are stored in dedicated memory pages. An IV-page of a given size can contain up to b IV-sets, each protecting one memory page of data of the same size. In $\mathcal{S}ec\mathcal{B}us$ the use of initialization vectors is optional[3]. When initialization vectors are not used they are replaced by the constant, zeroed, block ($IV(@(P_0)) = z(n)$) and, of course, the IV-sets are not stored in memory. The byte address of the CBC chain is involved in the computation of the first, not stored, block of the chain in order to make the enciphering function of each memory location unique and thus to prevent dictionary attacks.

2.3.3 Integrity checking against MACs

The integrity of RO memory pages is protected by MACs, computed in CBC-MAC mode with b-blocks long chains ($n \times b$ bits):

$$C_{-1} = E_K(@(P_0)||z(n - |@(P_0)]))$$
$$\forall\, 0 \leq i \leq b-1 \; C_i = E_K(P_i \oplus C_{i-1})$$
$$MAC(P_0|P_1|...|P_{b-1}) = C_{b-1}$$

The right-padded address of the chain is encrypted and the result used as initialization vector to render the MAC computation unique for each memory location and thus prevent splicing attacks. As for the initialization vectors, the MACs used to protect a memory page form a «MAC-set» and are stored in dedicated memory pages. A MAC-sets page of a given size can contain up to b MAC-sets, each protecting one memory page of data of the same size.

2.3.4 Integrity checking against Merkle trees

The integrity of RW memory pages is protected by Merkle trees, computed in CBC-MAC mode with b-blocks long chains ($n \times b$ bits). Depending on the different dimensioning parameters of a particular $\mathcal{S}ec\mathcal{B}us$-equipped platform, the Merkle trees can be

[3]Initialization vector are not supported in the currently available $\mathcal{S}ec\mathcal{B}us$ implementations

perfectly balanced, that is, end with a single root node ($b' = 1$), or not. Unbalanced Merkle trees have $1 < b' < b$ nodes (MACs) in their narrowest last level.

As for the IV and MAC-sets, the Merkle trees are stored in dedicated memory pages. A MAC-tree page of a given size can contain up to $b - 1$ different trees, each protecting one memory page of data of the same size. The narrowest last levels of the $b - 1$ trees form a $b' \times (b - 1)$-blocks long chain. The CBC-MAC of this chain is computed and the result used to protect the integrity of the MAC-tree page. It is stored in a dedicated $\mathcal{S}ec\mathcal{B}us$ data structure.

Chapter 3

Data structures

3.1 Notations

The HSM that enforces confidentiality and integrity protection of the external memory can use any cryptographic primitives, without major changes of its fundamental principles. Data structures are better discussed on a concrete example with well defined types and sizes. In the following we consider a typical SoC embedding 32 bits ARM CPUs, with 256 bits cache lines, and up to 4GB of external memory to protect. This example uses DESX, a lightweight block cipher primitive. This choice is well adapted to low and intermediate cost systems like ISP boxes, game consoles or set-top boxes. In other contexts, other, more expensive but stronger primitives, could be chosen.

All cryptographic operations performed by the HSM are based on the same DESX block cipher, a DES (Data Encryption Standard) variant that increases the nowadays too small DES key space at a very limited cost, using key whitening. The DESX block length is 64 bits and a DESX secret key is a 184 bits $\{k_1, k, k_2\}$ triple where k_1 and k_2 are two 64 bits words and k is a 56 bits word. DESX encryption is defined by:

$$C = DESX_{\{k_1,k,k_2\}}(M) = k_2 \oplus DES_k(M \oplus k_1)$$

where $\oplus$ is the bit-wise exclusive or. The DESX decryption is thus:

$$M = DESX^{-1}_{\{k_1,k,k_2\}}(C) = k_1 \oplus DES^{-1}_k(C \oplus k_2)$$

In the following, the members of a DESX key triple are denoted $key.k_1$, $key.k$ and $key.k_2$.

3.2 The Master Block (MB)

A system equipped with the HSM must allocate a contiguous region of its external memory to HSM control data structures. This memory area is called the Master Block (MB) and is split in three sub-regions:

- An array of Page Security Parameter Entries (PSPEs).
- An array Security Policies (SPs).
- The Master MAC Tree (MMT), a Merkle tree of MACs that protects the integrity of the MB.

