POLITECNICO
DI TORINO

Master's Degree in Electronic Engineering

Thesis

Secure FPGA bitstream management in reconfigurable mobile heterogeneous systems

Supervisors
Stefano Di Carlo
Paolo Ernesto Prinetto

Candidate
Carlo Alberto Cristofanini

Academic year 2017-2018

A tutti coloro che mi hanno supportato fino al raggiungimento di questo importante traguardo

Abstract

In these last years the mobile devices scenario has shown a huge spread all over the world. This has lead to a large diffusion of applications due to their availability, popularity and an easy accessibility. Another key factor that push them forward to a further evolution step, is the increasing computational power of mobile devices. Multicore architectures, heterogeneous computing, GPUs, ASICs have been key points in this evolution. Nowadays, another step forward can be achieved by mean of the reconfigurable computing devices, the FPGAs, which are fundamental parts of the embedded mobile devices of the future. They allow both to reduce the power consumption and to speed up complex operations.

The employing of reprogrammable devices introduces new security issues. FPGAs are programmed via bitstream files whose clear version must be kept secret. To achieve this, their confidentiality, integrity and authenticity must be preserved, defending them against threats by two types of attackers: MITM (man in the middle), which acts on the communication channel, and MATE (man at the end), which has physical access to the system.

The scenario assumed is quite close to the one used in the mobile application deployment. The end user buys an application on an application store; after the payment success confirmation it sends automatically, to the store, an ID to identify its FPGA. The store sends the request to the software provider, which sends both the bitstream file and the ID to the hardware vendor, which is the FPGA manufacturer. This encrypts the bitstream file with a key related to the FPGA ID and sends the ciphered bitstream back to the end user passing through the software provider and the store. The end user's device contains the key needed to decrypt the ciphered bitstream. The plaintext version of the bitstream is used by the device to program the FPGA.

To face the above mentioned problem, the *SEcube*^{TM} platform has been employed. It grants a good security level against MATE attackers; the FPGA and the microcontroller are embedded on the same SoC, therefore it's complex to gain

access to their interconnections.

This thesis focuses on the last passage of the scenario mentioned above, the secure of the FPGA programming phase: After the download, the ciphered bitstream files is stored into an external memory (e.g., SD card) aboard the end user's device. The internal device memory is limited, so a tradeoff between memory occupation and speed must be achieved to avoid the memory overflow. To overcome this issue, a "paging"-like solution has been adopted: the files have been virtually split in blocks. When the FPGA programming function is executed one of these blocks is read, decrypted and then stored into an internal buffer; at the same time the block sign is computed and compared to the given one to verify both integrity and authenticity. Then, the FPGA programming function accesses the buffer, which contains a portion of the plaintext version of the bitstream, to program the FPGA. The obtained clear version of the bitstream mustn't be readable in any way, neither by the end user, which is not reliable.

To protect the confidentiality, integrity and authenticity of the bitstream, both AES in CBC mode and a SHA algorithms have been employed. The CBC mode has been selected, instead of the ECB one, due to its simple decoding implementation and to avoid the main ECB drawback: any resident properties of the plaintext might well show up in the ciphertext using this mode. The internal buffer size has been chosen in order not to occupy all the available space; this is due to the assumption of other functions memory occupation and other free space needing.

This project is a mandatory tile in a bigger puzzle which is the FPGA bitstream securing panorama.

Acknowledgements

After long and intense months the day is finally arrived.
This words of thanks are addressed to all those that helped me with continuous support and patience, immense knowledge and encouragement, in the draft of my thesis, with suggestions criticism and observations. It has been a period of intensive learning, not only on a scientific level, but also on a personale level.

A special thanks is addressed to my thesis supervisors Prof. Stefano Di Carlo, Prof. Paolo Ernesto Prinetto and to the PhD Students for their great support, Alberto Carelli and Alessandro Vallero, because when I ran into a trouble spot or had a question about my research, the project or writing this thesis, they helped me in the right direction whenever I needed it, with sincere availability. I have been extremely lucky to have thesis supervisors who cared so much about my work.

My sincere gratitude is addressed to all of them.

Moreover I would like to thank my mother for her continued support, her encouragement since my childhood enabled me to reach this important goal.

Finally, I want to thank all my friends that have been close to me encouraging and understanding me, in good times and during the hardships and the discomfort throughout my years of study and through the process of searching and writing this thesis.

A heartfelt thanks to everyone.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Heterogeneous computing enters application era

In the last years the high-tech market is showing a huge expansion in terms of mobile devices, like multimedia players, smartphones, pads and latest wearable technologies as smart-watches and fitness trackers. This phenomenon is getting more and more important and involve each areas of everyday life.
The diffusion of these devices go with technology innovation and a significant mobile application market enlargement, leading to its immersion in a competitive environment and business landscape creating the opportunity to introduce new technological solutions based on these devices. Clearly, the mobile application market is starting to heavily affect the global business environment.
Some examples about this world are well known. Apple has always shown to the world its ability in spreading edge technology in its devices. It used to be the only force in mobile technology panorama. Its strength could lean on the iPhone, the iOS operating system, and the Apple App Store.
Nowadays, new rival devices, operating systems and application stores are growing and facing the old sector leader.
About 160 application stores offered by a range of companies, including device manufacturers, platform providers, mobile operators, and media conglomerates are today available.
The Apple App Store inventory, at the end of 2010, counted 300,000 apps, twice the number available the previous year, according to Distimo, a mobile analytics firm. Google Android Market has increased by six times the apps number available in 2009. The same effects can be seen in other mobile application stores (Nokia's Ovi Store, BlackBerry App World).
At the same time mobile apps sale is growing exponentially. The research firm

Gartner reported that mobile application stores delivered 17.7 billion downloads internationally in 2011, more than double 2010's 8.2 billion downloads [1].
Developer revenues came from app purchases, in-app purchases and mobile app advertising. Gartner expects revenue amount to surpass 29.5 USD billion in 2013, more than a fourfold increase over 2010. The Gartner's research director, S. Baghdassarian, doesn't think the app frenzy is just a fashion trend which, like many others, shall pass. She and her team strongly believe there's a bright future full of opportunities for application stores. Clearly applications need to grow up and deliver a superior experience to end users [2].
The opportunity to introduce new technological solutions aboard these devices is due to mobile applications immersion in a competitive environment and in a business landscape. Apps are getting more advanced, that means mobile devices need higher computational power.

The Moore's law has passed its half century lifetime and it's close to its end. At the same time, microprocessor speed has reached its higher limit. It's no longer worth to try to increase the speed given the cost in terms of power consumed and heat dissipated.
Dennard scaling law is related to Moore's law, and states that "at constant transistors power density, they are getting smaller, so the power used is in proportion with area occupied". This law is useful to create a relation between power and area: at fix power density, when transistors density increases, a power consumption reduction for the single transistor is obtained. It can be translated in voltage and current scaling with length. This is what has broken down: not the ability to etch smaller transistors, but the ability to drop the voltage and the current they need to operate reliably [3], [4]. But this law is getting close to its end too; voltage and current scaling with length is reaching its limits, since transistor gates have become too thin, affecting their structural integrity, and currents are starting to leak. Another drawback that must be addressed is thermal losses, which occur when a huge amount of transistors in a small area switch together several billion times per second. Faster transistors switching means that more heat is generated, and without proper cooling system the IC might be destroyed [5],[6]. Computational power reduction is still the main technology goal researchers and developers are trying to achieve.
Before continue with heterogeneous computing solutions it's mandatory to understand where those limitations came from and why they must be respected.
From digital electronic design theory, power dissipated by a device is made up of two contributions: static and dynamic power consumption. They are generated by two different conditions: *static power* tells informations about power dissipation due to leakage currents (I_{stat}) in function with voltage supply (V_{dd}), essentially

consists of the power used when the transistor (fundamental building block of modern electronic devices) is not in the process of switching;

$$P_{stat} = I_{stat} \cdot V_{dd}$$

dynamic power gives informations about power consumption during logical state transition at a specific speed (frequency f) and voltage supply (V_{dd}), with a known capacitive load(C_L).

$$P_{dyn} = f \cdot C_L \cdot V_{dd}^2$$

From the previous formula it's clear that increasing the speed entails a power consumption growth. The same applies to voltage supply but in much higher terms due to its quadratic relation with dynamic power. It affects static power too. Most common strategies to reduce power consumption consist in reducing both frequency and voltage supply splitting the circuit in two or more identical copies and using pipelining techniques. The main drawback is an increasing area occupation, so a trade off between area, frequency, voltage and power must be accomplished [7]. To face the clock rates stalled and the rising power consumption issues, different solutions, within heterogeneous computing, have been found in the last years.

Intel employs the speed/power trade off as fundamental theorem useful to explain the requirement of multicore processors and to describe them; that's the reason which moves the integrated system architectures to add two or more processing areas, or cores, on a single chip. A core is the main element of a processing system, it performs basic arithmetic/logical operations and controls input/output informations. Intel reports that under-clocking (slowing down microprocessor speed) a single core by 20% saves half the power while sacrificing just 13% of the performance. It means that, if a two cores microprocessor runs at an 80% clock rate, a performance enhancement of 73% for the same power is achieved and the heat is dissipated at two points rather than one. But it also means that the area occupied is double [5].

It's passed more than a decade since what has been called "the breakdown of Moore's law" and the switch to multicore processors instead of ever faster single chips. But it's not quite correct at all. Moore's law has not really broken down. Transistor count continues to increase. What has happened is that it is no longer possible to keep running these transistors at ever faster speeds; this is one of the reasons of the general-purpose multicore processing [4]. Parallel processing increases performance by adding more parallel resources (not just cores) while maintaining manageable power characteristics [8]. Analogously to the frequency case, where it's value couldn't be increased beyond a certain limit, the increase of cores number (in transistor budget terms) in multi-core architectures might not be the ideal strategy. Therefore a better design strategy would be to provide a

wider diversity cores capabilities. That diversity can come in the form of heterogeneous general-purpose cores, specialized cores, hardware accelerators, or even configurable fabric. Thread-level parallelism falls short of hardware parallelism, due to a lack of threads or to power constraints (dark silicon). This is the reason of heterogeneous computing requirement. An example that is easy to understand is when three threads are running on an eight-core homogeneous processor, those five idle cores provide no value whatsoever. But on a heterogeneous processor, even idle cores provide value; they could present a more efficient host for one of the running threads [9].

Graphics Processing Units (GPUs) are digital electronic circuits specialized in graphic computing. They've been used mainly as coprocessor to accelerate rendering graphic images creation. Modern GPUs are not only powerful graphics engines, but also highly parallel programmable processors featuring peak arithmetic and memory bandwidth that essentially outperform their CPU counterpart. A broad range of computationally demanding and complex problems have been entrusted to GPUs, due to their rapid increase in both programmability and capability [10]. This effort in general-purpose GPUs (GPGPU) engines allowed to separate regular, parallel, streaming segments of workload from the less parallel, control-intensive segments that still run most effectively on the CPU, positioning the GPUs as a compelling alternative to traditional microprocessors in high-performance computer systems of the future [9].
In "Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes", Johannes Langguth et al. analyze the performance on two platforms and show that combining the CPU and GPU execution capacity clearly provides a performance advantage over the GPU-only approach for irregular applications [11].
Using hardware optimized for specific functions is much more energy efficient than implementing those functions with general-purpose cores. On the other side this might represent an expensive investment for supercomputer customers because custom components, designed for high-end high-performance computing systems, isn't cheap at all. Therefore, high-volume GPU technology becomes a natural choice for energy-efficient data-parallel computing. Computing nodes that compose integrated CPUs and GPUs (called accelerated processing units, APUs, by AMD), along with the hardware and software support, enable scientists to run their scientific experiments on more advanced and sophisticated system [12].
Focusing on mobile world, GPUs allowed to move from a mobile device generation to a newer one. Nevertheless, today it's not powerful enough to manage the computational payload required without significant energy efficiency loss. About this, reconfigurable computing may represent a promising solution.
Hardware accelerators are digital electronic devices, known also as coprocessors,

designed to perform only specific functions which are performed more efficiently than in a general purpose CPU, where an huge amount of clock cycles are required to obtain the same result. Some other examples than GPUs are application-specific integrated circuit (ASIC), cryptographic accelerator, field-programmable gate array (FPGA), digital signal processor (DSP).

