PhD Thesis

At the
UNIVERSITY OF SOUTH BRITTANY
Under the governance of the

European University of Brittany

for the degree of

Doctor of the University of South Brittany

grade : *Electronics and Computer Science*

Hardware Core for Off-chip Memory Security Management in Embedded Systems

---

by
Romain Vaslin

submitted the 10/10/2008 to the PhD thesis committee

PhD thesis committee members

Committee reviewers

V. FISCHER	Professor at Jean Monnet University, St-Etienne, France
L. TORRES	Professor at Montpellier II University, Montpellier, France

Committee members

J-P. DIGUET	CNRS Researcher at UBS, Lorient, France (thesis supervisor)
G. GOGNIAT	Associate professor at UBS, Lorient, France (thesis advisor)
Y. TEGLIA	Senior Security Architect at STMicroelectronics, Rousset
R. TESSIER	Professor at University of Massachusetts, Amherst, USA
I. VERBAUWHEDE	Professor at Katholieke Universiteit Leuven, Leuven, Belgium

---

Laboratoire en science et technologies de l'information de la communication et de la connaissance
Lab-STICC CNRS UMR 3192 – European University of Brittany

Contents

List of Figures

List of Tables

Introduction

Context

Today there are around 3 billions of cell phone users and 50% of the earth population should have a cell phone by the end of the year. This figure illustrates the fact that electronic mobile devices have invaded our lives and cell phone is one of the most remarkable example. This list could be extended to GPS, mp3 players and so on. All theses mobile electronic devices are fully part of our lives. Some of them are used to store personal information or to exchange personal information [Arlot, 2008]. As an example, between 2007 and 2011 around 587 billions dollars will be exchanged through mobile phone transactions. As a cell phone user we really need to trust all these exchanges with remote servers.

The people's major fear when purchasing on-line or with their mobile phones is security. Can someone steal my credit card number? This is one of the main people concerns. But bank information is not the only sensitive data which are exchanged and/or stored on/with mobile devices. A lot of other personal information are also stolen. The identity theft is a new potential threat that people are not really taking care of. Someone can easily steal someone's information and spoof his real identity. With this new connected world companies are now obliged to work on these fields in order to make their customers confident in their products.

Companies are also facing new challenges. When a new mobile product is released on the market everyone is trying to bypass any protections and/or restrictions in order to execute specific software. Challengers can spy their technology innovations or hackers can jeopardize their image. The most significant example is the iPhone. Apple has applied so many restrictions on its product (phone provider restriction, software execution restriction) that after a few weeks (and even days for the 3G version), the iPhone has been jailbroken in order to bypass these restrictions. All the current protection mechanisms done in software are almost always broken by hackers. Moreover it is more difficult to protect a mobile device due to the small hardware and software capabilities of the system. In addition since the device is on the field, the system owner can almost do what he wants with his device, open it, make hardware or software modifications.

Some new reliable solutions are needed to protect the user and the companies from thieves. Companies wish to protect there intellectual properties and business. Users are only looking for a secure environment for their data and exchanges.

Problem

From this point, the number of potential threats on mobile device is huge. The first step toward a secure environment would be to guarantee the data and application protection from being stolen and modified. But is it really possible to have a mobile system with these features? As said previously mobile devices have limited hardware and software resources. In order to tackle the security issues, some new features need to be added to the system.

There are two ways to include these security primitives to the device. The first one is to try to add software protections. But due to the current capabilities of embedded systems, this approach does not seem to be the good one. Indeed the overhead created by these kinds of solution is still important. The user is not ready to have a cell which is stalled because an anti-virus is scanning the memory. The second solution is a more active one. The idea is to add some hardware primitives which are running on the fly and checking the system integrity. The cost of such solutions is not appearing on the performances of the system but mostly on its hardware cost and its power consumption. Energy is a critical point as soon as embedded systems are targeted especially when they are autonomous. Hardware architectures provide the most interesting solutions from an energy efficient point of view. For example, an hash algorithm consumes 80 times more energy in software than an hardware implementation.

Here the goal is to provide an efficient hardware solution, fully transparent for the user (no system stall) and also for the software designer which builds application for the device. The hardware primitives must bring the security primitive to guarantee the data confidentiality and the system integrity to prevent any unauthorized software modification. The challenge is to offer a new efficient solution which matches with the current hardware resources of small embedded systems.

Contribution

This document details a complete solution for secure application execution, system boot up and application update. A new efficient secure scheme is proposed in order to guarantee data confidentiality and integrity. This new scheme matches with the tight requirements of embedded systems (performance, memory footprint, logic area and energy consumption). Explorations have been done to evaluate the cost of the

different architectures for performance, memory and energy. The newly proposed scheme for data protection is compared with existing solutions which provide the same protection.

This document also details several features offered by the new scheme. It can be configured for different performances and security levels at a cost of memory and power consumption. This first step to a secure environment has been extended with new features which allow to manage on the fly the data security level to further increase the architecture global efficiency. These features provide to the user a maximum of flexibility to securely execute its own code on the secure architecture without using any specific processor instruction or modification of the operating system.

Any security scheme will fail if the boot up is not carefully designed, so an end to end solution with some specific schemes are also presented for a secure system boot up. The architecture has been extended with more capabilities which allow the system application to be securely updated on the field. Like any new contributions, some figures are provided in order to have an idea of security cost. All the work has been done in a way that each security block is independent of the others. For example, the security level management could be used with and other confidentiality and integrity checking scheme. Moreover, all the overhead due to these new security mechanisms match with the current resources offered by embedded systems.

Outline

Chapter 1 will introduce the basic notions of embedded systems and security. Memory data protection of a system is not a new topic and several solutions have already been proposed. The most interesting of them will be presented and discussed to evaluate if they are suitable or not for embedded systems.

Chapter 2 will be focused on the proposition of a new security scheme which matches with the current demands and resources of embedded systems. The solution will be explored from a security and cost point of view. The software performance and memory consumption will be analyzed and compared with existing solutions.

Chapter 3 will deal with new security features to manage the data security level on the fly. The data in memory will be protected with different security levels. Indeed, sometime data do not need the highest security level offered by the architecture, it depends on the application and the threats. This security management will help to improve efficiency of the system (performance, memory and energy).

In chapter 4, an end to end secure platform will be described. This new open

secure platform guarantees a secure environment from the system boot up to its update. These are common features an embedded architecture should provide to the user and the company which likes to keep its system up to date.

The aim of chapter 5 is to offer an update of the security scheme detailed in chapter 2. Indeed, during the writing of this document a weakness was found. The scheme has been updated in order to provide a fully secure environment. This update was the opportunity to offer more security features to the system. This chapter is a good example of the security area, each time a new solution is released a weakness is often found and the system must be updated with a stronger scheme.

In all these chapters, a complete set of results is presented. Several applications, several security policies and several architectures have been developed and implemented in order to provide the first detailed view of the security cost (performances, memory and energy). The document will end with some conclusions and perspectives for this work and future needs of embedded systems.

CHAPTER 1

Security & embedded systems

This chapter introduces some basic ideas of what is an embedded system and also some basic security principles such as confidentiality and integrity. Specifics attacks on embedded systems and a threat model are also detailed. The aim of a threat model is to define the potential attacks the system might face and to fully protect the architecture against them. Then this chapter focuses on solutions which have been recognized as secure against spoofing, relocation and replay attacks. Four well known solutions are finally presented: *XOM, PE-ICE, TEC-Tree* and *AEGIS*.

Contents

1.1 Embedded systems

The number of embedded systems in our life is constantly increasing every day. Cell phone, set-top box, PDA are some of the consumer electronic embedded systems which are widely spreading every where. "An embedded system is usually cost sensitive" [Berger, 2001]. It is particularly true for consumer electronics where each product is sold thousands of copies. An embedded system is built to fit with some specific requirements for a dedicated application or a set of applications. Now several functionalities are merging in one component. A good example is the iPhone which is now including cell phone capabilities, a GPS and the Ipod functionalities. By dedicating the architecture (hardware and software) to one specific task (image processing, signal processing), the designer minimizes his product cost. The main features which are characterizing an embedded system are: low business cost, low power cost, low memory footprint and low development time.

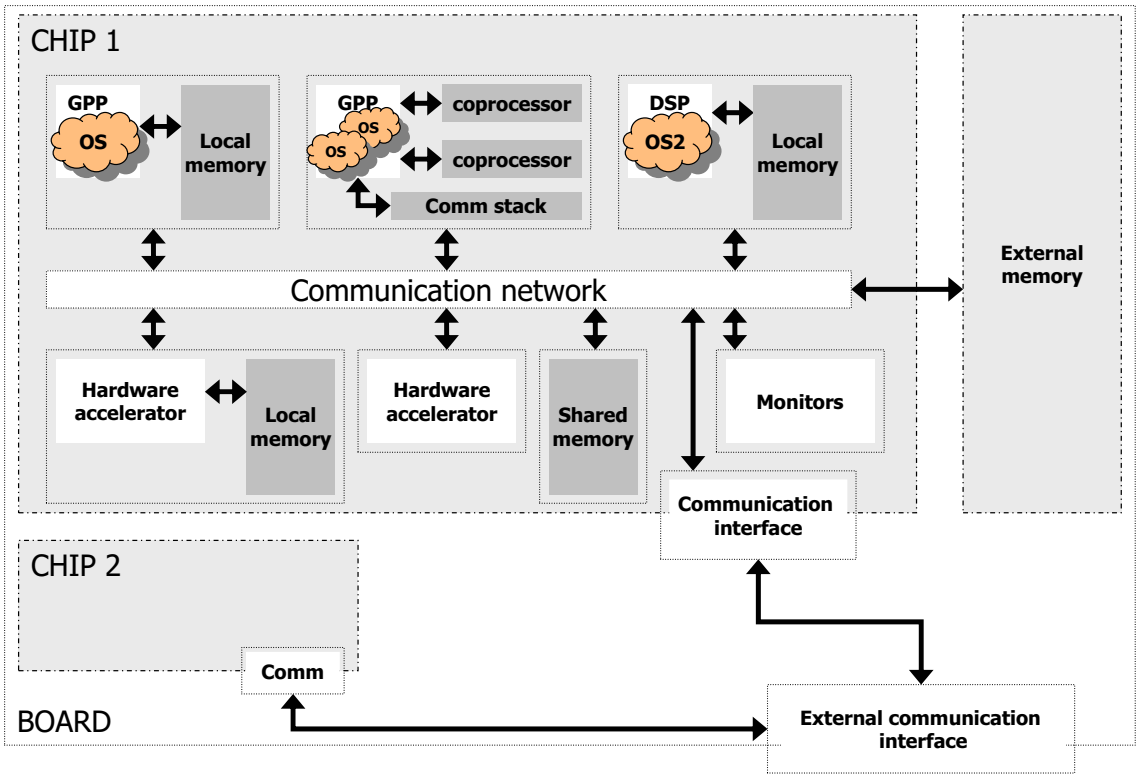

Figure 1.1: Embedded system overview

Figure 1.1 gives an example of what could be an embedded system nowadays. This figure will be used all along the chapter for several comments. One chip can integrate several processor cores. Each core can have its own hardware capabilities like coprocessors, instruction set extension and cache memories. Furthermore, inside the chip it is possible to find some dedicated hardware (accelerator) to specific tasks.

Some software tasks does not fit with the application constraints (throughput, time constraint) as a consequence dedicated hardware accelerators manage these tasks. The memory amount which can be included in the chip is also increasing. These memories can be locally used by one processor core or shared with other cores. All the cores inside the chip need communication network or buses. The application partitioning on the different cores leads to some result exchanges or synchronization events. In addition, several chips can compose a macro-architecture at a board level. Again the chips need to exchange some data between them on the board but also with some external entities out of the board. The complexity of such a system is high but it is always designed to exactly match with the applications/processes needs. Cost is the keyword associated with embedded systems.

But the notion of embedded system is not limited to hardware features. The software is also taking an important place. It is becoming usual to find embedded systems with one or several Operating System (OS) running on one or more processors. Even at a software level, the OS features need to fit with the specific requirements of the application. The software choices have an impact also on the hardware. For example the memory size will be mostly dependent on the way the software application is built and the number of OS running on the core.

All the features mentioned above are well known by embedded systems designers. But now a new feature is rising in this field, it is security. The need for security is emerging on different places [Ravi et al., 2004a] [Ravi et al., 2004b]. As said previously, the entity can communicate with external entities and this particular point can lead to some security issues. The data exchanged might be sensitive so they need to be protected with confidentiality and integrity for example. Moreover, the time, money and intellectual properties spent in the hardware and software development must not be stolen. It is important for a company to know that the software function or their algorithms can not be stolen too. With architectures including programmable logic such as FPGA, the configuration bitstream also needs to be protected for two reasons. First the IP included in the bitstream must not be stolen and also if the bitstream is modified the functionality of the system might be tampered. In the same way, the product and the service provided by the product must be available. If the system is the target of some attacks, the company wishes to guarantee that their product will keep running.

Some dedicated solutions for embedded systems are emerging. These new solutions have to fit with the tight requirements of embedded systems. Most of the propositions are currently relying on cryptographic principles to prevent the system to crash or to leak some information.

1.2 Security basis

1.2.1 Generalities

A system attacker has basically three choices. The first one is trying to extract some information: data or code from the memory or ciphering key from the hardware architecture. The second choice is to simply put the system out of order. The last possibility he has, would be to take the control of the system. In this case, the attacker is just trying to crash the system to stop it from providing a service. In order to answer these issues, the notion of *security* is based on five principles which are supposed to guarantee the correct execution of programs and communications:

- Confidentiality: only the entities involved in the execution or the communication can have access to the data.
- Integrity: the message must not be damaged during the transfer or storage time, or the program must not be altered for the execution, if so, the system must detect it.
- Availability: the message or the program must be available for the processor or the application for example.
- Authenticity: the entity must be sure that the message comes from the right entity or the system must trust the program source code.
- Non-repudiation: the entities implied in the exchange must not have the possibility to deny that they have taken part in the exchange.

Cryptography is the solution to these issues at least for confidentiality. The information encryption is used for confidentiality. For example, the encryption or the decryption key owners are the only entities able to communicate together. The most popular cipher algorithms are: RSA [RSA, 1998], ECC [ECC, 2002], AES [AES, 2001], 3DES [3DES, 1995]. RSA and ECC are asymmetric cipher algorithms. With asymmetric cipher algorithm, the key used to encrypt (public key) the message is different from the key used to decrypt the message (private key). As the key used to encrypt the message is public, everyone can send a ciphered message to the entities which own the private key to decrypt it. AES and 3DES are symmetric cipher algorithms. The key used for ciphering is the same as the one used for deciphering. In order to guarantee the data confidentiality, the key must be only known by the entities involved in the exchange.

The information hash is commonly used to check the message integrity by providing a signature which is unique for each message. Usually a message and its hash value are sent to the addressee. If the computed hash value matches with the received one, the addressee can consider that the message integrity is safe. The most known algorithms are MD5 [MD5, 1992] and SHA family [SHA-1, 1995] [SHA-2, 2001]. The

SHA family robustness varies according to the number of bits used to code the signature. In addition, Message authentication code (MAC) is used to authentication thanks to an algorithm relying on a key. A MAC value guarantees the message integrity and authenticity for the entities with also has the key. Most of all these algorithms are used in embedded systems. The next section shows some typical security implementations and applications.

1.2.2 Security primitives in embedded systems

The security issues in embedded systems can be found at different levels. At a software level first, it is possible to implement ciphering algorithms such as AES or RSA to protect some data that might be exchanged through an insecure channel with another entity. The advantage of such a solution is the simplicity and the time to have a functional result. On the other hand, it is known that the performance and especially the throughput of cipher algorithms in software is quite low. Indeed, it could be possible to integrate a coprocessor or an accelerator with the AES or RSA functionality. This way the throughput would be higher and the processor will be free for other critical tasks that must be done in software especially with hard real time applications.

Concerning the availability of the system, the data partitioning protection and the protection against system crash, some solutions exist at software level. The virtualization is an answer to some of these problems. Two operating systems can run together on the same target (Figure 1.1), one OS is dedicated to secure tasks and the second runs non secure tasks. By having two OS with two different memory spaces [TRANGO, 2008], the designer is sure that one task for OS2 will not access some sensitive data from OS1 for example. The use of two OS leads to some mechanisms to exchange data between the two kernels.

At a hardware level, security can also be guaranteed but in this case the cost of security will not be some extra memory to store cipher algorithms or several OS kernels. With hardware primitives, the security impact will appear on the design size and also the power consumption of the system. Many embedded systems are battery powered systems, so energy/power is a critical feature. These two points can be mitigated by the fact that the throughput of the hardware primitive will be higher than the same primitive implemented in software. Furthermore, some security issues can not be solved in software (data protection in the system memory for example). If a designer wants to cipher and hash the data from the memory on the fly, he had to use a hardware mechanism based on cipher algorithms if we does not want to decrease the global system performances. Usually, the AES algorithm is preferred. Indeed, the time to obtain the ciphertext based on the plaintext is shorter than with other algorithms such as RSA. For data hashing, MD5 or SHA-1 cores are relatively efficient solutions and guarantee data integrity. The implementation of security primitives in hardware leads to some new possible attacks on the systems.

The next section details the specific attacks on embedded systems.

1.3 Attacks survey in embedded systems

1.3.1 Hardware attacks

Attacks on embedded systems can be classified in two kinds. First of all, the hardware attacks are using some architectural features or leakage to attack or extract some information from the system. When the attacker wants to decrypt information, he needs to have the cipher key. A solution to get cipher keys is to observe leakage such as power consumption, electromagnetic emissions or noise. This kind of attacks are called *side channel attacks* [Guilley and Pacalet, 2004]. The most known relies on the algorithm power signature [Kocher et al., 1999]. By analyzing the algorithm power signature, it is possible to infer the algorithm round. Moreover, a differential analysis combined with a statistic study of the power signature can lead to a cipher key extraction [Kocher et al., 1999]. These two methods are called SPA: *Simple Power Analysis* and DPA: *Differential Power Analysis* [Kocher et al., 1999]. Similar solution also works with electromagnetic emissions [Agrawal et al., 2003] (*Differential Electromagnetic Analysis*). Instead of analyzing the power signature, the chip electromagnetic signature is studied and like *DPA* the ciphering key can be extracted. A significant remark concerns the cost of such attacks. It is especially cheaper than reverse engineering attack which needs an electronic microscope to study the structure. On the other hand, with attacks like DPA a lot of measurement are necessary usually thousands and even more.

Temporal analysis or timing attacks [Kocher, 1996] are other ways to catch cipher keys. The system temporal reaction leaks information which enables the extraction of cipher key or password. Like with the *DPA*, it is necessary to make assumptions concerning the information to be extracted. The software algorithm implementation knowledge, like the branch instructions in the program, can also help to obtain a secret since the algorithm model can be established [Kocher, 1996]. Indeed, timing hypotheses can be done as the program running on the target is often known. Thanks to statistic studies, sensitive information can be extracted (cipher key).

Fault injection [Liardet and Teglia, 2004] is the last way to obtain secrets with hardware attack. However, like reverse engineering, the material needed is more important than previous attacks. The fault injection into a system through a memory corresponds to a modification of one or several bits (laser or electromagnetic waves). The knowledge of the algorithm implementation is an important point to determine a secret. In most cases, the fault injection is done in the last round of the algorithm [Liardet and Teglia, 2004]. The reason is that the fault mark is more visible in the ciphered result. The attacker is not always looking to attack a specific part of the designer to get some information. Sometime he just want to modify the behavior of the system tanks to a fault injection. The memory is not the only design part which

can be the target of injection. Indeed, it is also possible to inject some errors in the processor datapath. Like with memory fault injection, the processor behavior will be different and information can be extracted or some instructions can be executed.

The goal of hardware attacks presented above, is to get secret information from the chip. *Denial of service* attacks are different and aim to put the system out of order. In autonomous embedded systems, power is an essential concern. It is one of the most important constraints on the system. As an example with a cell phone or a PDA, the attacker can perform a large number of requests which aim to activate the battery and to reduce the system lifetime [Nash et al., 2005] [Martin et al., 2004]. In wireless communication systems, another attack solicits the transmitter antenna in order to consume more power and reduce the system lifetime. Increasing the processor workload is also a good way to consume more power. Indeed, the workload is related to power consumption, so an assailant may try to force the processor to work more [Martin et al., 2004] [Nash et al., 2005]. As a consequence the lifetime will be affected. Other ways can be used to put a system out of order. Taking control of the temperature regulation system is a solution. Through the control of the regulator, it is possible to increase the temperature which will activate the overheat security mechanisms [Dadvar and Skadron, 2005]. Moreover, by increasing the system temperature, it also increases the power consumption of the processor which leads to a battery lifetime reduction, too.

The panel of attacks against a system is important and depends on several parameters: goal, budget and nature of the system. Hardware attacks represent an important threat against embedded systems.

1.3.2 Software attacks

The second kind of attack an embedded system might face, is software attack. Like computer systems (server and workstation), embedded systems are more and more affected by virus and worms [Dagon et al., 2004]. There is a difference between a virus and a worm. A virus needs the human help to infect a system and to spread contrary to a worm which does not need any human help. A worm is considered to be autonomous. All the computer science concepts can be transposed to the embedded system domain. The program substitution by a malicious one is a threat for the system security. The malicious program may try to get access to private data or shut down the system. Concerning secret data, cipher keys are the most sensitive as once the attacker knows the cipher keys, he has access to all the information in clear. Encrypting memory and protecting cipher keys correspond to classical solutions against these attacks. However protections used in computer systems are not suited for embedded systems (less computing power and memory). Thus various solutions dedicated to embedded systems are emerging (e.g. bus [Elbaz et al., 2005] or program monitoring [Mao and Wolf, 2007]).

Some software attacks are specific to the embedded system field. They are taking in account some weaknesses of embedded systems. As stated above, embedded systems are often battery powered architectures, and the battery is often seen as a weakness. For example a virus or a worm can be sent several times on a same system to launch the antivirus. Scanning all the system increases processor workload and thus decreases the battery lifetime. The combination of several attacks on a system can lead to some very complex detection [Ravi et al., 2004a]. The field of embedded systems extends the scope of potential threats.

1.4 Threat model

As shown in the previous section, the potential threats a system may face are really important. It is not possible to address all the embedded system security issues in one proposition. The architecture main memory is a weak part. Indeed, with external memory, an attacker can easily probe the bus interface between a processor and the memory or physically attack the memory itself with fault injection for example. Bus probing results in the collection of address and data values which can be used to uncover processor behavior. The encryption of data values using algorithms such as the Advanced Encryption Standard (AES) [AES, 2001] or Triple Data Encryption Standard (3DES) [3DES, 1995] prior to their external transfer guarantees data confidentiality. Data ciphered with these algorithms cannot be retrieved without the associated key. However, even ciphered data and their associated addresses leave memory values vulnerable to attack. Well-known attacks include the following:

- A *spoofing* attack occurs when an attacker places a fake data value in memory, causing the system to malfunction (Figure 1.2).

- A *relocation* attack occurs when a valid data value is copied to one or more additional memory locations (Figure 1.3). The main advantage of a relocation attack over a spoofing attack is the fact that if the memory is encrypted with the same scheme, the attacker will be able to modify and/or control the software behavior by swapping some encrypted values.

- A *replay* attack occurs when a data value which was previously stored in a memory location is substituted for a new data value which overwrote the old location (Figure 1.4). With a system protected against relocation attacks, the replay attack is a good way to overcome the relocation attacks protection. Instead of performing a spatial data swapping in the memory, the attacker is doing a timing data swapping. As shown on Figure 1.4, he registered a value at time 150 and just replays this value at time 550. This way, he simply overcomes the protection against relocation attacks and can then modify the software behavior.

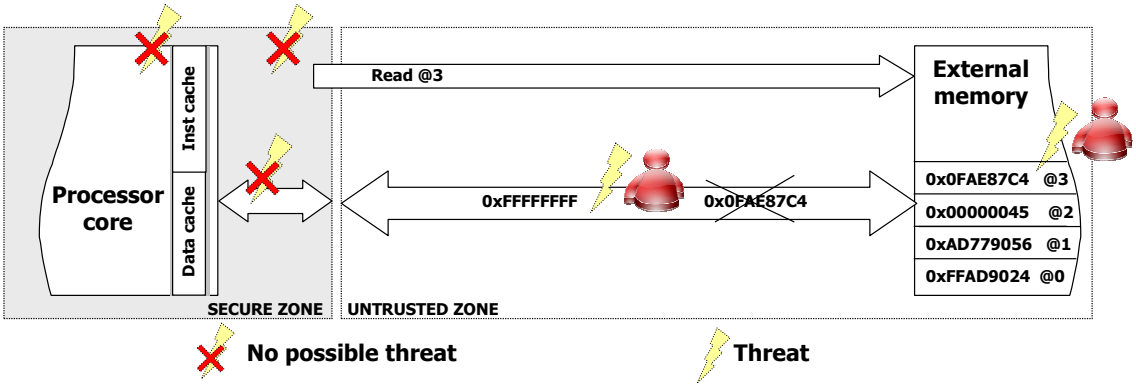

Figure 1.2: Example of a spoofing attack

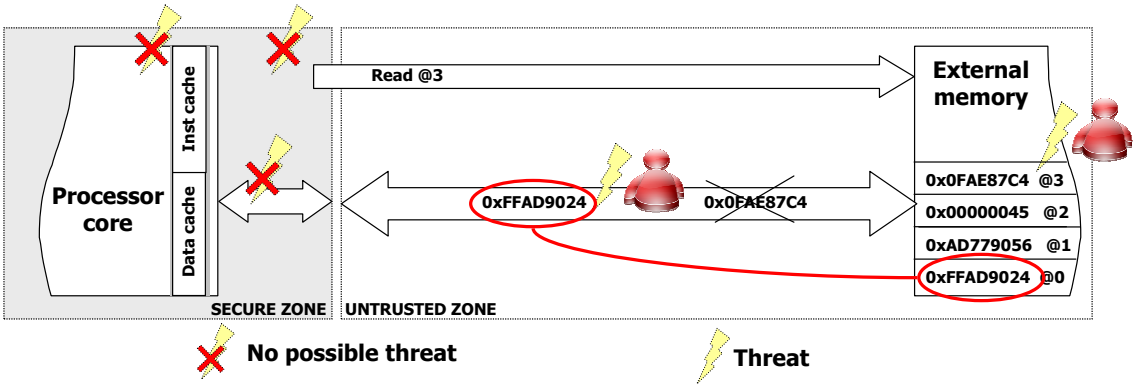

Figure 1.3: Example of a relocation attack

For all three cases, it may be possible for an embedded system to successfully decrypt data, but system behavior will be negatively impacted. Processor instructions are particularly vulnerable to relocation attacks since specific instruction sequences can be repeated in an effort to force a system to a specific state. Specific approaches that maintain the integrity of data from these types of attacks are needed to secure embedded systems.

In the following, attacks such as side-channel or fault injection in the processor hardware or in the security primitive hardware will not be handled. Solutions to partially prevent these attacks are existing (for example dual rail logic [Sokolov et al., 2005]) but these solutions are included in the synthesis flow and constrain the logic layout. The work presented here is at a system level as a consequence the processor core, cache and accelerator will be considered as included in a secure zone or trust zone that the attacker can not access (Figure 1.5). In the same way, the security primitive will be considered as protected against hardware attacks. Several works have been proposed in order to securely implement in hardware security primitives (dual rail [Sokolov et al., 2005], asynchronous logic). The external memory and the bus connecting the memory chip to the main chip (processor or

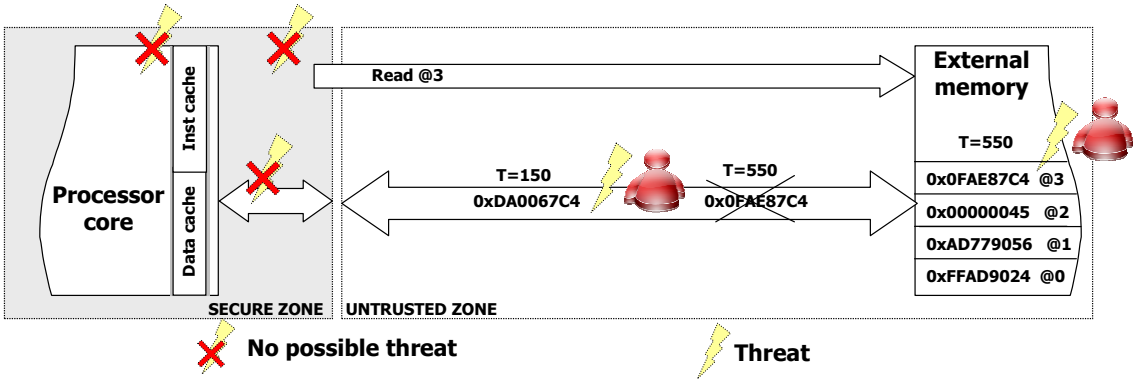

Figure 1.4: Example of a replay attack

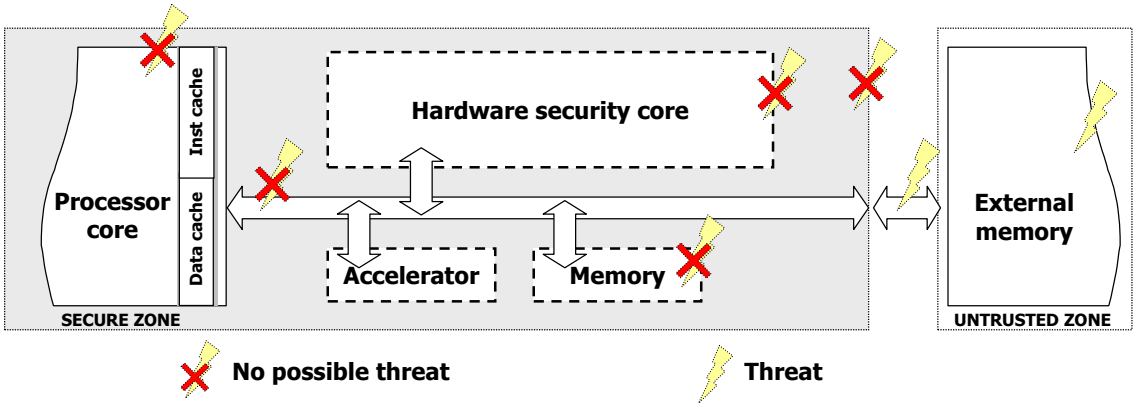

Figure 1.5: Threat model with secure zone and attacks of bus and memory

FPGA) will be the target in our threat model. The threat model and the secure architecture proposed have been used several time and can be considered as a typical case study. Usually, the typical architecture targeted is a set-top box. Indeed, the system is on field and the attacker can easily open the set-top box and spy the bus exchanges. In addition, it is really cheap for an attacker to spy the bus, no expensive material or specific knowledge is mandatory contrary to fault injection and so on. That is the reason why the chosen threat model seems to match with the on-field application. Each approach tries to fill the security gap with different solutions, but in all the cases the targeted attacks are the same because they cover a large number of security holes.

1.5 Existing solutions for data protection

The defined threat model limits the attacks to external memory. The best way to guarantee a secure execution of the application stored in the memory is to guarantee the confidentiality (thanks to ciphering) and integrity (thanks to hashing) of the application code and data. Based on that statement different schemes can be used to cipher and hash data (see Figure 1.6). Three operations scheduling are known:

- *Encrypt-then-Hash* (Figure 1.6.a): With this scheme, the deciphering and the integrity checking are parallelizable so the data can be ready and used faster but the tag will not be ciphered with the data.
- *Hash-then-Encrypt* (Figure 1.6.b): This scheme is the slowest one because neither the first step nor the second are parallelizable but this time the tag is ciphered with the data. And it can be considered as more secure.
- *Encrypt-and-Hash* (Figure 1.6.c): With this last scheme, the first step is parallelizable but the tag is not ciphered. Moreover, it is the plaintext which is used to obtain the tag. It means that the attacker will have a few more information about the plaintext.

In any of these schemes, the first step and the last step are not parallelizable. Some of them can be considered more secure depending on the information leakage due to the operation scheduling. The latency added by protection will be fully dependent on the way the ciphering and the integrity checking are implemented. In embedded systems, the memory fetching can be used to hide some ciphering computations for example. Figure 1.7 gives a comparison with a AES based ciphering solution and a One-Time Pad (OTP see next chapter for details) ciphering. With this type of architectural trick, it is possible to minimize the security latency penalty. Usually the AES mode used in this case is Electronic Code Book (ECB) mode. Indeed, values form the memory are accessed randomly which means that each block is ciphered and deciphered independently from other blocks that is why it is not possible to use mode using a chaining between the data. The next section will introduce four solutions which are known to match with the threat model established

before.