3.2.1 The Security Policy (SP)

A Security Policy defines a type of memory protection, both in terms of confidentiality and integrity. It contains:

- `ckey`, a confidentiality secret key used for enciphering and deciphering in counter mode when `cmode=ctr` or in Cipher Block Chaining (CBC) mode when `cmode=cbc`.
- `ikey`, a integrity secret key used for CBC-MAC computations when `imode=mac`.
  Note: for technical reasons that would be too long to explain here, when `imode=mac-tree` the CBC-MACs are computed with the Master Integrity Key (MIK) stored in the HSM internal registers.
- `cmode∈{none,ctr,cbc}`, a confidentiality mode indicator.
- `imode∈{none,mac,mac-tree}`, a integrity mode indicator.
- `val`, a boolean flag indicating whether the SP is usable or not. The HSM raises an interrupt when a memory access is performed in a page bound to an invalid SP.

The SPs are organized as an array and stored in a contiguous memory region protected by the HSM in confidentiality and integrity. The number of SPs is implementation dependent. In other data structures SPs are referred to by their index. The SP stored at the lowest address in memory has index 0.

Table 3.1: Security Policy fields

Name	Width	Long name
`val`	1 bits	VALid flag (0: invalid, 1: valid)
`imode`	2 bits	Integrity MODE (0: none, 1: MAC set, 2: MAC tree)
`cmode`	2 bits	Confidentiality MODE (0: none, 1: counter, 2: CBC)
`ckey0`	32 bits	32 LSBs of confidentiality KEY.K
`ckey1`	24 bits	24 MSBs of confidentiality KEY.K
`ckey2`	32 bits	32 LSBs of confidentiality KEY.K1
`ckey3`	32 bits	32 MSBs of confidentiality KEY.K1
`ckey4`	32 bits	32 LSBs of confidentiality KEY.K2
`ckey5`	32 bits	32 MSBs of confidentiality KEY.K2
`ikey0`	32 bits	32 LSBs of integrity KEY.K
`ikey1`	24 bits	24 MSBs of integrity KEY.K
`ikey2`	32 bits	32 LSBs of integrity KEY.K1
`ikey3`	32 bits	32 MSBs of integrity KEY.K1
`ikey4`	32 bits	32 LSBs of integrity KEY.K2
`ikey5`	32 bits	32 MSBs of integrity KEY.K2

3.2.2 The Page Security Parameters Entry (PSPE)

The PSPEs are not exactly organized in a hierarchy of tables as Memory Management Units (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 associ-