1.2 Reconfigurable on heterogeneous computing platforms

Heterogeneous computing architectures are integrated systems in which conventional and specialized processors work cooperatively.
Field Programmable Gate Arrays (FPGA) are semiconductor devices based on a configurable logic blocks (CLBs) matrix, connected via programmable interconnects. FPGAs are hardware-reconfigurable devices that can be redesigned repeatedly by programmers to solve specific types of problems more efficiently. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks.
Although one-time programmable (OTP) FPGAs are available, the dominant types are SRAM based which can be reprogrammed as the design evolves [13].
They have been used as programmable logic devices for more than a decade, but are now attracting stronger interest as reconfigurable coprocessors.
These peculiar devices can be adopted in HPC (high performance computing), enterprise environments and mobile devices thanks to their versatility and power efficiency [14] (Table 1.1).

FPGA Design	
Advantage	*Benefit*
Faster time-to-market	No layout, masks or other manufacturing steps are needed
No upfront non-recurring expenses	Costs typically associated with an ASIC design
Simpler design cycle	Due to software that handles much of the routing, placement, and timing
More predictable project cycle	Due to elimination of potential re-spins, wafer capacities, etc.
Field re-programability	A new bitstream can be uploaded remotely
ASIC Design	
Advantage	*Benefit*
Full custom capability	For design since device is manufactured to design specs
Lower unit costs	For very high volume designs
Smaller form factor	Since device is manufactured to design specs

Table 1.1. FPGA design advantages [15]

FPGAs can deliver orders of magnitude performance improvements over conventional processors on some types of applications [16]. They allow designers to

create a custom instruction set for a given application, and apply hundreds or even thousands of processing elements to an operation simultaneously. For applications that require heavy bit manipulation, adding, multiplication, comparison, convolution or transformation, FPGAs can execute these instructions on thousands of pieces of data at once, with low control overhead and lower power consumption than conventional processors. Systems made of CPU and FPGA, in which hardware acceleration and processing units run in parallel, enhance the total computational throughput of the system.
FPGAs had some historic issues that slowed down their spread; among them their speed (at the beginning they were quite slow devices) and their programming language, which is a Hardware Design Language (HDL). Although these languages are commonplace for electronic designers, they are completely foreign to most HPC system designers, software programmers and users. Nowadays simpler languages are emerging to let use this type of devices to a wide programmers range.
Eventually, as heterogeneous systems incorporating FPGAs become more widely used, it's common to believe they will allow users to solve certain types of problems much faster than anything that will be provided in the near future through Moore's Law, and even support some applications that would not have been possible before [17].
Some of their main applications are: ASIC prototyping, automotive, aerospace, defense, data centers, consumer electronics, industrial, medical, security, video and image processing, communications [18].

The basic structure of a CLB (clearly it is just a common example, they depend on producer strategy design) can be supposed to be made of (Fig.1.1):

- **Look-up table (LUT):** it's a particular logical device which gives in output a specified value as a function of its input;
- **D-FlipFlop (D-FF):** asynchronous set and clear flip-flop;
- **Mux:** multiplexer 2 to 1 used to bypass D-FF in case of pure combinatory cells.

Embedding FPGAs on reconfigurable platforms, gives the chance to create a new types of systems where frequent and remote hardware upgrade is required [19], [20].

FPGA market has grown rapidly in the last thirty years, covering a wide variety of applications in different industrial sectors, thanks to their advantages like their high flexibility.
A new concept of FPGA has born. *Dynamic partial reconfigurable FPGAs* offer new design space introducing some benefits, opening up new interesting applicative scenarios [21]-[22]: reduced configuration time, memory saving as the partial reconfiguration files (bitstreams) are smaller than full ones and possibility of run-time

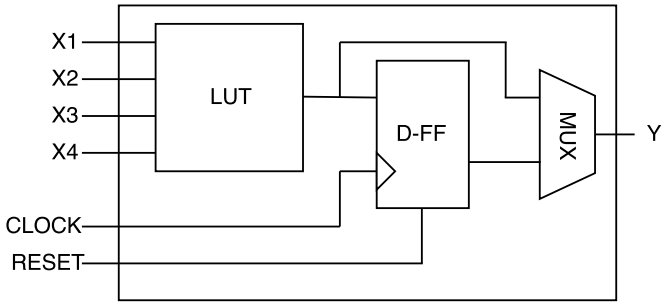


Figure 1.1. Elementary Configurable Logic Block (CLB)

reconfiguring selected portions of a device without affecting the remaining parts of the design. The DPR (dynamic partial reconfiguration) can be exploited in many application fields, for instance to fulfill space requirements in small portable systems, to create a system-on-a-chip with a very high level of flexibility, to realize adaptive hardware algorithms, and so on [23], [21], [24], [22]. It creates a possible scenario in which configurable hardware can be programmed at run-time with application specific computational cores to assist the software execution, leaving the processing unit available to operate new instructions while programmed FPGA works as hardware accelerator [25], [26].

1.3 IP Protection in FPGA-based reconfigurable computing

This innovative panorama introduces a new mobile application pattern that exploits hardware on demand to minimize computational resources with benefits on the overall system complexity. Final product should be an instance made of application specific hardware acceleration cores deployed together with the software application; it will introduce enhancements in mobile applications such as games, audio/video processing, secure communications, etc.
In addition to the benefits, several serious security threats have been introduced. Moving and storing hardware Intellectual Property (IP) (i.e., FPGA biststream files), to use reconfigurable computing technology, means to bring sensitive data over potentially insecure channels and repositories.
A bitstream file can be intercepted by an antagonist, which can spread it out to public domain, sell it for profit (violating the hardware block confidentiality) or may also tamper with the hardware block description trying to corrupt it, injecting malicious functionalities in the hardware core that may either prevent the correct

behavior of the system, or inject security threats in it.
Encryption features have been already provided by hardware vendors (FPGA seller) to help embedded system designers protecting the confidentiality of their products and their IP cores [27], [28], [29], [30].
There already exist some security mechanism, providing bitstream authentication and confidentiality, proposed in several publications [31], [32].
They concern a simple scenario in which the embedded system designer is the only entity entrusted to produce and deliver reconfigurable hardware descriptions to a remote system. In this thesis a more complex scenario is addressed.

1.4 Goal of the thesis

The considered scenario involves several independents parties like system designers, software and reconfigurable hardware providers, etc. This thesis addresses security issues due to the reconfigurable resource sharing by several applications from different vendors; integrity and confidentiality of the provided IP cores must be granted during these passages. Two different types of adversaries are considered as potentially critical to the system security:

- **Remote adversary:** acts on the communication channels between the application providers and the devices (known also as MITM, man in the middle);
- **Local adversary:** has physical access to the system (known also as MATE, man at the end),

For the envisioned structure, three aspects of security services are taken into account:

- hardware resources and system architecture to implement the required security primitives;
- high level software infrastructure needed to implement the required communication protocols;
- high level entrusting policies required among the involved entities.

This thesis project is focused on programming an FPGA using a given encrypted and signed bitstream.
For this purpose, the ciphered file will be stored in an external memory device (more precisely on a micro SD card); in order to gain access to the data, the microprocessor needs to interface to this auxiliary memory device. Encryption and signature type are hypothesized as already known by the firmware designers. The

main contribution to the global project are the development of the infrastructure required to:

- access to the encrypted data on the SD card;
- decrypt this file;
- program the FPGA.

Chapter 2

Cryptography

2.1 Introduction

Bruce Schneier, in his book "Applied Cryptography", introduces the concept of cyber security and cryptography using the following example: "If I take a letter, lock it in a safe, hide the safe somewhere in New York, then tell you to read the letter, that's not security. That's obscurity. On the other hand, if I take a letter and lock it in a safe, and then give you the safe along with the design specifications of the safe and a hundred identical safes with their combinations so that you and the world's best safecrackers can study the locking mechanism, and you still can't open the safe and read the letter, that's security." [33].

Years ago, cryptography was a military and governmental exclusive domain technology, but in last years its use, in everyday devices, has spread significantly. Computer security aims to protect an automated information system in order to attain the applicable objectives of preserving the integrity, availability, and confidentiality of information system resources (hardware, software, firmware, information/data, and telecommunications).
Suppose a sender wants to send a private message securely to a receiver, and wants to be sure that an eavesdropper can't read the message. An original message is called *plaintext* (known also as cleartext). The process of disguising a message, in order to hide message content, is called encryption. An encrypted, coded message is a ciphertext. The reverse process, of turning ciphertext back into plaintext, is decryption. The science of keeping messages secure is called cryptography. Cryptanalysis is instead the science of breaking ciphertext. Cryptology is the branch of mathematics encompassing both cryptography and cryptanalysis.

One of the most known ciphers of the history was the Caesar cipher by Julius Caesar. This cipher is quite simple and widely known, it consits on shifting each letter of the alphabet of three places further down the alphabet and it is wrapped

around, so that the letter following Z is A. For example:

Plaintext:	s	e	c	u	r	i	t	y
Ciphertext:	V	H	F	X	U	L	W	B

Table 2.1. Caesar's cipher, example

0	**V**	**H**	**F**	**X**	**U**	**L**	**W**	**B**
1	w	i	g	y	v	m	x	c
2	x	j	h	z	w	n	y	d
3	y	k	i	a	x	o	z	e
4	z	l	j	b	y	p	a	f
5	a	m	k	c	z	q	b	g
6	b	n	l	d	a	r	c	h
7	c	o	m	e	b	s	d	i
8	d	p	n	f	c	t	e	j
9	e	q	o	g	d	u	f	k
10	f	r	p	h	e	v	g	l
11	g	s	q	i	f	w	h	m
12	h	t	r	j	g	x	i	n
13	i	u	s	k	h	y	j	o
14	j	v	t	l	i	z	k	p
15	k	w	u	m	j	a	l	q
16	l	x	v	n	k	b	m	r
17	m	y	w	o	l	c	n	s
18	n	z	x	p	m	d	o	t
19	o	a	y	q	n	e	p	u
20	p	b	z	r	o	f	q	v
21	q	c	a	s	p	g	r	w
22	r	d	b	t	q	h	s	x
23	**s**	**e**	**c**	**u**	**r**	**i**	**t**	**y**
24	t	f	d	v	s	j	u	z
25	u	g	e	w	t	k	v	a

Table 2.2. Caesar's cipher, method

The main drawback is that the decryption algorithm is simple and if it's known, a

brute-force cryptanalysis is easily performed on a given ciphertext in Caesar cipher form, simply trying all the 25 possible keys [33], [34].
Three important characteristics of this problem enabled us to use a brute-force cryptanalysis (means try all the possible key to find the right one):

1. The encryption and decryption algorithms are known.

2. There are only 25 keys to try.

3. The language of the plaintext is known and easily recognizable.

In most cases the algorithms are known. The use of a large number of keys for an algorithm generally makes the brute-force cryptanalysis impractical. The third characteristic is also significant. If the language of the plaintext is unintelligible, then plaintext output may not be recognizable. Furthermore, the input may be abbreviated or compressed in some fashion, again making recognition difficult [35]. Cryptographic systems were based, at the beginning, on elementary operations as substitution and permutation. Basic types were simply calculated by hand but an advance in symmetric cryptography occurred when the rotor encryption/decryption machine was developed. It allowed a very complex cipher systems development. With the computer era arrival even more complex systems were devised.