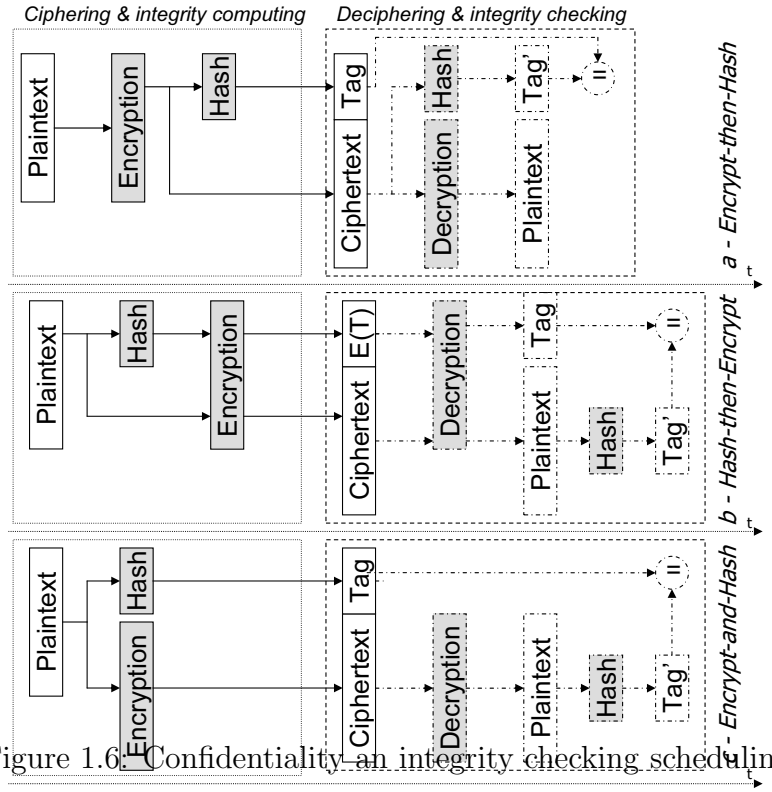

Figure 1.6: Confidentiality an integrity checking scheduling

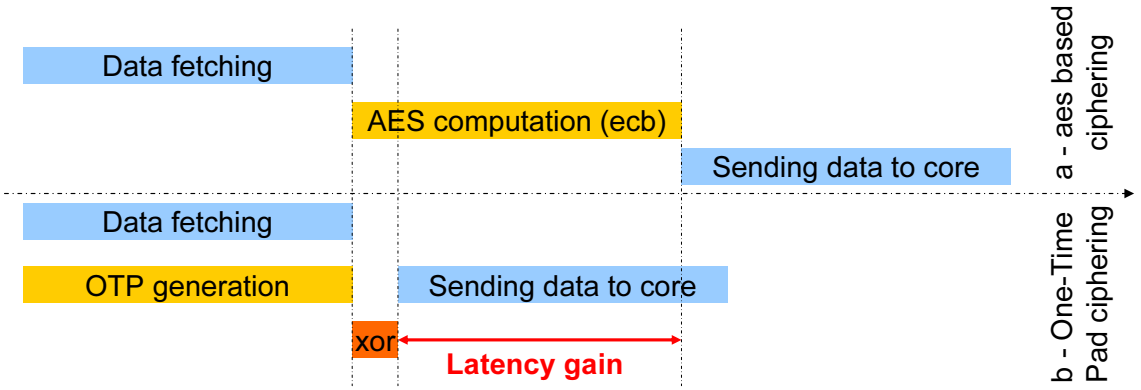

Figure 1.7: Latency comparison of two implementations of data ciphering

1.5.1 Execute-Only Memory (*XOM*)

XOM [Lie et al., 2003] is a security solution from Stanford University. The *XOM* approach, which provides memory protection, is based on a complex key management. The main *XOM* features are: data ciphering, data hashing, data partitioning, interruption and context switching protection. Figure 1.8 and 1.9 give an overview

of the *XOM* architecture and mechanisms. All the security primitives are included in the trusted zone. The only security information which are not in the trusted zone are the session keys. That is why *XOM* owns a complex key management to guarantee a secure architecture.

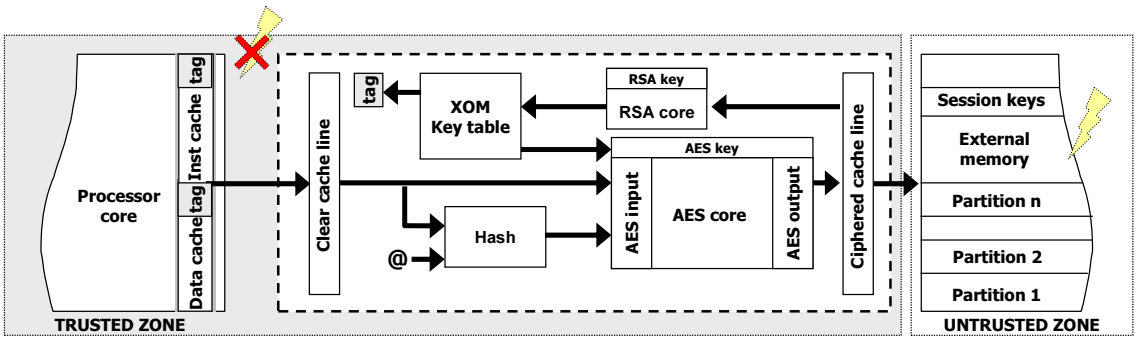

Figure 1.8: XOM architecture for write request

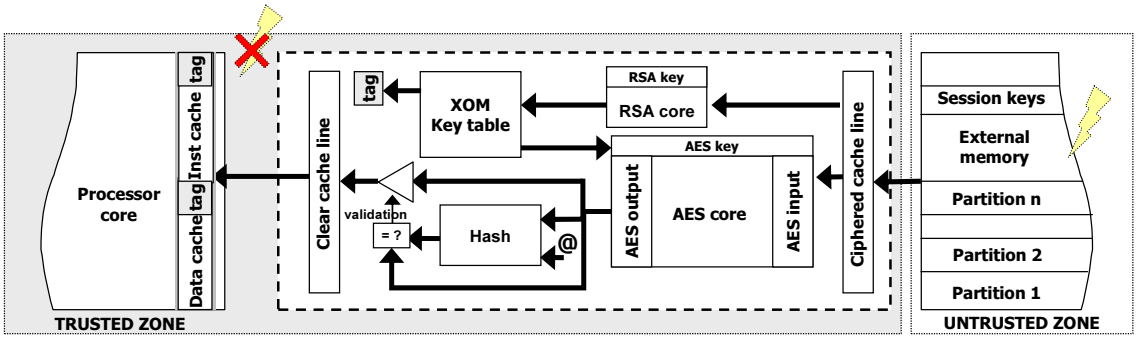

Figure 1.9: XOM architecture for read request

In order to guarantee the data confidentiality and integrity, each memory partition is associated with a session key which is needed to decrypt its content. Encrypted session keys are stored in main memory and can be decrypted using an asymmetric cipher algorithm (RSA in *XOM* case). Decrypted session keys are stored in the XOM key table (in the secure zone). The private key required for the asymmetric decryption is stored in the secure zone of the architecture (RSA key in Figures 1.8 & 1.9). The algorithm used for the symmetric deciphering is an AES 256 (256 bits key and 256 bits data input). For write requests, a hash value of the data and its address are concatenated with the data before ciphering with AES. The use of the address in the hash value is there to prevent the relocation attacks. When the core produces a cache miss for a read request, the 256 bits read from the memory need to be decrypted (Figure 1.9). Data integrity is ensured by a hash value relying on a MD5 computation. The hash of the deciphered data and its address are compared with deciphered hash value. If the new computed hash value

matches with the deciphered one the data is considered secure and can be used by the processor. The *XOM* security scheme is typically the one of scheme b in Figure 1.6.

In addition, the data stored in cache memory are associated with an identifier or tag in oder to guarantee the data partitioning at a cache level. When a task needs to use a data, the task identifier must be the same as the data, in that case it means the task is allowed to access the data. The tag value are provided by the *XOM* key table which also manages this part.

The last XOM security feature concerns OS preemption which has similarities with interruptions management (in both cases register values are *pushed* into the stack). During a context switching, if the architecture is running a secure application, the register values need to be stored securely (values are ciphered, hashed then stored in the external memory located in the untrusted zone). It is essential to store and protect the context in order to fend off an attack who aims to change some register values. *XOMOS*, which is the software extension of a non-secure OS, brings the software support for the new security primitives added in hardware.

All the protections added by this solution have a cost. The first one concerns the XOM implementation in an existing OS. A work is necessary on the OS kernel to add the instructions which help for the hardware security primitives use. All this work is transparent for the kernel user. According to the figures from [Lie et al., 2003], a real overhead appears in the cache management (cache miss raises from 10 to 40% depending on the application). This raise is mainly due to the information added in the cache to secure the data. Indeed, by adding some data tagging, some space in the cache memory is lost compared with a non protected solution. Moreover, all the security features are bringing some latency in the system to obtain the data in clear (data de/ciphering, hashing, tag checking). Even if these security primitives are done in hardware, the general architecture performances are slowed down. As shown on Figure 1.6.b the decryption needs to be done before the integrity checking. These two operations are not done in parallel, so some more latency is added. Some latency is also added to the software execution because of some software security primitive (secure context switching add some specific instruction for example).

The first proposed version of *XOM* [Lie et al., 2003] is known to have security holes like no protection against replay attacks. In [Yang et al., 2003], the authors extended the proposition and replaced the AES-based ciphering scheme (Figure 1.7.a) with a system based on OTP to guarantee protection against replay attacks and also to increase the performances of the system (Figure 1.7.b). Concerning the global security level of the *XOM* architecture, the attack possibilities are fully dependent on the integrity checking capabilities. To succeed, the attacker must be able to pass through the integrity checking in order to execute his own program or use his own data. He may exploit some collisions in the hash algorithm used. For example, with

MD5 the signature is 128 bits long. If he wishes to attack the system, he needs to find two inputs which will produce the same result with MD5. So he has one chance out of 2^{128} to get the same result. The security level of the *XOM* mainly depends on the hash algorithm used, because SHA-1 could be used for integrity checking. In this case the signature would be 160 bits long and the probability of success would be one out of 2^{160}.

1.5.2 PE-ICE / TEC-tree

PE-ICE architecture [Elbaz et al., 2006] has been proposed by the Montpellier Laboratory of Computer Science, Robotics, and Microelectronics (*LIRMM*). *PE-ICE* uses the spreading feature of block ciphering algorithms such as AES to provide system confidentiality and integrity.

Like *XOM*, a tag is added to the data before ciphering (scheme b in Figure 1.6). The main difference between *PE-ICE* and common solutions based on ciphering and hashing is that *PE-ICE* only relies on the AES ciphering scheme to guarantee integrity. There is no hardware hashing material added to the architecture (see Figures 1.10 & 1.11).

For read-only values, the tag used for integrity checking is the memory address to prevent relocation attacks. For read-write values, the address and a random value are included to prevent relocation and replay attacks. The main *PE-ICE* assumption is that due to the spreading feature of AES, if one memory bit is modified, a huge impact will appear in the deciphered value. Indeed, the output of an AES is directly influenced by the input. As the plaintext is composed of the data and the tag, when the system performs a comparison between the deciphered tag and the original one concatenated with the data (Figure 1.11), it can detect if the data integrity has been maintained or not. Like *XOM*, *PE-ICE* has an impact on memory read latencies since decryption can only be performed after the read of a full cache line from external memory (Figure 1.7.a). Integrity checking is performed using a comparator for the address and the tag, so the amount of logic needed to guarantee integrity is not significant.

TEC-Tree stands for Tamper-Evident Counter Tree [Elbaz et al., 2007]. *TEC-Tree* is a work which appears to be similar to the Merkle hash tree proposition [Merkle, 1980]. The Merkle tree solution relies on standard hashing algorithms such as MD5 or the SHA family. Figure 1.12.b shows an example of a hash tree. The main issues with this kind of algorithm are the latency required to obtain the message signature and the vulnerability to replay attacks. The latency to check the integrity of data depends on the number of nodes that need to be computed before reaching the root of the tree. The latency is directly affected by the hash algorithm used to travel from one node to another.

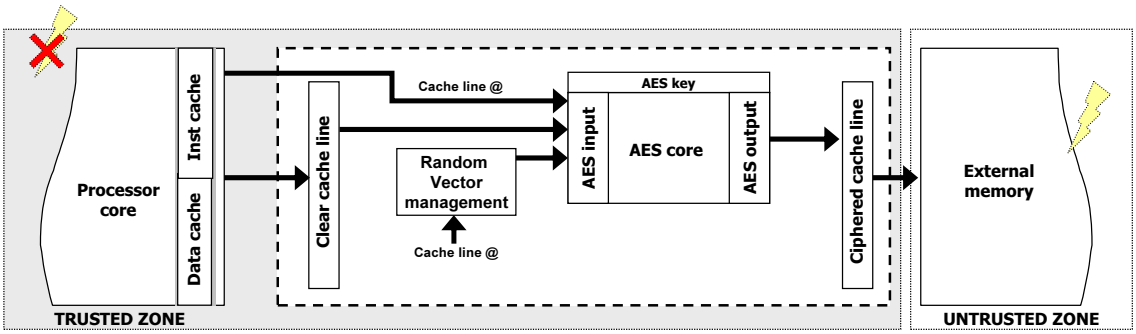

Figure 1.10: PE-ICE architecture for write request

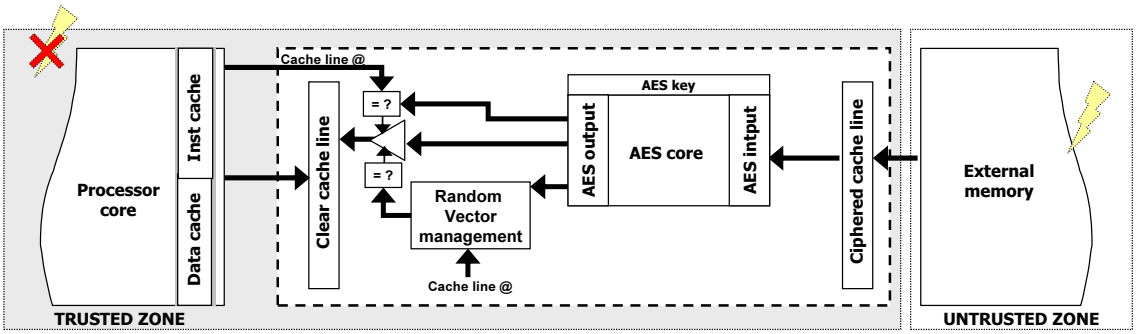

Figure 1.11: PE-ICE architecture for read request

In order to tackle these issues, the authors proposed in [Elbaz et al., 2007] to use the *PE-ICE* architecture instead of standard hash algorithm. Figure 1.12.a details the way the tree is built with *TEC-Tree*. Here the solution used to go from one node to another is *PE-ICE*. The use of *PE-ICE* in the *TEC-Tree* mechanism leads to a solution which guarantees the same protection for the data. A leaf of the tree corresponds to the ciphered data and its tag. A tag in *TEC-Tree* is composed of the address (to prevent relocation attacks) and a counter value (to prevent replay attack). For the tree node, several tags are concatenated together to create a new data which is also protected with the *PE-ICE* mechanism. The root of the tree is the last tag. One major benefit of *PE-ICE* is the way the solution is built naturally protect the system against replay attacks. In addition, the latency to obtain the integrity checking result is shorter than the Merkle hash tree. Indeed an AES computation requires from 11 to 14 cycles compared with MD5 or SHA-1 which are requiring from 64 to 80 cycles and which do not protect against replay attacks. Another benefit of *TEC-Tree* is that only the root of the tree is stored in the secure zone of the system. It minimizes the size of the secure zone. Moreover, contrary to *PE-ICE*, no on-chip memory is necessary to store the counter values because they are stored securely in the off-chip memory thank to the tree.

The security level of *PE-ICE* and *TEC-Tree* relies on the size of the tag. Indeed,

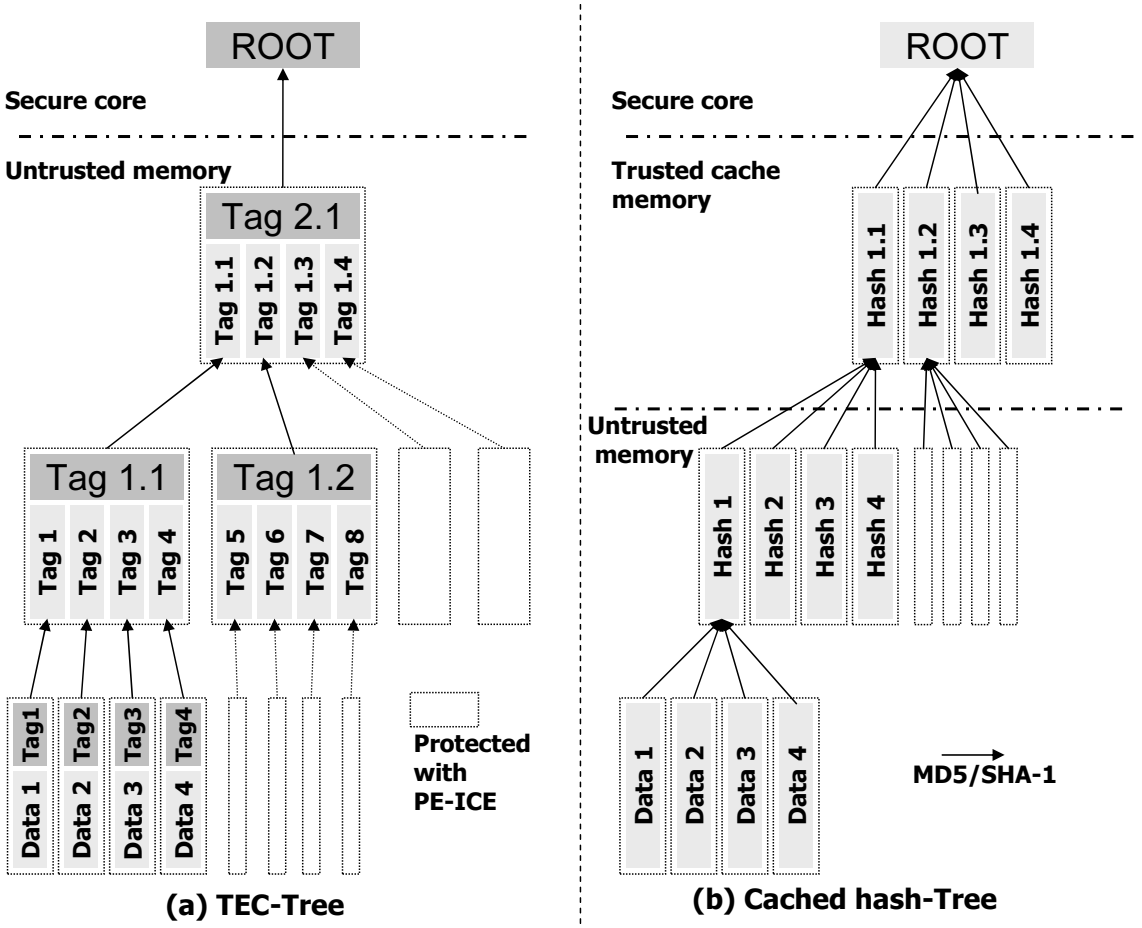

Figure 1.12: Secure tree implementation

to guarantee integrity, the system checks if the deciphered tag matches with the address and the random vector or counter value for read/write data. In this case, the possibility for an attacker to succeed is to obtain the same tag value for two different data. In most of the cases, PE-ICE uses a 32 bits tag which means for the attackers one chance out of 2^{32} to succeed. *TEC-tree* mainly relies on the *PE-ICE* mechanisms, so the security level can be considered as equivalent. With *TEC-Tree*, the tag is 64 bits, so the attack probability is one out of $2^{64.}$

1.5.3 AEGIS

AEGIS [Suh et al., 2008] [Suh et al., 2005] is an additional memory security solution from Massachusetts Institute of Technology. The confidentiality in the AEGIS solution relies on OTP encryption [Suh et al., 2003]. The mechanisms used in OTP will be detailed in Chapter 2. This encryption method has typically a small impact on memory latency (Figure 1.7.b) at the cost of memory space. The solution used by *AEGIS* for integrity checking is called *cached hash tree*. This hashing approach is similar to a *Merkle tree* [Merkle, 1980] but to increase the efficiency of the method

some hash tree nodes are stored in a cache memory included in the secure zone. The advantage is that instead of computing all the tree nodes to the root, the system just needs to compute the values until one value from the secured memory is reached. The only weakness of this solution is the architecture performance. It is fully dependent on the cache memory size used to store the secured nodes. The architecture performance also depends on the algorithm used for hashing. As said above, SHA-1 computation requires 80 cycles and MD5 only 64. The security scheme used by *AEGIS* is the one from Figure 1.6.c.

In addition to memory confidentiality and integrity protection, *AEGIS* also provides other security features. The *Physical Unclonable Function* (*PUF*) is an hardware mechanism which provides a unique secret associated to a chip. The propagation time within the chip corresponds to the base of the PUF. PUF is a random source used to create the secret which is based on a sequence of multiplexers giving a bit as a result. The fabrication process of integrated circuit is the source of the uniqueness of the propagation delay (each integrated circuit has its own delay). The multiplexer sequence makes the chip unique and the result of the sequence is very difficult to predict. The temperature is a good example of a parameter which can lead to a different behavior of the PUF. That is why a regulation system is required to limit the variation in the result of the sequence since the result sent by the PUF must always be the same (required for cryptography purpose). Moreover, PUF is associated with a hash algorithm to increase the complexity of the secret generation. In order to prevent model-building, one way hash is used to obfuscate the output of the delay circuit. Indeed, it should be easy for the attacker to build a timing model of the PUF and then guess the PUF result.

Memory protection is an important point as the memory is located in a non-secure zone of the architecture. Furthermore, the memory security is also obtained through the MMU (*Memory Management Unit*) which manages the security workspaces levels (user and superuser, secure or not). Each user can choose to cipher (or not) and to hash (or not) the data. Indeed, *AEGIS* also allows the selection of different security levels for different memory segments. This memory mapping must be identified during compilation through new instructions added to the instruction set. The new instructions handle the specific security hardware properties. Like *XOM*, some code needs to be added to the OS to use the hardware security primitives and manage different areas with different security levels. Overall, this approach adds complexity to the processor architecture, the operating system and the specific compiler which needs to be built for this architecture.

AEGIS seems to be a very complete solution to protect memory and program. The overhead is important in several domains. The silicon area increased by 1.9 [Suh et al., 2005]. The CPU core is the part which is the most affected by this overhead. Moreover, all the logic needed to control the specific mechanisms contributes to the increase of the area (OTP core and hash core). The global architecture per-

formances depend on parameters like the sizes of the protected memory and the cache memory.

For security concerns, *AEGIS* like *XOM* depends on the integrity checking capabilities for the hash algorithm used for the merkle tree. In [Suh et al., 2005], the authors use SHA-1 which will lead to a 160 bits signature. It means for the architecture that the success possibility of an attacker is one out of 2^{160}.

Conclusion

Embedded systems are the target of multiple threats (hardware and software). Some model of the potential threats has been defined to address some of them. The security needs for these embedded systems architectures are important and not always adapted to this particular field. Some propositions (*XOM, PE-ICE, TEC-Tree, AEGIS*) have been done to protect the system against the threat defined by the threat model. But the security cost of these proposition to several overhead: area, software execution losses and memory footprint. These three parameters have a direct impact on the power consumption of the architecture which is critical in embedded systems. Moreover, in some cases (*XOM* and *AEGIS*), the hardware architecture is not the only part which required some work. The OS and compiler modifications can be seen as major issues in the use of such solutions.

Table 1.1 gives a summary of the techniques used by each proposition. All these solutions require some extra memory for security. This extra memory is located inside the secure zone (on-chip memory) and depends on the way the security primitives are implemented in the untrusted zone (off-chip memory). TEC-Tree is the only solution with a low on-chip memory footprint (only the tree root). In order to guarantee confidentiality, all the existing solutions rely on AES algorithm. The AES implementation can vary (plaintext as an input of AES core, OTP). The main difference between all these four propositions comes from the integrity checking extension. In addition, it is the integrity checking which badly impacts the security memory footprint (hash value storage). *PE-ICE* is the only architecture where the integrity checking is fast (only the time to compare the tag). For the other propositions, latency to check integrity is really long from a memory latency point of view. This latency is mainly due to the use of slow algorithms like MD5 or SHA-1. For *AEGIS* and *TEC-Tree*, the use of tree to store and guarantee the data confidentiality badly impacts the system latency but the valuexs are stored off-chip. This off-chip storage counterbalances the latency cost.

Solution		XOM	PE-ICE	TEC-Tree	AEGIS
Memory	On-chip	XOM table/TS	Random value	Counter root	TS + Hash cache
	Off-chip	hash value	@ + RV	Counter node	Hash node
Confidentiality		AES/OTP	AES	PE-ICE	OTP
Integrity		hash	Tag checking	PE-ICE	Cached hash tree
Security		$2^{128}/2^{160}$	2^{32}	2^{64}	2^{160}

Table 1.1: Existing solutions summary

The ideal secure solution would be an architecture with a high security level (like XOM and AEGIS) which does not require any on-chip or off-chip memory at all for security data. Of course, the way the architecture should be implemented should not add any latency to obtain the clear data. In the next chapter, a new security scheme will be introduced with the idea of minimizing the security cost for the architecture.

CHAPTER 2

AES-TAC extended with integrity checking

The previous chapter introduced several threats and security solutions. Even if the solutions guarantee the protection against attacks targeted by the threat model, the overhead due to security is important. The aim of this chapter is to propose a security scheme which guarantees data confidentiality and integrity with a reduced overhead in area, memory footprint, performance and power consumption. The security scheme will be detailed and the different architectural features will be evaluated through experiments. The security level of the proposed solution will be estimated and it will be shown that security modularity has an impact on the architecture features.

Contents

2.1 AES-TAC principles

OTP or BASC (Binary Additive Stream Cipher) encryption was initially proposed by Gilbert Vernam during World War I [Anderson, 2001], but was only recently adapted to digital memory protection [Suh et al., 2003]. This protection approach is similar to the counter mode certified by the NIST [NIST, 2001]. The idea is to use the data fetching delay of the memory to compute a random key called keystream. After keystream generation, the result is XORed with the ciphered data to obtain the retrieved plaintext (the fetched data). Each keystream is created before a memory write and is used for encryption. The same keystream is used for subsequent decryption. Usually, the BASC is a random value generated on the fly by the architecture. As the BASC is used for ciphering and deciphering, the generation of the value must be reproducible or the BASC value used for ciphering must be stored in order to be available for the deciphering. With this second solution, the amount of memory needed to store all the BASC will be too much important (same size as the protected memory). Moreover, the BASC values need to be stored securely. Indeed, if an attacker accesses the BASC values, he will be able to decipher the ciphertext and the architecture confidentiality will be broken. As a consequence, the main issue of BASC is how to generate the values. In addition, the generation must be fast and secure.

In most systems, memory accesses require a significant latency to retrieve data from the external memory. As a result, the cache line read latency may be long enough to perform BASC computation. In [Suh et al., 2003], authors propose to use the AES algorithm to generate a keystream so the time to obtain the BASC will be the time to perform the AES computation.

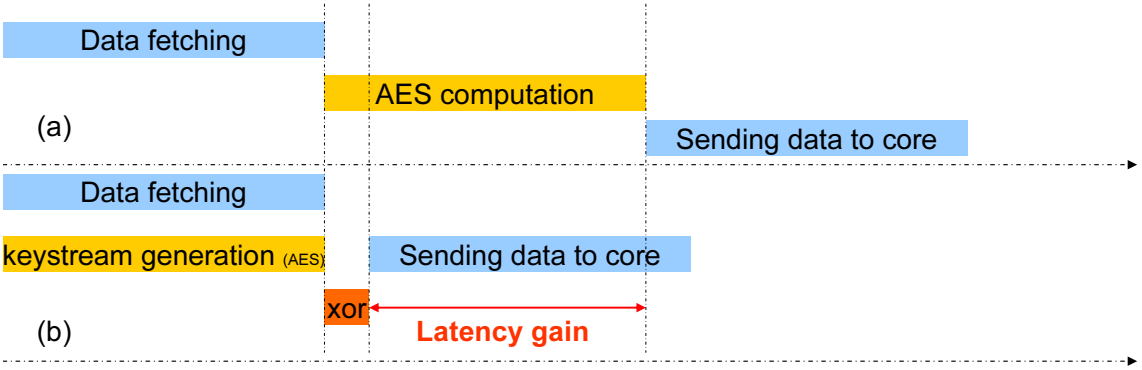

Figure 2.1: Data latency deciphering with short AES computation:
case a: AES-based solution
case b: BASC based solution

As shown in Figure 2.1.b, the latency added by encryption is reduced compared with *case a*. The *case a* typically corresponds to the latency obtained with *PE-ICE* and the first version of *XOM*. These previous solutions use the stored data as the

input for AES. The system needs to wait that data are retrieved from the memory to fill the AES input before starting the deciphering step. In *case b* of Figure 2.1, the latency added by BASC en/decryption is the latency of a logical XOR operation on the data. In general, the time needed to retrieve the data from the memory for decryption is longer than the time needed to compute the BASC with AES. Figure 2.2 gives an example where the data fetching time is shorter than the BASC computation. Even in this case, the figure clearly shows that the latency to obtain the deciphered data is shorter with the BASC scheme than the AES-based solution.

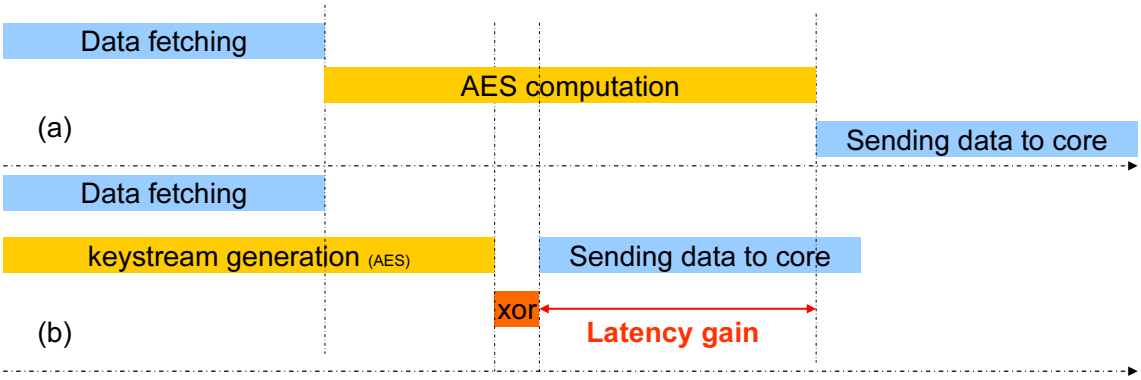

Figure 2.2: Data latency deciphering with long AES computation:
case a: AES-based solution
case b: BASC based solution

Figure 2.3 exhibits the latency cost for data write request. Again, the fact that the keystream computation is done in parallel with the processor request helps to reduce the latency compared with an AES based solution.

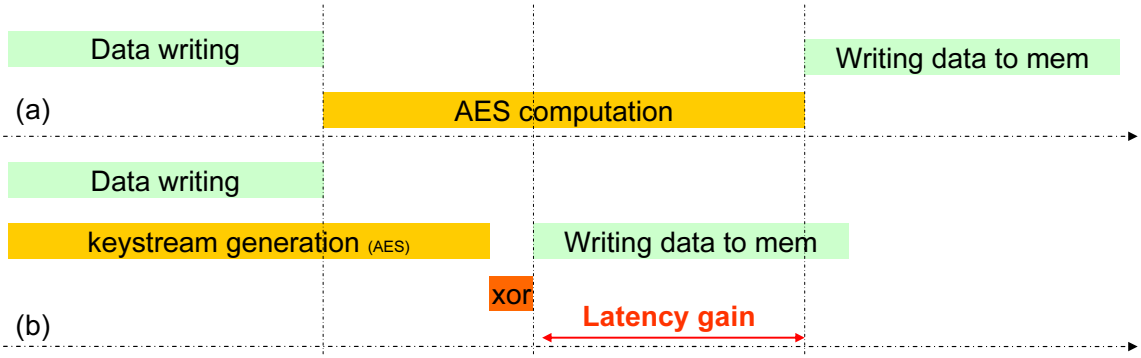

Figure 2.3: Exemple of security cost for data write request to the memory:
case a: AES-based solution
case b: BASC based solution

From a security standpoint, it is essential that the keystream is used only once. If the same keystream is used twice, it will be possible to compare data and see

if they have the same value or not. The keystream is obtained through AES, so the AES inputs also need to be used just one-time because the AES input directly impacts the AES output. There are no data collisions with AES. It means that each input value will have a unique output value (for a same ciphering key). Since keystream computation is supported by AES and since the keystream needs to be used one-time, the AES inputs must be determined the right way. Furthermore, the architecture needs to deal with the attacks defined in the threat model. To protect a system against relocation attacks, the data memory address is used as an AES core input for keystream generation (Figure 2.4). To prevent replay attacks, time stamps (TS) are used. In the following, the AES scheme with TS and address as inputs will be called AES-TAC (AES time address cipher). As shown in Algorithm 1, the TS value associated with each data address is incremented by 1 after each keystream generation. For each new cache line memory write request, the system will compute a different keystream since the value of TS is incremented, that is how the system guarantees protection against replay attacks. The TS values are stored in a memory included in the trust zone (Figure 2.4 and 2.5) for later use during memory read operations. During a read (Figure 2.5), the original TS value (Algorithm 2 operation 1) is retrieved from the time stamp memory and provided to AES during the read request. The result of AES will give the same keystream as the one produced for the write request and the encrypted data will become plaintext after being XORed (Algorithm 2 operation 4).

Algorithm 1 - ***Cache memory write request:***

$1 - Time\ stamp\ incrementation:\ TS(@) = TS(@) + 1$
$2 - keystream\ computation:\ keystream = AES\text{-}TAC\{TS(@), @, Padding\ value\}$
$3 - Ciphered\ data = plaintext \oplus keystream$
$4 - Ciphered\ data \Rightarrow memory$
$5 - Time\ stamp\ storage: TS(@) \Rightarrow TS\ memory$

Algorithm 2 - ***Cache memory read request:***

$1 - Time\ stamp\ loading:\ TS(@) \Leftarrow TS\ memory$
$2 - keystream\ computation:\ keystream = AES\text{-}TAC\{TS(@), @, Padding\ value\}$
$3 - Ciphered\ data\ loading: Ciphered\ data \Leftarrow memory$
$4 - Plaintext = Ciphered\ data \oplus keystream$
$5 - Plaintext \Rightarrow cache\ memory$

Read-only data does not require protection against replay attacks because these data are never modified. No TS values are needed for these data so the amount

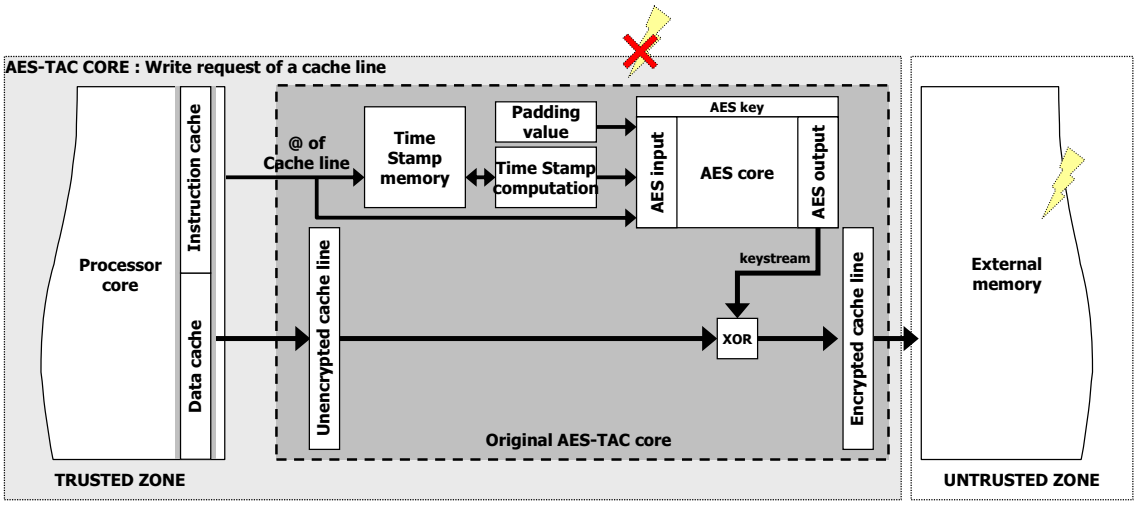

Figure 2.4: Architecture configuration for a write request

of TS memory space can be reduced. Read-only data may be the target of relocation attacks but the address used to compute the keystream guarantees protection against them.