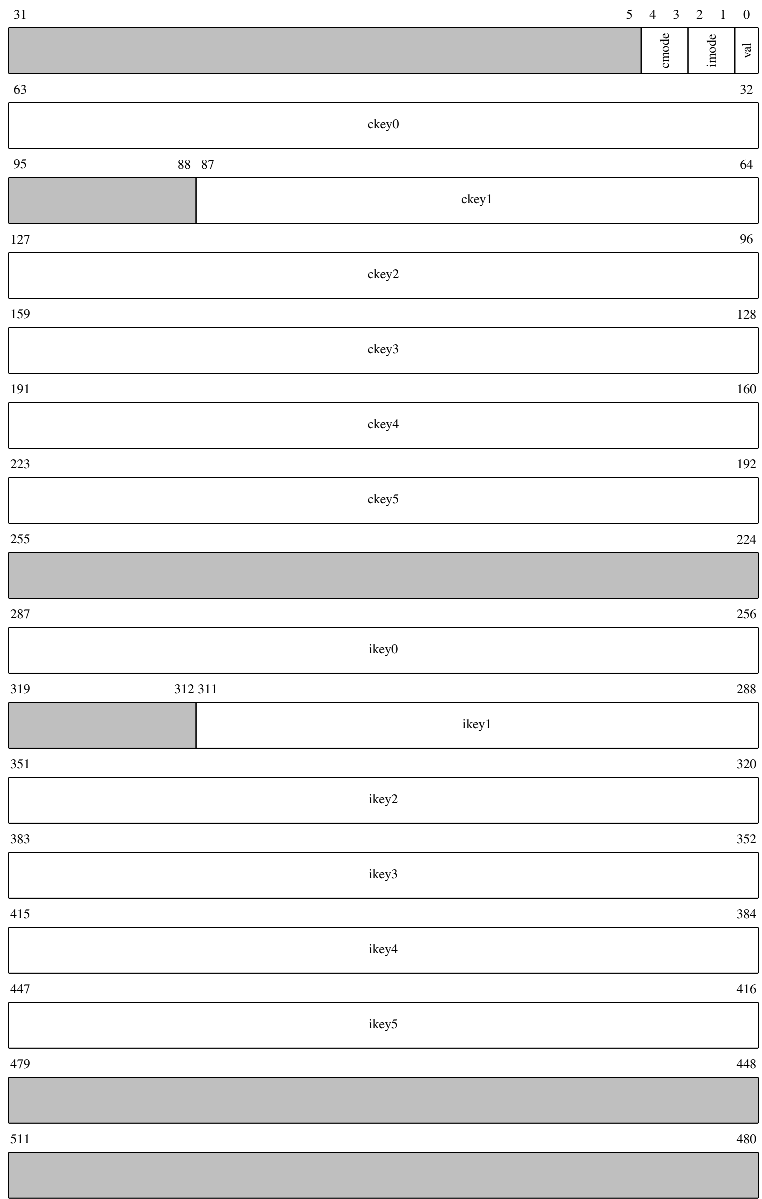


Figure 3.1: Security Policy layout

ated with. In a ARM-based system with four page sizes, 4kB, 64kB, 1MB and 16MB, a PSPE corresponding to a page aligned on a 16MB boundary can be that of a:

- 4kB page (the 16MB page is broken in 4096 4kB pages),
- 64kB page (the 16MB page is broken in 256 64kB pages),
- 1MB page (the 16MB page is broken in 16 1MB pages),

- 16MB page (the 16MB page exists).

The size indicator disambiguates this.

There are two different PSPE formats: master and slave. Master PSPEs bind regular memory pages to SPs. They contain the index of the bound SP. They also contain two addresses that point to:

- The MACs that protect the integrity (if the page is integrity-protected), that is:
  - A MAC set in a page of MAC sets (if the page is integrity-protected by a MAC set).
  - A MAC tree in a page of MAC trees (if the page is integrity-protected by a MAC tree).
- A Initialization Vector (IV) set in a page of IV sets (if the page is confidentiality-protected in CBC mode and IVs are enabled).

Slave PSPEs are associated to the pages of IV sets, MAC sets or MAC trees. The PSPE associated with a page of MAC trees contains the root of the MAC trees of the page. In the current version, the PSPEs associated with a page of IV sets or MAC sets is unused and can contain anything.

A one-bit flag indicates whether the page is unprotected (no confidentiality, no integrity) or not. It is redundant with the associated SP and exists for performance reasons.

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 an so on, until a PSPE is valid and its size indicator extends up to the requested address.

The PSPE are protected in integrity only because they do not contain any sensitive and confidential cryptographic material.

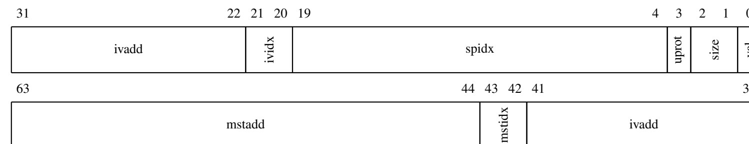


Figure 3.2: Page Security Parameters Entry layout

Table 3.2: Page Security Parameters Entry fields

Name	Width	Long name
`val`	1 bits	VALid flag (0: invalid, 1: valid)
`size`	2 bits	Page SIZE (0: 4KB, 1: 64KB, 2: 1MB, 3: 16MB)
`uprot`	1 bits	UnPROTected flag (0: protected, 1: unprotected)
`spidx`	16 bits	Security Policy InDeX

Name	Width	Long name
`ividx`	2 bits	IV set InDeX in IV page (0 to 3)
`ivadd`	20 bits	IV page ADDress (20 MSBs)
`mstidx`	2 bits	Mac-Set/mac-Tree InDeX in mac-set/mac-tree page (0 to 3/0 to 2)
`mstadd`	20 bits	Mac-Set/mac-Tree page ADDress (20 MSBs)

3.2.3 The Master MAC Tree (MMT)

As the security data structures (PSPE, SP) are stored in the external memory, their integrity must be protected. And as they are RW data they are protected by a Merkel MAC tree. The Merkel MAC tree protecting the PSPEs and SPs is the Master MAC Tree (MMT) and is also stored in the Master Block after the PSPEs and the SPs. The root node of the MMT is stored in an internal register of the HSM.