Information security purposes is to protect the CIA triad (confidentiality, integrity and availability) of data. These three security objectives for information systems are useful characterized in the following [33], [34]:

- **Confidentiality:** ensures limits or restricted access to the information, preserving authorized restrictions on its access and disclosure, including means for protecting personal privacy and proprietary information.

- **Integrity:** guarding against improper information modification, corruption or destruction, including ensuring information nonrepudiation and authenticity.

- **Availability:** ensuring timely and reliable access to and use of information.

Some people, in the security field, feel that additional concepts are needed to present a complete picture:

- *Authenticity:* the property of being genuine and being able to be verified and trusted; confidence in the validity of a transmission, a message, or message originator. This means verifying that users are who they say they are and that each input arriving at the system came from a trusted source.

- *Accountability:* the security goal that generates the requirements for actions of an entity to be traced uniquely to that entity. This supports nonrepudiation, deterrence, fault isolation, intrusion detection and prevention, and after-action recovery and legal action. Because truly secure systems are not yet an achievable goal, we must be able to trace a security breach to a responsible party. Systems must keep records of their activities to permit later forensic analysis to trace security breaches or to aid in transaction disputes.

- *Nonrepudiation:* a sender should not be able to falsely deny later that he sent a message.

Some issues may occur with associated risks and consequences. A loss of security definition is given for the three main categories [33], [34]:

- **Confidentiality:** a loss of confidentiality is the unauthorized disclosure of information.

- **Integrity:** a loss of integrity is the unauthorized modification or destruction of information.

- **Availability:** a loss of availability is the disruption of access to or use of information or an information system.

Cryptographic algorithms and protocols can be grouped into four main areas:

- *Symmetric encryption:* Used to conceal the contents of blocks or streams of data of any size, including messages, files, encryption keys, and passwords.

- *Asymmetric encryption:* Used to conceal small blocks of data, such as encryption keys and hash function values, which are used in digital signatures.

- *Data integrity algorithms:* Used to protect blocks of data, such as messages, from alteration.

- *Authentication protocols:* These are schemes based on the use of cryptographic algorithms designed to authenticate the identity of entities.

2.2 Theory about cryptography

Encryption and the decryption functions can be expressed likewise a mathematical functions. For example denoting the plaintext by M, for message, where it can be a stream of bits, a text file, a bitmap, a digital video image, whatever. M is the message to be encrypted and in this scenario it is simply binary data. The

ciphertext is denoted by C and it is binary data too: sometimes it has the same size as M, sometimes it is larger. The encryption function E operates on M in order to produce C, mathematically:

$$E(M) = C$$

In the reverse process, the decryption function D operates on C to produce M:

$$D(C) = M$$

Decryption D returns a message M, which has been encrypted before with E:

$$D(E(M)) = M$$

This is true because decryption D and encryption E are opposite functions.
A *cryptographic algorithm*, also known as cipher, is a logic method or procedure that express as a mathematical function used for encryption and/or decryption. It uses a *key* called K which assumes one of a large number of values. This range of possible key values is called keyspace. Both encryption and decryption operations use this key [33], [34]:

$$E_K(M) = C$$

$$D_K(C) = M$$

$$D_K(E_K(M)) = M$$

Algorithms that use different encryption and decryption keys do exist in this case:

$$E_{K1}(M) = C$$

$$D_{K2}(C) = M$$

$$D_{K2}(E_{K1}(M)) = M$$

2.2.1 Cipher types

A wide variety of algorithms compose the cipher panorama; they can be distinguished into two different key-based type:

- *Symmetric algorithms*;
- *Asymmetric algorithms* (public-key algorithms).

Symmetric algorithms

The encryption key, in a *symmetric algorithm*, can be computed from the decryption key and vice versa and in most cases they are exactly the same. These are also called single-key algorithms and require an agreement on a key, between sender and receiver, before they can communicate securely. It was the only encryption algorithm used before the development of public-key algorithm in 1970s. The security of this type of algorithms depends on the key secrecy. As long as no third parties know it, the informations are secured.

To better understand how all it works, a simple symmetric cipher model is described (Fig.2.1):

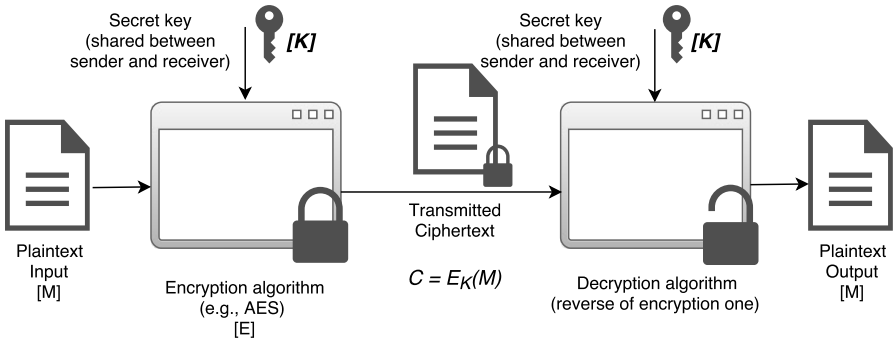


Figure 2.1. Simplified model of symmetric encryption

- **Plaintext:** is the original message or data in clear, ready to be used by the end user or as input for the cipher algorithm.
- **Encryption algorithm:** alters the plaintext contents performing various substitutions and transformations on it. The algorithm produces different outputs using different keys.
- **Secret key:** is the second input to the encryption algorithm; it's independent from both plaintext and encryption algorithm; it's used to obtain an univocal ciphertext, which can be retranslated into plaintext only using the same key.
- **Ciphertext:** is the scrambled message produced as encryption algorithm output. It depends on the plaintext and the secret key. For a given message, two different keys will produce two different ciphertexts. The ciphertext is an apparently random stream of data and it is unintelligible.

- **Decryption algorithm:** is the encryption algorithm reverse. It takes the ciphertext and the key and gives as output the original plaintext. This is true only if the right key is given (the one used for encryption).

Two requirements are mandatory to grant the information security:

1. An encryption algorithm is strong when an opponent, that knows it and has access to one or more ciphertexts, would be unable to decipher the ciphertext or figure out the key. Even if he or she is in possession of a number of ciphertexts together with the plaintext that produced each ciphertext.

2. Both sender and receiver must have obtained copies of the secret key in a secure way and must keep the key secure. If someone can discover the key and knows the algorithm used for the encryption, all communication using this key is readable.

Symmetric encryption is widely used due to its feasibility: it's assumed that is impractical to decrypt a message on the basis of the ciphertext plus knowledge of the encryption/decryption algorithm. In other words, the algorithm doesn't need to be kept secret, only the key must remain secret. This is a great advantage for manufacturers that can develop low cost chip implementations of data encryption algorithms. The principal security problem concerns maintaining the secrecy of the key. The essential elements that compose a symmetric encryption system are reported in the following scheme:

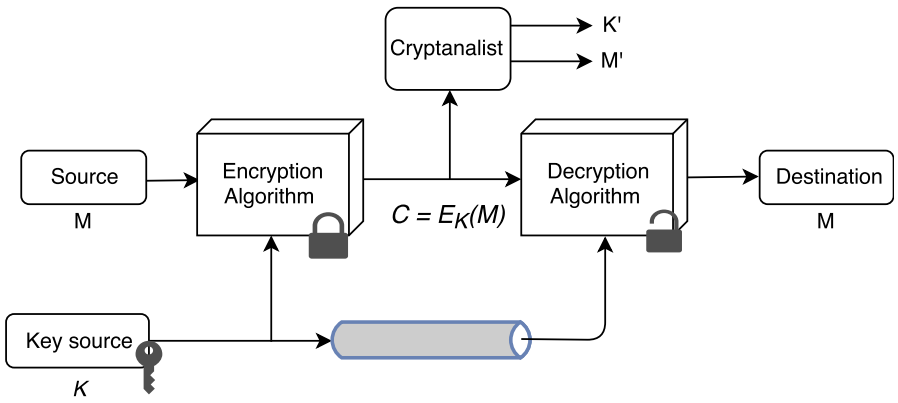


Figure 2.2. Simplified model of a symmetric cryptosystem

In the example (Fig.2.2), the key is generated at the message source, then it must also be provided to the destination by means of some secure channel. Another

solution is available: a third party could generate the key and securely deliver it to both source and destination.
An opponent (cryptanalist), which is assumed to know the encryption (E) and decryption (D) algorithms, observing the cyphered informations (Y) but not having access to the keys (K) or the plaintexts (X), may attempt to recover X or K or both. If the opponent is interested in only this particular message, then the focus of the effort is to recover X by generating a plaintext estimate Xn. However, often the opponent is interested in being able to read future messages as well. In that case an attempt is made to recover K by generating an estimate Kn.

This kind of algorithms can be split into two categories:

- *Stream ciphers:* operations executed on a single bit or byte at a time (Fig. 2.3);
- *Block ciphers:* operations executed on a group of bits (Fig. 2.4). Produce an output block for each input block. On modern computers, algorithms use typically block size of 64 bits.

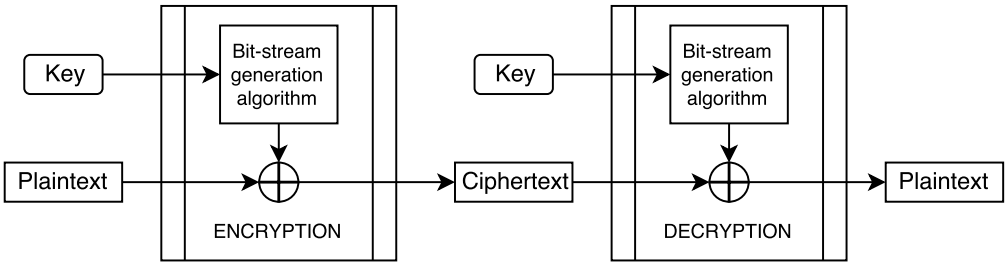


Figure 2.3. Stream cipher using algorithm bitstream generator [34]

Symmetric ciphers are also used for key management [33], [34].

Asymmetric algorithms

"For practical reasons, it is desirable to use different encryption and decryption keys in a crypto-system. Such asymmetric systems allow the encryption key to be made available to anyone while preserving confidence that only people who hold the decryption key can decipher the information" [36]. The development of public-key cryptography is the greatest and perhaps the only true revolution in the entire history of cryptography.
Asymmetric encryption is a form of cryptosystem in which encryption and decryption are performed using different but related keys, a public key and a private key.

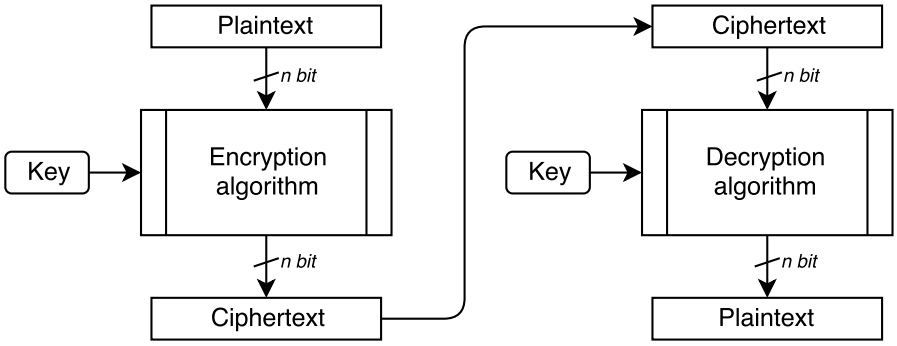


Figure 2.4. Block cipher [34]

The most widely used public-key cryptosystem is RSA. The difficulty of attacking RSA is based on the difficulty of finding the prime factors of a composite number. Public-key algorithms are based on mathematical functions rather than on substitution and permutation only. The use of two keys has a deep impact in the areas of confidentiality, key distribution, and authentication.
A common misconception is that public-key encryption is more secure from cryptanalysis than is symmetric encryption. In fact, the security of any encryption scheme depends on the length of the key and the computational work involved in breaking a cipher. There is nothing in principle about either symmetric or public-key encryption that makes one superior to another from the point of view of resisting cryptanalysis.
A second misconstruction is that public-key encryption is a general purpose technique that has made symmetric encryption outdated. On the contrary, because of the high computational demand of public-key encryption schemes, there is not consistent reasons that symmetric encryption will be abandoned.