The size of the address and the TS might not be long enough to completely fill the AES core input, so a padding value may be necessary. The value used for padding has no impact on the security feature of the AES-TAC scheme. Even if the attacker has the TS, address and padding values, he will not be able to obtain the generated keystream since he does not know the secret key used by the AES core.

The use of TS and data addresses for AES-TAC protects a system against replay and relocation attacks. If a data value is replayed, the TS used for ciphering will differ from the one used for deciphering. If a data value is relocated, its address will differ from the one used to generate the keystream. In both cases, the deciphered data will be invalid. To use this information, the secure memory access system must be able to detect that the deciphered data is incorrect or has been modified. The AES-TAC implementation is efficient because it performs keystream computation (operation 3 in Algorithm 2) in parallel with memory data requests (operation 4 in Algorithm 2). Figures 2.1 and 2.2 provide a view of this benefit.

2.2 Integrity checking extension for AES-TAC

Thanks to the AES-TAC encryption method, the system can produce wrong value if memory values are modified or relocated. Some mechanisms are necessary to detect these errors and to stop the system from executing the wrong instruction or processing the wrong data. Additionally, integrity checking must be performed with a negligible overhead to minimize latency from a memory latency point of view.

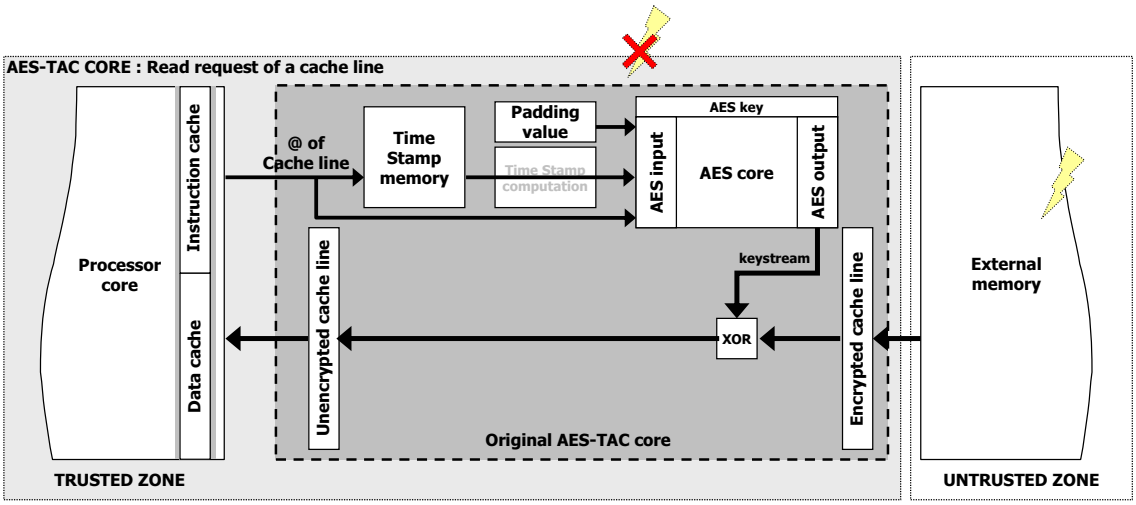

Figure 2.5: Architecture configuration for a read request

The solution to this issue involves the use of a Cyclic Redundancy Check module (CRC) [Peterson and Weldon, 1961].

Algorithm 3 - ***Cache memory write request:***

$1 - \textit{Integrity computation} : CRC\,(@) = CRC\{\textit{plaintext}\}$
$2 - \textit{Time stamp incrementation} : \; TS\,(@) = TS\,(@) + 1$
$3 - \textit{keystream computation} : \; \textit{keystream} = \textit{AES-TAC}\{TS\,(@), @, \textit{Padding value}\}$
$4 - \textit{Ciphered data} = \textit{plaintext} \oplus \textit{keystream}$
$5 - \textit{Ciphered data} \Rightarrow \textit{memory}$
$6 - TS\,(@) \Rightarrow TS\ \textit{memory}$
$7 - \textit{Integrity value storage} : CRC\,(@) \Rightarrow CRC\ \textit{memory}$

Prior to keystream generation, the CRC of the cache line to be encrypted (operation 1 in Algorithm 3) is stored in a cache (operation 7 in Algorithm 3). Later, when the processor core requests a read, the CRC result of the final XOR operation is compared with the CRC value stored in the memory (operation 6 in Algorithm 4). If the data is changed following storage, the CRC of the retrieved value will differ from the stored value, so the attack is detected. As previously stated, the decryption results following a replay or relocation attack will differ, so the CRC will differ. The latency added to the original AES-TAC solution by the integrity checking extension is the latency of CRC computation and checking. This CRC computation can be completed in one clock cycle. With the extended AES-TAC, the minimum latency added to a memory access is the time to obtain the result of the XOR and the CRC check. For a write request, the CRC computation is done in parallel with the ciphering so no latency will be added to this step compare with the AES-TAC

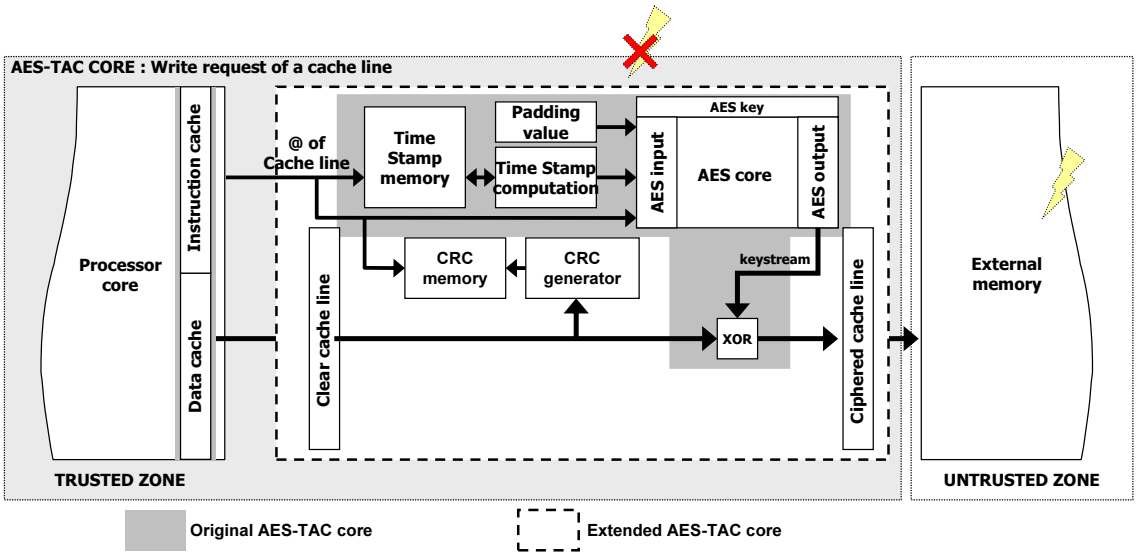

Figure 2.6: Architecture configuration for a write request with integrity checking

encryption process.

Algorithm 4 - ***Cache memory read request:***

$1 - Time\ stamp\ loading:\ TS(@) \Leftarrow TS\ memory$
$2 - Integrity\ value\ loading:\ CRC(@) \Leftarrow CRC\ memory$
$3 - keystream\ computation:\ keystream = AES\text{-}TAC\{TS(@), @, Padding\ value\}$
$4 - Cipered\ data\ loading:\ ciphered\ data \Leftarrow memory$
$5 - Plaintext = Ciphered\ data \oplus keystream$
$6 - Integrity\ checking:\ CRC(@) \equiv CRC\{plaintext\}$
$7 - Plaintext \Rightarrow cache\ memory$

Highlighted operations are only available for the extended AES-TAC solution proposed here with integrity checking

Figure 2.6 and 2.7 show the modification applied to the architecture to handle the integrity checking design. Like the TS memory, the CRC memory is included in the secure zone. Indeed, if the attacker accesses the CRC values or modifies it, the architecture security can not be guaranteed.

2.3 Security level & architecture overhead

In the solutions presented in chapter 1 section 1.5, the security level depends on the integrity checking capabilities of the architecture. It is the same for the proposition

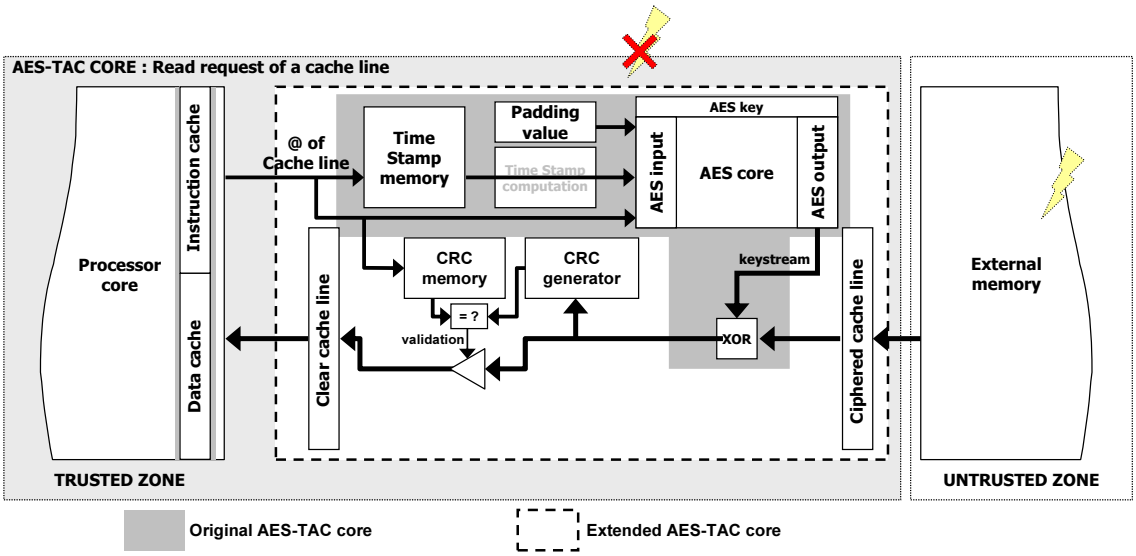

Figure 2.7: Architecture configuration for a read request with integrity checking

described hereafter. But it will be shown in the following that it is possible to have different security levels. The architecture memory footprint and performance will be fully dependent on the security requirements. More memory dedicated to security will lead to better performance and security for the architecture.

2.3.1 Security level

In general, an integrity checking approach based on CRC could be considered as weaker than approaches based on MD5 and SHA-1 algorithms. Although the implementation presented here mitigates the weakness. It is assumed that the attacker does not have access to the plaintext. The CRC computation is performed on a clear cache line and only the ciphered data is stored in memory. Due to AES-TAC encryption, there is no direct link between the result of the CRC computation and the data in memory. In this case, the attacker does not have any information to attack the CRC results because he only has access to the ciphered data.

Previously, the security level of the other solutions was estimated. In the case of this CRC approach, the security level will depend on the CRC result size. To succeed, the attacker must find two ciphertexts which produce the same CRC results. For example, if a CRC 8 bits is used, the chance of success for the attacker will be one on 2^8. Now if the designer wishes to provide an architecture with a higher security level, he will need to use a CRC which produces a longer result (32, 64, 128 bits). The choice of a longer CRC will impact the architecture security level but also other parameters such as the memory to store the CRC results.

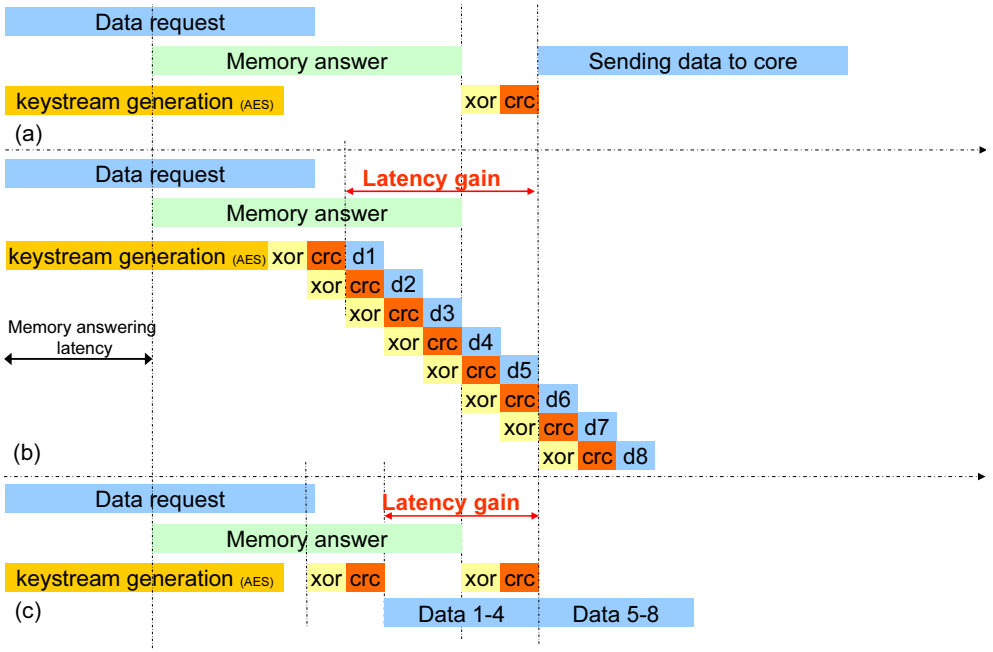

Figure 2.8: Data deciphering and integrity checking with short memory latency:
case a: Non pipeline version (256 bits cache line / 256 bits CRC input)
case b: Pipeline version (256 bits cache line / 32 bits CRC input)
case c: Half pipeline version (256 bits cache line / 128 bits CRC input)

2.3.2 Dependencies between security level & architecture overhead

If the CRC is performed on a full data cache line, the operation can only be done when all the data have been read from the external memory and XORed (*case a* in Figures 2.8 and 2.9). A way to decrease the data retrieval latency is to perform the XOR and CRC on shorter word like the half of the cache line (*case c* in Figures 2.8 and 2.9) or for each word of the cache line (*case b* in Figures 2.8 and 2.9). The Figures 2.8 and 2.9 show that depending on size of the word for the CRC input, the data availability varies.

For the solution with one CRC per cache line (*case a* in Figures 2.8 and 2.9), the latency to get the checked data is long but the memory required to store the CRC will be small because there is only one CRC per cache line. A long latency to get the data means a loss in the execution performance of the architecture. For the solution with a CRC per word (*case b* in Figures 2.8 and 2.9), the latency saved is important but the number of CRC to store will be important (8 CRC results for a cache line for example). As the latency to obtain the checked data is short, the performance penalty on the architecture will be small.

Figures 2.8.c and 2.9.c show a trade-off solution were the word used by the CRC

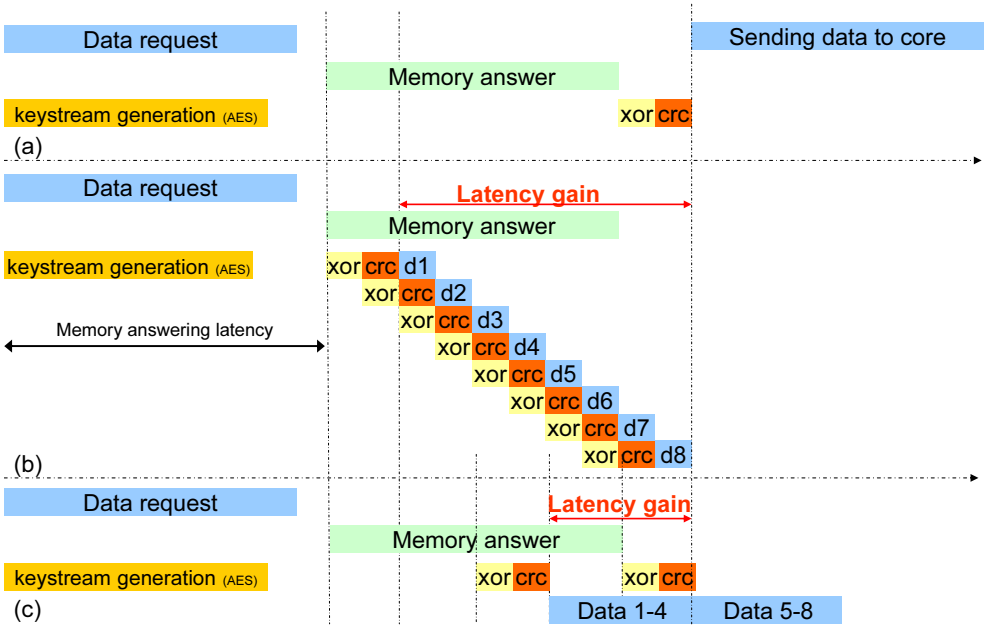

Figure 2.9: Data deciphering and integrity checking with long memory latency:
case a: Non pipeline version (256 bits cache line / 256 bits CRC input)
case b: Pipeline version (256 bits cache line / 32 bits CRC input)
case c: Half pipeline version (256 bits cache line / 128 bits CRC input)

is the half of a cache line. In this case the latency to obtain the checked data is average. Here it clearly appears that a trade-off between the performance, the security memory footprint and the security level. The designer has the choice.

A very efficient solution with a low memory footprint is possible but the security level of this architecture will be low too. On the other hand, a strong architecture with a good security level and good performances are also possible but at the cost of a bigger memory footprint. However in some cases, the increase of the security level can also be associated with a performance increase. For example, if in *case a* Figures 2.8 and 2.9, the CRC size is 32 bits for a full cache line. If the same CRC size is used in *case c*, as a consequence the total CRC size will be 64 bits for the cache line so the security level of the architecture is higher. Moreover, as the deciphering and the CRC checking can be done earlier, the data will be available sooner so the global performance of the architecture will be higher. But in this case the memory footprint of the *case c* architecture will be twice more important than the one from *case a*. All these architectural parameters give the opportunity to the designer to build a secure system which fits with the requirements of his application.

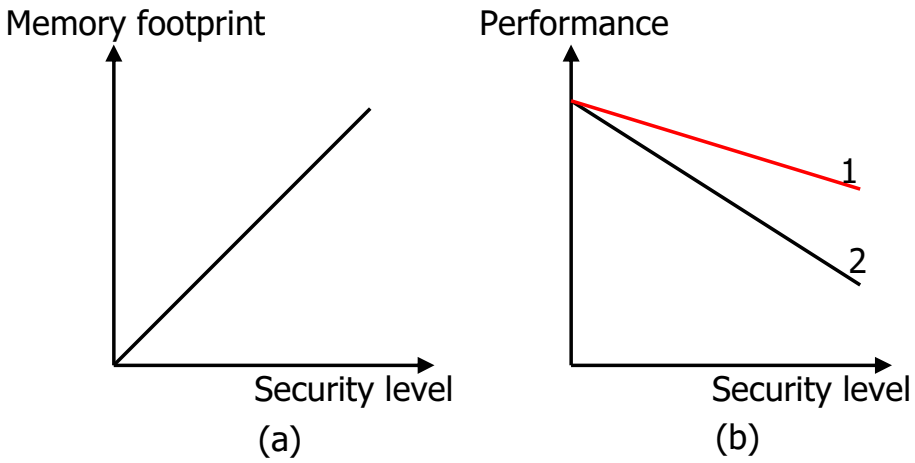

Figure 2.10: Summary of security overhead depending on security level
a: Security memory footprint depending on security level
b: Architecture performance depending on security level for 2 associated memory footprint (memory footprint 1 > memory footprint 2)

Figure 2.10 gives an overview of the different possible trade-offs. Figure 2.10.a shows how the security memory footprint is directly linked with the security level. Indeed, a higher security level means a bigger CRC so more storage for the CRC results. Concerning the architecture performance, Figure 2.10.b indicates that the performance is linked to the security level too but also linked to the memory allocated to the system. If more memory is allocated, it will be possible to produce some pipelining in the data management so the latency to obtain the data will be lower. The proposed solution offers to the designer a few parameters to customize his core depending on his security needs and the architecture features.

In [Drimer, 2008], the author underlines the two system designers concerns which are : cost-conscious and security-conscious. The cost-conscious designer system is looking for a cheap system, with short term protection and generally trusted. The security-conscious designer is looking for a system where security is the priority with long term protection and proof of the security scheme. The work presented here is cost-conscious since embedded systems are mostly cost sensitive. Even if the approach mainly focuses on minimizing the security cost, the proposed architecture also offers the possibility to the designer to choose the right security level.

2.4 Experimental results & cost of security

2.4.1 Experimental architecture

An embedded platform based on an Altera NIOS II soft-processor [Altera, 2008a] has been used to validate this new memory protection approach. The NIOS II configuration includes instruction and data caches, each with 512 total bytes and 256

bits cache lines. Figure 2.11 shows the way, the NIOS processor is interconnected to the Hardware Security Core and how the different hardware IPs are connected together. The data bus size between all the components is 32 bits. Two Avalon buses are needed to build the system. The first one handles the connection between the Avalon master from cache memory and the hardware security core. The second Avalon bus manages the connection between the core and the DDR SRAM IP core. The frequency of the system is around 120 Mhz. The hardware security core does not contain the architecture frequency critical path so it does not bring any loss to the system frequency.

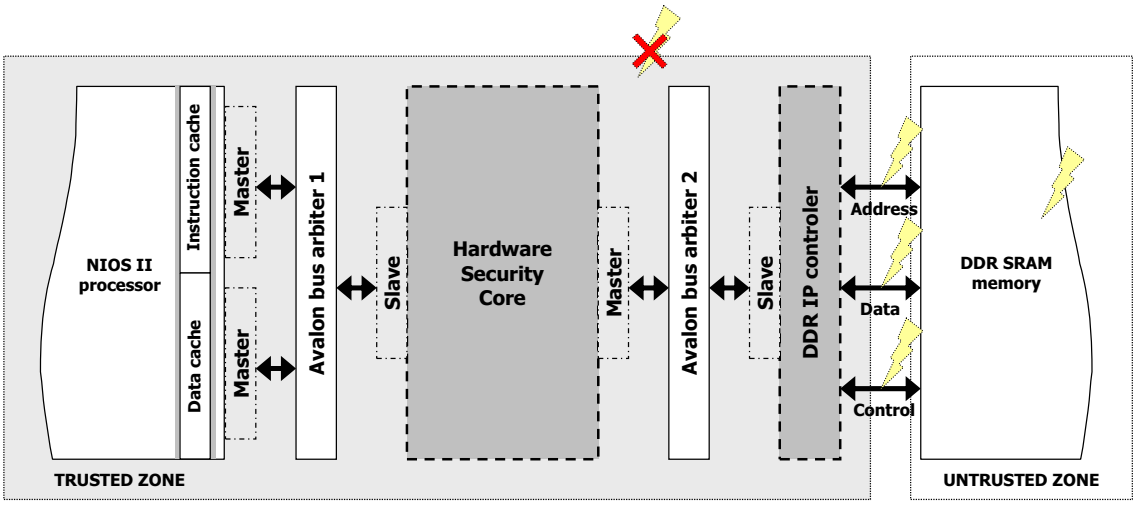

Figure 2.11: NIOS II architecture with security core and memory IP

In the following, three architectures will be evaluated. The first one called *AES-TAC/CRC32* is an AES-TAC encryption associated with a CRC 32 checking on the full cache line. The architecture temporal behavior is typically the one shown in Figures 2.8.a and 2.9.a. The second called *AES-TAC/CRC8* corresponds to an AES-TAC encryption associated with a 8 bits CRC. In this case, each 32 bits word of the cache line will have its own 8 bits CRC result so the decryption and integrity checking could be pipelined (Figures 2.8.b and 2.9.b). The last one called *AES-TAC/CRC32*^{2} is an intermediate solution with AES-TAC and a CRC 32 bits result for each half of cache line (Figures 2.8.c and 2.9.c). As stated previously, these three architectures correspond to three typical cases a designer might face:

- Low performance, low memory footprint, average security level: *AES-TAC/CRC32*.
- High performance, high memory footprint, low security level: *AES-TAC/CRC8*.
- Average performance, high memory footprint, high security level: *AES-TAC/CRC32*^{2}.

Architecture	Total		HSC		AES-TAC unit		CRC unit		Control	
	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs
AES-TAC/CRC32	8352	4887	3443	1336	1580	434	662	472	1201	430
AES-TAC/CRC8	8172	4623	3265	1072	1581	433	398	202	1286	437
AES-TAC/CRC32^2	8266	4651	3357	1098	1579	434	497	264	1281	400

Table 2.1: Detailed breakdown of AES-TAC core resource usage

2.4.2 Area overhead

As it can be seen in Table 2.1, the security has an important impact on the architecture area. A complete NIOS II soft-processor with peripheral requires around 4900 ALUTs and 3550 registers/flip-flops. So the average overhead depending on the solution is around 68% (70% for *AES-TAC/CRC32*, 66.5% for *AES-TAC/CRC8* and 68% for *AES-TAC/CRC32*^{2}).

The main difference in the core size comes from the integrity checking part. Depending on the choice made for security, the amount of logic to manage the CRC checking varies. Indeed, the logic area required to implement a CRC32 with a 256 bits input is more complex than a CRC8 with a 32 bits input. Nevertheless, the logic required with the CRC8 to manage the pipeline scheduling also adds some logic complexity to the system. Concerning the AES core, there is no size change in any of these architectures, the AES-TAC logic size is constant around 1580 ALUTs. The AES-TAC logic includes two 128 bits AES cores. The cache line size is 256 bits that is why two AES cores are mandatory to guarantee good performance. A 256 bits AES could have been used but the latency to obtain the 256 bits would have been much longer (14 cycles instead of 11). The two AES cores are using the same ciphering, TS and padding value. But the addresses used as an input of the AES core are different in order to prevent the generation of 2 identical keystreams for the two 128 bits plaintext.

It is difficult to compare the area overhead with the solutions presented in chapter 1 due to the lack of information. Nevertheless, all the solutions proposed require one or several AES cores. The main logic overhead difference comes from the way integrity checking is added to the architecture. For *PE-ICE*, the extra material for integrity checking is very low since a simple 32 bits comparator is necessary to check the deciphered tag. For *AEGIS* and *XOM*, the use of algorithm such as MD5 or SHA-1 will have a strong impact on the area. In [Ducloyer et al., 2007], authors show that a MD5 core size is around 676 ALUTS at a frequency of 90 Mhz and a SHA-1 core size is around 1034 ALUTs at a frequency of 111 Mhz. In the case of MD5, the main problem would be the frequency. MD5 would imply a 33% loss from a frequency point of view. For SHA-1, the main issue is the size of the core. In the proposition made here, the size of the CRC checker is around 662 ALUTs in the

	Base NIOS	NIOS AES-TAC/CRC32		NIOS AES-TAC/CRC8		NIOS AES-TAC/CRC32^2	
			cost		cost		cost
Read latency (cycles)	0	11 (8+3)	+11	4 (1+3)	+4	7 (4+3)	+7
Write latency (cycles)	0	12 (8+4)	+12	12 (8+4)	+12	12 (8+4)	+12

Table 2.2: Latency due to security compared with non protected NIOS architecture

worst case compared with the 1034 ALUTs needed for the SHA-1 core. It corresponds to a 44% decrease of the logic dedicated to integrity checking with the CRC checking approach. For *AEGIS* two cached hash tree implementations are possible. The first one with only one SHA-1 core, in this case the SHA-1 core will be used to compute each tree node as a consequence the latency to check the data may be really long. The other implementation uses several SHA-1 cores in order to compute all the tree nodes to the root at the same time. The logic area increase for that last implementation would be very high but the time to obtain the checked value would be much smaller than the first implementation. Concerning the logic dedicated to confidentiality, *XOM* needs one 256 bits AES core and *AEGIS* two 128 bits AES cores.

The tight requirements of embedded systems lead the designer to choose the architecture with the smallest area impact. Based on all these statements, it is expected that the proposed solution should be a good alternative from a logic area point of view compared with other solutions. Furthermore, a logic or memory area increase also impacts the power consumption.

2.4.3 Architecture performance

As shown in Figures 2.8 and 2.9, the way the solution is implemented has an impact on the system latency. Furthermore, with the NIOS architecture configuration and the Avalon buses, there is a delay between the time the request is sent over the bus and the time the memory provides the data. In the case of the NIOS architecture, the memory latency is long enough to perform the keystream computation so the latency due to security only relies on the deciphering and integrity checking part (Figure 2.9).

2.4.3.1 AES-TAC latency

Table 2.2 gives an overview of the security cost from a latency point of view compared with a non protected solution (the memory answering latency is excluded from

these values as it is always the same for all the cases). Depending on the way the security is applied, the latency to obtain the data is lower. With *AES-TAC/CRC32*, the system needs a full cache line to perform the CRC checking. As soon as, the eight 32 bits words composing the cache line are fetched (8 cycles), the deciphering and the CRC checking are performed in 3 cycles. So in this case the latency to obtain the checked data is 11 cycles (8+3). A standard NIOS architecture does not need to wait for a full cache line to keep running the application so the latency is considered to be 0.

Table 2.2 also shows that the way the security is implemented minimizes the latency. With the *AES-TAC/CRC8*, as soon as the first 32 bits word is loaded (1 cycle), the security core can check the value (3 cycles). In this case, only 4 cycles for security checking are mandatory (1+3). Another possible architecture presented here is the half pipeline version where only one half of the line (4 cycles) is necessary to start the security checking (3 cycles). So in this case, only 7 cycles are required. These figures clearly prove that depending on the security level wished by the designer, the impact on the system latency can vary especially for the read requests.

For write request, the keystream computation through the AES core is not completely overlapped by the data fetching from the cache. The CRC computation can be done in parallel with the data fetching from the cache but during a write request, the critical path is due to the ciphering part (see Figure 2.3). Before sending the data to the external memory the plaintext must be XOR'ed with the keystream which leads to a total of 12 cycles. 11 cycles are required compute the keystream thanks to the AES core. As soon as the keystream is read the plaintext is encrypted and sent to the memory. So 4 cycles are needed over the 8 cycles required to fetch the data from the cache.

2.4.3.2 Security latency comparison with existing solutions

The security latency is the latency needed to obtain a deciphered and checked instruction or data from the external memory. Table 2.3 and 2.4 give the latency required by security for different existing solutions. In these two tables, the presented architectures are compared with a NIOS architecture protected with an AES-based solution. All the latencies include the time to fetch the data from the memory and the latencies required by the different security schemes. The total latency is then compared with the AES based solution. In all the cases, the *AES-TAC/CRC* solution proposed here minimizes the latency needed to get the data.

For *PE-ICE*, the latency is really close to a NIOS based architecture protected with an AES. The extra cycles required come from the time needed to check the tag value. The AES ciphering can only start when the data are fetched from the cache. The main *PE-ICE* time penalty is due to the non parallelization of the time

to fetch the full cache line from the memory and the AES deciphering. For the write request, the security latency is almost the same as for the read request.

For *XOM*, the main issue is the hash value which is done on the deciphered data. The time penalty is important for deciphering (8 cycles for data fetching and 14 cycles for AES computation) but the penalty due to the MD5/SHA-1 computation is much bigger (64/80 cycles). For write requests, *XOM* has the biggest latency. The hash value is ciphered with the data so 64 to 80 cycles are necessary before the AES computation. With the last *XOM* version [Yang et al., 2003], a few cycles are saved (around 14) since they use the AES-TAC method for ciphering. So the keystream generation is overlapped by the hash value computation.

For *AEGIS*, the evaluation of the latency penalty is more difficult. Indeed, the time to check the data integrity depends on the number of tree nodes which are needed to be computed before reaching a value in the secure zone. The hash algorithm used is SHA-1 which requires 80 cycles. So the latency for *AEGIS* is 80 cycles multiplied by number of nodes to compute. As *AEGIS* is using an AES-TAC technique for data ciphering, the keystream generation time is overlap by the hash tree computation. For write request, contrary to *XOM*, the result hash node is not ciphered with the data, so the hash computation can be done in parallel with the data ciphering. As a consequence, the security latency to cipher data is limited to the time to generate the AES-TAC and perform the xor operation.

The latency added by hash algorithms such as MD5 and SHA-1 may be prohibitive but they guarantee a higher security level than the *AES-TAC/CRC* proposition and *PE-ICE*. However, for embedded systems, this integrity protection may be sufficient, especially when one considers the requirements of MD5 and SHA-1. The minimum input data size required for these algorithms is 512 bits, which may be prohibitive. Many processor-based embedded architecture do not support 512 bits cache lines. In addition, the output of these algorithms is 128 or 160 bits and the memory overhead required to store information is substantial (around 30% more than a non protected solution).