The integrity protection of memory pages by Merkle MAC trees is thus organized in two-layers:

- A memory page is integrity-protected by a tree stored in a page of MAC trees. The page of MAC trees has the same size as the protected page and contains 3 different trees. A super-MAC is computed on the narrowest levels of the 3 trees of a page of MAC trees. This super-MAC is stored in the slave PSPE corresponding to the page of MAC trees, in the part of the Master Block that contains the table of PSPEs.

- The slave PSPEs (and also the master PSPEs and the SPs) are, in turn, integrity-protected by the MMT which root is stored in the HSM, that is, inside the secure perimeter of the SoC.

3.3 The layout of external memory

3.3.1 The memory pages dedicated to integrity protection

There are two types of memory pages dedicated to integrity protection: MAC set and MAC tree pages. MAC sets are used to protect read-only memory pages (pages in which memory locations are written once and read many) while MAC trees protect read-write data. In the considered example implementation the block cipher is DES-X (64 bits blocks, 184 bits secret key) and the MACs are computed over groups of four consecutive blocks of data ($4 \times 64 = 256$ bits, that is, a cache line of the ARM core). So, a MAC set page of size $S \in \{4kB, 64kB, 1MB, 16MB\}$ contains up to four MAC sets, each protecting one read-only page of size S. As the MAC trees are quaternary, a MAC tree page of size S contains up to three MAC trees, each protecting one read-write page of size S.

The integrity pages are stored in the memory region covered by the HSM and defined at boot time in its `cfg` interface register. For each integrity page, a slave PSPE is defined. The management of the integrity pages is performed by the SSM.

3.3.2 The Initialization Vectors (IV) memory pages

The read-write pages are enciphered by the DES-X block cipher in CBC mode with four-blocks CBC chains. In order to ensure semantic security (also called ciphertext indistinguishability), Initialization Vectors (IV) are required. The use of semantic security is optional because semantic security is not always necessary and it has a cost: the memory pages in which the IVs must be stored. When a memory page must be confidentiality-protected in CBC mode, an IV set must thus be allocated. For the same reasons as for integrity, an page of size S dedicated to IV sets can store four IV sets, each protecting a page of size S. IVs are not implemented yet.

3.3.3 The global picture

Figure 3.3 shows a typical memory layout with the Master Block. Several RW pages have been allocated with integrity protection, they are linked to their MAC tree pages thanks to the PSPE. The root MAC of the MAC trees of the same page of MAC trees is stored in the associates slave PSPE, in the PSPE table of the Master Block. The integrity of the Master Block is itself protected by the MMT. There is also a RO page protected by a MAC set.

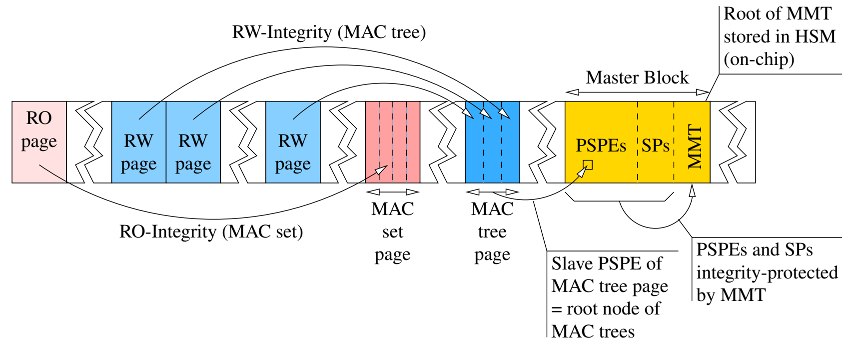


Figure 3.3: Memory layout with integrity pages

Chapter 4

HSM interface registers

The HSM is controlled through a set of interface registers and a set of data structures stored in external memory. The HSM low-level software driver offers a small set of software primitives to access both. Before listing these primitives we will explore the interface registers and explain their role. In the following the interface registers are read-write, unless otherwise stated. An unused register's field is represented as a grey area. Reading an unused field always returns a zero value and writing it has no effect. When reading or writing a register with unused fields it is recommended to assume zero values and to write zero values in unused fields because if future versions make use of these fields the zero value will always be the default one, corresponding to the current behaviour.

4.1 The configuration register

The *configuration register* (`cfg`, figure 4.1 and table 4.1) defines the global configuration of the HSM (address of the Master Block in external memory, various enable flags, definition of the protected memory area). It is mainly used at HSM initialization. The interrupts enable flag can also be set/unset during execution.