There is a feeling that key distribution is trivial when using public-key encryption, compared to the rather cumbersome handshaking involved with key distribution centers for symmetric encryption. The procedures involved, the need for a central agent to use certain protocols are not manageable and clearly nor more efficient than those required for symmetric encryption. The public-key cryptography concept evolved from an attempt to attack two of the most difficult problems associated with symmetric encryption.

1. The first problem concerns the key distribution for symmetric encryption. As mentioned before, it requires either that two communicants already share a key, which somehow has been distributed to them, or the use of a key distribution center.

2. The second problem concerns the "digital signatures". Since the use of cryptography became widespread unto commercial and private purposes, then electronic messages and documents would need the equivalent of signatures used in paper documents.

The asymmetric algorithms have the following important characteristic.

- Giving only knowledge of the cryptographic algorithm and the encryption key must be computationally unfeasible to determine the decryption key.

Some algorithms, like RSA, also exhibit the following additional characteristic.

- Either of the two related keys can be used for encryption, with the other used for decryption.

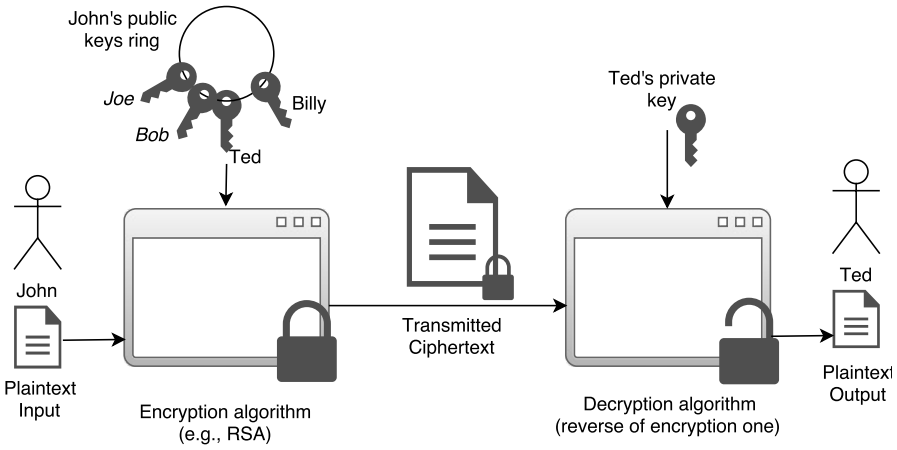


Figure 2.5. Simplified model of asymmetric encryption

These public-key algorithms (Fig.2.5) have the following important characteristic:

- **Plaintext:** is the original message or data in clear, ready to be used by end user or as input for the cipher algorithm.
- **Encryption algorithm:** alters the plaintext contents performing various transformations on it using the given public key.
- **Public and private keys:** are the pair of keys that have been selected; one is used for encryption and the other is used for decryption. The exact transformations performed by the algorithm depend on the public or private key that is provided as input.

- **Ciphertext:** is the scrambled message produced as encryption algorithm output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts. The ciphertext is an apparently random stream of data and it is unintelligible.
- **Decryption algorithm:** takes the ciphertext and the private key and gives as output the original plaintext.

The essential steps to create a public-key working system are the following:

1. Each user generates a pair of keys (public and private) to be used for the encryption and decryption of messages.
2. Each user places the public key in a public register or other accessible file. The companion key is kept private. Each user maintains a collection of public keys obtained from others users.
3. If A wants to send a confidential message to B, A encrypts the message using B's public key.
4. When B receives the message, he decrypts it using her private key. No other recipient can decrypt the message because only B knows his private key.

With this technique, all participants have access to public keys; private keys are generated locally by each participant and therefore are never distributed. As long as a user's private key remains protected and secret, the incoming communication is secure. If it's needed, the system can change its private key but has to publish the companion public key to replace its old public key [33], [34].

Key-based algorithms summary

Table 2.3 summarizes some of the important aspects of symmetric and public-key encryptions. To discriminate between them, the key used in symmetric encryption is called secret key, the others used for asymmetric algorithms are called private and public key.

2.2.2 Attack types

The main goal cryptography wants to achieve, is to keep a message secret avoiding it readability by third undesired parties. Eavesdroppers (also called adversaries, attackers, opponents) are assumed to have complete access to the communications channel between the sender and receiver. So they may intercept the encrypted data. The science based on message plaintext recovering without access to the key

Conventional Encryption	**Public-Key Encryption**
Needed to Work: 1. The same algorithm with the same key is used for encryption and decryption. 2. The sender and receiver must share the algorithm and the key.	*Needed to Work:* 1. One algorithm is used for encryption and a related algorithm for decryption with a pair of keys, one for encryption and one for decryption. 2. The sender and receiver must each have one of the matched pair of keys (not the same one).
Needed for Security: 1. The key must be kept secret. 2. It must be impossible or at least impractical to decipher a message if the key is kept secret. 3. Knowledge of the algorithm plus samples of ciphertext must be insufficient to determine the key.	*Needed for Security:* 1. One of the two keys must be kept secret. 2. It must be impossible or at least impractical to decipher a message if one of the keys is kept secret. 3. Knowledge of the algorithm plus one of the keys plus samples of ciphertext must be insufficient to determine the other key.

Table 2.3. Conventional and Public-Key Encryption comparison

is called *cryptanalysis*. An attempted cryptanalysis is called an *attack*. Successful cryptanalysis may recover the plaintext or the key, it depends mainly on the adversary's objective. It also may find weaknesses in a cryptosystem that eventually lead to the previous results. The key loss through noncryptanalytic means is called a compromise. Auguste Kerckhoffs, a Dutch linguist and cryptographer, professor of languages in Paris, in the 19th century, first enunciated a fundamental assumption in cryptanalysis: the secrecy of a cryptosystem must reside entirely in the key. Kerckhoffs assumes that the cryptanalyst has complete details of the cryptographic algorithm and implementation. Even that the encrypted message is stille secure. Actually attackers don't always have all these detailed information. If an algorithm can't break, even with knowledge of how it works, then they certainly won't be able to break it without that knowledge.

In literature terms *attack* and *threat* are commonly used to mean more or less the same thing. Below their definition taken from *Internet Security Glossary* [37]:

- *Threat:* a potential violation of security, which exists when there is a circumstance, capability, action, or event that could breach security and cause harm. It is a possible danger that might exploit a vulnerability.
- *Attack:* an assault on system security that derives from an intelligent threat. This act is aimed at evading security services and violating the security policy of a system.

Different ways to define and classify the attacks do exist. All of them are useful to better understand how an attack is performed and why. The previous distinction helps to distinguish among the attackers approach type.

Attacks can be classified in two categories (Fig.2.6):

- **Passive attack:** the eavesdropper's goal is to obtain the transmitted information. There exist two types of passive attacks:

- *Release of message contents:* prevent an opponent from learning sensitive or confidential informations, took from learning the contents of a transmitted file.
- *Traffic analysis:* masking a communication, using the encryption (for example), prevents an opponent to extract informations from the message. Using the encryption isn't enough to stop an adversary to observe the pattern of these messages. Location and identity of communicating hosts could be determined; frequency and length of messages exchanged could be observed. These informations all together might be useful in guessing the nature of the communication that is taking place.

This type of attacks don't involve any alteration of the data, for this reason is very difficult to detect them. Passive attacks are very difficult to detect, because they do not involve any alteration of the data. Usually neither the sender nor receiver is aware that a third party has read the messages or observed the traffic pattern. The solution to face these attacks is prevention. Use an encryption algorithm can be the right means against them.

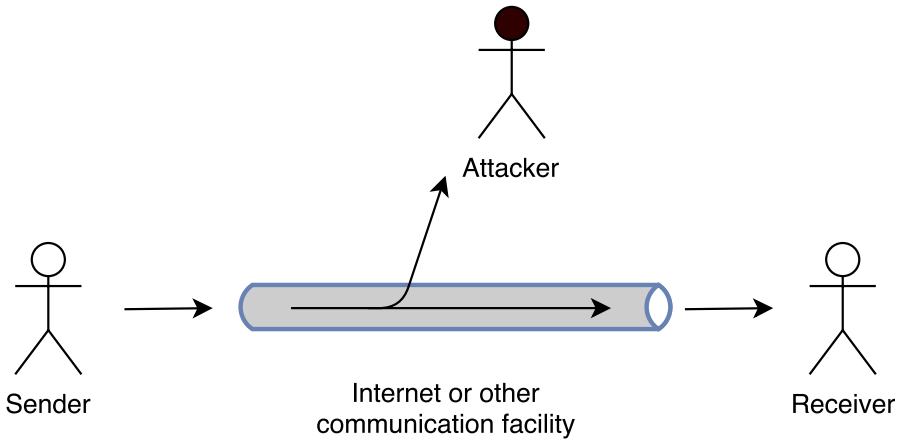


Figure 2.6. Passive attack simplified model

- **Active attack:** involves every type of message alteration for any purpose, performed by an adversary. They can be subdivided into four categories:
  - *Masquerade:* an entity pretends to be a different one (path 2 of Fig.2.7 is active). Usually this type of attack goes with other forms of active attacks.

- *Replay:* a transmitted data and its subsequent retransmission are passively captured in order to produce an unauthorized effect (paths 1, 2, and 3 of Fig.2.7 are active)
- *Modification of messages:* some portions of the original message is altered, or is delayed, or reordered, to produce an unauthorized effect (paths 1, and 2 of Fig.2.7 are active).
- *Denial of service:* normal use or management of communications facilities is prevented or inhibited (path 3 of Fig.2.7 is active). This type of attack may have a specific target; for example, an entity may suppress all messages directed to a particular destination (e.g., the security audit service). Another form of service denial is the disruption of an entire network, either by disabling the network or by overloading it with messages so as to degrade performance.

They exhibit the opposite characteristics of passive attacks. They can be easily detected but they're difficult to prevent completely because of the wide variety of potential physical, software, and network vulnerabilities. Instead, the idea is to detect active attacks and to recover from any disruption or delays caused by them. If the detection has a deterrent effect, it may also contribute to prevention.