For *TEC-Tree*, the latency to get a deciphered data also depends on the number of nodes to compute before reaching the root. Contrary to *AEGIS*, *TEC-Tree* has to compute the full tree (even if it could be possible to have a cache memory for the tree). Compared with *AEGIS*, *TEC-Tree* should be a more efficient solution. Indeed, *TEC-Tree* is mainly based on *PE-ICE* to go from one node to another so the latency to compute one node is much shorter (11/14 cycles for AES, 80 cycles for SHA-1). The fact that *TEC-Tree* uses *PE-ICE* to fully travel the tree minimized the fact they do not use a cache memory for some nodes. That is why *TEC-Tree* should provide better performances than *AEGIS*. Nevertheless, like *AEGIS*, the *TEC-Tree* performance depends on the number of *PE-ICE* cores. The use of several *PE-ICE* cores will lead to a parallelization of the tree node computation as underline previ-

	NIOS AES	NIOS AES-TAC/CRC32		NIOS AES-TAC/CRC8		NIOS AES-TAC/CRC32^2	
			gain		gain		gain
Read latency (cycles)	22 (8+14)	11 (8+3)	-11	4 (1+3)	-18	7 (4+3)	-15
Write latency (cycles)	22 (8+14)	12 (8+4)	-10	12 (8+4)	-10	12 (8+4)	-10

Table 2.3: Latency due to security compared with an AES based secure architecture

	XOM AES/Hash		PE-ICE AES		AEGIS AES-TAC/hash tree		TEC-Tree PE-ICE/hash tree	
		cost		cost		cost		cost
Read latency (cycles)	86 (8+14+64) 102 (8+14+80)	+64 +58	25 (8+17)	+3	≈(SHA-1) x node nb	>80	≈25 x node nb	>25
Write latency (cycles)	86 (8+14+64) 102 (8+14+80)	+64 +80	26 (8+18)	+4	12 (8+4)	-10	26 (8+18)	+4

Table 2.4: Latency of existing solutions compared with an AES based secure architecture

ously with *AEGIS*.

2.4.3.3 Secure software execution comparison with existing solutions

After having a detailed view of the security cost at a cycle level, Figure 2.12 shows some application performances. The aim of this figure is to evaluate the software performance decrease depending on the security solution. Of course, the performance loss is dependent of the application cache miss numbers and the memory latency. The first analysis provides by Tables 2.2, 2.3 and 2.4 is confirmed by Figure 2.12. The average loss for the most efficient architecture (*AES-TAC/CRC8*) is around 10%. The 100% corresponds to the execution of a non protected solution. The values presented by Figure 2.12 correspond to the software performance compared with a non protected execution. For example, a 97% value corresponds to a 3% decrease of the software execution performance with the protected solution. All the figures have been obtained after profiling the software execution (numbers of cache miss, latency between two cache misses). Based on the applications profiling results and the security scheme latencies, the secured software execution behavior has been inferred and the performances have been computed and compared.

Again the software performance is directly impacted by the security level. As said previously, the *AES-TAC/CRC32*^{2} is considered as more secure than the others

(*AES-TAC/CRC8* and *AES-TAC/CRC32*) but the performance penalty is higher (20% performance loss on average). The *AES-TAC/CRC32* architecture can be considered as the slowest architecture due to the fact that the full cache line must be fetch from the memory before starting the deciphering and integrity checking but the following will prove that this low performance is counterbalance by a memory footprint reduction. Nevertheless, even if *AES-TAC/CRC32* is the slowest architecture based on AES-TAC, the performances of this architecture are better than an architecture with AES only. Indeed, an average of 13% is saved compared with an AES based solution.

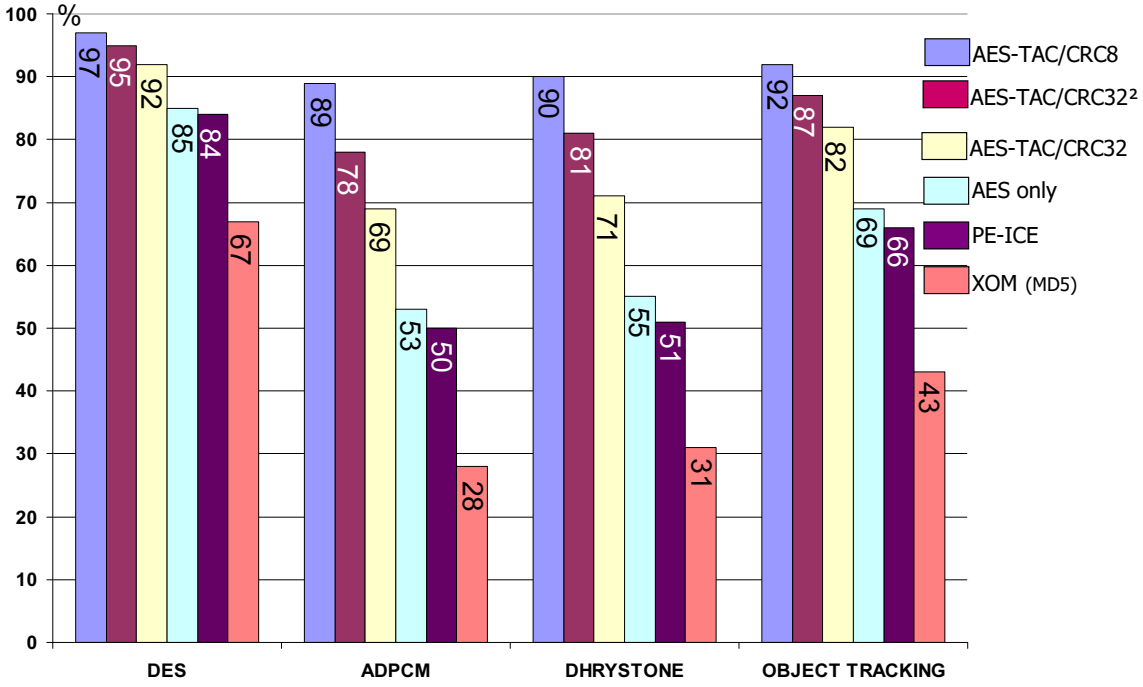

Figure 2.12: Performance overhead based on the number of cache miss for different application executions

For *PE-ICE*, the performances are worst than the AES solution, it comes for the time to perform the tag verification. But the *PE-ICE* architecture guarantees confidentiality and integrity contrary to the AES only architecture which only guarantees confidentiality. *XOM* proposes the worst solution from a performance point of view (more than 40%). The latency due to the integrity checking with MD5 is really high as stated in the previous section (around 64 cycles) and it badly impacts the global software execution. No evaluation has been done for *TEC-Tree* and *AEGIS* because there is no simple way to extract the behavior of the architecture.

The results of Figure 2.12 are for a NIOS processor with 512 bytes caches. In the case of 2 kB caches or more, the global performance would be better. With 2 kB caches, the cache miss number would be lower so less data would be exchanged with

	AES-TAC/CRC32	AES-TAC/CRC32^2	AES-TAC/CRC8
CRC code (kB)	32	64	64
CRC data (kB)	32	64	64
TS data (kB)	32	32	32
Total (kB)	96	156	156

Table 2.5: Detailed on-chip security memory overhead for 256kB of code and 256 kB of data

the external memory and so less latency would be added by the security core. With some applications like **DES** and **object tracking**, the *AES-TAC/CRC8* architecture performance may be close to the performance of a non protected solution. The security latency would be fully hidden from a software execution performance point of view. Nevertheless, the next section will show that security based on AES-TAC will always have an impact on memory footprint.

2.4.4 Security Memory footprint

2.4.4.1 Memory overhead with AES-TAC

Table 2.5 exhibits the memory cost for the 3 different architectures used in all the previous results. The figures of the table are obtained through Equation 1. It shows a good example of the security impact and performance on the memory. As said previously, *AES-TAC/CRC32*^{2} can be considered as more secure because 64 bits are used for integrity checking. As a consequence, the CRC storage is twice more important for *AES-TAC/CRC32*^{2} (128 kB) than *AES-TAC/CRC32* (64 kB). For *AES-TAC/CRC8*, the memory cost does not come from security but from the performance. As a matter of fact, the *AES-TAC/CRC8* architecture is built to minimize the latency (pipeline version) to obtain data at the cost of a lower security level (see Figures 2.8 and 2.9). In any case, the protection against replay attacks cost is the same (32 kB of TS). The security memory cost of data is more important than the code. The data are vulnerable to replay attacks contrary to code which is read-only data so no TS are needed.

These memory costs are dependent on the CRC size (*AES-TAC/CRC32*^{2} and *AES-TAC/CRC8* for example) but it is also linked with the architecture features such as the cache line size. In Equations 1.1 and 1.4 the cache line size is taken in account. If the architecture has a cache line which is two time bigger, the security cost in term of memory overhead will be two time less important. Three parameters are appearing in the equations to obtain the memory security cost.

- The cache line size
- The CRC input size

- The CRC output size

Equation 1 - *Security memory equations*

Size of CRC memory for code:

$$1 - CRC\ overhead = \frac{\frac{cache\ line\ size}{CRC\ input\ size} * CRC\ output\ size}{cache\ line\ size}$$

$$2 - CRC\ code = Total\ code * CRC\ overhead$$

Size of CRC memory for data:

$$3 - CRC\ data = Total\ data * CRC\ overhead$$

Size of TS memory for data:

$$4 - TS\ overhead = \frac{TS\ size}{cache\ line\ size}$$

$$5 - TS\ data = Total\ data * TS\ overhead$$

Example with 256kB of code and 256 kB of data for AES-TAC/CRC32[2]:

$$CRC\ code\ overhead = \frac{\frac{256}{128} * 32}{256} = 0.25$$

$$CRC\ code = 256kB * 0.25 = 64kB$$

$$CRC\ data = 256kB * 0.25 = 64kB$$

$$TS\ data\ overhead = \frac{32}{256} = 0.125$$

$$TS\ data = 256 * 0.125 = 32kB$$

Most of the time, the designer has only access to the CRC parameters except with soft-processor where the cache line size can be parametrized. It means that his only chance to control the security cost is through the two last parameters. If he increases the CRC output size, he will increase his architecture security. If he wishes to increase his architecture performances, we will have to decrease the CRC input size. So the memory needed to have the same security level with more performance will be more important. Here it clearly appears that a trade-off between security, performance and memory size appears. The designer choices will be guided by the application requirements and the hardware cost he is allowed to reach.

2.4.4.2 Memory overhead comparison with existing solution

As it is shown in Table 2.6, the memory overhead is located in different places depending on the solution (on-chip and off-chip). With *AES-TAC/CRC*, all the data dedicated to security are in the trust zone (on-chip memory). The base system chosen for the experiments has a 512 kB memory which leads to a total of 610 kB system for *AES-TAC/CRC32* (32+64+512). The memory cost varies from 18.75% to 30.4% for *AES-TAC/CRC* architectures. These values can be much higher with a system where each 32 bits data would have a 32 bits CRC.

If *PE-ICE* is applied to the same architecture, the memory overhead is around 54.7%. With *PE-ICE*, random vector for data are stored in the on-chip memory, and the data ciphered with the tag are stored in the off-chip memory. Like *AES-TAC/CRC*, the security cost for data is more important because of the vulnerability to replay attacks.

For *TEC-Tree*, the memory overhead is around 76%. The security cost for the code is less important than the cost for data because it is not mandatory to use the hash tree for code. Indeed, the PE-ICE solution is fully protecting the read-only data against all attacks. For the data, *TEC-Tree* helps to save the on-chip memory required with *PE-ICE* at the cost of a bigger impact on off-chip memory. For the same amount of data (256 kB), the cost of data with *TEC-Tree* is twice more important than code. It comes from the fact that all the nodes from the hash tree are stored off-chip.

The *AES-TAC/CRC*, *PE-ICE* and *TEC-Tree* are suitable with the basic NIOS architecture chosen for experiment. *XOM* and *AEGIS* are based on hash algorithms (MD5 and SHA-1) which have a 512-bit input. Here the cache line size is 256 bits it means that the algorithm input would not be completely filled. If these two solutions were used with NIOS architecture, the security cost would be really important (around 50% for *XOM* and 91% for *AEGIS*). The fact that the input is to large enforces the idea that these solutions are not adapted for the architecture chosen here. As mentioned above, the cache line size impacts the memory overhead. With a longer cache line, the memory overhead with *XOM* and *AEGIS* would be much lower. For *TEC-Tree*, the high memory overhead is counterbalance by the higher software performances that the designer might expect compared with *XOM* and *AEGIS*.

All the solutions presented in Table 2.6 do not have the same security level. Nevertheless, the results of this table show that if the designer wants more security for his architecture, the *AES-TAC/CRC* solution provides a good compromise between security level and memory. In any case the security memory footprint of *AES-TAC/CRC* is smaller than other solutions.

2.4.5 AES-TAC with other processors

All the results presented in the previous sections were generated with a NIOS based architecture. The aim of this section is to extend the analysis to other processor architectures and try to guess what could be the results with these processor cores.

Concerning the area overhead, it is difficult to evaluate. The AES-TAC will not need any major changes with another processor. Indeed, the main difference would be the wrappers used to interconnect the AES-TAC core with the processor and the

	AES-TAC/ CRC32		AES-TAC/ CRC32^2		AES-TAC/ CRC8		PE-ICE		TEC-Tree		XOM		AEGIS	
	code	data	code	data	code	data	code	data	code	data	code	data	code	data
on-chip	32	64	64	96	64	96	0	24	0	0	0	0/32	0	32
off-chip	0	0	0	0	0	0	128	128	128	262	128	128	160	276
total	610		666		666		792		902		768/800		980	
overhead	18.75%		30.4%		30.4%		54.7%		76.2%		50%/56%		91%	
security	2^{32}		2^{64}		2^{8}		2^{32}		2^{64}		2^{128}		2^{160}	

Table 2.6: Detailed on-chip security memory overhead in kB for 256 kB of code and 256 kB of data

memory. So the AES-TAC logic amount should be slightly different. For the global architecture size (processor + AES-TAC), the overhead will be fully dependent on ratio between the processor core size and the AES-TAC core size.

If the AES-TAC scheme was applied to another processor architecture such as MicroBlaze [Xilinx, 2008a] for example, the software execution results should lead to the same conclusion. The intrinsic latency of the AES-TAC is the smallest one compared with other solutions. As a consequence the result trend should be the same and the AES-TAC should provide the most interesting results. The main differences would come from the application cache miss rate and also the logic latency of the IP used to interconnect the blocks (cache with the AES-TAC and AES-TAC with memory see Figure 2.11).

For the memory overhead, in section 2.4.4 it was underlined that several parameters are taken in to account to evaluate the footprint. The main difference of NIOS with other processors will be the cache line size. Even if in some particular case, the cache line can be configured, it is the major feature which could impact the memory overhead.

2.5 Alternative for memory footprint improvement

The last section presented the security cost of *AES-TAC/CRC* architecture. It has been shown that this cost is still high even if some hardware and architectural tricks are used to minimize it. Even if the *AES-TAC/CRC* approach gives a low memory footprint, the amount of on-chip memory required is not negligible compared with the actual ASIC or FPGA capabilities. The other existing solutions use both on-chip and off-chip memory to merge towards a trade-off.

Figure 2.13 introduces the architecture of an *AES-TAC/CRC* approach with an

off-chip memory storage of the TS and CRC values. If some TS and CRC are stored off-chip it means that they can be attacked so they must be protected too. The basic idea here is to apply the same cache system as proposed in [Yang et al., 2005] and to extend it for security purpose.

The TS and CRC stored in the off-chip memory are protected with the *AES-TAC/CRC* mechanisms presented in this chapter. A small cache memory is used to store the deciphered and checked TS and CRC from the external memory. It will help to minimize the global architecture performance decrease. In order to decipher and check the TS and CRC values from the external memory, some small on chip storage is needed for the TS and CRC used for protecting the external TS and CRC. They will be called TS^2 and CRC^2. When a data is fetched from the memory, if the TS and CRC are already present in the cache, they are used to decipher and check the data values. If not, the TS and CRC cache is updated securely through the AES-TAC mechanism but with the TS^2 and CRC^2.

The advantage of such a solution is the reduction of the on-chip storage required. Indeed, it will lead to a reduction by 8 of the on-chip memory need (one TS^2 can protect 8 TS and one CRC^2 can protect 8 CRC) but an increase of the needed logic. In systems where the on-chip memory is a critical issue, this architecture might be a good alternative. There will be a small increase of the external memory size because the TS and CRC are stored there (see Figure 2.13). Nevertheless, the logic complexity to implement this solution would be higher compared with the original *AES-TAC/CRC* architecture. The approach presented here requires a cache memory, a cache management and a more complex configurable datapath which will increase the logic size. Additionally, the architecture performance will be fully dependent on the size of the TS and CRC cache memory. If each time a data is fetched from the memory, the system must decipher and check the TS and CRC before using it, it will badly impact the global performance. So to guarantee a good execution performance, the size of the cache must be larger. A large cache means a large amount of on-chip memory to build the cache.

With such an analysis, it is not really clear if the TS and CRC off-chip storage combined with a cache system is a good solution or not. The logic complexity, the performance loss and the small memory gain are strong drawbacks to use such a solution. The main AES-TAC goal is to take advantage of the on-chip memory, as soon as an architecture is built in a way that it removes the on-chip memory, the AES-TAC becomes inappropriate.

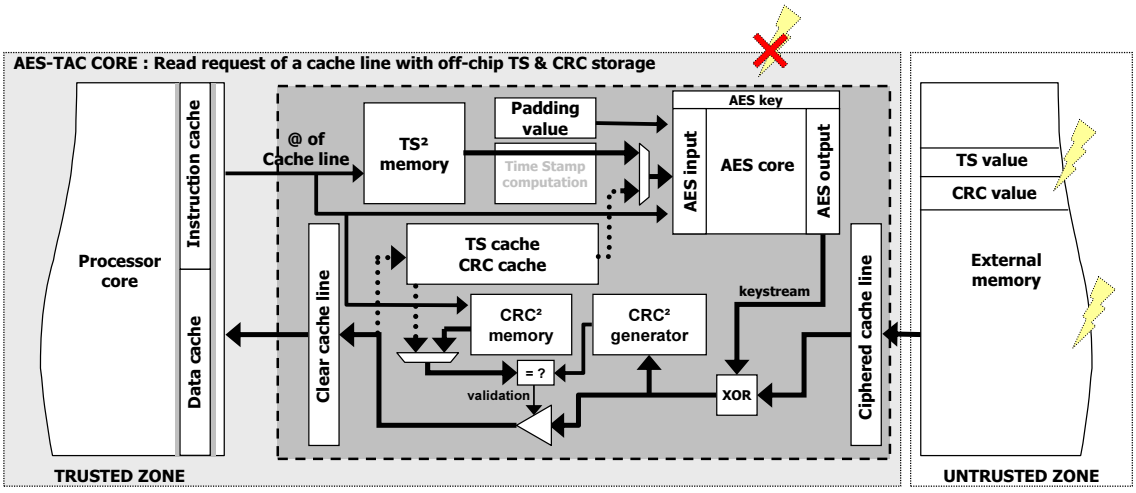

Figure 2.13: AES-TAC architecture with an off-chip TS & CRC storage

Conclusion

Through this chapter the *AES-TAC/CRC* mechanisms have been presented in detail. It has been shown that the security cost impacts several architecture parameters (area, memory and performance). Moreover, due to the possibilities offered by the integrity checking based on CRC, the designer wishes can also influence the architecture overhead.

In average the size of a NIOS architecture is increased by 68% compared with a non protected solution. Concerning the software performance execution loss, it fully relies on the designer choice. According to the set of applications used for analysis, the performance loss is around 25% in the worst case but the loss is limited to 10% in the better case. To obtain a high efficient software execution, the system will need more memory for security purpose. The memory overhead will be affected by two parameters, the performance designer wishes and the security level. More security and more performance mean more memory. In the example presented above the memory overhead goes from 18.75% to 30.4%.

All the features and results for the *AES-TAC/CRC* have been compared with existing solutions. The figures obtained thanks to the experimental results lead to think that the *AES-TAC/CRC* proposition is a good alternative to the existing solutions. The execution performances are higher and the security memory footprint is limited. The following will introduce new opportunities to improve the global architecture efficiency.

CHAPTER 3

Hardware security level management

In the previous chapter, it has been underlined that the embedded system tight requirements lead to build dedicated security solutions. Even if these solutions are designed in an optimized way, the overhead due to security is still high. The main issue is the amount of memory mandatory to guarantee a good security level. In this chapter, a security level management is introduced to choose what memory parts should be protected in order to minimize the security memory cost. By choosing the right security policy, the application designer will have the opportunity to build a system with better performance and to limit security overhead (area, memory, performance and energy).

Contents

3.1 Optimization based on security-oriented application mapping

3.1.1 Principles

The use of an OS in embedded system is becoming ubiquitous. The applications are becoming so complex and the application constraints so strong that the only way to match with all the requirements is to use an OS. The main idea here is to exploit the OS features in order to see if any savings can be done for security. In the last chapter, the memory security cost for the architecture have been detailed and even if it has been reduced thanks to *AES-TAS/CRC*, the amount of on-chip memory for TS and CRC storage is still high. One statement is that maybe the whole memory does not need to be protected. With a multi-tasking application, it is possible to extract some tasks which are not critical for information security or for the secure behavior of the application. Some tasks may also not require the integrity checking, for example a company might just want to protect its software from be stolen but does not care if the system crashes sometime. The goal here is to reduce the data amount to be protected in order to limit the associated on-chip memory storage.

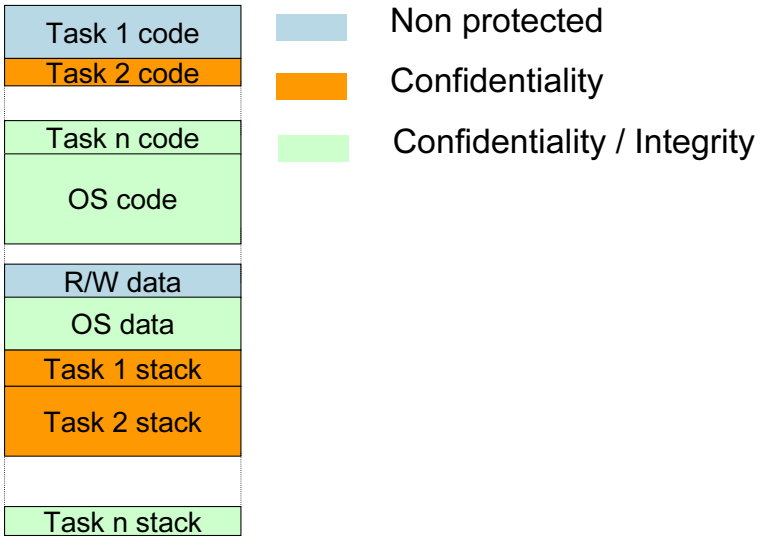

Figure 3.1: Example of security memory mapping

The use of an OS provides a natural partitioning of application code and data. Based on this memory partitioning, developers can specify a desired security partitioning for each task. Figure 3.1 gives an example of a typical partitioning. The application designer can build the security mapping of his application the way he wishes. The application instructions and stack data of Task 1 of Figure 3.1 have different security levels for example. In this case, the application designer may wish to keep the data used by task 1 secret. The application code may be less sensitive,

eliminating a need for security. It is essential to note that we suppose that the OS code and data are protected the right way. As a matter of fact, if the OS is corrupted, it means that all the tasks may be corrupted too.

Our system is designed to be used in conjunction with a memory management unit (MMU). This unit ensures that a task will not read or write memory segments that are not associated with it, creating a security risk if security levels differ. The availability of configurable security levels provides benefits. Compared with a whole memory protection, the on-chip memory necessary to apply the right security mapping will be reduced. Additionally, the latency and dynamic power of unprotected memory accesses is minimized since unneeded security processing is avoided.

3.1.2 Security memory mapping

In order to produce the memory mapping, the compiled code is analysed to extract the memory segment corresponding to each task. Depending on the security strategy of the software designer, the security segments are built. The code example below details the typical look of a code after being disassembled. It is easily possible to extract the information necessary to built the security memory mapping (SMM).

Application assembly code example

```
0x8000020 <alt_exception>:
   8000020: deffed04  addi sp,sp,-76
   8000024: dfc00015  stw ra,0(sp)
   ...
0x80001d0 <task1>:
 80001d0: 800eff80  call 800eff8 <OSFlagPend>
 80001d4: 800ca400  call  <alt_timestamp_start>
 80001d8: 1004403a  cmpge r2,r2,zero
 ...
0x80002e8 <task2>:
 80002e8: defffb04  addi sp,sp,-20
 80002ec: dfc00415  stw ra,16(sp)
 80002f0: df000315  stw fp,12(sp)
 ...
0x8000424 <task3>:
 8000424: 800eff80  call 800eff8 <OSFlagPend>
 8000428: 01020074  movhi r4,2049
 800042c: 2110b704  addi r4,r4,17116
 ...
0x80006ac <task4>:
 80006ac: e0800245  stb r2,9(fp)
 80006b0: e0800243  ldbu r2,9(fp)
 80006b4: 10801de8  cmpgeui r2,r2,119
 ...
```

```
0x80007cc <task5>:
 80007cc: 800ff340  call 800ff34 <OSMboxPend>
 80007d0: e0800415  stw r2,16(fp)
 80007d4: e0000305  stb zero,12(fp)
 ...
0x8016530 <task1_stk>:
 ...
0x8018530 <image_contour>:
 ...
0x801aab0 <task4_stk>:
 ...
0x801cab0 <image_prec>:
 ...
0x801f030 <image_generated>:
 ...
0x80215b0 <task2_stk>:
 ...
0x80235b0 <task3_stk>:
 ...
0x80255b0 <image_binary>:
 ...
0x8027b30 <task5_stk>:
 ...
0x8029b30 <OSFlagTbl>:
   ...
```

Each segment is defined by 4 parameters:

- The segment base address

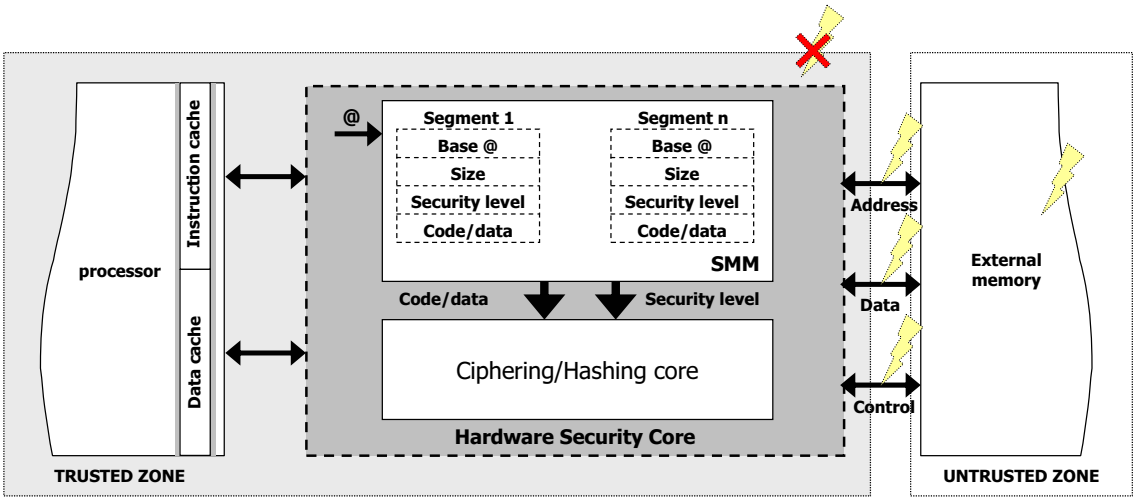

Figure 3.2: Architecture with SMM

- The segment size
- The segment security level
- The kind of segment: code (real-only) or data (read-write)

All this information will be located in the *Security Memory Mapping* (*SMM*) block. It is important to note that contrary to other solutions like *XOM* and *AEGIS* no specific instructions are used when the system starts to execute secure tasks. Everything is managed in hardware by the *Hardware Security Core* (*HSC*), which includes a ciphering and hashing core (*CHC*) and the *SMM* (see Figure 3.2). The *HSC* is totally independent from the software. No instructions are needed to configure it, it is reducing the potential threat on the system in the case of software attacks. Furthermore, as the *HSC* is directly working with the memory segment address, no specific compiler or compiling steps are necessary. The use of a SMM leads to a solution which is very simple to integrate in any architecture even with a high security level due to the independence between the hardware configuration of the *SMM* and the software (OS).

As shown of Figure 3.2, the SMM is a very simple unit. As soon as a request is made by the processor, the SMM looks at the address request to see what policy should be applied to the data. Based on all the information hard-coded in the SMM, the security level to apply to the data is selected. Only two signals are needed to control the *CHC* in order to configure it the right way to apply the right security level to the data. The two information needed are the security level and the kind of date (code or data).

In the following, some results will be presented in an architecture where the *SMM* is combined with the *AES-TAC/CRC* architecture but the idea to apply a

specific security level to each memory segment can be extended to any other solutions dedicated to the external memory security.

3.2 Architecture evolutions

The architecture based on *AES-TAC/CRC* has been extended to support the security level management. Figure 3.3 and 3.4 exhibit the architecture datapath configuration for read and write requests. Three security levels are selected for this architecture:

- No protection: the data are not protected
- Confidentiality only: the data are ciphered with the AES-TAC mechanism
- Confidentiality and integrity: the data are ciphered with the AES-TAC mechanism and the integrity checking is guaranteed by the CRC

The integrity-only security level is not considered as a security level. It is supported by the architecture but the integrity checking with a CRC is too weak to be considered as secure (see Chapter 5 for details). In the current scheme the plaintext is not known by the attacker so he has no clue to attack the system. In an integrity only scheme, the attacker will have access to the plaintext and he can easily find another values which leads to the same CRC. The CRC is a very simple function. For this reason, it is not really suitable for the integrity only scheme which is targeted here. For security solutions like *XOM* and *AEGIS* integrity only can be added to the security level of the architecture. The algorithms such as MD5 and SHA-1 used for integrity checking in these solutions are strong enough to be used alone compred with the CRC checking introduced in the previous chapter.

Of course the *SMM* is included in this architecture (Figures 3.3 and 3.4). It will select the security level that the *CHC* has to apply to the data. As soon as the *CHC* receives a new security level to apply, the datapath is configured to handle the chosen security level.

One major change in the HSC introduced in the previous chapter is due to the *SMM*, the OS use and the AES input management. As shown in Figure 3.3 and 3.4, the padding value has been replaced by a new block called Segment ID. This new scheme will be called AES-TASC (AES in time address segment cipher). The need for unique time stamps creates a problem if the time stamp generation counter rolls over and starts reusing previously-issued timestamps. As stated previously, the system must not use twice the same keystream value for ciphering. Usually, when this case happens, the system must change the AES key et reencrypt all the memory. This operation will stall the system during the whole reencryption process. For critical real time system (with hard time constraints), it might be an issue

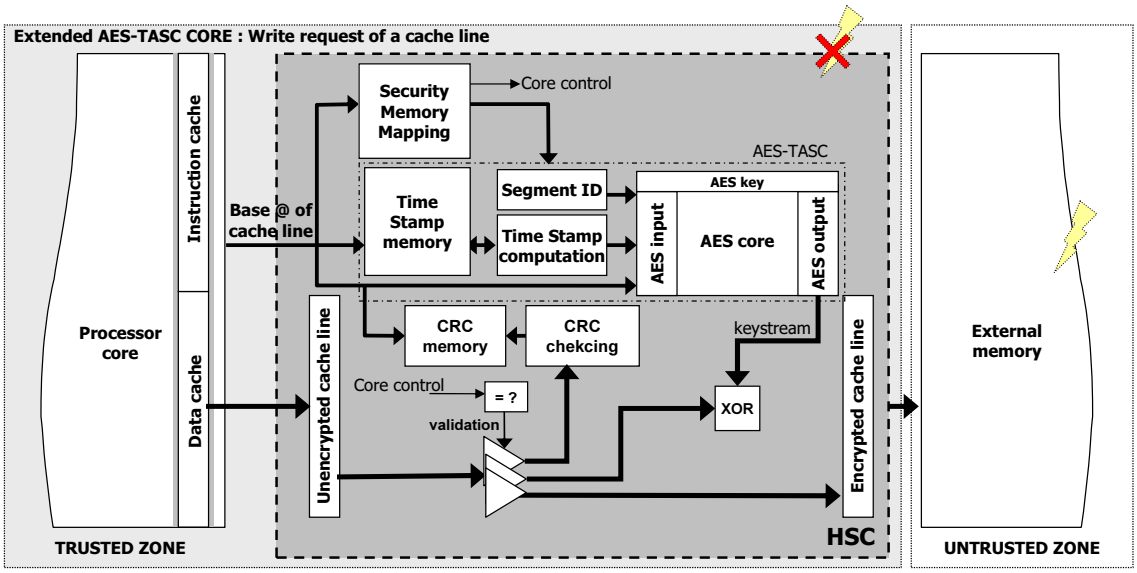

Figure 3.3: AES-TASC architecture with SMM for write request

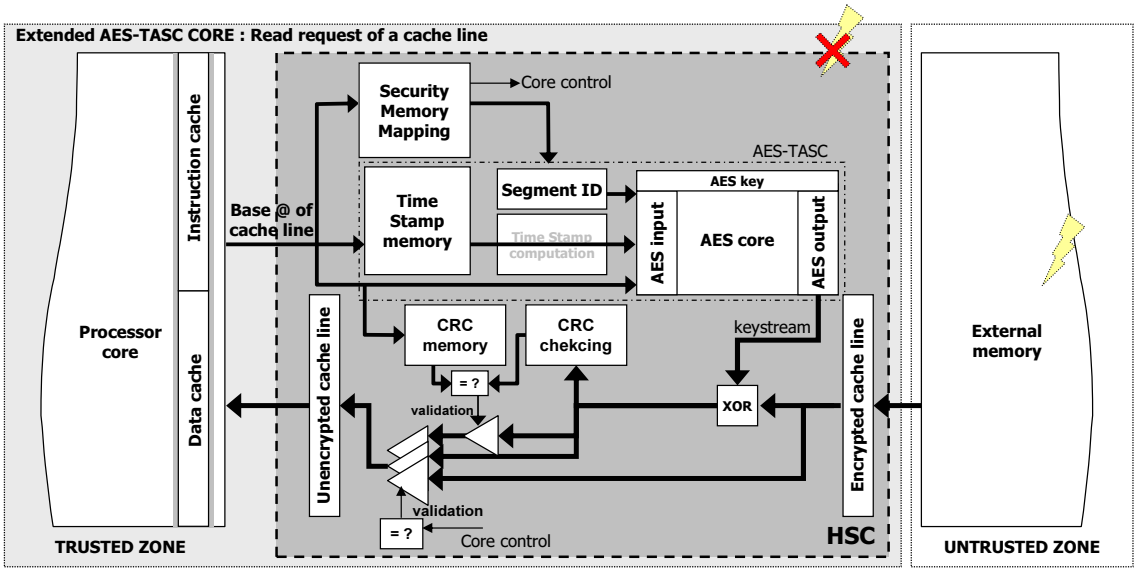

Figure 3.4: AES-TASC architecture with SMM for read request

because the process may not reach the application time constraint. A typical solution to this issue involves the reencryption of stored data with new time stamps [Yan et al., 2006]. Recently, a keystream solution which uses multiple time stamp generator counters [Yan et al., 2006] was proposed. If a time stamp counter reaches its maximum value, only about half the data must be reencrypted. With the new security approach proposed here, the same idea of multiple time stamp can be applied. Each memory segment is associated with a counter. This counter is called Segment ID. If a TS rollover is needed, the Segment ID will be incremented for reencyrption. The use of the Segment ID in keystream generation helps avoiding the issue of matching time stamp values in this case. If reencryption due to counter rollover is needed, only a portion of the external memory is affected. This feature can help in reducing the embedded system down time (stall time the system when the process is freezed). Moreover, the probability that the Segment ID counter rolls over is very low compared with the system lifetime. For example, with a 32-bit TS and a 32-bit address, 64 bits are still needed to fill the 128-bit AES input. It means that the Segment ID counter will be 64 bits which gives a total counter of 96 bits (TS 32 bits + Segment ID 64 bits). A total of 2^{96} accesses to one specific address is possible. Another advantage of this solution is that if the Segment ID never rolls over, no new AES key will be needed. So it also helps to save some hardware to manage the generation of a new AES key. Indeed in the previous solution using the same kind of approach [Suh et al., 2003], if a TS rolls over, the whole memory is decrypted then reencrypted with a new AES key. This scheme requires some extra hardware to generate an AES key on the fly.