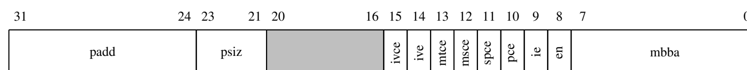


Figure 4.1: Configuration register layout

Table 4.1: Configuration register fields

Name	Width	Long name	Description
`mbba`	8 bits	Master Block Base Address	Aligned multiple of 16MB. 8 MSBs only. Must be set prior use of external memory.
`en`	1 bits	hsm ENable	0=disable, 1=enable.
`ie`	1 bits	Interrupt Enable	0=disable, 1=enable.

Name	Width	Long name	Description
pce	1 bits	Pspe Cache Enable	0=disable, 1=enable.
spce	1 bits	SP Cache Enable	0=disable, 1=enable.
msce	1 bits	Mac Set Cache Enable	0=disable, 1=enable.
mtce	1 bits	Mac Tree Cache Enable	0=disable, 1=enable.
ive	1 bits	IV Enable	0=disable, 1=enable.
ivce	1 bits	IV Cache Enable	0=disable, 1=enable.
psiz	3 bits	Protected SIZe	Size of protected memory area: 1=64MB, 2=256MB, 3=1GB, 4=4GB.
padd	8 bits	Protected ADDress	Start address of protected memory. Aligned multiple of 16MB. 8 MSBs only.

4.2 The status register

The *status register* (`status`, figure 4.2 and table 4.2) is read only. It contains indicators about the current state of the HSM. Reading the status register clears the pending interrupts flag.

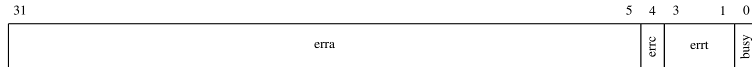


Figure 4.2: Status register layout

Table 4.2: Status fields

Name	Width	Long name	Description
busy	1 bits	BUSY flag	0=idle, 1=busy.
errt	3 bits	ERRor Type	If not 0 on HSM interrupt, indicates the type of error: 0=none, 1=PSPE invalid, 2=SP invalid, 3=integrity violation (MAC sets), 4=integrity violation (MAC trees).
errc	1 bits	ERRor Cause	Type of access that caused error: 0=read, 1=write.
erra	27 bits	ERRor Address	Address of group which access caused an error (27 MSBs).

4.3 The master integrity key register

The *master integrity key register* (`mik`, figure 4.3 and table 4.3) is write only and is used at start-up to set the key used to compute the MAC nodes of the MAC trees (Master

MAC tree and MAC trees protecting regular memory pages).

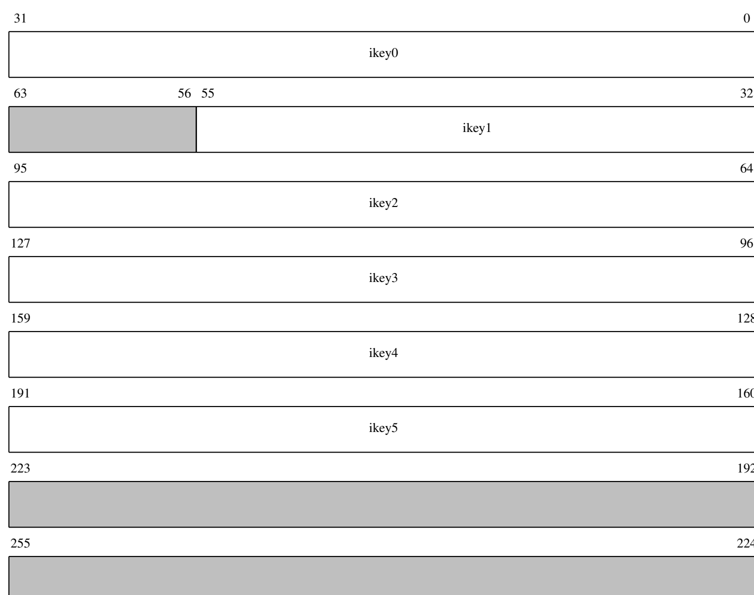


Figure 4.3: Master integrity key register layout

Table 4.3: Master integrity key fields

Name	Width	Long name	Description
`ikey0`	32 bits	Integrity KEY0	32 LSBs of MIK.K.
`ikey1`	24 bits	Integrity KEY1	24 MSBs of MIK.K (most significant byte ignored).
`ikey2`	32 bits	Integrity KEY2	32 LSBs of MIK.K1.
`ikey3`	32 bits	Integrity KEY3	32 MSBs of MIK.K1.
`ikey4`	32 bits	Integrity KEY4	32 LSBs of MIK.K2.
`ikey5`	32 bits	Integrity KEY5	32 MSBs of MIK.K2.