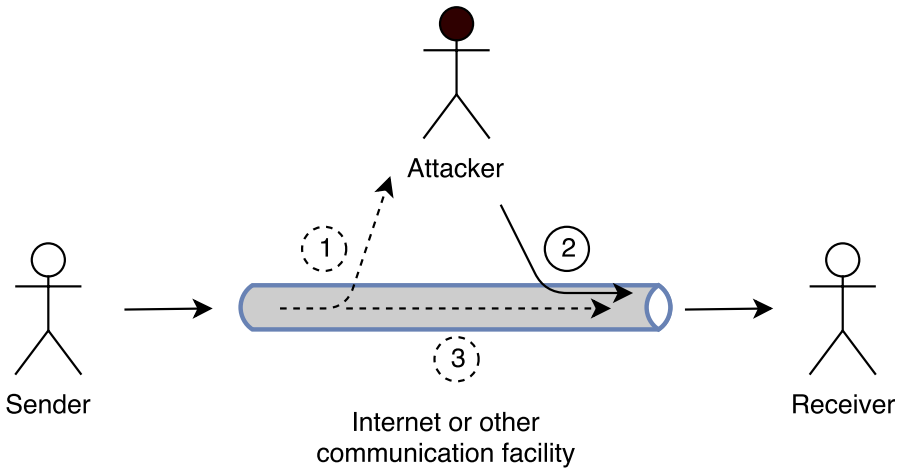


Figure 2.7. Active attack simplified model

The first possible solution to find out a password (or a passphrase) consists on asking it directly to the person who knows it and, if needed, using persuasive,

unconventional and maybe non legal approaching methods. Of course this isn't a suitable solution for our purpose [33], [34].
Brute-force attack is a direct type of cryptanalytic attack that consists in trying many different keys hoping to guess the right one. It can be used to, theoretically, find the key and decrypt encrypted data. Obviously it is not a good approach for long keys, where the mean time needed to guess the right key, computing every possible combinations, increases with its length.
In these cases, an efficiency solution, if applicable, is using a **side-channel attack**. Its concept is based on gain information from the physical system implementation, rather than from the algorithm weaknesses. It can be done, for example, having access to temporary memory (cache like; it requires technical knowledge of the system), measuring power consumptions or electromagnetic leaks, and so on. An example of side-channel attack is simple power analysis (SPA); it attempt to interpret the power consumption behavior of a device and deduce information about its performed operations (Fig. 2.8) [38], [39].

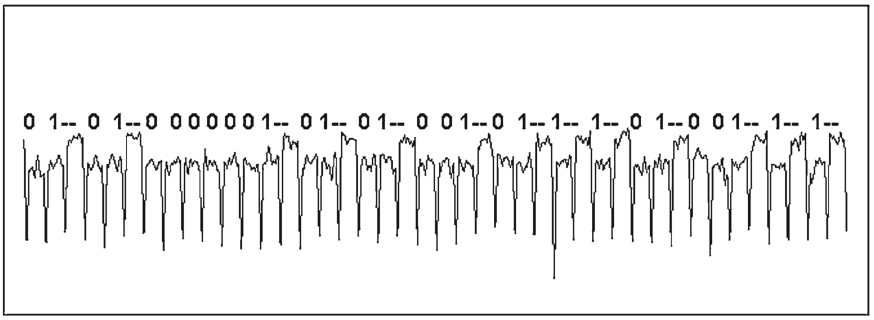


Figure 2.8. SPA leaks from an RSA implementation [39]

In the following four general types of cryptanalytic attacks are described, in order to account some of the basic techniques to get access to ciphered informations. Each of them assumes that the cryptanalyst has complete knowledge of the encryption algorithm used [33], [34]:

1. **Ciphertext-only attack:** the adversary has the ciphertext of several messages, encrypted with the same encryption algorithm. His goal is to obtain the plaintexts, or better to deduce the key.

2. **Known-plaintext attack:** the adversary has both ciphertext and plaintext of several messages. His goal is to deduce the key or an algorithm to decrypt any new message encrypted with the same key.

3. **Chosen-plaintext attack:** the adversary has both ciphertext and plaintext of several messages and chooses the plaintext that gets encrypted. It's clearly more powerful than *known-plaintext attack* because the attacker can choose the plaintext block to encrypt which can give him the more amount of information to deduce the key. Again his goal is to deduce the key or an algorithm to decrypt any new message encrypted with the same key.

4. **Adaptive-chosen-plaintext attack:** the adversary has both ciphertext and plaintext of several messages and chooses the plaintext that gets encrypted; moreover he can also change his choice based on the previous encryption results. This differs from the previous type, in which a single large block of plaintext can be chosen to be encrypted. In *adaptive-chosen-plaintext attack* a smaller plaintext block can be chosen to be encrypted, then another one, on the results of the first, can be encrypted.

2.3 Cryptographic hash function

Hash functions are a peculiar type of functions that, given an arbitrary size data in input (called pre-image), give a fixed size (generally smaller) data in output (called hash value). No matter how large the input is. An example of simple hash function would be a function that takes the input data and returns a byte consisting of the XOR of all the input bytes.
In this thesis only one-way hash functions are considered. They are functions in which f is known and doesn't require any secret information to operate. Given y, in the range of f, it is hard to find an x (is computationally speaking complicated to obtain a value for x, it may require a very high number of computational steps to be obtained, it is translated in terms of time) such that

$$y = f(x)$$

on the other hand, y needs to be easily computable [40]. One-way hash function is also known with many different names which describe clearly its role: compression function, message digest, fingerprint, cryptographic checksum, manipulation detection code. It covers a central position in modern cryptography and is an essential part for many protocols. Cryptographic hash functions are widely used: with public-key algorithms for both encryption and digital signatures, to check the integrity (any input bit or bits change, with high probability, cause a change to the hash code) and for the authentication. Because of its typical "many-to-one" mechanism, the two strings equality can't be determined with certainty, but a reasonable assurance of accuracy can be achieved. However, a good one-way hash function is also collision-free: it's hard to generate two pre-images with the

same hash value. An hash function can be used for both cipher and stream cipher. About the first, either the single block and the whole message can have an hash value.

A message authentication code (MAC) is a certain type of one-way hash function in which a secret key has been added. The hash value is a function of both the pre-image and the key. The theory mentioned above, about classic hash functions, is still valid, except that only someone with the key can verify the hash value.

The cryptographic hash functions are used in conjunction with symmetric ciphers for digital signatures. In addition, hash functions are used also for message authentication [33], [34].

Chapter 3

Related Works

3.1 Introduction

To grant the security of an information transmitted between two different entities, by mean of a potentially non secure communication channel, different mechanisms could be employed (Fig 3.1):

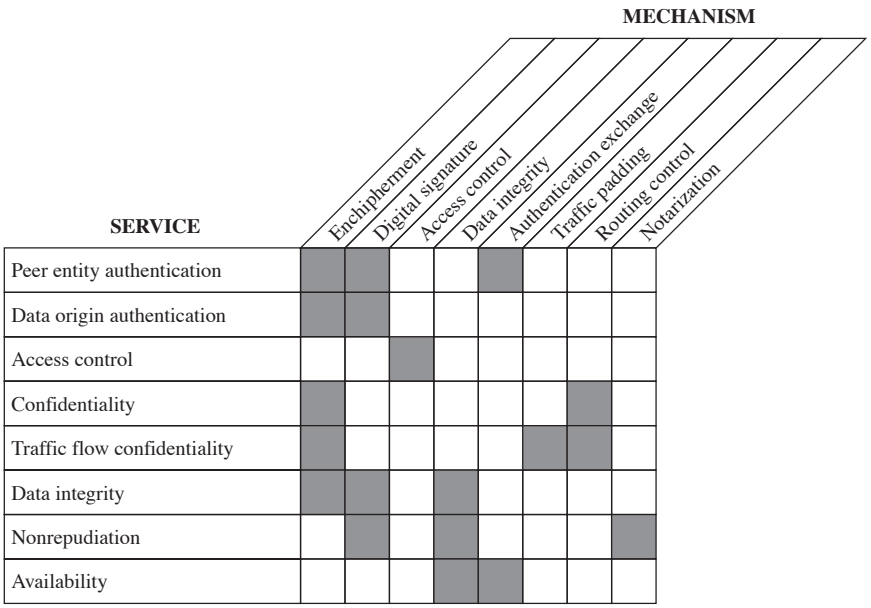


Figure 3.1. Relationship between security services and mechanisms [34].

The potential security issues about remote update of FPGAs, employed in

embedded systems, has already been addressed by previous related works. T. Wollinger et al. proposes in his article a state-of-the-art description of related risks and how to prevent them. It provides also a list of open research problems, explains the advantages of reconfigurable hardware for cryptographic applications and summarizes both public and symmetric-key algorithm implementations on FPGAs [41].

Nowadays, most FPGA vendors (as Microsemi, Lattice, Altera, Xilinx, etc) offer bitstream confidentiality through bitstream encryption facilities [27] - [30], [42], [43].
An Altera report analysis shows common security problems related to their specific type of FPGAs; these security risks (listed below) can be spread also to each other vendors FPGAs.

SRAM-based FPGAs store configuration file in an external volatile support memory; it introduces three security risks [28]:

- *Copying (cloning):* it consists in make an identical copy of the FPGA content without understanding how it works. These informations can be caught by either reading the memory content or intercepting them at power-up, when they're sent from memory to FPGA. This technique is a primary form of IP theft.

- *Reverse engineering:* the original design, in register transfer level (RTL) or in schematic form, is recreated analyzing the configuration file. This type of IP theft requires an high amount of time and resources and is reasonably more complex than copying. Sometimes more work than create a design from the scratch is required. But, on the other hand, the recreated design can be enhanced, gaining a competitive edge.

- *Tampering:* it's when the original design in the device has been modified or replaced with different one. Tampered device may cause system malfunction, security breach or steal sensitive data.

3.2 Bitstream confidentiality

The IP confidentiality is preserved encrypting the bitstream with a symmetric key shared between the FPGA and the system designer. The key is usually stored by the system designer on a volatile memory on the device. Obviously, regarding to the three risks listed above, this memory is designed to prevent physical attacks; not only the FPGA needs protection against attacks.
Bitstream encryption has proved to be an effective solution protecting designer's IP

against *clonation* or *reverse engineering* and *IP disclosure*. Encryption methodology, suitable for *dynamic partial reconfiguration* (DPR), requires user logic to decode encrypted partial bitstreams.

3.3 Bitstream integrity

Bitstream integrity is grant by vendors company whose systems need to be able to:

- Confirm that the configuration data stored in an FPGA device is correct;
- Alert the system to the occurrence of a configuration error.

Usually they employ cyclic redundancy checks (CRC) for error detection. It determines if data received have been corrupted during transmission. To accomplish this, before the transmission, a function is used to calculate a checksum value for the data and it attaches the checksum to the original data. At the receiver, the same computation is done in order to get as result a checksum value of the original data, which is compared to the attached one from the transmitter. If both checksum values are the same, then the received data frame is correct and no data corruption occurred during transmission or storage [44], [45].

But CRCs purpose is not to identify bitstream alteration in cryptographic terms, it has been developed in order to detect unwanted data modification caused by noisy or damaged transmission or storage media, this doesn't include changes by an intelligent third party like malicious attacker. Error detecting and correcting codes were invented to detect these errors. These codes calculate a value from the set of data and transmit or store it with data. Any hash function can accomplish the role of error detection; one of them is exactly the cyclic redundancy check (CRC) which isn't a cryptographically secure hash and therefore can't reliably detect malicious changes in transmitted data, but it can provably detect some common accidental errors like one or two bit or burst errors and can be implemented very efficiently. Even if it's coupled with encryption, it doesn't grant adequate security levels [46]. Therefore, some solutions based on cryptographic hashing primitives have been developed [47], [48], [49].

3.4 Bitstream authenticity

A good technique to assemble a message authentication code, for implementing a bitstream authentication mechanism, has been found in the dual-pass counter with the Cipher Block Chaining Message Authentication Code (CBC-MAC in CCM mode). Its main drawback is an increasing of the configuration process time due

to the separate authentication and encryption procedures [48]. Another step to improve the system security has been done by providing both authentication and confidentiality using two symmetric encryption cores running in parallel sharing resources for efficient implementation. Authentication provides cryptographic-level assurance of data integrity and its source. When the hash functions process incorporates a secret key, the outcome is called a Message Authentication Code (MAC) and allows the receiver to verify that the message is authentic: it has not been tampered with and the sender knows the key [49].

3.5 Further solutions

For latest FPGA functionalities as "*partial dynamic reconfiguration*" and "*self reconfiguration*" a solution that aims to improve security (for SRAM FPGAs in the case treated in [42]) through flexible bitstream encryption has been proposed.