3.3 Experiments

3.3.1 Experimental approach

3.3.1.1 Architecture presentation

In order to validate the benefits of the approach, an architecture based on an Altera NIOS II soft processor [Altera, 2008a] was developed. The security core and associated memory were implemented in FPGA logic and interfaced to the processor via an Avalon bus. In separate sets of experiments, the NIOS II was first allocated instruction and data caches of 2 KB bytes and then 512 bytes. The core is interconnected the same way as in Figure 2.11 chapter 2. The current NIOS II implementation does not use a MMU. So the architecture explored in the following of the work will no protect the system against tasks, which write or read data in another memory space. The aim of the experiments in this section is to evaluate the benefit of a security memory mapping.

Concerning the *CHC*, the next results focus on an architecture with *AES-TASC/CRC32*[2] which was introduced before (chapter 2 section 2.4.1). The main features of *AES-*

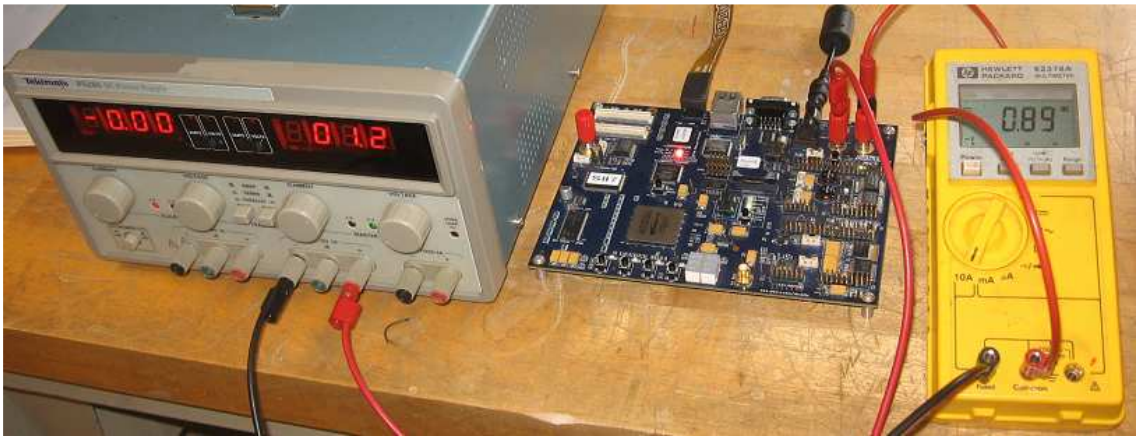

Figure 3.5: NIOS II development kit board with Stratix II 2S60 FPGA

TASC/CRC32[2] are: good security level (64 bits CRC results), average memory overhead (30.4%) and average performance loss (less than 20%). With the *AES-TASC/CRC* architecture, there will be three security levels. As stated in Section 3.2, the availability of a security core which allows different security levels with different memory segments leads to a tradeoffs between security and resources. In our analysis we consider three specific scenarios:

- No protection (NP) - This is the base NIOS II configuration with no memory protection.
- Programmable protection (PP) - This NIOS II and security core configuration provides exactly the security required by each application memory segment, as discussed in Section 3.1.1.
- Uniform protection (UP) - This NIOS II and security core configuration provides the highest level of security required by a memory segment to all memory segments. Since all segments use the same security level, the SMM is smaller.

The widely-used MicroC/OS-II [LaBrosse, 2008] embedded real time operating system was used to validate our approach. MicroC/OS-II is a scalable, preemptive and multitasking kernel. The OS can be configured by the designer during application design to use only the OS features that are needed. A priority-based scheduling approach is used to evaluate which one of up to 64 tasks runs at a specific point in time. MicroC/OS-II uses a hardware timer to produce ticks which force the scheduler to run. MicroC/OS-II also provides common primitives such as semaphore, mailbox, event and so on.

All applications were successfully implemented on a Stratix II Development Kit board (Figure 3.5) containing a 2S60 FPGA device. Area and embedded memory blocks were determined following compilation with Altera Quartus II. Power measurements were determined using the experimental setup shown in Figure 3.5. Core

Application	Tasks	Mem Segs.	Total mem (kB) Code	Data
Image	5	12	80	59
VOD	7	10	152	431
Comm	6	4	71	68
Hash	5	2	92	55

Table 3.1: Application memory overview

power for the FPGA was provided by a Tektronix power supply. A multimeter configured as an ammeter was used to measure the current. Other board peripherals (flash memory, DDR SDRAM memory) were powered with a standard power supply input.

3.3.1.2 Application description

To evaluate the impact of the security management on performance, area, and power consumption, 4 multi-task applications were used. These applications include:

- **Image processing** - This application selects one of several values for a pixel and combines the pixels into shapes. Pixel groups that are too small are removed from the image. This process is called morphological image processing [Dougherty and Lotufo, 2003].
- **Video on demand (VOD)** - This application includes a sequence of operations needed to receive transmitted encrypted video signals. Specific operations include Reed Solomon (RS) decoding [Reed and Solomon, 1960], AES decryption, and MPEG-2 decoding with artifact correction [Kristensen et al., 2006].
- **Communications** - This application includes a series of tasks needed to send and receive digital data. Specific operations include Reed Solomon decoding, AES encryption, and Reed Solomon encoding.
- **Hash** - This application can perform selective hashing based on a number of common algorithms. Supported hash algorithms include MD5, SHA-1 and SHA-2.

3.3.1.3 Applications security policies

Each of these applications can benefit from a selective memory security policy. For the **Image processing** application, image data and application code used to filter data are protected, but data and code used to transfer information from the system

App.	Confidentiality and Integrity						Confidentiality						No protection					
	Code			Data			Code			Data			Code			Data		
	kB	T	S	kB	T	S	kB	T	S	kB	T	S	kB	T	S	kB	T	S
Image	25	2	5	33	2	3	7	2	1	10	2	1	38	1	1	16	1	1
VOD	26	5	3	113	6	4	58	1	1	0	0	0	68	1	1	318	1	1
Comm	71	6	1	28	0	2	0	0	0	40	6	1	0	0	0	0	0	0
Hash	0	0	0	0	0	0	92	5	1	0	0	0	0	0	0	55	5	1

Table 3.2: Application memory protection details by protection level (T: number of Tasks, S: number of Segments)

are not protected. The application processing core is protected but not the communication task.

For the **VOD** application, deciphered image data and AES specific information (e.g. the encryption key) are considered critical so the highest security level is applied to them (confidentiality and integrity). Also, the MPEG algorithm is considered proprietary and its source code is ciphered, while MPEG data and RS code and data are left unprotected. In this case, the software designer wants to protect secret (AES key) and its intellectual property (customized MPEG with artifact decoder [Kristensen et al., 2006]).

For the **Communications** application, all data are considered sensitive and worthy of protection (confidential). In order to guarantee correct operation, the code must not be changed so confidentiality and integrity checking is applied to all the code. Application data is only protected for confidentiality.

Hash application code is only ciphered (confidentiality) and application data has no protection. For example, a company may decide to protect its code from visual inspection. Since there is no need for integrity checking in this application, no storage for time stamps or CRC values is needed. The TS input to the AES core, shown in Algorithms 1 and 2 chapter 2, are set to zeroes for this case. In all applications except **Hash**, the operating system code and data use both confidentiality and integrity checking. For the **Hash** application, only confidentiality is used for the OS instructions.

Tables 3.1 and 3.2 summarize the task, external memory count, and number of memory segments for the applications. Note that tasks only represent application tasks, not operating system tasks, but memory segments include application and OS external memory segments. Since the security memory mapping is extracted after the compilation step, sometime tasks are concatenated together into the same segment. For example, if task 1 and task 2 are protected with the same security level

	Uniform protection				Programmable protection			
Architecture	NIOS II + HSC		HSC		NIOS II + HSC		HSC	
	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs
Image proc	8147	4662	3325	1113	8342	4714	3505	1159
VOD	8301	4674	3441	1126	8335	4703	3489	1153
Comm	8150	4670	3316	1116	8289	4677	3450	1135
Hash	7326	4405	2553	854	7086	4397	2295	848

Table 3.3: Architectural parameters for different security levels

and if there are following each other in the memory after compilation it will lead to only one memory segment. One good example in Table 3.2 is the 5 tasks protected through 3 segments for the code confidentiality and integrity of application **VOD**. It will lead to only 3 memory segments for the *SMM*. This helps to save some logic area in the hardware. This point will be stressed later in this chapter. On the other hand, it is also possible to have more memory segments than tasks number if the compiler splits the code. In example from Table 3.2, for **Image processing** code with confidentiality and integrity, there are 2 tasks and 5 segments are needed.

3.3.2 Area overhead of security

For comparison purposes, a NIOS II based system without a security core was compiled to a Stratix II 2S60 FPGA. This basic configuration includes data and instruction caches, a timer, flash memory controller, DDR SDRAM memory controller and a JTAG interface. The Quartus II compiler has reported that the basic cinfoguration with 512-byte caches 4909 ALUTs and operates at a clock frequency of 121 MHz. A basic configuration with 2 KB caches required 4 additional ALUTs and operated at the same clock frequency.

As shown in Table 3.3 for configurations with 512-byte caches, in most cases the *HSC* logic required for programmable protection is greater than for uniform protection due to the inclusion of the *SMM*. Note that for the **Hash** application, integrity checking is neither performed for either uniform nor for programmable protection. The uniform protection area overhead is around 65% except for the **Hash** application were the overhead is limitied to 50% because for this last architecture the integrity checking material is not needed. For the **VOD**, the area security cost raises to 70 % since the amount of on-chip memory to manage is much more important compared with the other applications. In average, the area cost of the programmable approach is around 5% higher compared with the uniform protection.

A detailed breakdown of the individual units size in the *HSC* is provided in

Table 3.4 for both uniform and programmable protection. The security applied to the application impacts the amount of logic. The average size of logic required for AES-TASC management is around 1550 ALUTs and 430 FFs for both uniform and programmable protection. The only difference concerns the **Hash** application were the AES-TASC logic is reduced to 1282 ALUTs for the programmable protection. In this particular case, there are no data protected for confidentiality (see Table 3.2). It means for the *HSC* that no on-chip storage is necessary for the TS and for their management. These 2 points help to save some logic in the AES-TASC core.

Furthermore, depending on the security policy, some security tools are not necessary. Typically if the architecture does not provide integrity checking guarantee, the CRC logic and storage can be removed. Again with application **Hash**, the CRC logic was taken away. The average size of the CRC checking block is 495 ALUTs and 250 FFs. For the **VOD** application, this size is increased to 549 ALUTs. The main reason is that the CRC amount to manage is more important than with other application and some extra logic is required (memory address bus, memory bank management).

For the **Image** and **Communication** applications, the area overhead between uniform and programmable protection comes from the *SMM*. For the uniform protection architecture the logic needed to implement the *SMM* is really small because only 2 memory segments are necessary (one code and one data segment with confidentiality and integrity checking). For the programmable approach, the *SMM* logic size depends on the numbers of memory segments. In Table 3.1, it appears that 12 memory segments are needed for the **Image** application and this large number of segment impacts the *SMM* size. It goes from 15 ALUTs and 3 FFs for the uniform protection to 144 ALUTs and 31 FFs for the programmable protection.

Through the figures of Table 3.3 and 3.4, the security management impact is clear. The security policy chosen by the software designer to protect his application has a direct link with the area overhead due to security. Additionally, the way he will partition the security application will also determine the hardware design size (more secure memory segments imply more logic needed for the *SMM*).

3.3.3 Performance cost of security

The security management will help to improve the software execution performance. The run time of each NIOS II-based system for each application was determined using counters embedded within the FPGA. Table 3.5 shows the run time of each application and configuration and an assessment of performance loss in comparison with the base configuration. Experiments were performed for the 3 architectures *NP*, *UP* and *PP*. With the 512-Byte architecture, the average performance loss is around 20.5% for the *UP* approach and 13.75% for the *PP*. The 7% saved by the *PP* are due to the security mapping. The fact that the whole memory is not protected

	Uniform protection									
Application	Total		AES-TASC		CRC		SMM		Control	
	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs
Image	3325	1113	1501	433	495	247	15	3	1314	430
VOD	3441	1126	1561	440	549	249	19	3	1312	434
Comm	3316	1116	1503	436	492	247	13	3	1308	430
Hash	2553	854	1518	438	0	0	14	3	1021	413
	Programmable protection									
Application	Total		AES-TASC		CRC		SMM		Control	
	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs
Image	3505	1159	1584	434	496	264	144	31	1281	430
VOD	3489	1153	1586	432	497	264	108	27	1298	430
Comm	3450	1135	1608	439	497	253	41	10	1304	430
Hash	2295	848	1282	434	0	0	8	1	1005	413

Table 3.4: Detailed breakdown of hardware security core (HSC) resource usage

	No protection	Uniform protection		Programmable protection	
	(ms)	(ms)		(ms)	
Image 512	93.3	121.4	-23%	108.7	-14%
Image 2k	65.5	80.4	-18%	73.5	-11%
VOD 512	8186.5	10307.7	-21%	9405.6	-13%
VOD 2k	5369.0	6387.0	-14%	6076.0	-12%
Comm 512	42.8	53.1	-20%	49.0	-13%
Comm 2k	26.4	29.5	-10%	28.8	-8%
Hash 512	5.3	6.4	-18%	6.2	-15%
Hash 2k	3.9	4.5	-15%	4.3	-14%

Table 3.5: Application execution time and performance reduction

means that for some data read request, the *HSC* mechanism is totally or partially skipped and the read operation latency to obtain the data is reduced accordingly. For the 2 kB architecture, the benefit of the *PP* approach is around 6 %. The performance loss due to security is higher for configurations which include smaller caches. This could be expected, since smaller caches are likely to have a larger number of memory accesses, increasing the average fetch latency. With a 2 kB cache, the cache miss number is less important so less read requests and less secure read requests will be necessary so the security impact on the software execution is smaller.

Note that for the 2 kB cache versions of the **communication** application, the performance loss for the programmable protection version is smaller by only 2% in comparison with the uniform protection version. The performance loss falls to 1% for the **Hash** application. The security policy applied for these 2 applications is much less aggressive than the one applied to **Image** and **VOD**. So the software designer who chooses the security memory mapping and the security policy, will have enough information to build a system which matches with the time requirements of his application.

3.3.4 Memory cost of security

As underlined in section 3.1, the main issue with the *AES-TASC/CRC* proposition is the amount of the on-chip memory required to protect the memory. Thanks to the security management, the designer has the possibility to limit this amount of memory. Table 3.6 exhibits the detailed cost of on-chip memory for the PP solution compared with the UP one. The major point is that in all cases some amount of on-chip memory is saved. Even better, in some particular cases no on-chip memory is necessary at all to protect the system.

As in all the cases presented in the previous sections, the benefits obtained through the security memory mapping rely on the way the software designer has secured his application. For the **Image** and **VOD** applications, the gains are 52.6% and 75.5%. Of course, these kinds of gain are only possible if some parts of the memory are not protected or protected only for confidentiality (see Table 3.2). For the **Communication** application, the memory gain is limited to 23.1% but the global security level of this architecture can be considered higher than for the 3 other applications. As a matter of fact, all the memory is protected. For the **Hash** application, with the PP architecture no on-chip memory is required. With the *UP* approach, the code and the data were protected for confidentiality only. For the *PP* solution, the code is the only memory partition where confidentiality is guaranteed. As a consequence, no CRC values are stored. Moreover since the code can not be attacked with replay attacks, no TS and no on-chip memory are needed.

	Image		VOD		Comm		Multi-hash	
	UP	PP	UP	PP	UP	PP	UP	PP
CRC code (kB)	20	6.3	38	6.5	17.8	17.8	0	0
CRC data (kB)	14.8	8.3	107.8	28.3	17	7	0	0
TS data (kB)	7.4	5.4	53.8	14.1	8.5	8.5	6.8	0
Total (kB)	42.2	20	199.6	48.8	43.2	33.2	6.8	0
Gain		52.6%		75.5%		23.1%		100%

Table 3.6: Detailed on-chip security memory overhead in kB for different applications

3.3.5 Energy overhead of security

Figures 3.6 and 3.7 provide an energy comparison of the four applications, as physically measured in the lab. Energy, instead of power, is used here to take into account the different applications run times and different security policies. Cache size has a significant impact on the results since larger cache sizes require fewer memory transactions. Our security core is designed to avoid dynamic power consumption via clock gating when the core is not active. The energy saved by programmable protection versus uniform protection depends on the application. Since **VOD** uses a substantial amount of unprotected memory it exhibits only a 28% energy increase versus the base configuration versus a 57% energy increase for uniform protection with a 512 byte cache. The **communications** application contains no unprotected memory and therefore shows only a few percentage points energy savings between *PP* and *UP* implementations. With larger cache memories, the overhead is less important. The **Hash** application does not show large savings since the amount of on-chip memory is relatively small. Most of the energy savings come from a reduced number of *HSC* accesses and the reduced amount of on-chip memory.

Results shown on Figures 3.6 and 3.7 can be analysed based on the comments made in [Senn et al., 2008]. The energy values are obtained and evaluated at a core level. If the analysis was done at a chip or board level the results could be different. Based on the information given in [Senn et al., 2008], it can be considered that the power/energy consumption overhead is almost negligible at chip level and even more negligible at board level. Indeed, if the power consumed by the input/output (*IO*) is included in the analysis, the power/energy consumed by the core is only around 20% of the global power/energy consumed by the chip and even lower at a board level. This last comment leads to think that solution based on on-chip storage for security tag will have the lowest energy overhead due to security.

Moreover, the *AES-TASC/CRC* approach does not store any security data off-chip so no extra off-chip communications with the memory are added by security mechanisms. The request to the external memory solicits the chip IOs and the external memory which consumes a lot of power. Solutions like *PE-ICE*, *TEC-Tree*

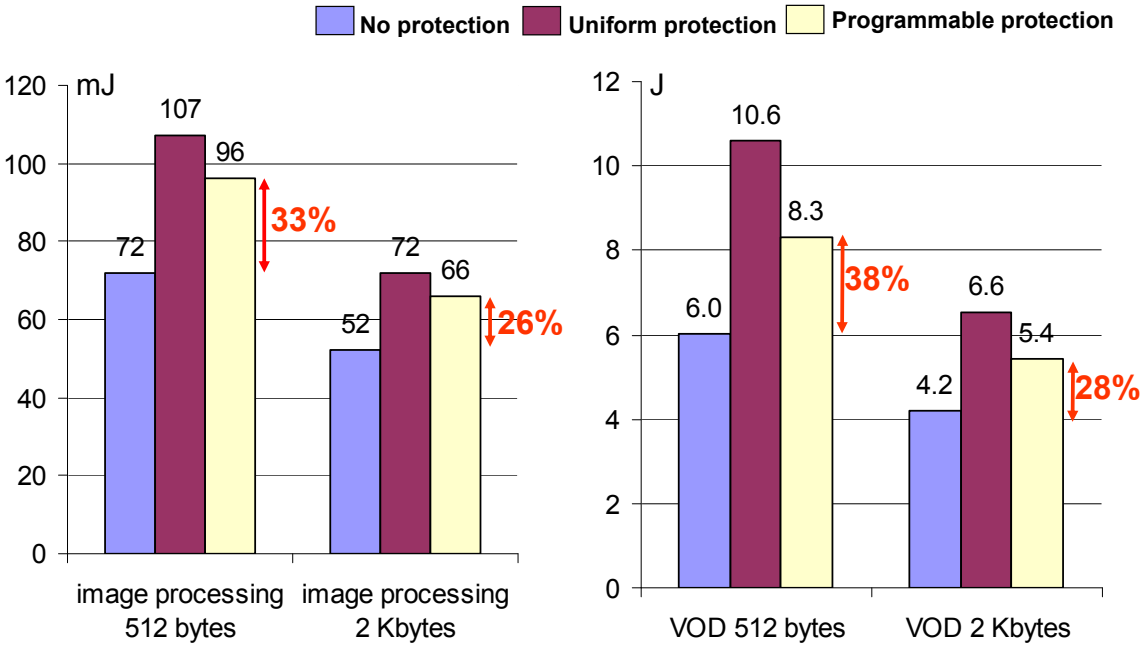

Figure 3.6: Energy consumption for **Image processing** and **VOD**

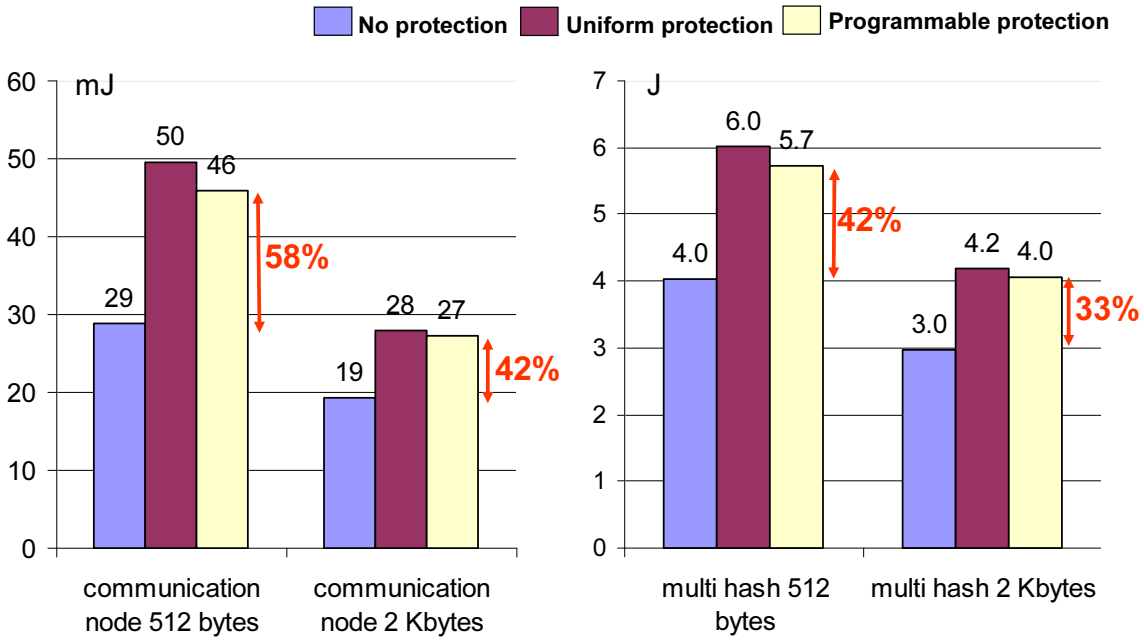

Figure 3.7: Energy consumption for **Communication** and **Multi hash**

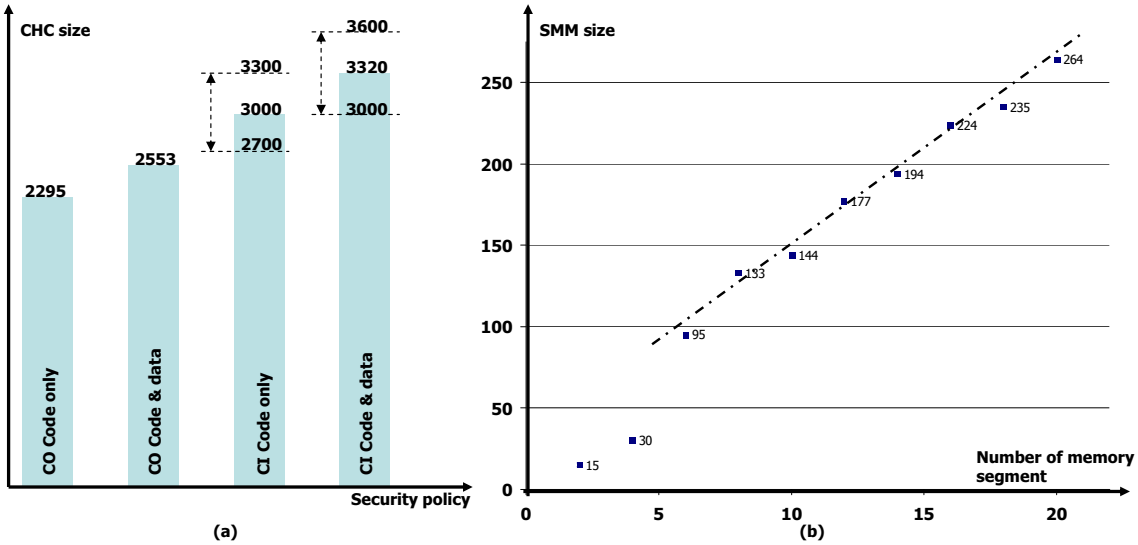

Figure 3.8: Security logic cost
a: Base AES-TASC core size
b: SMM size depending on the number of memory segments

or *AEGIS* which store security data off-chip should consume more power compared with the *AES-TASC/CRC* architecture. If the targeted embedded system is a battery powered one, the energy may become a critical issue. The use of a solution based on on-chip memory may be a good opportunity to extend the battery lifetime compared with solutions where security data are stored off-chip.

3.4 Security architecture resources exploration

The aim of this section is to provide to the software designer the keys and parameters to perform a resource exploration. Security mainly impacts 3 architecture features which have been detailed all along the previous section: area, memory and software performance.

Figure 3.8 illustrates the logic area evolution depending on the security policy. The logic amount is influenced by the policy but also by the memory segments number. Figure 3.8.a introduces the CHC size. There are 4 different CHC:

- CO code only: the only secure zones are code with confidentiality only
- CO code & data: the code and data are only protected with confidentiality only
- CI code only: the only secure zones are code with confidentiality and integrity
- CI code & data: the code and data are only protected with confidentiality and integrity

For the CHC including integrity checking, the amount of logic required can vary from -/+10%. It is due to the way the CRC checking is added to the AES-TASC encryption (pipeline, half pipeline). For the logic added by the *SMM*, Figure 3.8.b shows that the logic cost is linear with the numbers of memory segments. The equation is given by the curve : *SMM = 11.6 * Number of memory segments + 32.* Based on the equation and the value from Figure 3.8, the software designer can have an idea of the security hardware cost depending on his security wishes.

The second feature that might be evaluated by the application designer is the security memory cost. Equation 2 is an evolution of the Equation 1 chapter 2. The difference is that with the security level management approach, only some specific parts of the code and data are protected. The designer is able to have an estimation (and even the right values) of these protected segment size. So he can easily evaluate the cost of his security policy. Of course, the choice made for the CRC checking will also impact the security memory size (see section 2.4.4 in chapter 2 for details).

For performance evaluation, the exploration is more difficult and requires the execution profiling of the non protected application. In section 2.4.3, a detailed view of the security cost at a cycle level has been introduced. If the designer is able to obtain the cache miss number and the data address which caused the cache miss, he should be able to establish a map of the cache miss memory location. Based on this application profiling, he should be able to estimate the performance impact of his security policy based on the Equation 3. He has to add the right latency due to the security policy applied to the data. It is not possible to evaluate the performance penalty based on the time required by the task without protection. The security penalty is fully linked to the cache miss number and the data security level.

Equation 2 - *Security memory equation*

Size of CRC memory for code:

$1 - CRC\ overhead = \frac{\frac{cache\ line\ size}{CRC\ input\ size} * CRC\ output\ size}{cache\ line\ size}$

$2 - CRC\ code = code\ size\ with\ CI * CRC\ overhead$

Size of CRC memory for data:

$3 - CRC\ data = data\ size\ with\ CI * CRC\ overhead$

Size of TS memory for data:

$4 - TS\ overhead = \frac{TS\ size}{cache\ line\ size}$

$5 - TS\ data = data\ size\ with\ CO\ or\ CI * TS\ overhead$

Equations 3.1, 3.3, 3.5 give the read latency due to security for different security

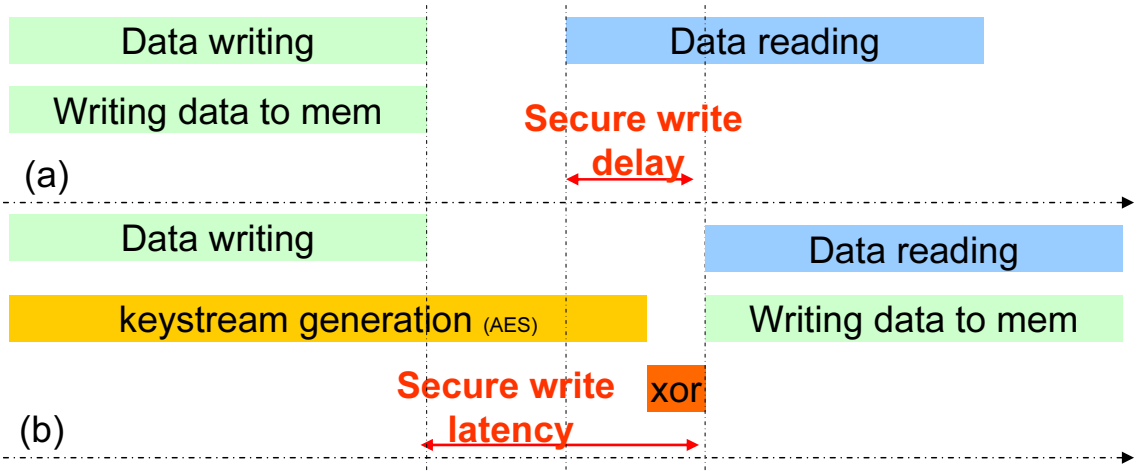

Figure 3.9: Read after write sequence
a: Non protected architecture
b: Protected architecture

level: Confidentiality and integrity (CI), Confidentiality Only (CO) and Integrity Only (IO). As underlined in the first chapter part, the idea of data filtering and security level management can be applied to any other solutions providing for ciphering and hashing data. All the equations below are proposed to help the designer to evaluate the performance cost of the security policy he would like. The Integrity Only security level has been introduced because some security approach can offer this level contrary to the AES-TASC approach. Some explanations are needed for the CI, CO, IO write penalty. Usually security adds some latency to the data fetching.

At first sight this point is considered as the main issue. But the latency due to a secure write can also impact the global performance of the system. Figure 3.9 exhibits an example where the secure data write adds some latency to a read after a write sequence. For example, if the secure write latency is 10 cycles. With a non protected architecture if a read occurs after a write to early (less than 10 cycles), this read will be delayed with the secure architecture (see Figure 3.9.b). If the designer wants to have a really accurate evaluation of his system performances, he will need a detailed profiling of his application.

Equation 3 - ***Performance cost equations:***

Performance penalty due to confidentiality and integrity

$1 - CI\ read\ penalty = cache\ miss\ number\ in\ CI\ segment * CI\ read\ latency$

$2 - CI\ write\ penalty = cache\ miss\ number\ in\ CI\ segment\ after\ write\ request * CI\ write\ latency$

Performance penalty due to confidentiality only:

$3 - CO\ read\ penalty = cache\ miss\ number\ in\ CO\ segment * CO\ read\ latency$

$4 - CO\ write\ penalty = cache\ miss\ number\ in\ CO\ segment\ after\ write\ request * CO\ write\ latency$

Performance penalty due to integrity only:

$5 - IO\ read\ penalty = cache\ miss\ number\ in\ IO\ segment * IO\ read\ latency$

$6 - IO\ write\ penalty = cache\ miss\ number\ in\ IO\ segment\ after\ write\ request * IO\ write\ latency$

Global system execution:

$7 - Total\ read\ penalty = CI\ read\ penalty + CO\ read\ penalty + IO\ read\ penalty$

$8 - Total\ write\ penalty = CI\ write\ penalty + CO\ write\ penalty + IO\ write\ penalty$

$9 - Total\ read\ penalty = Total\ read\ penalty + Total\ write\ penalty$

Additionally, this performance evaluation is necessary if he wishes to obtain the energy overhead. Energy is fully linked with the application execution time. If the designer adds more security segments, it will slow down the application and the time required to run the task will be longer. As a consequence the energy consumed by the architecture will be higher. The more he will protect the data, the more he will use the *HSC* and the more the system will also consume power. Nevertheless, it is difficult to efficiently estimate the power consumed by the application based on the previous section figures.