4.4 The master confidentiality key register

The *master confidentiality key register* (`mck`, figure 4.4 and table 4.4) is write only and is used at start-up to set the key used to encipher / decipher the Security Policy area of the Master Block.

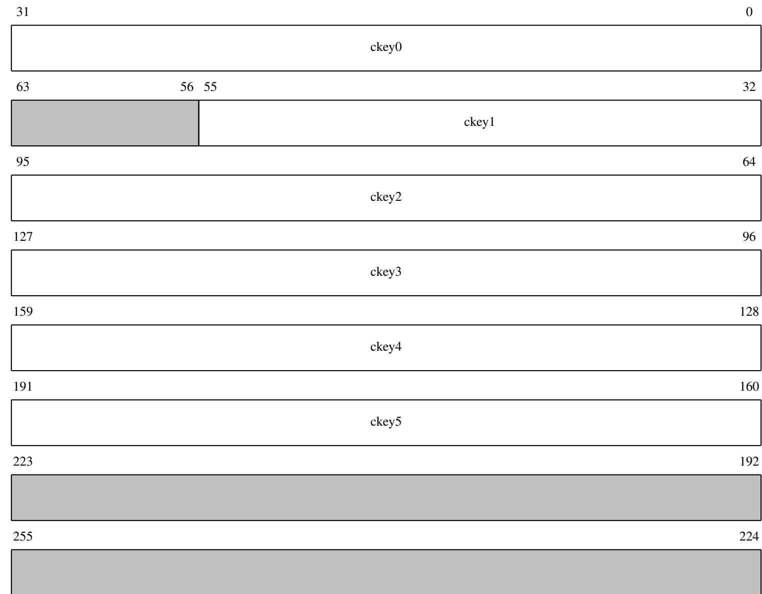


Figure 4.4: Master confidentiality key register layout

Table 4.4: confidentiality key confidentiality key fields

Name	Width	Long name	Description
`ckey0`	32 bits	Confidentiality KEY0	32 LSBs of MCK.K.
`ckey1`	24 bits	Confidentiality KEY1	24 MSBs of MCK.K (most significant byte ignored).
`ckey2`	32 bits	Confidentiality KEY2	32 LSBs of MCK.K1.
`ckey3`	32 bits	Confidentiality KEY3	32 MSBs of MCK.K1.
`ckey4`	32 bits	Confidentiality KEY4	32 LSBs of MCK.K2.
`ckey5`	32 bits	Confidentiality KEY5	32 MSBs of MCK.K2.

4.5 The group (or block) atomic read-write operations

The HSM offers atomic operations to securely access an aligned 64-bits double word or an aligned 256-bits group in external memory. The 256-bits atomic accesses are required for proper initialization of read-only memory pages protected by the block cipher in counter mode (confidentiality) and / or MAC sets (integrity). They are the only way to guarantee the write-once property[1]. The atomic accesses are also used to efficiently access PSPEs (64-bits) and SPs (2×256-bits). Atomic accesses in the PSPE area of the Master Block are always 64-bits. Accesses elsewhere in memory are always 256-bits. Requesting an atomic access is done by setting a set of interface registers (see below); writing the `agrwcmd` register launches the access (and must thus be the last register setting of a request). Upon read accesses the read 64 or 256 bits are retrieved from the `agrwdata` register. When the HSM performs the requested atomic access it automatically applies the defined Security Policy, based of the target address, as for regular load-store operations. Note: regular load-store accesses in the Master Block are forbidden. Accessing the Master Block must absolutely be done through the atomic operations.

The same set of registers is also used to initialize the MAC tree of a newly allocated read-write memory page that must be integrity-protected. The only relevant parameter for the MAC tree initialization is the byte base address of the protected regular page. The associated PSPE and SP provide all the other parameters. Two different commands are dedicated to this MAC tree initialization:

- If the MAC tree to initialize is the first of its page of MAC trees, the topmost levels of the other MAC trees in the same page of MAC trees are not verified when computing the root MAC of the page of MAC trees.
- If the page of MAC trees already contains initialized MAC trees, the topmost levels of the other MAC trees in the same page of MAC trees are verified when computing the root MAC of the page of MAC trees.