Another related work concerns employing SHA-1 and AES algorithms (implemented as C program) for authentication and encryption phases. This approach shows that the total processing time, in both schemes, is not sufficient for practical *dynamic partial reconfiguration* [50].

The same goal (grant both confidentiality and authenticity) has been achieved by encrypting a partial bitstreams with AES-GCM cipher [51].

3.6 Available solutions drawbacks

Other security issues can't be treated if the FPGA design itself isn't secure. Using an unsecured device embedded in a security system is not security-efficient [42].

As can be seen in section 3.2 the confidentiality of a DPR FPGA is grant by encrypting its bitstreams; however, this requires user logic and computational power.
For the integrity some solutions based on cryptographic hashing algorithm have been used even if the CRC technique is not really suitable for this purpose; the CRC requires some adjustments in order to recognize voluntary data alteration.
For the authenticity, more time is required to the FPGA for the configuration process.

The main drawback of the above mentioned solutions is that they concern a single secret shared between the system designer and the target device. Only one entity can provide updated bitstreams, this entity is the system designer. But in this thesis, the proposed scenario needs to let generate and distribute dedicated bitstreams by software providers. The confidentiality and integrity of these bitstreams must be preserved.

Chapter 4

System architecture and attack model

4.1 Assumptions and models

The scenario assumed in this thesis is very close to the one used to describe the mobile application deployment. In this proposed panorama, three main participants are involved: the **software provider** (SP), the **hardware vendor** (HWV) and the **end user**.
End user buys the embedded system from the *hardware vendor* on which he (or she) wants to run an application made by the *software provider*. A *software provider* develops applications and sells them directly or through one or more application stores. The purpose of the application stores is to reach the higher number of possible costumers, who may have different platforms with different OS and can access different application stores. This happens because application stores are more handy than a specific *software provider*, thanks to their accessibility (for example by web servers), they're widely known and commonly used by *end users*. Stores are linked to payment gateways that allow different type of payments (e.g., credit cards, PayPal) to the costumers. In this way the *end users* may pay for the software they want to purchase.
The applications considered in this thesis are made of two parts: the software executable code, and an FPGA bitstream file. The first one represents the classic software used for current applications. The second one is an additional file used to describe one or more IP cores required to improve the application performances.

A simple example could be a video player application which let *end users* to watch video encoded in the HEVC (High Efficiency Video Coding or H.265) standards format. This is one of the latest video compression standard which, compared to the previous one (H.264), at the same video quality, gives twice as much

the data compression. This gain is payed in terms of power computing required for data decompression. In this case, an hardware accelerator, which is made specifically to execute HEVC decompression algorithm, is mandatory to avoid high CPU usage. For this reason, an FPGA implementation could be a clever solution (as said in chapter 1.2). However, it requires an additional file containing the algorithm to be implemented aboard itself. This is the bitstream [52]. Every time the application is executed the second file (bitstream) is dynamically reconfigured in the FPGA.

In this thesis, the *hardware execution platforms* (HEPs) are systems on chip (SoCs) composed of microprocessors and FPGAs featuring partial reconfiguration and basic bitstream encryption mechanism. The *hardware vendor* designs and sells the *hardware execution platform* on which *software providers* application will run. It has all the knowledge about its products, this concerns security mechanism like cryptographic keys. Another assumption is that *hardware vendor* offers services by a web server (for example) to *software providers* and *end users*, like hardware product authenticity, firmware updates, etc.

4.2 Security requirements

Software providers goal is to preserve the authenticity, the integrity and the intellectual property of bitstreams deployed with its software (and of the SW itself, obviously, but this is out of the scope of this thesis. Literature is already full of available methodologies to approach this problem).

Preserve authenticity and intellectual property requires to satisfy the following security requirements:

- **Bitstream confidentiality:** only *software provider* must be able to read in cleartext a bitstream. The *end user* must notbe trusted because he may have malicious intentions. Also other *software providers* aren't trusted and the application stores too. The only entity that may be reliable is the *hardware vendor* due to legal binding contracts or Non-Disclosure Agreements (NDAs).

- **Bitstream authentication:** to use an FPGA bitstream, the *end users* must have bought an original copy of the software. Use a bitstream file means that the *hardware execution platform* can configure the onboard FPGA to support the related software execution. Nevertheless the *end user* in no case must be able to access bitstream in cleartext.

- **Bitstream integrity:** *software provider* in no cases wants that corrupted bitstreams are delivered to *end users*, or used to configure the FPGA (producing further security risks)

If only one of these requirements mentioned above isn't met, the *software provider* looses its aim: an attacker may be able to read the bitstream, start reverse engineering or compromise its authenticity and integrity.

4.3 Attack model

For the purpose of this thesis, direct attacks to the physical system (*hardware execution platform*) act to damage it, or make it inoperative (service denials for example), are not considered. However, it is assumed that the reconfigurable hardware, and other blocks associated to it, are resistant to physical attacks even if *hardware execution platform* is situated in an hostile environment.

Two type of attacks are considered:

- **Intellectual Property attacks:**
  - *Adversary's goal:* violate the confidentiality of the bitstream, to make illegal copies of the hardware.
  - *How it works:* gain access to the cleartext version of the bitstream. There are two ways to implement it: over the network links used to deploy the applications; on the *hardware execution platform* tampering external memory devices or interconnections.
- **Integrity attacks:**
  - *Adversary's goal:* introduce malfunctions in the execution of the related software application or completely replace it with a previous version (downgrade) to avoid, for example, security updates.
  - *How it works:* integrity corruption of the bitstream, modifying its contents or completely replacing it.

4.4 Adversary model

The *attack model* above identify two type of adversaries:

- **Remote adversary:** its attack aims at the network links, which connect the application store to the *hardware execution platform* (Fig. 2.6 and 2.7.
- **Local adversary:** is a malicious *end user* (usually might be competing firms) which has physical access to the *hardware execution platform.*

Both of them are well known as attackers by the *software providers*. They have to be considered technically skilled, with considerable knowledge of the system and the device they are attacking and with an high amount of resources, which are also limited (for example in terms of time).

Chapter 5

Protocol and secure bitstream exchange

5.1 Introduction

This chapter presents the solution to describe the secure reconfigurable computing scenario used. More precisely, it analyzes the data exchange between the entities (listed below) and the required protocols used to identify the hardware structures.

Both scenarios show 5 entities:

1. *Software provider (W):* it produces the application, made of the software executable code and the FPGA bitstream file associated to it.

2. *Hardware provider (H):* it is the *hardware execution platform* manufacturer. On its hardware execution platform the application, provided by the software provider, will be executed. It is the only entity that has all the knowledge about the hardware and knows the secret keys of the FPGAs.

3. *Store (S):* it is the means by which the application can be sold and deployed.

4. *Client host (C):* it is the *hardware execution platform* itself. It is identified, in this thesis, with the *end user*, more precisely it represents the hardware system the user has to interact with.

5. *Configuration manager:* it is the set of instructions that are used to program the FPGA. The FPGA is the physical device on which the bitstream part of the application is configured. It is surrounded by other hardware devices it may need (e.g., external SRAM).

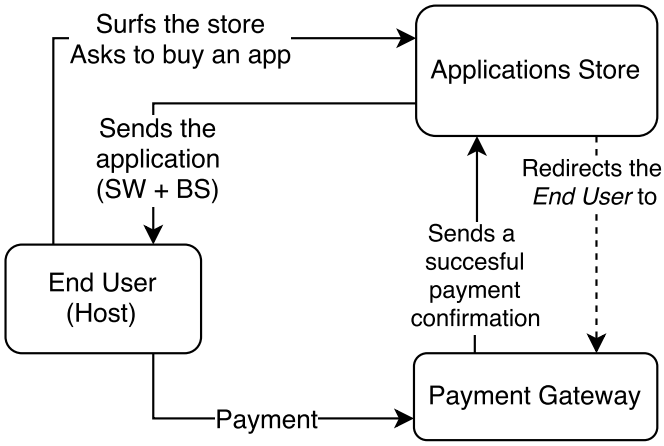


Figure 5.1. End user's point of view of the application store structure

The end user doesn't need to know all che complex structure that exists behind the application market. During an application purchase he comes directly in touch with just two entities: the application store and the payment gateway (Fig 5.1).

The entities listed before are attached to the following assumptions:

- the *end user* has an account on the *store*;
- the *end user* has access to different payment systems (e.g., credit card, PayPal);
- the *store* is linked to one or more payment systems;
- the *FPGA* is uniquely identified by an identification number called id_{FPGA};
- the *FPGA* and the *hardware vendor* share the *FPGA key* secret;
- *end user*, *hardware vendor*, *store* and *software provider* are able to create secure channels to share informations;
- the *software provider* knows the *hardware vendor*, or has an account on its servers.

Two different scenarios have been proposed to face security threats. A simpler one (*simple scenario*), which satisfies minimum security requirements but it needs also minimum hardware resources and complexity. A more complex one (*complex scenario*), which fulfills the entire set of security requirements but needs trusted computing hardware.

5.2 Simple scenario

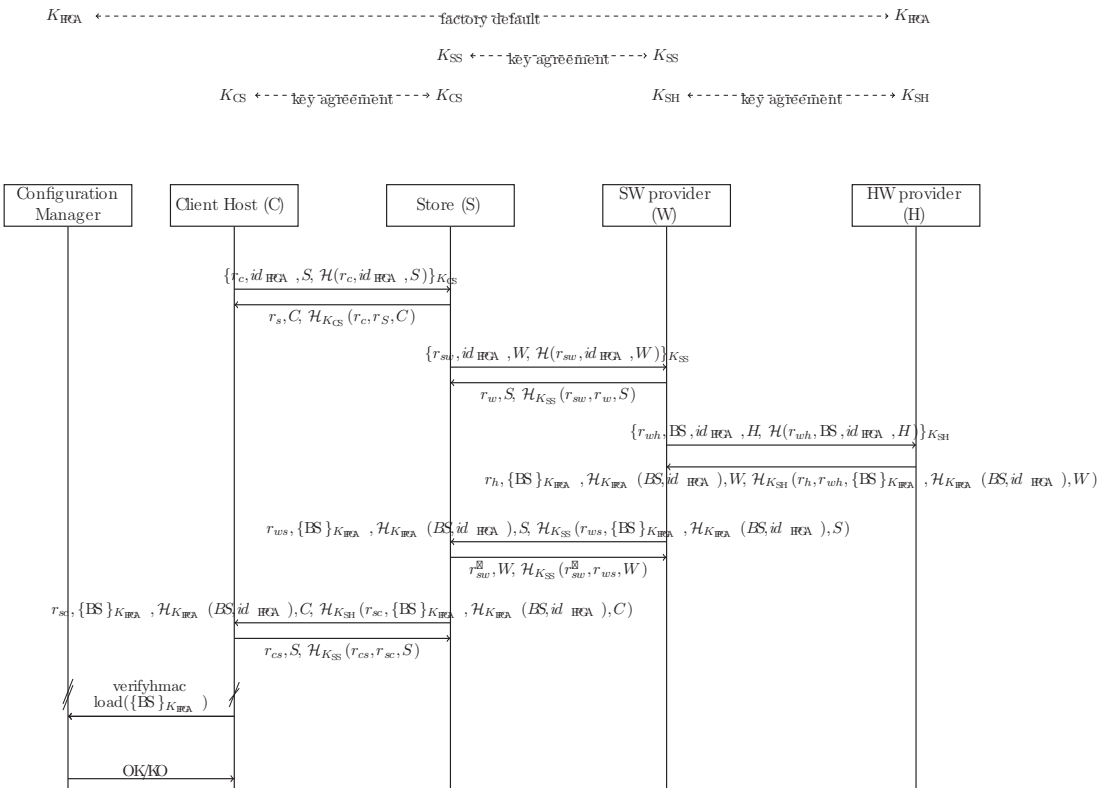


Figure 5.2. Simplified model of a symmetric cryptosystem

- id_{FPGA}: identification value to identify a physical FPGA device;
- $C,\ S,\ W,\ H$: entity receiver identifier;
- $r_{...}$ and $r_{... ...}$: random and secret number for challenge-response identification (strong authentication)[1][53];
- $H(...)$: hash value;
- BS: bitstream file;
- $\{...\}_{K_{...}}$: encrypted information with a specific key;

In this scenario (Fig. 5.2) the following procedural steps are considered:

1. The user decides to buy (download) an application, developed by the software provider, from the store. This application is made of software executable and bitstream file which will run on the FPGA. Store redirects the user to the payment gateway based on the payment typology chosen. When the payment procedure is successfully completed, the payment gateway notifies the store. The store initializes the bitstream sending procedure, asking it to the software provider. To do that the id_{FPGA} is mandatory to identify the FPGA authenticity. Obviously to share this kind of information (e.g., credit card number, id_{FPGA}) a secure channel which uses an agreed symmetric key K_C is needed. Confidentiality, data integrity and authentication must be ensured.