Conclusion

The security memory mapping brings some new opportunities for the designer. He now has the possibility to build a solution which matches with his wishes. The security mapping idea also reduces the global cost of security for the architecture. As said previously, embedded systems are cost sensitive and the security cost may at first sight be prohibitive but with this new approach security seems more accessible for embedded systems. In addition, the idea of a security mapping is not limited to the *AES-TASC/CRC* architecture. It could be extended to any other solutions and allow these solutions to be more interesting for embedded systems. Figure 3.10

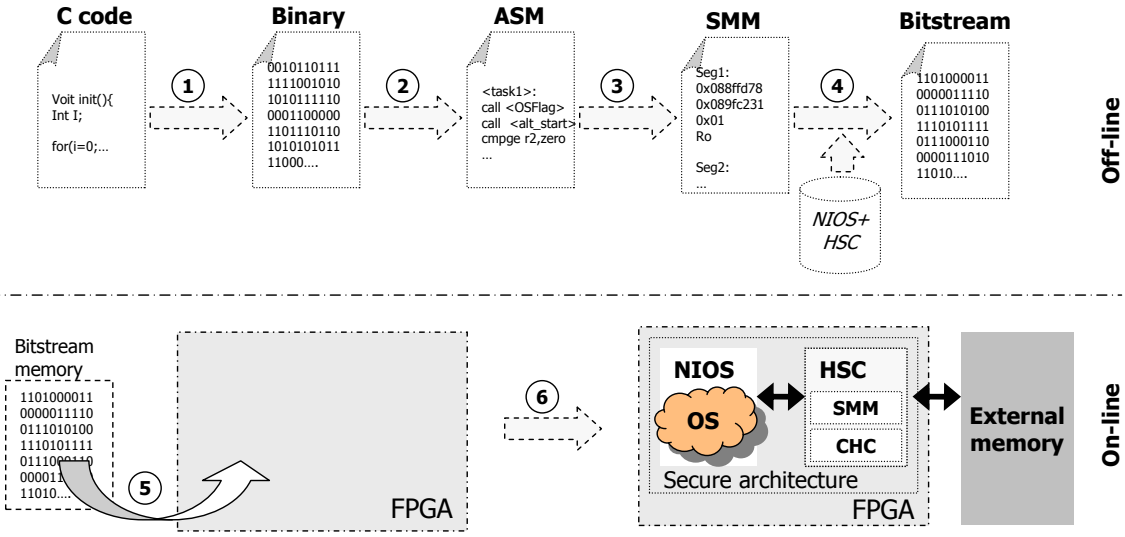

Figure 3.10: Proposed flow to build the secure architecture

Off-line

1 - Standard C code compilation step.
2 - Binary code disassembling to get the assembly code.
3 - SMM parameters extraction.
4 - Architecture synthesis including the SMM parameters to obtain the right security policy for the application.

On-line

5 - FPGA is configured with the generated architecture
6 - The secure generated architecture securely executes the application

exhibits an overview of the global flow from the software compilation to the architecture synthesis to the last step which is the secure software execution.

The essential point that the application designer must have in mind is that the security policy he chooses will directly impact the architecture. Through this chapter several important points have been underlined as critical: area, memory, performances and energy. Most of them are linked together.

In all the previous results and analysis, the application were considered to have been defined at the conception step of the development. Now with the complexity of software applications and architectures it is often happening that the system needs patch or update. The next chapter will introduce some mechanisms to offer a secure system were it will be possible to handle these new embedded systems features (application patches and updates).

CHAPTER 4

Toward an end to end solution and secure application update

In all the previous chapters, the work was focused on security schemes for software execution. The link between the OS and the secure hardware for data protection has led to a more complete secure solution. In this chapter, the goal is to go further in the security scheme. Indeed, before running code on the target, the system must boot up. Moreover, software update and remote software update are common features in actual embedded systems. The system boot up and application update are the main features that need to be added to a secure architecture in order to provide an end to end solution to the user.

Contents

4.1 What's an end to end solution?

In many cases, a secure architecture mainly focuses on the secure program execution. Here the goal is to introduce an end to end solution from the architecture boot up to the secure software execution and the secure application update. Figure 4.1 exhibits a typical architecture for an embedded system. At power up the code is in a flash memory. The code is then loaded from the flash memory to the main memory often a RAM memory. Usually the code execution is not done from the flash memory due to the intrinsic high latency to obtain data from such kind of memory. RAM memories such as DDR RAM memory are low latency memories and will guarantee high software performances for the architecture. The flash content copy is usually done in software as soon as the processor registers are configured. The time needed by the boot copier operation depends on the code size to copy in the main memory and the flash data fetching latency.

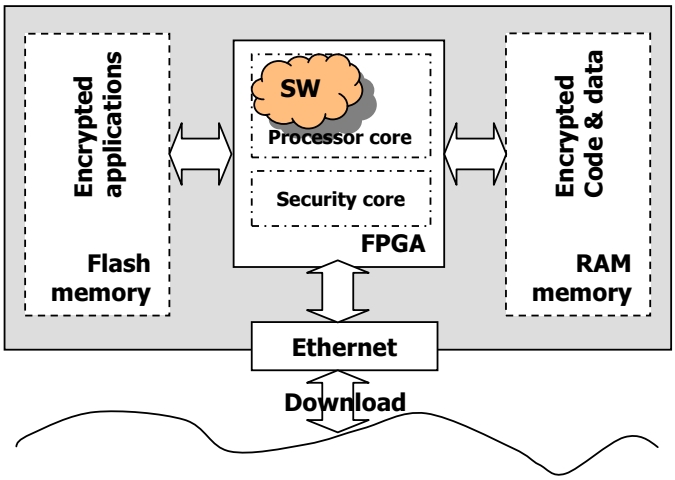

Figure 4.1: Application download and boot loader update

In this secure scheme, the data must also be protected in the flash. The flash memory content must be ciphered to prevent an attacker to access the code. The code integrity must also be guaranteed to ensure the correct application execution. A company wants to ensure that the executed code is the one that has been developed and not another one. Figure 4.1 also introduces the possibility for the architecture to exchange some data with a remote entity. The idea is to securely download a new firmware or applications for the system. Again the download, the storage and the loading of this new software must be secured.

These secure boot up and secure application update must be like all the work done until now: cost-conscious. The goal is to have hardware solutions with a low logic overhead. The system boot up is an infrequent task in the system lifetime as a consequence the feature added for the boot will not be used. In this chapter, the

logic cost of the approach will be the main concern.

4.2 AES ciphering with authentication

In this section specific AES modes with authentication checking are presented. This AES modes will be used to securely load the code from the flash to the RAM memory. The chosen solution must be a very low overhead one. Indeed, this hardware mechanism will be used only for code loading. This operation is not often used by the system (boot up and application loading). If the AES mode implies some extra hardware material just for boot up or the application loading, the solution will not be interesting. It means for example, that if the area is increased just for the mode, the whole architecture will pay the power price of this mode during execution. The chosen AES mode might be the one which has the lowest logic overhead for the architecture.

4.2.1 Known AES modes

Several AES modes are available to guarantee data authenticated encryption. From a security point of view these modes offer the same level of protection. The first one is AES-OCB [Rogaway et al., 2003]. This AES mode deciphers and checks the data authenticity in only one pass, this means that the AES deciphering is only applied one time on the whole message. The main issue with this scheme is that it has been patented. Two other AES mode CCM (Counter with Cipher Block Chaining Message Authentication Code) [NIST, 2004] and GCM (Galois/Counter Mode) [NIST, 2007] have been standardized by the NIST [NIST, 2004] [NIST, 2007]. The problem with the AES-CCM mode is that 2 passes are necessary to provide both confidentiality and authenticity compared with AES-GCM which requires only one pass. This leads to a choice between AES-OCB and GCM. AES-GCM is interesting because standardized and not patented contrary to AES-OCB. The main difference between these two algorithms is the hardware necessary to implement the solution. AES-GCM is a little bit more logic consuming due to the Galois field multiplier. From a performance standpoint AES-OCB and AES-GCM are almost equal. In both schemes, the Initialization Vector (IV) and the tag result (used for authenticity) are stored in clear in the memory. If one change is made in the IV, the tag or the data to check, the result will not match and the system will detect the problem. AES-GCM is the one which offers the most interesting features for our need because it is not patented and matched with the performance and area requirements.

Figure 4.2 and 4.3 detail the scheme for AES-GCM. The main feature which characterizes this AES mode is the way authentication is guaranteed. The scheme relies on the multiplication based on the Galois field. For the ciphering part, the GCM approach mainly relies on the counter mode introduced by the NIST [NIST, 2001]. A counter is used to generate a keystream which is then XORed with the plain-

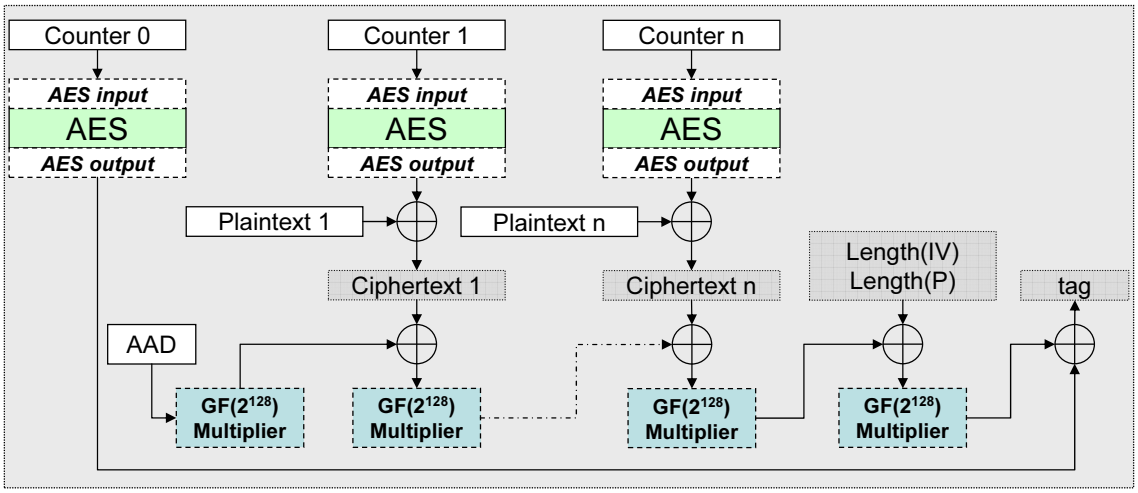

Figure 4.2: AES-GCM encryption and tag computation

text to obtain the ciphertext. The fact that a counter is used for ciphering leads to pattern masking in the ciphertext. This pattern masking helps to remove any plaintext information leakage (code repetition for example). The tag computation is done on the ciphertext and not on the plaintext (Figure 4.2). In order to obtain the tag, a multiplication in the Galois field is used ($GF2^{128}$). All the results are chained together. This way if one bit is changed in one ciphertext, the error will be propagated in all the next deciphered values. The Initialization Vector does not require any protection. With AES-GCM, the initialization vector is also called Additional Authenticated Data. The deciphering scheme is almost the same (Figure 4.3). The ciphertext and plaintext are just swapped in the scheme. At the end of the deciphering, the tag stored with the ciphertext is compared with the tag newly computed during the deciphering step. If the tags match, it means that the message has not been modified and both confidentiality and authenticity are guaranteed.

4.2.2 AES Invert Cipher Block Chaining (AES-ICBC)

The main drawback with the AES-GCM mode concerns the extra material needed to perform the GF multiplication. Another approach would be to use the AES CBC but in an invert way. The standardized ciphering scheme is applied for deciphering and vice-versa. Figure 4.4 shows the ciphering and deciphering scheme of this AES mode which will be called AES-ICBC. The ciphering step is done off-line and the deciphering in done on-line at power up. The main advantage of such a mode is the way the deciphering mechanism is built helps to guarantee errors detection. Indeed, if one error bit is injected in the ciphertext, the error will widely spread in all the chain and modify several output bits. After the last round, the last deciphered value is compared with the value stored in the memory to check if any change has occurred. In this approach, the Initialization Vector (IV) and a Random Value (RV) are stored in clear in the flash memory. The random value is here to help keeping

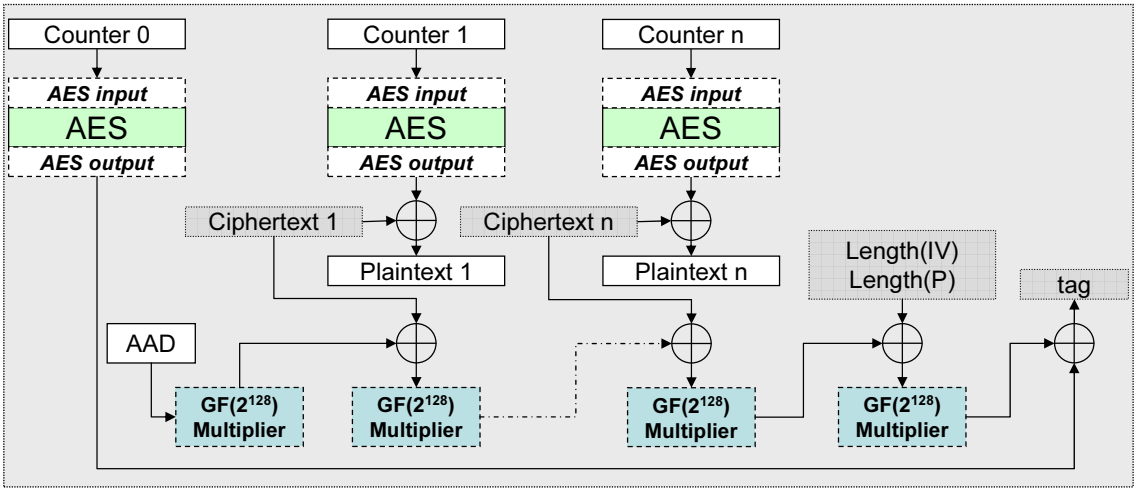

Figure 4.3: AES-GCM decryption and tag computation

the confidentiality of the whole data because the designer does not want to keep a real value of his program in clear in the memory.

This scheme is mainly interesting because of the low logic overhead needed to implement it. Indeed, only a XOR operator is required. Nevertheless, there is a drawback to this low logic overhead. Due to the way the ciphering mechanism is done, some patterns could appear after ciphering. From a security point of view it could be seen as a weakness. In order to tackle this issue, a counter value or the data address is added to the code to cipher. In this case, the plaintext would be concatenated with a value (counter or address) to hide any pattern after the ciphering step. The value size concatenated with the plaintext may vary (32 bits or even less). This kind of tricks will have an impact on the final ciphertext size which will be larger than the plaintext. Another drawback is the extra AES en/decryption which are needed compared with the AES-GCM due to the extra data added to the original code. As said previously, the boot up scheme will not be widely used by the architecture so it must be as small as possible. The memory footprint overhead and the longer latency to perform the deciphering are not major issues is this case. Of course the goal is to minimize as much as possible the solution cost but as mentioned earlier this mechanism and hardware will not be used often. The AES-ICBC mode has been added to the *AES-TASC/CRC* to evaluate the hardware cost. The AES-ICBC requires around 187 ALUTs and 317 FFs more, which is a very limited overhead for this solution (less than 2% compared with the 8300 ALUTs of a complete architecture). This very limited hardware overhead was the main motivation for the exploration of this solution.

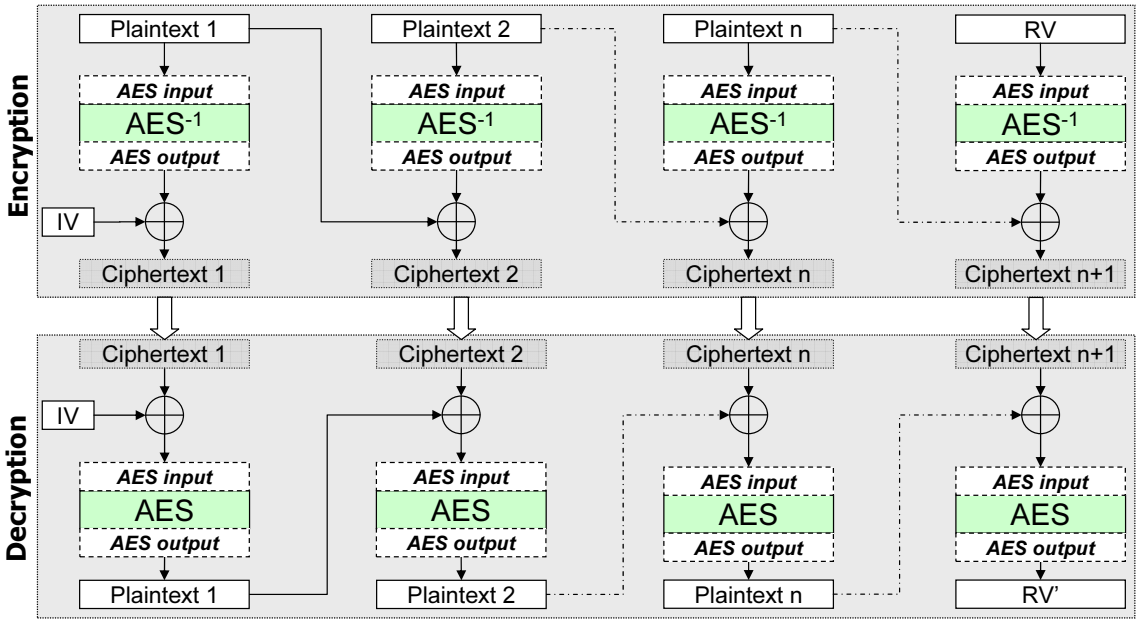

Figure 4.4: AES-ICBC mode

4.2.3 AES modes comparison

Table 4.1 offers an overview of the AES modes introduced before. As underlined previously, the hardware cost of the proposed approach is expected to be lower compared with the AES-GCM. On the other hand, this low hardware overhead is counter balanced by the cost in memory and time.

	AES-GCM		AES-ICBC	
Hardware	GF multiplier 2 xor		1 xor	
Counter size	128 bits		16 bits	32 bits
Memory	$AAD + N + tag$ $[N+2] * 256\ bits$		$IV + (N + RV) * 9/8 + RV$ $[(N+1) * 9/8 + 2] * 256\ bits$	$IV + (N + RV) * 5/4 + RV$ $[(N+1) * 5/4 + 2] * 256\ bits$
Time	$N + AAD + len$ $[N+2] * AES$		$(N + RV) * 9/8$ $[(N+1) * 9/8] * AES$	$(N + RV) * 5/4$ $[(N+1) * 5/4] * AES$
Flash size	image	80 kB	90 kB	100 kB
	VOD	152 kB	171 kB	190 kB
Time at 85 MHz	image	150.5 μs	169.3 μs	188.1 μs
	VOD	285.7 μs	321.7 μs	357.5 μs

Table 4.1: AES modes comparison for N*128 bits message

For the AES-ICBC mode, the cost in time and memory can vary. Indeed, the counter used for the encryption to hide the potential patterns after ciphering is not

necessarily on 32 bits. With a 16 bits counter, it allows the architecture to protect up to 1 MB of data ($2^{16} * 128\ bits = 1$ MB). With a reduced counter concatenated with the data, it helps to reduce the memory size. Due to the memory overhead produced by the counter, the total data amount to decipher is bigger and so the time to decipher these data compared with the AES-GCM is longer. A 32 bits counter leads to around 25% overhead in time and memory. With the 16 bits counter this overhead is limited to 12,5 %. These 12.5% and 25% memory footprint overhead concern the off-chip memory. The flash memory size is not a real problem with embedded systems (several MB available).

Table 4.1 provides the equations to evaluate the memory and time cost of the AES-GCM and AES-ICBC modes with different parameters. For the memory, the equations in gray are normalized with 256 bits. N is the number of 256 bits words composing the code. All the other parameters (IV, ADD, tag) are also 256 bits values. For AES-ICBC, the IV and the RV (ciphered and clear value) are stored in the flash memory. The encrypted value with the AES-ICBC costs more memory due to the counter concatenated with the data to hide patterns that is why a factor is apply to N and RV. For the AES-GCM only the code (N) and the initialization vector are ciphered and stored in the flash memory. For the time, the results in gray are normalized with the time necessary to perform an AES decryption. In this case, only the parameters which are included in the ciphertext are taken in account. For AES-ICBC, it is the code and the RV which are ciphered, with AES-GCM it is the code, AAD and the len value.

The main advantage with the AES-ICBC mode is to provide a very low area solution which guarantees confidentiality and integrity of the deciphered data. With a maximum of 25% memory and time overhead, this approach is suitable for system boot up. The time to copy the code from the flash to the RAM memory will be longer but this operation is only done at the system boot up. The amount of flash memory will also be higher. A small amount of hardware is added to implement the AES-ICBC especially if the architecture already includes an AES core. It means that the static power consumed by this extra hardware will be negligible during the software execution. Indeed, the AES-ICBC is used only for the system boot up, during the execution step it will only consume a very small amount of power thanks to the small area overhead.

Table 4.1 gives an example of code size for the different AES modes and for 2 applications used in the previous chapter (image processing and video on demand applications). The GCM mode does not require any extra memory contrary to the ICBC approach. Table 4.1 also exhibits the time needed to perform the AES computation at a frequency of 85 MHz. These values only include the computation time, further in this chapter some figures will be presented including the time to access the data from the flash which adds some latency to the system.

4.3 Secure Hardware Data Loader

4.3.1 Principles

The aim of the Secure Hardware Data Loader (SHDL) is to provide a hardware mechanism which is able to securely load some data from the flash to the RAM memory. In addition to the data fetching from the flash, it must also store the data in the RAM memory with the right security policy. In the following, the AES scheme used for the data deciphering from the flash can be the AES-GCM or the AES-ICBC. Results will be proposed for both approaches. On top of the two AES modes, there are two possibilities for the Secure Hardware Data Loader. Figure 4.5 and 4.6 describe the two different processes. The major differences for the two schemes are the data amount to load and the numbers of passes needed to perform the load. Scheme 1 needs 2-pass but no extra memory, contrary to scheme 2 which needs extra memory but only one pass.

4.3.1.1 The two SHDL schemes

With the 1-pass scheme, the application code is already stored in the flash with the AES-TASC ciphering scheme. A simple copy of the application code in the RAM memory is enough contrary to the 2-pass scheme where the whole application is ciphered with AES-ICBC or AES-GCM. So the application must first be deciphered then reciphered with the AES-TASC scheme to generate the tag or RV needed later when the data will be read during the application execution. That is the reason why two passes are needed. With the 1-pass scheme, only the Integrity Checking value (CRC) are protected with AES-ICBC or AES-GCM. So it should limit the number of deciphering. As soon as the IC tags are deciphered they are loaded in the on-chip memory of the HSC for IC tags for later used.

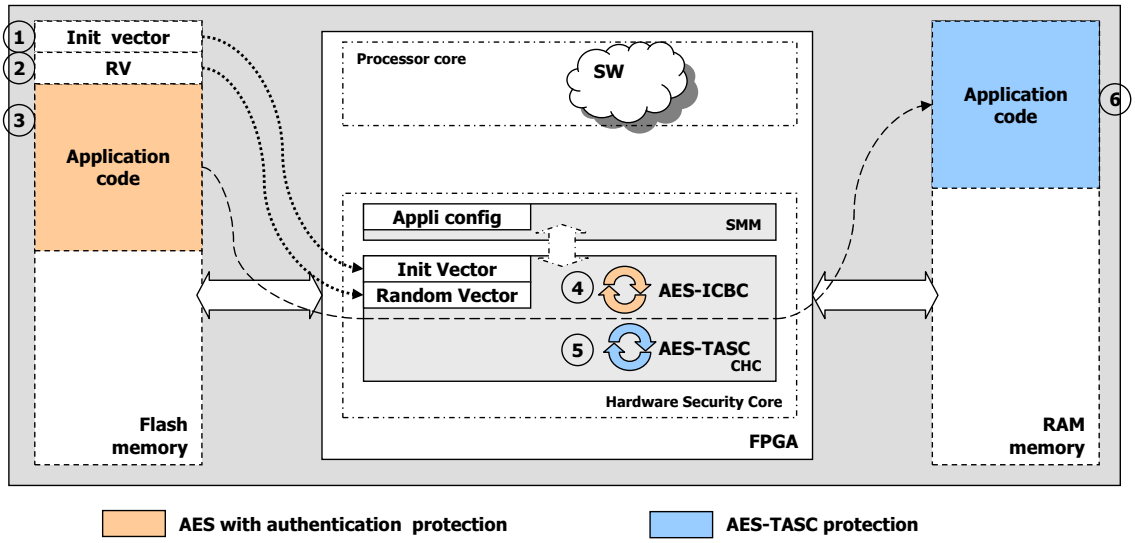

Figure 4.5: Secure hardware code loader with 2-pass scheme

Figure 4.5 details the 2-pass scheme:
1 - The IV is copied in the HSC.
2 - The RV is copied in the HSC.
Loop: for all application code:
3 - The encrypted data is copied in the HSC.
4 - The data is decrypted with the AES-ICBC scheme.
5 - The decrypted data is encrypted with the AES-TASC mechanism.
6 - The encrypted data is copied in the RAM memory.
End loop

The operations 3,4,5 and 6 are repeated until the whole application is loaded into the RAM memory. Due to the use of the AES-ICBC the application code integrity is checked at the end of the copy.

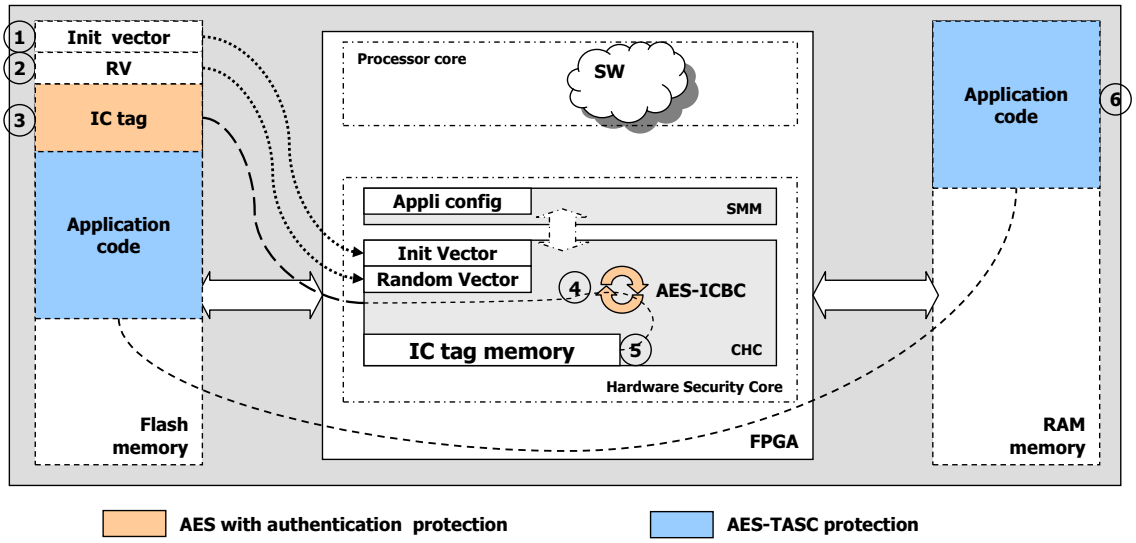

Figure 4.6: Secure hardware code loader with 1-pass scheme

Figure 4.6 details the 1-pass scheme:
1 - The IV is copied in the HSC.
2 - The RV is copied in the HSC.
Loop: for all application code:
3 - The encrypted data is copied in the HSC.
4 - The IC tag is decrypted with the AES-ICBC scheme.
5 - The decrypted data is stored in the Integrity checking (IC) tag memory.
End loop
6 - The rest of the application code is copied in the RAM memory.

The operations 3,4 and 5 are repeated until the IC tags are stored in the memory. If this scheme is applied to the AES-TASC/CRC architecture, the IC tags correspond to the CRC values.

4.3.1.2 Security and memory footprint comparison

From a security standpoint, the two solutions are almost equivalent. The main drawback with the 1-pass approach is that the system will detect an error at the execution. Contrary to the first scheme where the application is checked at the end of the data copy from the flash. With the 1-pass scheme, the processor will load some data from the RAM memory and at this point the HSC might detect an error in application code that may have been corrupted earlier in the flash. With the 2-pass scheme before starting the execution of the loaded code, the system will know if the application has been corrupted or not in the flash. It will prevent a system stall during the execution that will happen if the 1-pass scheme is chosen. Another drawback of the 1-pass scheme is the extra external memory needed to store the IC

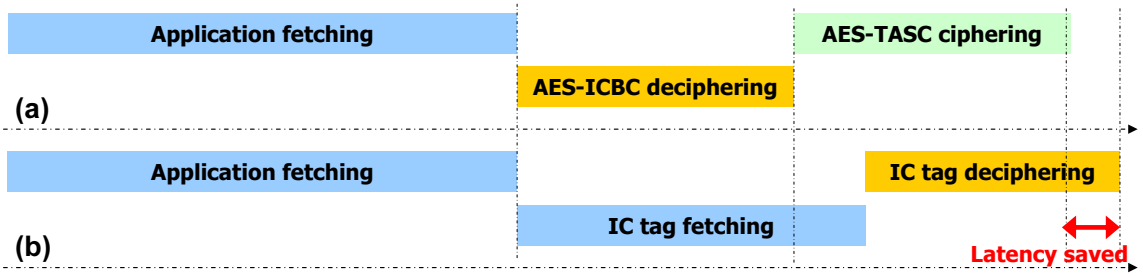

Figure 4.7: Latency loading and deciphering with long latency flash memory:
a - 2-pass scheme
b - 1-pass scheme

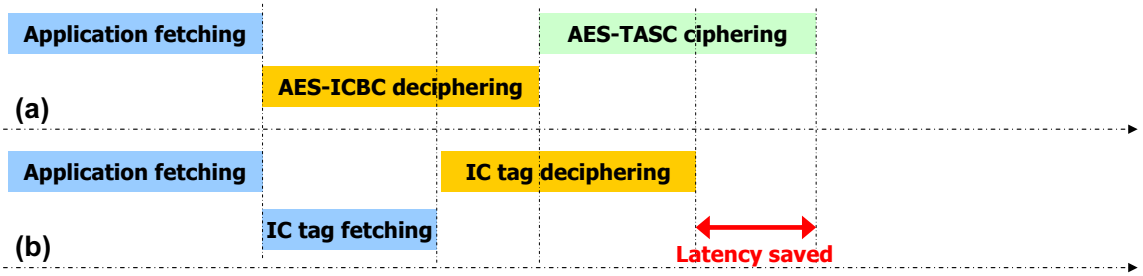

Figure 4.8: Latency loading and deciphering with short latency flash memory:
a - 2-pass scheme
b - 1-pass scheme

tags.

These drawbacks of the 1-pass scheme is counter balanced by the fact that the time to perform the application loading should be lower. Indeed, with the 2-pass scheme the whole application is first deciphered with the AES-ICBC and then ciphered with the AES-TASC. With the 1-pass scheme, the IC tags are protected with AES-ICBC and the application code is already ciphered with the AES-TASC. The IC tag memory is much smaller than the application size so the time to decipher it will be lower. The ratio between the application and the IC tag footprint size depends on the IC tag size. With large IC tags a lot of memory will be needed for the tag storage in the flash (from 12,5% to 25% as mentioned above). However the time needed to load the application will not always be longer with the 2-pass scheme as explained in the following. Indeed, depending on the flash latency one scheme or another can be more efficient (see Figures 4.7 and 4.8). The next section details the equations which help to evaluate the case in which the first scheme is more interesting than the second one.

4.3.2 Case study for boot time & memory footprint

The aim of this section is to evaluate the different schemes used for the SHDL. Two cases must be discussed. The first one is the case of a uniformed protected application with confidentiality and integrity checking. The second case is when the SHDL is applied on the programmable protection. Depending on these two cases, the most interesting scheme that should be chosen to load the data is not always the same.

Equation 4 - ***Boot time equation***

2-pass scheme:

$1 - Application\ fetching\ time = Block\ fetching\ latency * Nb\ of\ blocks$

$2 - AES - ICBC\ time = Block\ deciphering\ latency * Nb\ of\ blocks$

$3 - AES - TASC\ time = Block\ ciphering\ latency * Nb\ of\ protected\ blocks\ with\ confidentiality$

$4 - Loading\ time = Application\ fetching\ time + AES - ICBC\ time + AES - TASC\ time$

1-pass scheme:

$5 - Application\ fetching\ time = Block\ fetching\ latency * Nb\ of\ blocks$

$6 - IC\ tag\ fetching\ time = Block\ fetching\ latency * Nb\ of\ IC\ tag\ blocks$

$7 - AES - ICBC\ time = Block\ deciphering\ latency * Nb\ of\ IC\ tag\ blocks$

$8 - Loading\ time = Application\ fetching\ time + IC\ tag\ fetching\ time + AES - ICBC\ time$

Equation 4 shows the detailed equations to evaluate the application loading time for the 1-pass and 2-pass schemes. The equations 4.4 and 4.8 only represent the total latency to load code. Each equation is composed of 3 subequations. In both cases, the application needs to be loaded from the flash (Equation 4.1 and Equation 4.5 also see Figures 4.7 and 4.8). Then some data are deciphered through AES-ICBC: the application code for the 2-pass scheme (Equation 4.2) or the IC tags for the 1-pass scheme (Equation 4.7). Since the behavior of the two solutions is a bit different, the difference also appears in the equations to compute the latency. For the 2-pass scheme, the latency to perform the application encryption with AES-TASC must be included (Equation 4.3). In the case of the 1-pass scheme, the latency to fetch the extra data (IC tags) from the flash memory which must be taken in account (Equation 4.6). Based on these equations, figures of Tables 4.2 and 4.3 have been generated. The flash memory latency and the RAM memory latency has been obtained through experiment with the Stratix II 2S60 FPGA board at a frequency of 85 MHz. Table 4.2 introduces the figures for the applications presented in the previous chapter with the uniform protection (UP) approach. For Table 4.3, the programmable protection is applied to the applications, so the time to boot is affected because all the data in memory are not protected. Furthermore, the tables have been provided for both AES-GMC and AES-ICBC in order to have an overview of the different loading times offered by the different scheme combinations.

Uniform protection

With the uniform protection, the 2-pass scheme is the most efficient one. Indeed as underlined previously, no memory overhead is necessary contrary to the 1-pass scheme where an increase of 12.5% to 25% of memory is required to store the encrypted IC tags of 16 and 32 bits. This penalty will be further increased when using AES-ICBC compared with AES-GCM since AES-ICBC requires to store the data related to the counter values. It implies an additional increase of memory from 12.5% to 25% for 16 and 32 bits counters. Concerning the boot time, the 2-pass scheme limits the loading time overhead to 5.7% with the AES-GCM mode. With the AES-ICBC the loading time overhead rises to 18.9%. From Table 4.2, it clearly appears that AES-GCM is more interesting for the uniform protection with the 2-pass scheme. The data loading time is lower and less memory consuming. In the case of 2-pass scheme, the memory difference only comes from the AES mode. Contrary to the 1-pass scheme where the loading time overhead depends also on the IC tag size. With the 1-pass scheme the memory overhead is almost the same between AES-GCM and AES-ICBC. With 32 bits IC tags, the overhead is 26.4% and 13.2% with 16 bits tag for AES-GCM. Again the penalty is more important with the AES-ICBC, 29.7% for the 32 bits IC tag and 14.8% with the 16 bits IC tag. Here it clearly appears that the 2-pass scheme is the most efficient solution with the AES-GCM but it is not always true with AES-ICBC. Also these figures confirm the fact that AES-GCM mode is more efficient (load time and memory) than AES-ICBC at the price of more logic area. Nevertheless, the loading time difference is small (between 200 and 500 μs) which is almost negligible compared with a 2.5 or 5 ms second boot time. From this first analysis with the uniform protection, it is really difficult to plan what will be the most interesting solution. The user will need to perform a preliminary evaluation based on the Equation 4.