4.6 The atomic group read-write address register

The *atomic group read-write address register* (`agrwadd`, figure 4.5 and table 4.5) is used to set the byte address of the 64-bits double word or 256-bits group to access atomically.

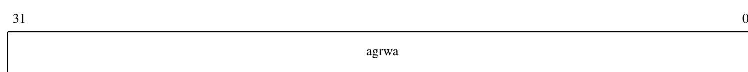


Figure 4.5: Atomic group read-write address register layout

Table 4.5: Atomic group read-write address fields

Name	Width	Long name	Description

[1] If the initial write of the 256-bits group was not atomic, it could lead to multiple enciphering and / or MAC computations with a partly initialized group.

Name	Width	Long name	Description
agrwa	32 bits	Atomic Group Read-Write Address	Group's or block's byte address for atomic group read-write operations. Aligned on group's or block's boundary: LSBs are ignored. Block atomic access if address falls in PSPEs, else group atomic access.

4.7 The atomic group read-write data register

The *atomic group read-write data register* (agrwdata, figure 4.6 and table 4.6) is used to store the data to write or to retrieve the read data of an atomic access. Upon 64-bits accesses (PSPEs), only one quarter of this 256-bits register is used and the quarter used depends on the alignment of the 64-bits double word in the 256-bits group.

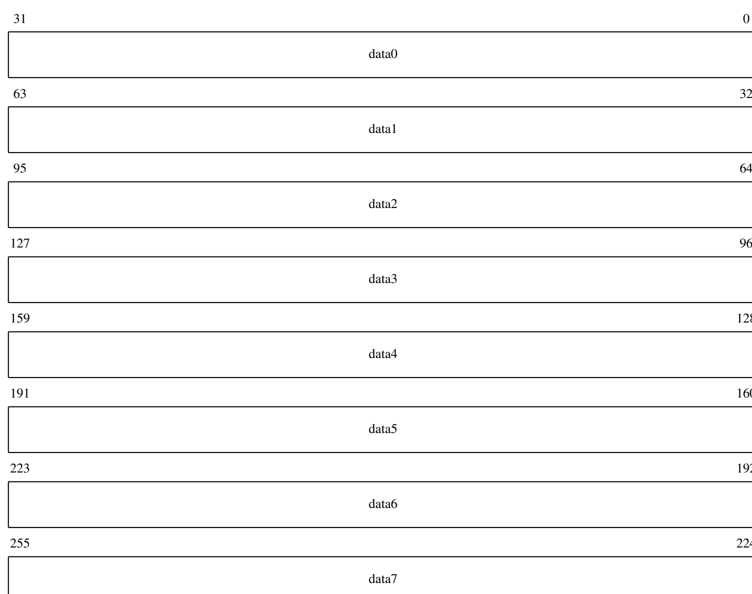


Figure 4.6: Atomic group read-write data register layout

Table 4.6: Atomic group read-write data fields

Name	Width	Long name	Description
data0	32 bits	DATA0	Read data or data to write (lowest address in memory).
data1	32 bits	DATA1	Read data or data to write.
data2	32 bits	DATA2	Read data or data to write.
data3	32 bits	DATA3	Read data or data to write.
data4	32 bits	DATA4	Read data or data to write.
data5	32 bits	DATA5	Read data or data to write.
data6	32 bits	DATA6	Read data or data to write.

Name	Width	Long name	Description
`data7`	32 bits	DATA7	Read data or data to write (highest address in memory).

4.8 The atomic group read-write command register

The *atomic group read-write command register* (`agrwcmd`, figure 4.7 and table 4.7) is used to set the requested command:

- read (of a 64-bits PSPE or a 256-bits group),
- write (of a 64-bits PSPE or a 256-bits group).
- initialize first MAC tree of a page of MAC trees
- initialize a MAC tree that is not the first of its page of MAC trees

Figure 4.7: Atomic group read-write command register layout

Table 4.7: Atomic group read-write command fields

Name	Width	Long name	Description
`cmd`	3 bits	Atomic Group Read-Write CoMmanD	0: none, 1: read, 2: write, 3: init, 4: continue. HSM applies SP defined for target group or block.