2. The store notifies the software provider about the user purchase; the FPGA informations (id_{FPGA}) are forwarded to the SP via a secure channel using an agreed symmetric key K_S.

3. The software provider sends, by a secure channel, the bitstream and the id_{FPGA} to the hardware vendor. The symmetric key used for this communication is called K.

4. The hardware vendor encrypts the bitstream using the FPGA key (related to the id_{FPGA}) and sends back the ciphered bitstream $\{BS\}_{K_{FPGA}}$ to the software provider. For this communication the same channel established before using the key K can be used.

5. The software provider sends back the ciphered bitstream $\{BS\}_{K_{FPGA}}$ to the store. Also in this step the channel has already been established before so this one is used for the communication, it uses the key K_S.

6. Both the ciphered bitstream $\{BS\}_{K_{FPGA}}$ and the software executable are available and the end user can download them from the store.

An enhancement hypothesis concerns the chance of the hardware vendor to locally store the bitstream to avoid re-sending of it from the software provider to reduce communication costs.

[1]The claiming entity proves to the verifying entity its identity by demonstrating knowledge of a secret associated with the verifying entity (this secret must not be revealed during the protocol). This is accomplished providing a response, which depends on both the entity's secret and the challenge, to a time-variant challenge. Time-variant parameters can be used in identification protocols to counteract some type of attacks (e.g., replay attack) [53].

As happens with firmware, operating systems and, more in general, with software, the bitstream might need to be updated, due to bug fixing or new functionality additions. So, if the software provider develops a new bitstream version, it has to notify the involved infrastructures in order to send to the end user the new bitstream file version.

1. The application connects to the software provider to look for updates.
2. If a new version of the application is available the host sends its id_{FPGA} to the software provider that checks if it corresponds to a valid customer.
3. Software provider and hardware vendor establish a secure channel and the first sends the updated version of the bitstream file and the id_{FPGA} to the second one. For the communication the symmetric key K is used.
4. The hardware vendor encrypts the bitstream using the FPGA key (related to the id_{FPGA}) and gives the ciphered bitstream back to the software provider. The secure channel is the one created in the step before (3).
5. The host downloads the updated application which includes the ciphered (updated) bitstream $\{BS\}_{K_{FPGA}}$.

This presented scenario requires that software provider trusts the hardware vendor, this is true thanks to legal contracts. Hardware privacy can't be achieved because the hardware vendor knows which software is bought by an user which has a specific id_{FPGA}.

5.3 Full scenario

A possible solution based on FPGA which includes functionalities like a trusted platform module, is described in the following scenario. It is based on the *direct anonymous attestation* (DAA) protocol, it grants the authentication preserving the host privacy.

In this case the idea is that the FPGA, by the DAA protocol, is recognized from the hardware vendor as genuine and obtains a valid DAA credential. This time, the request is made by means of the certificate ($AIK-cert$) created in this way instead of using the id_{FPGA} which gives informations about the host. After the purchase of an application by the user (passing through a payment gateway), the software provider checks the FPGA genuineness verifying, with the hardware vendor, the certification using the DAA verify protocol. If it's not marked as "rogue" the SP proceeds sending the bitstream file encrypted by an encryption key called K_S, used to secure the channel, and the key K_S itself using the ($AIK - PUB$) extracted

from the $(AIK - cert)$ and gives them back to the store which let them available for the download by the user.

Chapter 6

Platform Implementation

6.1 Introduction

In the supposed scenario (chapter 5.2, Fig. 5.2), an end user buys an application found on an application store. After the buying process ends, the application is downloaded from the store to the end user device (commonly an external memory device like an SD card). This application is made up of two different parts: a software program and a FPGA bitstream file. When the user launches the installed application, the software part will run on the embedded microprocessor, while the bitstream is used to program the FPGA device. Then, every time the software running on the CPU needs to execute some complex operations, which have been designed to be executed by an hardware accelerator, it relies on the programmed FPGA. The execution of these operations are, in this way, speeded up; during this period of time, the CPU is available to execute other operations until the FPGA finishes its duty.

The bitstream considered in this thesis is made up of two files that contain respectively the *algorithm* and the *data* used to program the FPGA. These are called VME files and they're compressed binary files created by the deployment tool starting from an XCF file[1] (Fig. 6.1). They allow a system containing an FPGA device to program it via JTAG by using the embedded microprocessor. The CPU, according to the VME files interpretation, manipulates the JTAG signals of connected target device in order to program it [54].

This thesis project main objective is to secure the bitstream granting its confidentiality during the FPGA programming operation and verifying its authenticity

[1] "An XCF file is a configuration file used by Diamond Programmer and for programming devices in a JTAG daisy chain. The XCF file contains information about each device, the data files targeted, and the operations to be performed." [54]

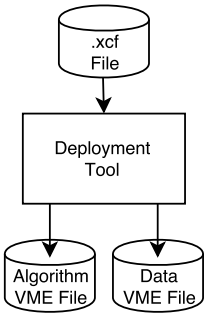


Figure 6.1. VME file generation flow [54]

and integrity. It means that the ciphered bitstream files, after being stored into the SD card, must be securely managed by the software to program the target device, and their plaintext version mustn't be in any case, even if in portion, stored into the external memory that's considered non secure. At the same time an hashing function checks both integrity and authenticity of the informations.

Code in C has been written in order to program the FPGA in a secure way, reading the ciphered bitstream files from the SD memory card. As can be seen in the following sections, the embedded system firmware[2] has been updated (employing a STMicroelectronics tool) and their low-level commands have been used to create other high-level functions (to manage, decrypt, check and elaborate data).

To complete the case analysis, of the securing FPGA bitstream management, the following assumptions have been done:

1. the FPGA and all the other blocks related to it are considered resistant to physical attacks (the end user can't physically gain access to the FPGA or its interconnection links);

2. the bitstream is made up by two files *algorithm* and *data*; these are generated from an *hardware description language* (HDL) like VHDL writes by an hardware designer.

3. the FPGA is linked to a microcontroller which let the partial reconfiguration of it;

4. the ciphered bitstream is downloaded and stored into an external memory (an SD card);

[2]The firmware is a set of data and low-level commands that allow the device to perform specific tasks. Firmware also provide a hardware-independent environment to realize more complex software [55].

5. the bitstream is quite big, so it can't be entirely decrypted and stored aboard an internal memory (RAM or Flash);

6. the bitstream is encrypted in blocks;

7. the encryption algorithm used is the advanced encryption standard (AES) [56] employed in *cipher block chaining* (CBC) mode with 16 bytes *key* length and 16 bytes *initialization vector* (IV) length:

8. the *secure hash algorithm* (SHA) [57] used to analyze the bitstream, is SHA-256 [58]. It returns a digest value of 256 bit (32 Byte) length;

9. to grant the bitstream security its plaintext version must never be saved into the external memory (SD card), even if it's just a block or a portion of it.

6.2 Hardware architecture

The hardware system must be resistant against physical attacks. This means that an adversary (or whoever may have direct access to the device) can't easily gain access to the data stored into the FPGA and its support memory (e.g., the SRAM for SRAM-based FPGAs), and to the interconnection links, which connect the FPGA to other devices too.

To achieve these requirements a unique embedded SoC, which integrates aboard microprocessor, memories and FPGA, could be a good solution (other information about the connection between the FPGA and the microcontroller can be found in appendix A). For this reason the SEcube^{TM} development kit board [59] has been chosen (Fig. 6.2). At the same time this chip grants a good level of security against adversaries attacks [60].

6.2.1 SEcube^{TM} development kit board

The SEcube^{TM} (Secure Environment cube) Open Security Platform is an open source security-oriented hardware and software platform. The board hardware has been designed by Blu5 Group [61]. Whereas the software libraries have been developed among five international research institutions[3].

[3] *Blu5 Labs Ltd*, Blu5 Group, Ta Xbiex, Malta - Reference: Antonio VARRIALE
CINI Cyber Security National Lab, Torino, Italy - Reference: Paolo PRINETTO
Lero, The Irish Software Research Centre, University of Limerick, Limerick, Ireland - Reference: Tiziana MARGARIA

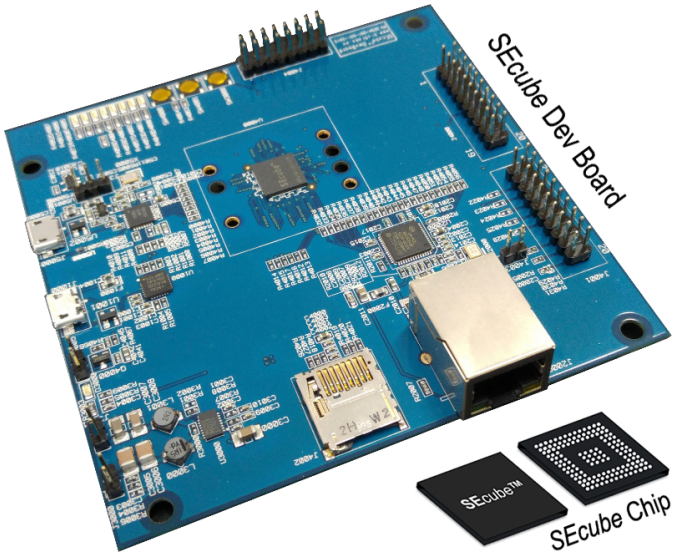


Figure 6.2. SEcube^{TM} development kit board and BGA chip [59]

The software libraries allow non expert developers to produce software using provided security functions, they can experience the SEcube^{TM} platform as a high-security black box. On the other hand, experts in cyber security can enjoy the openness to verify, change or rewrite the pre-existing software code at any system level.

The SEcube^{TM} device family consists of:

- the integrated circuit, *SEcube^{TM} Chip* or *SEcube^{TM}* (Fig. 6.2);
- the development board, *SEcube^{TM} DevKit* (Fig. 6.2);
- the USB stick, *USEcube^{TM} Stick (Fig. 6.3).*

The *USEcube^{TM} Stick* (Fig.s 6.4, 6.5) has been designed to be compatible with any Operating System and the *SEcube^{TM}* functionalities are accessible to applications and services without installing any driver. The SD card memory slot aboard *USEcube^{TM} Stick* let the end user to decide its capability and speed. The previous developed firmware is injected by mean of an embedded secure bootloader because the *USEcube^{TM} Stick* isn't provided of JTAG interface has *SEcube^{TM} DevKit* does.