Programmable protection

With the programmable protection approach, the 2-pass scheme is not always the most efficient. It is dependent on the security policy applied by the software designer. Indeed, if a large part of the data is not protected or protected with confidentiality only, it means that the protected IC tag memory will be smaller and will require less time to be securely loaded. In addition, for non protected data since the 2-pass scheme needs to decipher the whole data some latency is added to this scheme compared with the 1-pass scheme where the data are already stored in clear in the flash. As shown on Figure 4.6, the data are ready to be used by the AES-TASC mechanism. It means that in the case of a programmable protection approach if a data is supposed to be non protected in the RAM memory, this data is not protected neither in the flash. A good example is the load time of **Hash** application where all the application code is protected with confidentiality only. As a consequence, no IC tag memory is needed so the data load time is the time needed to copy the data from the flash to the RAM memory. Concerning the comparison between AES-GCM and AES-ICBC, in the case of the 2-pass scheme, the AES-GCM mode is more interesting in all the cases. With the 1-pass scheme the loading time and the

AES-GCM	Application	Non protected		2-pass		1-pass 16 bits IC tag		1-pass 32 bits IC tag	
		time (ms)	size (kB)	time	size	time	memory	time	memory
	Image	2.18	80	2.30	80	2.47	90	2.76	100
	VOD	4.15	152	4.38	152	4.69	168	5.24	190
	Comm	1.94	71	2.04	71	2.19	80	2.45	88.8
	Hash	2.51	92	2.65	92	2.51	92	2.51	92
AES-ICBC	Application	Non protected		2-pass		1-pass 16 bits IC tag		1-pass 32 bits IC tag	
		time (ms)	size (kB)	time	size	time	memory	time	memory
	Image	2.18	80	2.59	90	2.50	91	2.83	102
	VOD	4.15	152	4.93	171	4.76	170	5.37	194
	Comm	1.94	71	2.30	80	2.22	81	2.51	91
	Hash	2.51	92	2.98	103	2.51	92	2.51	92

Table 4.2: Application loading time and flash memory size with uniform protection

memory footprint are almost equivalent due to the security policy applied by the designer which reduces the amount of external memory protected. Like the uniform approach, with the 1-pass scheme the AES-GCM and AES-ICBC modes are not really different from a performance point of view.

From all the figures provided by the 4 tables, there is no clear rule which can define which solution would be better. In order to make the right choice, the user must use the equations to evaluate the loading time. Nevertheless, Figure 4.9 gives the global trend of the boot time for the uniform protection. It is not possible to extract a trend for the programmable approach since the results are fully dependent of the security policy chosen by the software designer. To conclude this analysis, if large parts of the memory are not protected or protected with confidentiality only, the 1 pass scheme should be more interesting. Moreover, these 2 schemes can be applied to any HSC based on other ciphering mechanisms. Concerning the AES-GCM and AES-ICBC comparison, it appears that the difference is always under 500 μs which is really small from a system boot up time point of view. In a FPGA based system, the FPGA configuration time must also be included in the system boot up time.

	Application	Non protected		2-pass		1-pass 16 bits IC tag		1-pass 32 bits IC tag	
		time (ms)	size (kB)	time	size	time	memory	time	memory
AES-GCM	Image	2.18	80	2.27	80	2.27	83.15	2.36	86.3
	VOD	4.15	152	4.34	152	4.24	155.25	4.33	158.5
	Comm	1.94	71	2.04	71	2.19	80	2.45	88.8
	Hash	2.51	92	2.65	92	2.51	92	2.51	92
	Application	Non protected		2-pass		1-pass 16 bits IC tag		1-pass 32 bits IC tag	
		time (ms)	size (kB)	time	size	time	memory	time	memory
AES-ICBC	Image	2.18	80	2.54	90	2.28	83.5	2.38	87
	VOD	4.15	152	4.86	171	4.25	155.25	4.33	159
	Comm	1.94	71	2.28	80	2.22	81	2.51	89
	Hash	2.51	92	2.98	103	2.51	92	2.51	92

Table 4.3: Application loading time and flash memory size with programmable protection

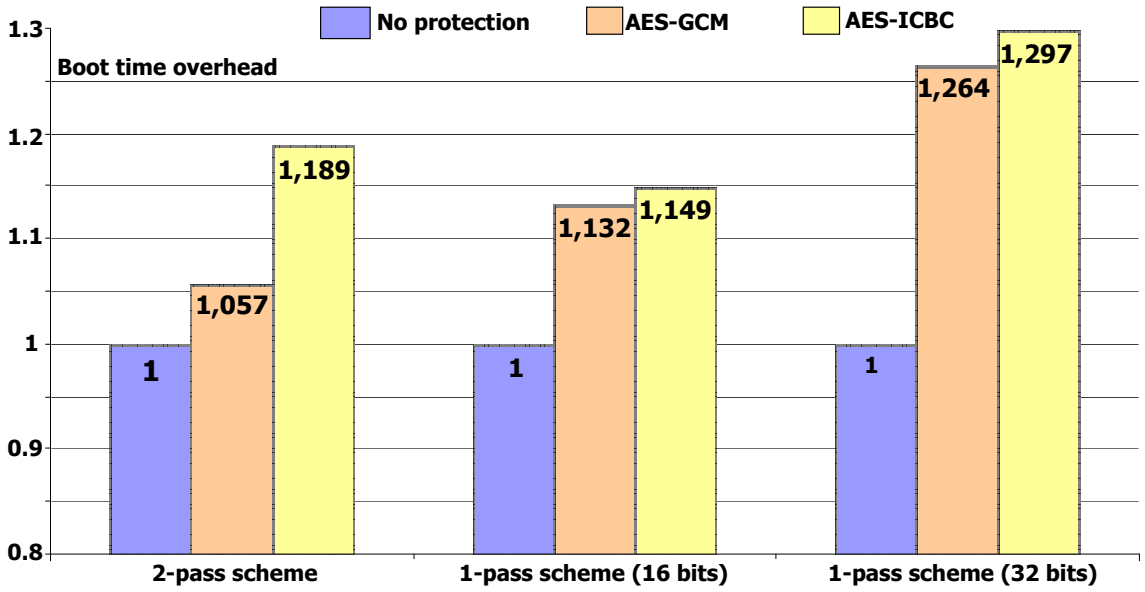

Figure 4.9: Trend of boot time overhead for different boot configuration with an uniform protection

4.3.3 Typical use with an FPGA architecture

Here the typical system boot up will be presented based on the mechanisms detailed above. Figure 4.10 describes the boot in the typical case with an FPGA architecture. First of all, the FPGA must be securely configured. Altera and Xilinx provide several solutions in order to securely configure an FPGA [Altera, 2008c] [Altera, 2008b] [Xilinx, 2008b]. The FPGA bitstream can be stored in a flash memory and then securely loaded into the FPGA thanks to FPGA supplier mechanisms. The FPGA bitstream content is: NIOS II processor, some peripherals and the HSC. The SMM configuration introduced in the previous chapter section 3.1.2 is hard coded in the bitstream. As the bitstream is securely stored in the flash memory the SMM configuration can not be modified and/or attacked. This way as soon as the FPGA is configured, the secure loading of the application can be launched. Since the FPGA logic owns the SMM configuration, the application code will be loaded in the RAM memory with the right policy provided by the SMM. The mechanism used to load the application is one of the solutions presented in the previous section (SHDL). When the FPGA and the application have been securely loaded, the system can start the secure application execution. Here the system is protected from the FPGA configuration to the application execution.

The main issue with this approach is the fact that the SMM configuration is hardcoded in the FPGA bitstream. It means that if the user wishes to run another application on the FPGA, he needs to change the content of the flash. Indeed, a new application will have a different SMM so a new bitstream with the new FPGA configuration must be provided. Several bitstreams could be stored in the flash memory even if the bitstream size is quite large compared with an application (4 MB and even more for a bitsream). Another possibility would be to have a specific architecture to handle several applications with only one bitstream. This solution would be for example the concatenation of all the application SMM. But this solution is not really interesting because the system would require a very large amount of on-chip memory to store all the TS and IC tags for all the possible applications. Furthermore, the logic SMM size would be also much bigger than the values presented in chapter 3 section 3.3.2. In the following an open secure platform which can handle several applications with several security policies without changing the bitstream is described.

4.4 Open Secure Platform

In the previous section, it has been underlined that each application needs a dedicated bitstream. Indeed, each application needs a specific SMM to correctly handle the security so each application needs a bitstream. With the Open Secure Platform, one bitstream is able to handle several applications. In order to do that the SMM will be modified to support configuration. The hardware architecture will only be able to handle applications with a security policy which matches with the hardware

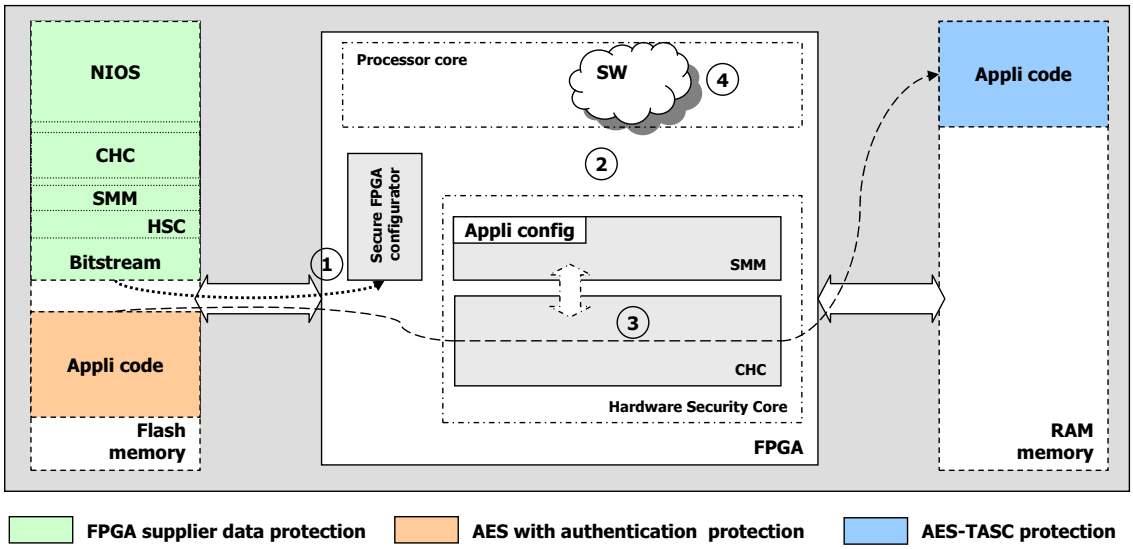

Figure 4.10: Typical system boot of a PFGA based architecture

1. The FPGA configuration is securely load with the appropriate secure mechanism.

2. The FPGA is configured with the NIOS processor and the HSC and its SMM configured for the application.

3. The application code is securely load in the RAM memory with the SHDL scheme.

4. The application is securely executed from the RAM memory on the secure architecture loaded in the FPGA.

architecture characteristics. In the case of an *AES-TASC/CRC* architecture, the limitations are: the TS memory size, the CRC memory size and the number of memory segments the SMM is able to store. For example, if an application code is very big, the amount of CRC to store might be to big for the CRC memory capacity of the HSC.

The architecture for the open secure platform relies on an HSC with a configurable SMM and a CHC core. As mentioned in section 4.3, before storing the data in the RAM memory, the right security level must be applied to the data during application loading. This security level is obtained thanks to the information stored in the SMM. In the case of a generic architecture which is supposed to handle several applications, the SMM must be first configured at boot up before starting the application loading. That is why each application will have a header. This header contains the SMM configuration which needs to be securely loaded prior to the application loading. The next section details the way the application memory block is built and what is exactly included in the application header. Then section 4.4.2 presents the architecture boot scheme with the SMM configuration prior to the application loading.

4.4.1 Memory block layout

Prior to the application loading the SMM configuration is mandatory. The SMM configuration is stored in the flash memory in the application header (see Figure 4.11). As the flash memory is accessible by the attacker the SMM configuration can be attacked and so needs to be protected. The SHDL scheme introduced in section 4.3 is applied to the SMM configuration. Figure 4.11 exhibits the layout of a memory block. A memory block is composed of a protected application header and the protected application. The header contains information which are necessary to perform the application loading. The SMM configuration, the application size and the RAM memory address where the application code must be loaded. The header size will vary depending on the size of the SMM configuration:

- Header initialization vector (256 bits)
- Application address (32 bits)
- Application size (32 bits)
- SMM configuration (64 bits * number of segments)
- Header Random Value (256 bits)

The SHDL scheme is relying on the AES-GCM or AES-ICBC mode to decipher and authenticate data. These two modes are using an IV/AAD and a RV/tag. As mention in sections 4.2.1 and 4.2.2, the IV/AAD and the RV/tag are stored in clear

in the memory. Since the header is protected through the SHDL scheme these two values are necessary to securely load the header data. Since the application is also loaded thanks to the SHDL, the application block also has a specific IV/AAD and RV/tag for its secure load.

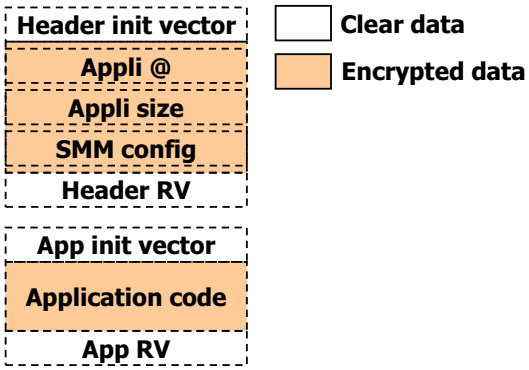

Figure 4.11: Protected memory block

4.4.2 Boot up & code loading scheduling

4.4.2.1 Operations scheduling

Figure 4.12 describes the operation schedule. Several applications can be stored is the system flash memory. Each application has its own application header in order to correctly configured the SMM and correctly launch the application loading. The boot loader code is in charge of launching the right application depending on input parameters (for example sensor, jumper). Like any other application, the boot loader code needs to be securely loaded in the RAM memory before execution. So a header is also mandatory for the boot loader code.

The boot loader code loading is done in two steps. The first step is the loading and the checking of the boot header. The header contains some information about the loading (address, size) and also the SMM configuration (security level, segment address) of the boot loader code. The header secure load is done using one of the schemes presented before (1 or 2 passes SHDL). As soon as the SMM configuration is checked thanks to the SHDL scheme, the boot configuration is loaded in the SMM. Indeed, the SMM configuration must be checked before being loaded in the hardware in order to skip any potential misconfiguration and potential threats for the architecture. If an attacker modifies the data in the flash memory, the header must be checked before being loaded in the SMM to prevent any misconfiguration. The SMM must be configured with the right security policy to guarantee the boot loader code protection in the RAM memory. Like any application, the boat loader is an application which needs to be protected so a security policy is mandatory to guarantee a secure execution.

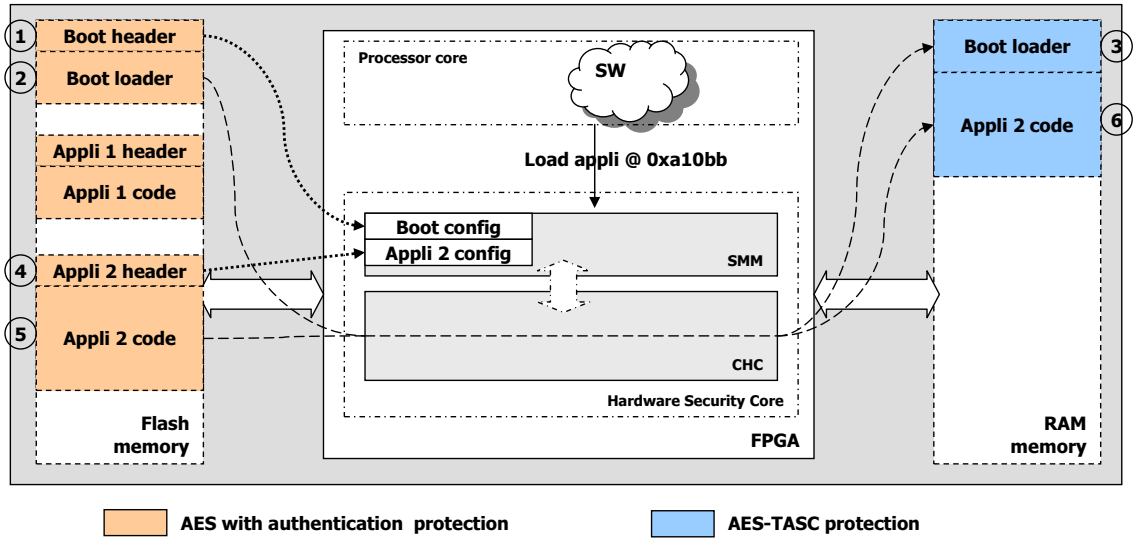

Figure 4.12: Boot and application loading scheduling

Figure 4.12 shows the detailed schedule of operation.

1. The boot loader header is securely loaded with the SHDL scheme.
2. The boot loader code is securely loaded with the SHDL scheme.
3. The boot loader code is securely stored in the RAM memory
4. The boot loader code is securely executed and launches the application secure loading.
5. The application header is securely loaded with the SHDL scheme.
6. The application code is securely loaded with the SHDL scheme.
7. The application code is securely stored in the RAM memory
8. The application code is securely executed

At this point the hardware architecture is loaded and is ready to start the boot loader code loading which is the second step. When the boot code header is checked and the SMM configured, the SHDL is launched to load the boot loader code. Then the boot loader code is securely executed from the RAM memory thanks to the SHDL scheme. As soon as the the secure loading of the code is done, the secure boot execution can start. During the secure boot loader execution, an instruction will be sent to the HSC to launch the application loading. This specific step will be presented in the following section.

The application secure loading is done the same way as the boot loader code based on the SHDL scheme. Two steps are also necessary, the application header is securely loaded first in order to configure the SMM correctly for the application and then the application code is loaded. The application can then be securely executed. This whole scheme relies on the fact that the FPGA configuration has been done securely thanks to the FPGA supplier strategy.

4.4.2.2 Cost of the scheme

The software designer is in charge of the right choice concerning his applications policy. Indeed, the hardware for security is limited, it means that he will have to take into account the hardware possibilities of the architecture to shape his security policy. It will be the designer responsibility to have a software which matches with the hardware capacities.

This approach will imply two costs. The first one is the memory required by the boot loader. Based on the experiments, the boot loader size will be negligible compared with the application code size. The boot loader code is usually included in all embedded system softwares. The extension to a standard boot loader will be the piece of code needed to chose which application should be launched. So this part should be minor compared with the application size. The second cost of this approach concerns the hardware necessary to have a configurable SMM. As it was shown in chapter 3 section 3.3.2, the SMM size impacts the architecture area. This configurable SMM is a little bit different. Some logic is required to configure it and also a small on-chip memory (less than 2kB) to store the SMM configuration. The logic is in charge of detecting which security level should be applied depending on the requested data by the processor. This logic size is totally dependent on the number of segments the SMM manages. The first experiment leads to an increase of 5 to 15% of the design size based on the results presented in chapter 3 section 3.3.2. The 15% overhead is in a case of a 15 segments SMM.

4.4.2.3 Application update

Nowadays, it is common to have software update of the system (firmware or new applications download). This section details how the approach presented above can be extended to support this kind of update. As said in the previous section, it is the boot loader which takes care of the application launch. It means that if a new application is remotely downloaded in the flash memory the current boot loader code will not be able to handle this new application at boot up. The basic idea is to download a new boot loader code which has a new code which is able to launch the new application. It means that the system needs to be reboot to launch the new application. It could be possible to launch one application from another application code but this idea will imply some difficulties in the SMM configuration management or a bigger SMM which should be able to manage several applications at the same time. Moreover, the user expects to use his new downloaded application at the next system boot up so this new application must be available to be launched by the boot loader code. Each time a new application will be downloaded in the system a new boot loader will also be needed. As mentioned in the last section the size of the boot loader code is almost negligible compared with the application code size. As a consequence, this extra data download is not a real issue.

4.4.2.4 Secure Application Launcher

The boot loader code has the possibility to launch any application loading from the flash memory. This operation is a little bit critical because it will imply a change of the SMM configuration and also modify the behavior of the system since a new application might be loaded. Figures 4.13 and 4.14 give an overview of the operation call scheduling to the Secure Application Launcher (SAL).

In order to authorize the application loading from the flash memory, the HSC must first be unlocked with a secret key (Ukey). As soon as the Ukey has been entered in the HSC and compared with the hard coded key (Hkey), the processor allows one access to the configuration register for only one write operation to launch the application loading. When the write operation to the secure register is done, the HSC will automatically lock itself and erase the Ukey from the register. If another application must be loaded, the processor must enter the Ukey a second time to access the configuration register.

The Ukey and Hkey are 128 bits long. The Ukey is securely stored in the RAM memory in the boot loader code (see Figures 4.13 and 4.14). The boot load is protected with the highest security level (confidentiality and integrity) by the hardware security core. So if the attacker tries to change the Ukey in the RAM memory, the system will detect it. Moreover, since the RAM memory has the confidentiality protection the attacker can not know the Ukey. It means that he will have to guess the Ukey and due to the system latency the time to try almost all the possibilities should overcome the system lifetime. Of course the Ukey size can be extended to 256 bits

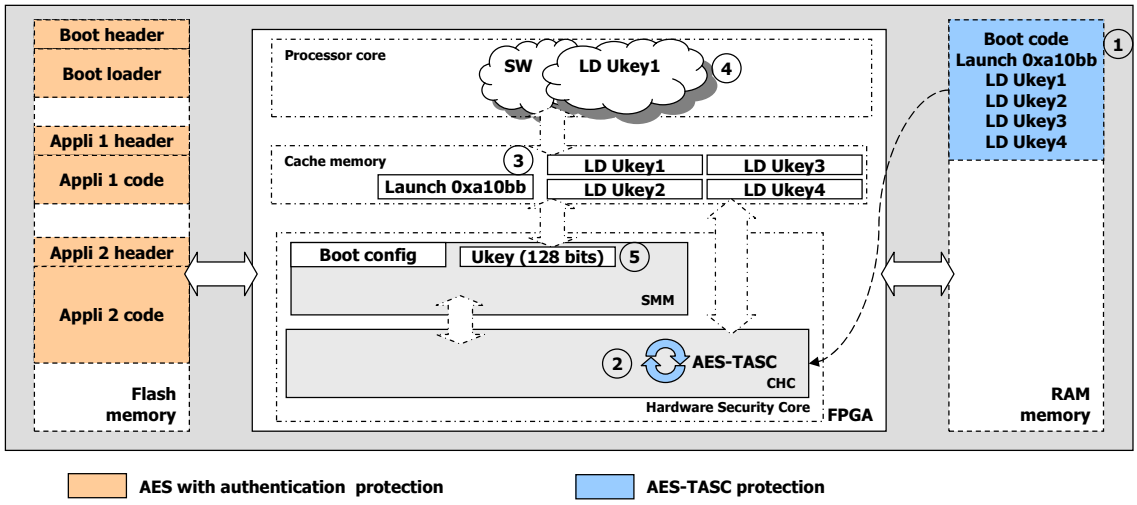

Figure 4.13: Secure application loader sequence (1/2)

Figure 4.13

1 - The boot loader code including the load instruction for the key is securely stored in the off-chip memory.
2 - The *load key* instructions are fetched from the external memory and checked with the *AES-TASC/CRC* scheme.
3 - The *load key* and *launch* instructions are moved in clear in the cache.
4 - The processor executes the *load key*.
5 - The processor writes the key value in the SMM in order to unlock the system.

Figure 4.14

6 - The SMM checks the registered key (*Ukey*) with the hard coded key (*Hkey*), if the keys match the access to the application launch register is allowed.
7 - The processor executes the *launch* and the register of the SMM are updated with the address of the application to securely load.

for a higher security level. The Hkey is hard coded in the FPGA bitstream and as a consequence protected through the secure FPGA configuration. The mechanism is fully hardware. The instructions used for the communication between the processor and the HSC are not added to the processor instruction set. The HSC is seen like a standard peripheral by the processor with some accessible registers (for the Ukey). In order to modify these registers, a simple "load" assembly instruction is enough.

The secure application launcher will add more hardware to the global architecture. The interface for the processor communication with the HSC and the secure register access requires around 350 ALUTs and 240 FFs. In the case of an architecture with a 256 bits secret key, more area would be required to implement the solution. From an execution latency point of view, the time needed to load the instructions from the RAM memory and to execute these instructions are taking less than 100 cycles. The time to execute the secure application launcher is negligible compared with the execution time of the application and the system lifetime.

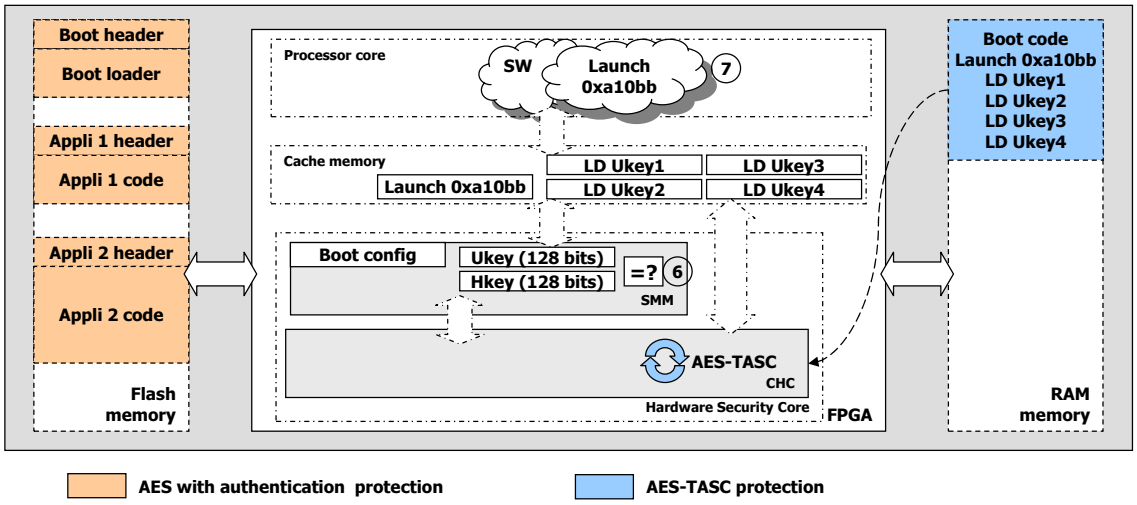

Figure 4.14: Secure application loader sequence (2/2)

4.5 Benefits of the Open Secure Platform

The main interest of the Open Secure Platform is the flexibility offered to the software designer. With this new end to end architecture, he can securely boot his system, execution several applications without changing the bitstream. The flexibility and security has a price: a bigger area and also more time to boot the system. The extra time required by the secure boot must be compared with the FPGA configuration time. Based on the equation provided by the FPGA suppliers [Altera, 2008a] [Xilinx, 2008c], the time to configure the FPGA in the ideal case is: size of the bitstream * loading frequency * width of the bus. In the case of a 4 MB bitstream with an 8 bits data width flash memory at a frequency of 50 MHz, the time to load a bitstream should be around 80 ms. However, this value is optimistic. Indeed, during the experiment made to evaluate the application time, it clearly appears that the flash memory latency is much longer than the system frequency. Based on the measured latency of the flash memory with the board already used for previous experiments (see section 3.3.1), the real configuration time of the bitstream is around 850 ms. In section 4.3, the results show that in the worse case the time to load an application is around 5 ms. The application loading time is low compared with the FPGA time configuration. It means that even if the application loading time takes more time, it will still be negligible compared with the FPGA configuration time. In addition, the 850 ms of configuration is the value in the case of a non secure FPGA configuration. In all the schemes presented in this chapter the assumption was that the FPGA configuration is done securely. In this case, the time needed to configure the FPGA will be longer than the expected 850 ms because of the deciphering step which adds some latency to the process.

The configuration capabilities of the new architecture perfectly matches with the requirements of embedded systems. One of the main concern was to offer a system

which supports the field update. With the current security approach, the update is fully supported. The software application can be updated and the hardware architecture is configured at boot up in consequence. Moreover, as the solution was mainly thought to fit with reconfigurable embedded systems (FPGA based), it is interesting to notice that the update is now only dedicated for the software and not for the FPGA bitstream. The FPGA bitstream is never changed. So the solution could be applied to ASIC. The reason of using an FPGA based architecture is in the case of an application which will be to big for the architecture resources with an ASIC (performance, memory footprint, energy consumption). In this particular case, in addition to the application update a bitstream update in the flash would also be mandatory.

Conclusion

This chapter has detailed the last secure features that need to be added to a system in order to be fully operational on the field. The major concern is the system boot. Moreover, all the specific features to manage the boot have been extended in order to support on field update. With the extended communication between systems this last feature is essential to match with the current demand of embedded architectures.

All these new possibilities offered have a price. The time to boot and also the small flash memory overhead are not real issues due to the current capabilities of embedded systems. The main point is the hardware overhead. For the code loading, the extra logic is quite small. For the configurable SMM, the logic price is more important. This price is counter balanced by the flexibility offered to the software designer. He now has the key to build a secure system based on a hardware which is as much as possible independent of the software needs.

CHAPTER 5

Security scheme update

This chapter presents an update of the security scheme introduced in chapter 2. Indeed a weakness due to the CRC was discovered [Borisov et al., 2001]. Like any other solutions weaknesses are found (XOM for example and the replay attacks) and the approach needs to be updated. It is a classical process in the security field. People find weaknesses and engineers try to correct the errors. The new integrity security scheme relying on an AES round is fully detailed and a new set of results has been generated in order to evaluate the costs and benefits of this new scheme. All the results of chapter 3 are updated and some more results are presented since this new security scheme offers some new security possibilities to the architecture

Contents

5.1 CRC weakness

In chapter 2, the integrity checking is guaranteed through the CRC checking. Equations 5 shows a possible weakness due to the linearity of the CRC function. It leads to a possibility for the attacker of changing the ciphertext without being detected by the integrity checking unit.

Equation 5 - *CRC linearity*

Lineraity property:
$1 - CRC(A \oplus B) = CRC(A) \oplus CRC(B)$

Possible modification due to linearity:
$2 - Ciphertext = Plaintext \oplus Keystream \text{ and } P = C \oplus K$
$3 - CRC(P) = CRC(C \oplus K) = CRC(C) \oplus CRC(K)$
$4 - P' = C' \oplus K \text{ with } C' = C \oplus Modification \text{ added by the attacker}$
$3 - CRC(P') = CRC(C' \oplus K) = CRC(C') \oplus CRC(K)$
$6 - CRC(C') = CRC(C \oplus M) = CRC(C) \oplus CRC(M)$
$7 - If\ CRC(M) = 0\ Then\ CRC(C') = CRC(C)$
$8 - So\ CRC(P') = CRC(P) \text{ with } P' \neq P$

In a real case application, it should be easy to find a value which provides 0 at the CRC output. Nevertheless, it does not mean that the attackers will be able to find a good value which will help him to do what he wants with the system. It opens the door to some potential threats and thus must be addressed. It is still a challenge to find value which will match with the expected CRC result and which will also match with the behavior the attacker is expected to get with the third part code he is trying to execute on the architecture.

5.2 New integrity checking approach

The main interests of the CRC checking approach were the very low logic amount necessary to implement the function and the low latency time to obtain a CRC result (one clock cycle). The goal now is to find a function with the same properties but this function must not be linear. A very well known non linear function which has been proved to be secure is the AES round. It is also key dependent which leads to more difficulties for the attackers to break the secure mechanism. Figures 5.1 and 5.2 show the architecture evolutions. There are no major changes. The architecture is always composed of the AES-TASC scheme for de/ciphering. The CRC integrity checking block is replaced by a new one. The CRC memory is now called IC tag memory. The global architecture behavior is still the same.

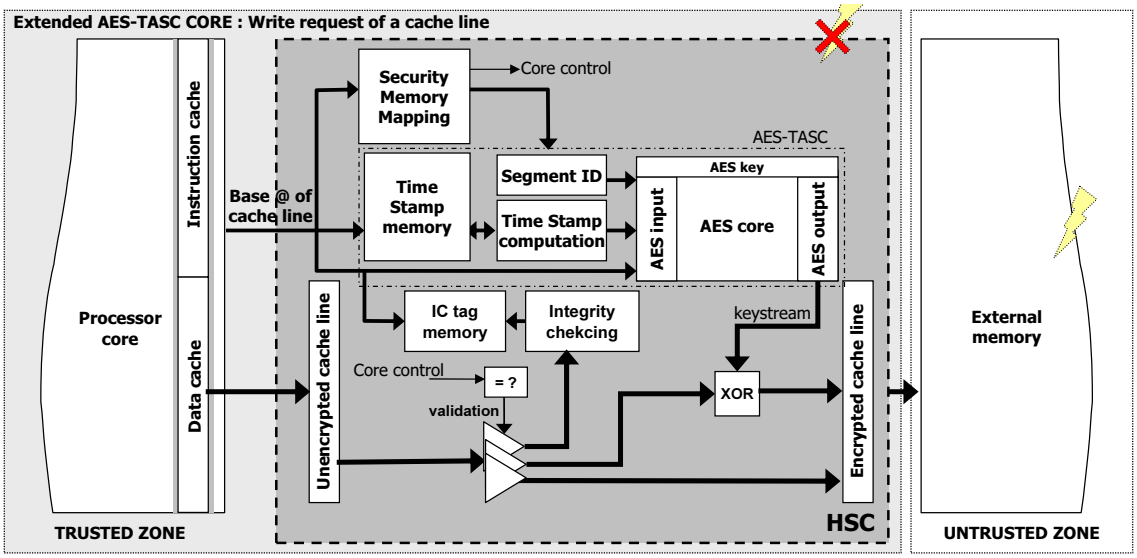

Figure 5.1: AES-TASC architecture for write request

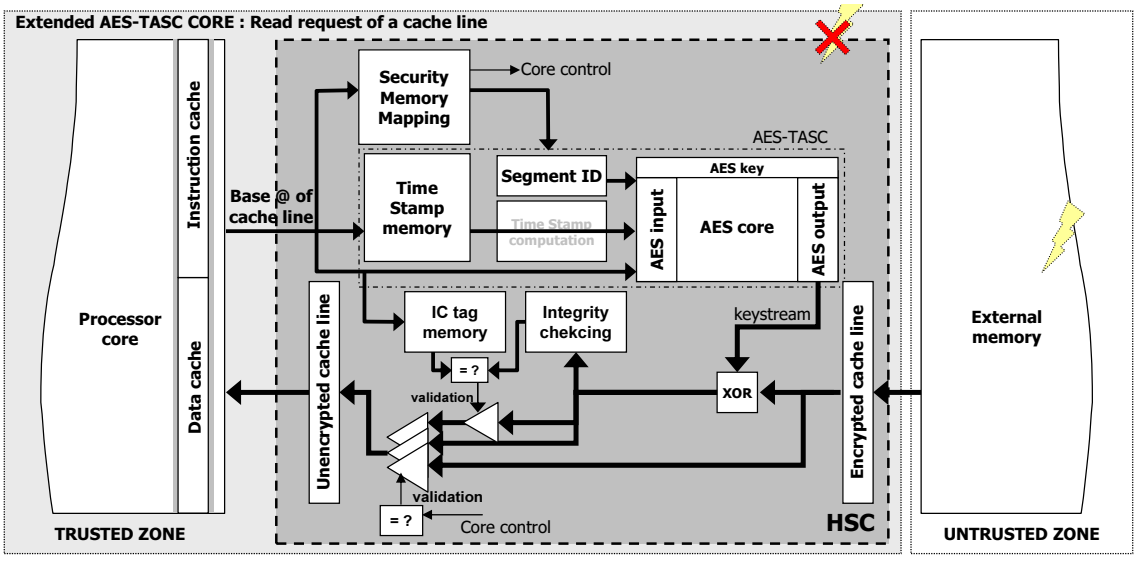

Figure 5.2: AES-TASC architecture for read request

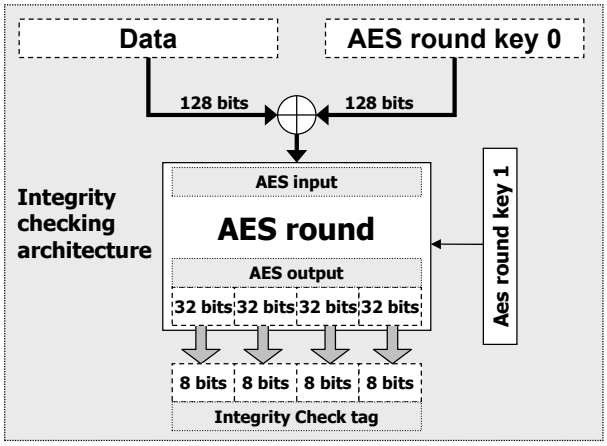

Figure 5.3: Integrity checking architecture with one AES round

5.2.1 AES round for integrity checking

The idea is to use the spreading feature of the AES round. In [Daemen and Rijmen, 2002], the authors underline that AES is built in a way that if one input bit is changed, after two AES rounds half of the output bits will be dependent of the first round input bits. In order to have a full diffusion 4 AES rounds are necessary. The goal here is not to use AES for cipher or for its resistance against linear and differential cryptanalysis but for this specific spreading characteristic.

Figure 5.3 gives a detailed view of the scheme used for integrity checking. It is a typical AES first round. The key point is the way the output bits are selected. As said above, after two rounds half of the bits should be dependent of the initial values. It comes from the way the AES round is built. For the integrity checking application, the output bits selected as part of the tag must be carefully chosen.

Figure 5.4 exhibits the way an AES round is built. One round is composed of 4 operations: shiftRows, subBytes, MixColumns and addRoundKey. By studying the way the shiftRows operation works, it appears that each 32 bits output word of the AES round is dependent on 4 specific 8 bits words of the input as shown on Figure 5.4. For example, the first 32 bits of the output depends on the 8 bit words numbered 0, 5, A and F. In order to detect a one bit change in any on these 16 words of 8 bits of the AES inputs, several bits of each 32 bits output must be selected. This choice is due to the way the computation is done in the MixColumns operation. The solution to increase the security level of the approach is to use a bigger tag (64 bits and even more).

The integrity scheme will be same as the one presented in chapter 2. During a write request, an IC tag of the data to be encrypted will be computed and stored in the IC tag memory for later comparison. When the data will be requested for read, the IC tag stored in the on-chip memory will be compared with the IC tag newly

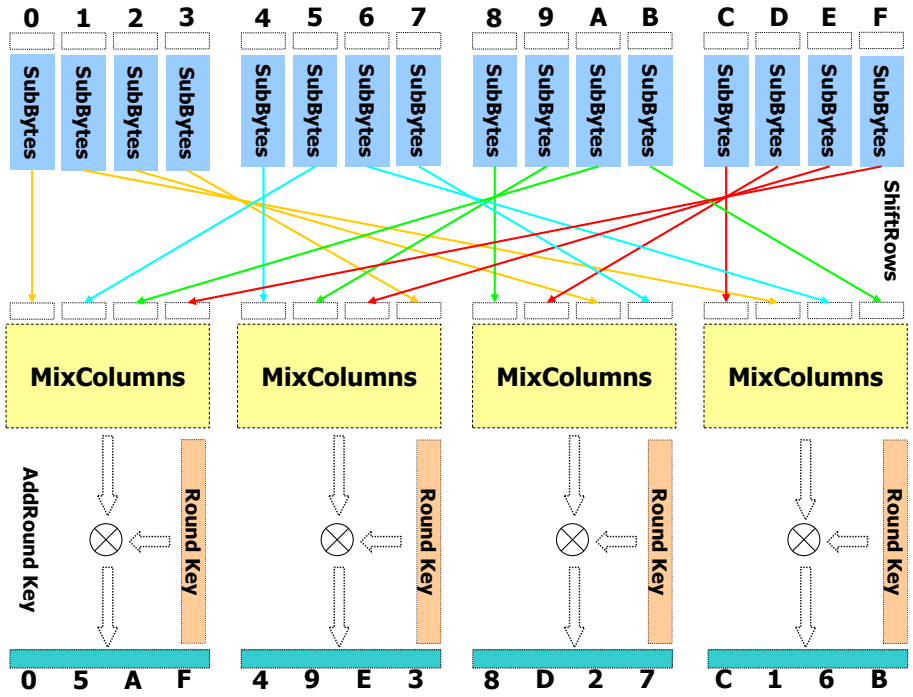

Figure 5.4: AES round function

computed with deciphered value. If the two IC tags match it means that the data integrity is right. The new integrity scheme is still very efficient because like the CRC checking it can be done in only one clock cycle. Furthermore, like the CRC approach the IC tag size can be reduced to limit the amount of on-chip memory for the IC tags. In chapter 2 Figure 2.4.3 shows that a pipeline version of the integrity scheme leads to better performances, with this new scheme this pipeline approach can also be applied. The last critical point is the size of this new integrity checking block in hardware. The next section will show that it is even smaller than the previous approach with the CRC checking.

5.3 Results tables update

This section provides a very complete set of results for the new architecture with the AES used for integrity checking. All the results presented in chapter 3 have been updated and several other results are provided. Concerning memory overhead, the new approach do not change anything. The IC tag size can be managed the same way as the CRC results so no new memory footprint figures are needed.

First of all the area of the architecture remains almost the same. Around 180 ALUTs and 100 FFS are saved compared with the previous core version with the CRC for integrity checking. The AES round is as expected a solution which is low area consuming and it is an essential point. In addition, some new architecture

	Uniform protection CI				Programmable protection			
Archi	NIOS + HSC		HSC		NIOS + HSC		HSC	
	ALUTs	FFs	ALUTs	Fs	ALUTs	FFs	ALUTs	FFs
Image	7971	4569	3149	1020	8165	4603	3328	1048
VOD	8159	4602	3299	1054	8157	4592	3311	1042
Comm	7975	4576	3141	1022	8112	4584	3273	1042
Hash	8073	4245	3129	999	7086	4397	2295	848

Table 5.1: Detailed breakdown of hardware security core (HSC) resource usage

	Uniform protection CO				Uniform protection ICO			
Archi	NIOS + HSC		HSC		NIOS + HSC		HSC	
	ALUTs	FFs	ALUTs	Fs	ALUTs	FFs	ALUTs	FFs
Image	7297	4379	2517	833	6489	4111	1393	566
VOD	7386	4404	2592	853	6513	4145	1468	581
Comm	7278	4381	2509	832	6510	4124	1431	573
Hash	7326	4405	2553	854	6490	4124	1441	572

Table 5.2: Detailed breakdown of hardware security core (HSC) resource usage

results have been added in Tables 5.2 and 5.3. As underlined earlier, the CRC was not strong enough to provide the integrity checking only (ICO) protection. Now with the new integrity checking scheme this protection level is available and the architecture sizes for different applications are proposed. In this particular case, the global architecture size is even lower than with the CO protection level.

A complete new set of results has also been generated for the performance application. As mentioned above, the latency of the AES round integrity checking is the same as the CRC. It means that all the performance results including the new integrity scheme are the same as before. Again some extended figures have been provided for the ICO and CO protection. In almost all the cases the programmable approach keeps being the most efficient solution. Concerning uniform protection with CO, the performances are really interesting. It comes from the fact that as soon as a data is fetched from the memory, it can be deciphered immediately. While an architecture with integrity checking requires to wait for one half of the cache line to check the line integrity and send it to the processor core. That is why the performance of the CI and ICO protections are almost the same. In both cases, the performance penalty comes from the delay of waiting the half of the cache line because of the integrity checking.

The last point which has been updated is the energy consumption. As stated above, for the programmable approach and for the architecture including the integrity checking the global design size has not changed that much. Moreover the performances are the same. It means that the energy consumption will also remain the same. The small area reduction has not really impacted the energy consump-

	Programmable protection									
Application	Total		AES-TASC unit		IC unit		SMM		Control	
	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs
Image	3328	1048	1584	434	319	153	144	31	1281	430
VOD	3311	1042	1586	432	319	153	108	27	1298	430
Comm	3273	1042	1608	439	320	154	41	10	1304	430
Hash	2355	898	1282	434	0	0	8	1	1005	413
	Uniform protection CI									
Application	Total		AES-TASC unit		IC unit		SMM		Control	
	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs
Image	3149	1020	1501	433	319	154	15	3	1314	430
VOD	3299	1054	1561	440	407	177	19	3	1312	434
Comm	3141	1022	1503	436	317	153	13	3	1308	430
Hash	3129	999	1502	431	317	153	14	3	1306	412
	Uniform protection CO									
Application	Total		AES-TASC unit		IC unit		SMM		Control	
	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs
Image	2517	833	1507	433	0	0	14	3	996	397
VOD	2592	853	1555	436	0	0	15	3	1022	414
Comm	2509	832	1502	431	0	0	12	3	995	398
Hash	2613	904	1518	438	0	0	14	3	1021	413
	Uniform protection ICO									
Application	Total		AES-TASC unit		IC unit		SMM		Control	
	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs
Image	1393	566	0	0	337	151	12	2	1044	413
VOD	1468	581	0	0	404	153	15	3	1049	425
Comm	1431	573	0	0	355	151	14	3	1062	419
Hash	1441	572	0	0	340	151	12	3	1089	418

Table 5.3: Detailed breakdown of hardware security core (HSC) resource usage

	No protection	Uniform protection CI		Uniform protection CO		Uniform protection ICO		Programmable protection	
	(ms)	(ms)		(ms)		(ms)		(ms)	
Image 512	690	894	-23%	785	-12%	890	-22.5%	794	-13%
VOD 512	8227	10360	-21%	9164	-9.8%	10234	-19.3%	9405	-13%
Comm 512	42.8	53.1	-20%	46.2	-7.4%	49.4	-14.2%	49.0	-13%
Hash 512	5.3	7.5	-29.3%	6.4	-17.2%	7.3	-27%	6.2	-14%

Table 5.4: Application execution time and performance reduction

	No protection	Uniform protection CI		Uniform protection CO		Uniform protection ICO		Programmable protection	
	(mJ)	(mJ)		(mJ)		(mJ)		(mJ)	
Image 512	556	909	63%	795	43%	870	56%	800	44%
VOD 512	7434	12419	67%	10227	37%	10315	38%	9717	30%
Comm 512	36.4	57.9	59%	502	37%	48.34	32%	53.5	46%
Hash 512	4.7	8.37	77%	7	48%	6.92	47%	6.62	40%

Table 5.5: Application energy consumption

tion. Indeed, due to the lack of precision in the power measurement it is not possible to detect any variation. Nevertheless, it was underlined that a small change is the power consumption of the core is really negligible compared with the power consumption of the chip I/O.

Some new figures are provided for the confidentiality only and integrity checking only architectures. It appears that the CO protection has the lowest cost in energy (even lower than the programmable approach). For the ICO protection, the energy penalty is really important even if the core size is the smallest one. This energy overhead is due to the performance penalty. The global ICO architecture power is lower but as the execution time is much longer then the energy is higher.

Conclusion

The architecture weakness has been detected and successfully removed. The new approach relying on the AES round offers the same capabilities as the CRC core and a more secure architecture. There are no extra costs due to this security update of the scheme. The new scheme logic area is even lower than the previous one. The scheme relying on an AES round still needs to be study to see if several round are necessary or not.

The complete new set of results gives an overview of the cost of all the security policy the architecture can handle. For the results, it has be shown that even though the security update, the security cost remains the same and slightly lower. The approach described in this chapter still fits with the requirements of embedded systems (low area, low memory footprint, low energy consumption).

Conclusion & perspectives

Conclusion

All along this document the goal was to propose the first hardware security scheme which fits with the embedded systems requirements. Among the chapters, the solution was incrementally built to finish with the proposition of an end to end solution. All the results obtained through implementations and experimentations lead to think that the solution matches with the statements established in introduction and during the first chapter. Thanks to the work presented in this document, an architecture can securely boot up, execute and update the application with the right security policy.

As underlined in the introduction, the main interest of hardware architecture is the good performance obtained compared with software approaches. However the system logic size is the point which is the most badly impacted . If all the scheme is included in one architecture (AES-TASC, integrity checking, boot up, secure application loading), the core size compared with a non protected solution would be twice more important. For the software performance execution, the designer security policy is the key. Depending on his choices, the application execution will be efficient or not. If he wishes to have a very secure architecture, the application performance will be lower than with a security policy where large parts of application code and data are not protected. Concerning the memory, the designer should also carefully choose his policy. In addition depending on his choice for the boot up scheme and the time to load data, the data amount in flash memory may vary. There are a lot of parameters to take into account when building a secure application. In the document several sections with results have been added to give some general trends of the security cost. For some features, memory for example, it is possible to know exactly what will be the on-chip cost related to the security policy. For performances, a complete profiling is needed to obtain more accurate results.

The cost of security may still seem to be high. At one point, Digital Signal Processor were introduced in embedded system in order to respect the time constraint of real time applications. Maybe in the future, secure processors will be considered as part of the design and the overhead due to security will not be compared anymore to

the general purpose processor. Moreover, it has been shown through all the results and analysis that it is quite complex to build an architecture and the software in a secure way. Since there are special software designers for signal processing, there might be soon special designers for secure applications. In this document, the steps needed to evaluate the cost of a secure architecture have been proposed. Now it would be interesting to have a tool which gives a set of expected results based on the designer policy wishes. It would help for the resource exploration, performance and memory cost. The work done here offered enough information to built this kind of tool.

The proposed approach has been built based on the threat model established in chapter 1. From a security point of view, the protection offered by the new architecture is guaranteed with reasonable resistance against attacks. The integrity protection is limited by the possibility offered by the tag size (2^{32} for example). The tag size choice depends on the security and performance wished by the designer. For a fast and secure execution a lot of IC tag storage will be needed. The scheme has been built on blocks which are mostly relying on the AES core. There might still be some open doors for attacks. There are several security issues which have not been taken in account (side channel attacks for example). The protection against side channel attacks would require more area to implement counter measures. The threat model extension to these kind of attacks should lead to the reduction of theses potential security issues but at the cost of more hardware.

The work presented in this document has lead to the proposition of a secure architecture which is configurable from a security level point of view. This architecture has been characterized (area, performance, memory, power consumption and security) in order to offer to the designer the idea of the security cost he might expect if he considers this solution. The results have been obtained through several applications, the strengths and the weaknesses of the proposed solution has be discussed and sometimes have lead to updates (chapter 5 for example). Some parts of this work have been introduced in a way that schemes are not limited to the proposed architecture but can be extended to other secure architecture. A good example is the SMM use which is not limited to the AES-TASC/CRC architecture, it could be used with any other security solutions which support datapath configuration to offer several security level. In the same way, the schemes and methods presented for the secure boot, secure update and the open secure platform are not limited to the AES-TASC architecture but could be used or modified to work with any other approaches.

Perspectives

In the whole document, the reduction of the security cost was one of the main concern. The limitation of the design area is one of these costs. In the current core

version several AES round cores are used. It could be possible to only use one core in order to minimize the area. Several logic optimizations of this kind could be done with the goal to reduce again the security area cost and so the power consumption. In addition, there is one security issue which needs to be fix. If an attacker spies the addresses bus sequence and if he has a deep knowledge of the processor instruction-set, he can infer the code which is executed and then bypass the confidentiality. Some work has already been done to protect a system against this kind of attacks. It should not reduce the system efficiency but increase a little bit the core size.

In the embedded system presentation in chapter 1, Figure 1.1 shows a detailed architecture with several processors in a single chip. One perspective for the next secure architectures would be to study how is it possible to handle security for these kinds of multi-processors architectures? Will one processor be in charge of the secure software execution? Must the security be spread all over the architecture? How the processor should internally communicate to exchange sensitive data? There are a lot of challenges and issues to tackle with these kinds of architectures. Moreover since the complexity and the application numbers are growing on embedded systems, the number of dedicated processors inside the chip will also increase in order to support all the expected functionalities and constraints.

Recently a new approach is emerging. Operating system virtualization is a new hot topic in embedded systems. Indeed, instead of having one processor associated with one OS, several OS are running on the same processor. It offers new perspectives for security. Indeed, by having two operating systems running on one processor, it means that the processor can execute applications from two different memory spaces. Then it is easy to imagine to have one of the two OS which is dedicated to the execution of secure applications. Thanks to the two memory spaces, the non secure operating system can not access the data of the secure one. Nevertheless, there is still a need for memory protection. In addition, the communication between the two OS must be studied. The non secured OS may need some sensitive data. Some interesting challenges are appearing in that field and new perspectives for security but also for attackers.

Other perspectives based on this work would be the exploration of the security cost with other ciphering and hashing solutions. Indeed, the SMM idea and the open platform can be applied to other security schemes (PE-ICE and AEGIS for example). It will be interesting to see what is the impact on system performances and security level. Maybe solutions which are not suitable for embedded systems will be suitable for implementation with the SMM which helps to reduce the security penalties. In addition, the work was mainly focused on AES core. It might also be interesting to evaluate the security cost with other AES final candidates. Maybe the performances and area may be better than the current version of the hardware security core proposed in this document.

All along the document, security is linked with the operating system and the memory layout due to task. Moreover, it has been underlined that the OS context switching is naturally protected due to the way the solution is built. There is one specific aspect which has not been studied. Indeed, hardware architectures often include hardware accelerators which are used by tasks to speed up the process. But in software when a switch context occurs, the data task are protected when the registers are securely moved in the memory. This secure software switch context should be extended to the accelerator registers used by the task. The HSC will do a secure hardware switch context for these registers. It could be mapped in the SMM like any other memory segment in order to apply the right security policy on the accelerator data.

All the results presented in this document have been generated with FPGA. But the most interesting feature of FPGA has not been used. The reconfigurable capacity is something that needs to be explored for security. Furthermore, the partial and dynamic reconfigurable features should offer more perspectives. For example, with the approach presented in chapter 3, the SMM might be reconfigured on fly to handle a new application. But in order to do that, a secure partial reconfiguration scheme should be introduced because it is not available right now. Moreover, the reconfigurable capabilities must be analysed to be sure that it will not bring new weaknesses to the architecture.

People have done a lot of research on security from a software and hardware point of view. The next step may be to merge these two fields in order to have a secure software running on secure hardware which would offer secure primitives to speed up the software execution. The combination of software and hardware for security should lead to very complex architectures and as soon as the complexity increases the possible attacks and weaknesses too. More material to protect a system also means more potential threats for the system.

Publications

Journal

A Security Approach for Off-chip Memory in Embedded Microprocessor Systems
Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet, Eduardo Wanderley, Russell Tessier, Wayne Burleson
Journal on Microprocessors and Microsystems, 2008

Conferences

Memory Security Management for Reconfigurable Embedded Systems
Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet, Russell Tessier, Deepak Unnikrishnan, Kris Gaj
International Conference on Field-Programmable Technology, 2008

High-efficiency protection solution for off-chip memory in embedded systems
Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet, Russell Tessier, Wayne Burleson
Engineering of Reconfigurable System and Algorithms, 2007

Low latency solution for confidentiality and integrity checking in embedded systems with off-chip memory
Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet, Eduardo Wanderley, Russell Tessier, Wayne Burleson
Reconfigurable Comminucation-centric SoCs, 2007

Trusted Computing - A New Challenge for Embedded Systems
Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet
Internationnal Conference on Electronics, Circuits and Systems, 2006

Secure architecture in embedded systems: an overview
Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet
Reconfigurable Comminucation-centric SoCs, 2006

Bibliography

[3DES, 1995] 3DES (1995). Triple data encryption standard (3des), rfc 1851. In *http://www.faqs.org/rfcs/rfc1851.html*.

[AES, 2001] AES (2001). Advanced encryption standard (aes), fips 197. In *http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf*.

[Agrawal et al., 2003] Agrawal, D., Archambeault, B., Rao, J., and Rohatgi, P. (2003). Side channel attack methodology - multi-channel attacks. In *Lecture Notes in Computer Science of the University of Madrid*.

[Altera, 2008a] Altera (2008a). Altera website. In *http://www.altera.com/*.

[Altera, 2008b] Altera (2008b). Design security in stratix ii gx devices. In *http://www.altera.com/products/devices/stratix-fpgas/stratix-ii/stratix-ii-gx/features/security/s2gx-security.html*.

[Altera, 2008c] Altera (2008c). Fpga design security solution using a secure memory device reference design. In *http://www.altera.com/support/refdesigns/sys-sol/indust_mil/ref-des-secur-mem.html*.

[Anderson, 2001] Anderson, R. (2001). Security engineering: A guide to building dependable distributed systems.

[Arlot, 2008] Arlot, P. (2008). La sécurite dans les téléphones mobiles passe aussi par la virtualisation. In *Electronique international, http://www.electronique.biz*.

[Berger, 2001] Berger, A. (2001).

[Borisov et al., 2001] Borisov, N., Goldberg, I., and Wagner, D. (2001). Intercepting mobile communications: The insecurity of 802.11. In *Conference on Mobile Computing and Networking*.

[Dadvar and Skadron, 2005] Dadvar, P. and Skadron, K. (2005). Potential thermal security risks. In *21st IEEE Semiconductor Thermal Measurement and Management Symposium*, pages 229–234.

[Daemen and Rijmen, 2002] Daemen, J. and Rijmen, V. (2002). Aes - the advanced encryption standard. In *The Design of Rijndael*.

[Dagon et al., 2004] Dagon, D., Martin, T., and Starner, T. (2004). Mobile phones as computing devices: The viruses are coming! In *IEEE Pervasive Computing*, pages 11–15.

[Dougherty and Lotufo, 2003] Dougherty, E. and Lotufo, R. (2003). In *Hands-on Morphological Image Processing*.

[Drimer, 2008] Drimer, S. (2008). Securing sram fpga designs in distribution and in operation. In *CryptArchi 2008*.

[Ducloyer et al., 2007] Ducloyer, S., Vaslin, R., Gogniat, G., and Wanderley, E. (2007). Hardware implementation of a multi-mode hash architecture for md5, sha-1 and sha-2. In *DASIP 2007 : Workshop on Design and Architectures for Signal and Image Processing*.

[ECC, 2002] ECC (2002). Use of elliptic curve cryptography (ecc) algorithms in cryptographic message syntax, rfc 3278. In *http://www.faqs.org/rfcs/rfc2437.html*.

[Elbaz et al., 2007] Elbaz, R., Champagne, D., Lee, R. B., Torres, L., Sassatelli, G., and Guillemin, P. (2007). Tec-tree: A low cost and parallelizable tree for efficient defense against memory replay attacks. In *CHES'07: Workshop on Cryptographic Hardware and Embedded Systems*, pages 289–302.

[Elbaz et al., 2005] Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M., and Martinez, A. (2005). Seca: security-enhanced communication architecture. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 78–89.

[Elbaz et al., 2006] Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M., and Martinez, A. (2006). A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 506–509.

[Guilley and Pacalet, 2004] Guilley, S. and Pacalet, R. (2004). Socs security: a war against side-channels. In *Annales des télécommunications*, pages 998–1009.

[Kocher, 1996] Kocher, P. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113.

[Kocher et al., 1999] Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *Proceedings of the 19th Annual International Cryptology*, pages 388–397.

[Kristensen et al., 2006] Kristensen, M., Soren, S0rensen, LeMoullec, Y., and Koch, P. (2006). Efficient algorithm and system architecture for the suppression of mpeg artifacts. In *24th Norchip Conference*, pages 293–296.

[LaBrosse, 2008] LaBrosse, J. (2008). Microc/os-ii the real-time kernel, 2002. In *micrium website http://www.micrium.com/*, pages 179–190.

[Liardet and Teglia, 2004] Liardet, P.-Y. and Teglia, Y. (2004). Fault resistance: from reliability to safety. In *Workshop on fault diagnosis ans tolerance in cryptography*.

[Lie et al., 2003] Lie, D., Thekkath, C., and Horowitz, M. (2003). Implementing an untrusted operating system on trusted hardware. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 178–192.

[Mao and Wolf, 2007] Mao, S. and Wolf, T. (2007). In *Design Automation Conference (DAC)*.

[Martin et al., 2004] Martin, T., Hsiao, M., Ha, D., and Krishnaswami, J. (2004). Denial-of-service attacks on battery-powered mobile computers. In *RCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, page 309.

[MD5, 1992] MD5 (1992). The md5 message-digest algorithm, rfc 1321. In *http://www.faqs.org/rfcs/rfc1321.html*.

[Merkle, 1980] Merkle, R. (1980). Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, page 122.

[Nash et al., 2005] Nash, D., Martin, T., Ha, D., and Hsiao, M. (2005). Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices. In *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 141–145.

[NIST, 2001] NIST (2001). Recommendation for block cipher modes of operation: Methods and techniques. *National Institute of Standard and Technology Special Publication 800-38A*.

[NIST, 2004] NIST (2004). Recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality. *National Institute of Standard and Technology Special Publication 800-38C*.

[NIST, 2007] NIST (2007). Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. *National Institute of Standard and Technology Special Publication 800-38D*.

[Peterson and Weldon, 1961] Peterson, W. and Weldon, E. (1961). Cyclic codes for error detection. In *Proceedings of the IRE*, pages 228–235.

[Ravi et al., 2004a] Ravi, S., Raghunathan, A., Kocher, P., and Hattangady, S. (2004a). Security in embedded systems: Design challenges. *Transaction on Embedded Computing System*, pages 461–491.

[Ravi et al., 2004b] Ravi, S., Raghunathan, A., Kocher, P., and Hattangady, S. (2004b). Tamper resistance mechanisms for secure embedded systems. *International Conference on VLSI Design*.

[Reed and Solomon, 1960] Reed, I. and Solomon, G. (1960). Polynomial codes over certain finite fields. In *Industrial and Applied Mathematics*, pages 300–304.

[Rogaway et al., 2003] Rogaway, P., Bellar, M., Black, J., and Krovetz, T. (2003). Ocb: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transaction on Information and System Security*, pages 365–403.

[RSA, 1998] RSA (1998). Rsa cryptography specifications version 2.0, rfc 2437. In *http://www.faqs.org/rfcs/rfc2437.html*.

[Senn et al., 2008] Senn, E., Laurent, J., Juin, E., and Diguet, J.-P. (2008). Refining power consumption analysis in the component based aadl design flow. In *Forum on specification and design languages*.

[SHA-1, 1995] SHA-1 (1995). Secure hash standard ,fips 180-1. In *http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf*.

[SHA-2, 2001] SHA-2 (2001). Secure hash standard ,fips 180-2. In *http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf*.

[Sokolov et al., 2005] Sokolov, D., Murphy, J., Bystrov, A., and Yakovlev, A. (2005). Design and analysis of dual-rail circuits for security applications. In *IEEE Trans. Comput.*, pages 449–460.

[Suh et al., 2003] Suh, E., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S. (2003). Efficient memory integrity verification and encryption for secure processors. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 339.

[Suh et al., 2008] Suh, E., O'Donnell, C., and Devadas, S. (2008). Aegis: A single-chip secure processor. In *IEEE Design & Test of Computers*.

[Suh et al., 2005] Suh, E., O'Donnell, C., Sachdev, I., and Devadas, S. (2005). Design and implementation of the aegis single-chip secure processor using physical random functions. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 25–36.

[TRANGO, 2008] TRANGO (2008). Trango virtual processor website. In *http://www.trango-vp.com/*.

[Xilinx, 2008a] Xilinx (2008a). Microblaze processor. In *http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm*.

[Xilinx, 2008b] Xilinx (2008b). Secure chip aes bitstream encryption/decryption technology. In *http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/designsec.htm*.

[Xilinx, 2008c] Xilinx (2008c). Xilinx website. In *http://www.xilinx.com/*.

[Yan et al., 2006] Yan, C., Rogers, B., Englender, D., Solihin, Y., and Prvulovic, M. (2006). Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the International Symposium on Computer Architecture*, pages 179–190.

[Yang et al., 2003] Yang, J., Zhang, Y., and Gao, L. (2003). Fast secure processor for inhibiting software piracy and tampering. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 339.

[Yang et al., 2005] Yang, J., Zhang, Y., and Gao, L. (2005). Fast secure processor for inhibiting software piracy and tampering. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 351–360.

Abstract

Hardware Core for Off-chip Memory Security Management in Embedded System

Nowadays embedded systems are part of our lives. Most of them are storing or allowing people to exchange sensitive data. As a user we wish that no one will be able to steal any information. For companies which are developing product, they do not want to see their intellectual properties stolen or their product modified in order to execute any third-party applications. The system security and its correct behavior associate the users and companies against possible threats for the system. Dedicated and suitable security solutions for embedded systems is a new challenging topic. Solutions coming from computer science are not adapted for embedded systems. Indeed, there are less available resources and also new constraints like the power consumption for autonomous systems.

We offer a secure hardware architecture for system boot up, secure software execution and on field update. A new scheme is presented to guarantee data confidentiality and integrity for off-chip memories. The architecture capabilities are extended to support on the fly security level management of data. The goal is to minimize the overhead due to security like logic area, performance, memory footprint and power consumption for the architecture. After careful evaluation through real time applications execution with this secure architecture, the next step was to provide an end to end solution. Toward this solution, a secure boot up mechanism is proposed in order to securely start applications from a flash memory. More techniques are also introduced to allow on field software update for later secure execution with the architecture.

A complete set of results has been generated in order to underline the fact that the proposed solution matches with the current needs and constraints of embedded systems. For the first time the security cost in area, performance, memory and power has been evaluated for embedded systems with an end to end solution.

Résumé

Architecture matérielle pour la protection dynamique des données en mémoire externe dans les systèmes embarqués

Les systèmes embarqués ont envahi notre quotidien. La plupart d'entre eux stockent ou permettent d'échanger des données sensibles. En tant qu'utilisateur nous souhaitons être assurés qu'aucun vol d'information ne peut être possible. De même, les sociétiés qui conçoivent ces systèmes souhaitent ne pas voir leurs propriétés intellectuelles volées ou leurs appareils modifiés grâce à des applications tierces. La sécurité du système ainsi que son bon fonctionnement sont deux aspects qui unissent l'utilisateur et le concepteur dans une lutte contre les possibles attaques à l'encontre du système. L'apport de solutions pour la protection d'un système embarqué est un nouveau challenge qu'il va falloir relever. Les solutions de l'informatique classique ne sont pas viables dans le domaine de l'embarqué. En effet, les ressources disponibles sont moins importantes et de nouvelles contraintes apparaissent notamment celle de la consommation énergétique.

Nous proposons une architecture matérielle sécurisée du démarrage du système en passant par l'exécution des applications jusqu'à sa mise à jour sur le terrain. Une nouvelle technique afin de garantir la confidentialité et l'intégrité des données en mémoires est présentée et évaluée dans un premier temps. L'architecture proposée est alors étendue avec de nouvelles fonctionnalités qui permettent de gérer à la volée le niveau de sécurité spécifique à la donnée. Ceci ayant pour but de minimiser au maximum les coûts engendrés par la sécurité et notamment la surface, la performance, la consommation mémoire et énergétique de l'architecture. Cette base étant évaluée au travers de différentes applications temps réel s'exécutant sur l'architecture sécurisée, l'étape suivante est la mise en oeuvre complète d'un système. Pour cela une méthode de démarrage sécurisée est également proposée afin de lancer les applications depuis une mémoire flash. D'autre mécanismes sont également introduits afin de permettre une mise à jour des applications contenues dans la flash et leur exécution par la suite sur l'architecture sécurisée.

L'ensemble des résultats générés ont pour but de montrer que la solution proposée correspond aux besoins et aux capacités des systèmes embarqués. Pour la première fois le coût de la sécurité a été évalué sur l'ensemble des caractéristiques spécifiques au domaine des systèmes embarqués (surface, performance, consommation mémoire et énergétique) pour une chaine totalement sécurisée.