LIRMM, CNRS, Montpellier, France - Reference: Giorgio DI NATALE
TU Dortmund, Dortmund, Germany - Reference: Bernard STEFFEN

Figure 6.3. *USEcube*^{TM} Stick final commercial product [59]

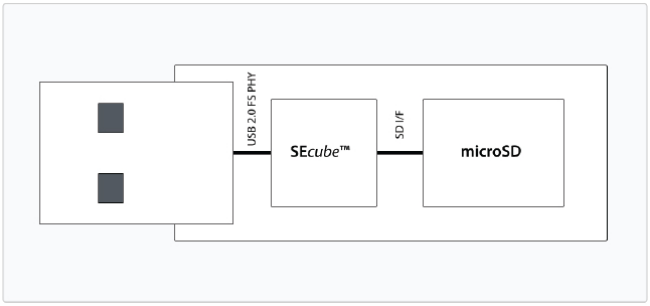


Figure 6.4. *USEcube*^{TM} Stick internal blocks scheme [62]

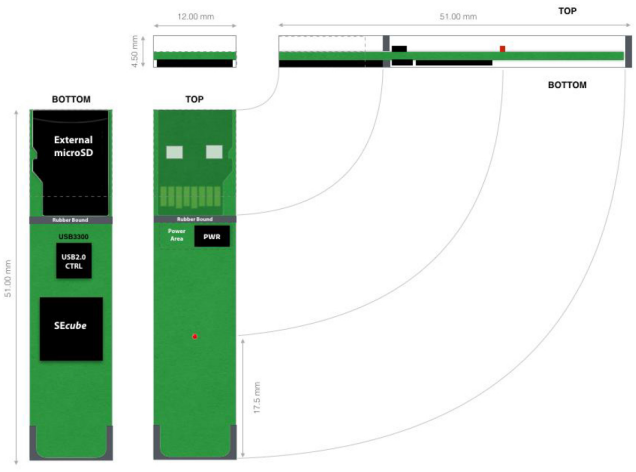


Figure 6.5. *USEcube*^{TM} Stick dimensions and internal physical structure [59]

To develop the software for SEcube^{TM}, the *DevKit* has been employed; it has been linked to a PC by mean of an USB cable; a SD external card memory has been plugged into the board; to program the SoC, the JTAG interface has been used by mean of the *St-Link/v2* (Fig. 6.6), which has been connected to both the *DevKit* board and the PC.

Figure 6.6. *St-Link/v2* kit: device and cables [59]

Focusing on the hardware part of SEcube^{TM}, it is composed of three security embedded elements, which give a versatile security environment in a single chip (Fig. 6.7):

- STM32F4 MCU based on ARM 32-bit Cortex®-M4 processing unit;
- MachXO2-7000 FPGA;
- SLJ52G certified security controller (Smart Card);

Suitable for any high-end design solution, it delivers integration of a flexible, configurable and certified secure element [63].

In this thesis the *Smart Card* above mentioned is not used, for this reason it will be no longer treated in the follow. The main information about the system devices have been summarized:

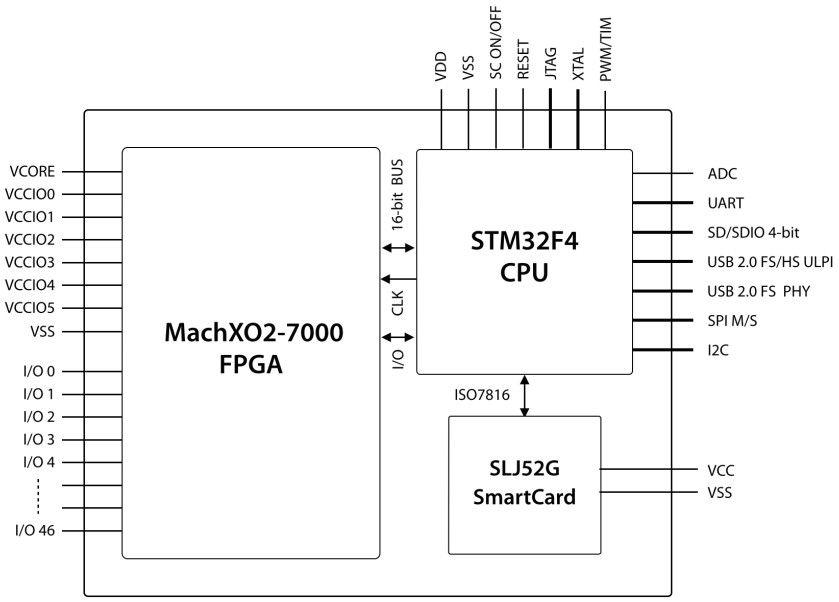


Figure 6.7. *SEcube*^{TM} internal blocks scheme [62]

CPU hardware features

- *Core*: ARM 32-bit Cortex®-M4 CPU with FPU, Adaptive real-time accelerator, frequency up to 180 MHz, MPU and DSP instructions;
- *Memories*:
  - *Flash memory*: 2 MB organized into two banks allowing read-while-write;
  - *SRAM*: 256+4 KB, including 64-KB of CCM (core coupled memory) data RAM;
- *Clock*: 4-to-26 MHz crystal oscillator, internal 16 MHz factory-trimmed RC, internal 32 kHz RC with calibration;
- *Other specs* are: Low power: Sleep, Stop and Standby modes, Timers: up to 17; SPI: Master/Slave configurable; USART; I2C interface; SD/SDIO interface: up to 48MHz, 1bit-4bit modes supported; True random number generator; CRC calculation unit; RTC; USB Connectivity; Connections to SmartCard; Connections to FPGA.

The CPU provides also a standard JTAG interface useful for programming and debugging. This interface can be permanently disabled once the development cycle is over, protecting the device from physical hardware lock.

FPGA hardware features

- 6864 LUTs and 47 I/Os, which may be used as a 32-bit external bus able to transfer data at 3.2 Gib/s;
- Embedded and distributed memory:
  - 240 Kbits SysMEM^{TM} embedded blocks RAM;
  - 54 Kbits distributed RAM;
  - Dedicated FIFO control logic;
- 256 Kbits On-Chip User Flash Memory;
- *Other specs* are: Flexible I/O Buffers; Wide Frequency range (10 MHz to 400 MHz); Non-Volatile infinitely reconfigurable; In-field logic configuration while system operates; Ultra low power device.

The FPGA JTAG is connected only to the embedded CPU, which manage both debug and programming operations (Fig. A.1).

6.3 Development flow

6.3.1 FPGA

FPGA programming code analysis

The SEcube^{TM} software is already equipped with libraries, which contain functions useful to program the FPGA, starting from a bitstream file.

The API (application programming interface) function given to the software developer is called *B5_FPGA_Programming*; recall this function let starts the FPGA programming process.

In this first part of the project, the FPGA programming algorithm has been analyzed: the bitstream is given in *"TEST_FPGA.h"* file in which algorithm and data parts are split into two arrays, *__fpga_alg[g_iAlgoSize]* and *__fpga_data[g_iDataSize]*, where *g_iAlgoSize* and *g_iDataSize* are the array lengths given from the bitstream file dimensions.

B5_FPGA_Programming function (Fig. 6.8) does something more than just programming the FPGA, it checks the VME version, starts and stops the hardware for the data communication with the FPGA.
The VME version is verified reading the first 8 bytes of the *__fpga_alg[]* array[4].
The FPGA programming part is executed by *ispProcessVME* function. This is based on the *switch-case* statement and it remembers a FSM (finite state machine) structure (Appendix C.1, Fig. C.1). The sequence of states of this "FSM" is provided by the *algorithm bitstream array*, whereas the other array, the *data bitstream array*, contains the informations used to program the FPGA.

The *ispProcessVME* function gets the data from the arrays by mean of another function called *GetByte*.

GetByte function (Fig. 6.9) checks which of the two bitstream arrays it needs to access and then returns exactly a single byte (*uint8_t*). Obviously this function needs some parameters to work correctly. They are: vector *index* (useful to get the righteous byte from the bitstream array) and bitstream type (mandatory to distinguish between algorithm ora data arrays).

Testing the proposed bitstream

After the algorithms analysis, some modifications have been introduced to the project: the FPGA files have been added, and the function *B5_FPGA_Programming* has been recalled directly from the *main* of the project.
A first test has been launched on the board in order to verify the properly functioning of both, the software and the FPGA.
The result has been achieved, as expected reading the VHDL source code: the 8 LEDs provided on the *SEcube*^{TM} DevKit have been turned on in sequence, one after another (Fig. 6.10).

6.3.2 SD card file system

The SD card has been employed to achieved one of the previous assumptions (in chapter 6.1, the 4th element of the list): to download the bitstream files, *"algo.vme"* and *"data.vme"*, on the device and to store them somewhere. Hence the need to introduce a file system aboard the *SEcube*^{TM} software.

A file system contains all the criteria used to manage and organize data stored in a storage device. It is essential, for example, to distinguish where a file ends and another one begins, but also to define filenames and directories specs, to

[4]Afterwards in the thesis, the buffer arrays content will be filled reading the data from two files (*"algo.vme"*, *"data.vme"*); the VME version is stored at the top 8 bytes of *"algo.vme"* file.

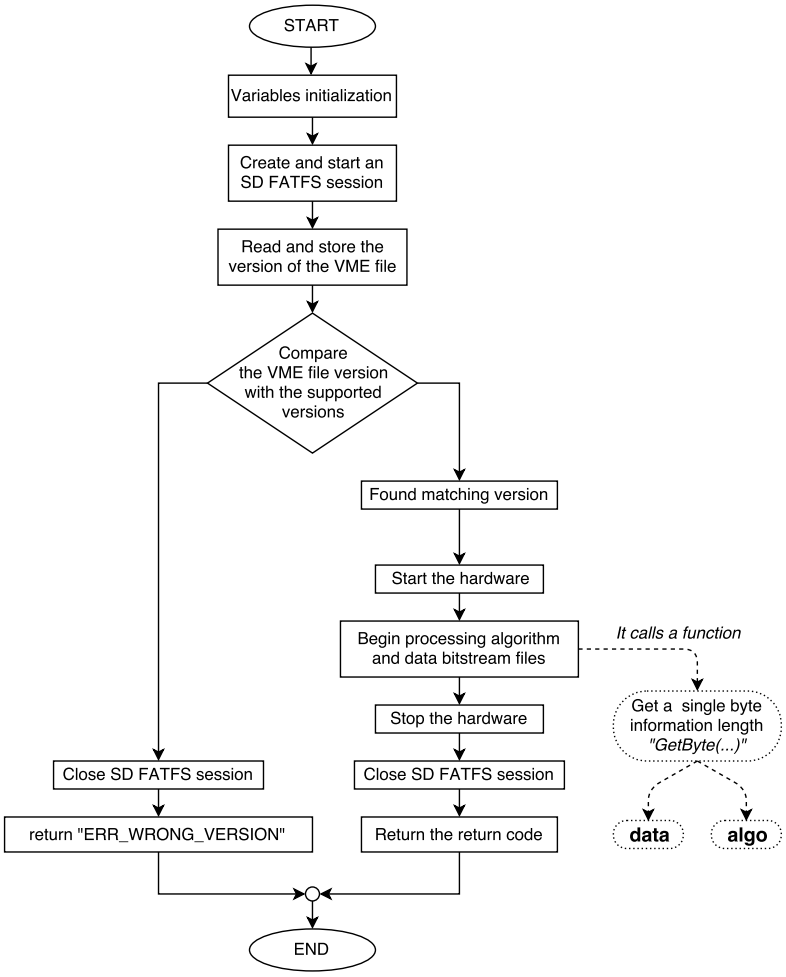


Figure 6.8. *"B5_FPGA_Programming"* function block diagram

manage the space, to restrict the access, to maintain the integrity and much more. Nowadays a wide variety of FSs (file systems) type exist, they have been developed in order to grant different advantages, for example, according to the hardware device type [64].

FAT file system

Among the different types of FS, the one used in the project to manage the SD card memory is *FatFs* (File Allocation Table File System) by elm-chan [65]. It is a generic FAT file system module for embedded systems. It is written in compliance with ANSI C (C89) and completely separated from the disk I/O layer, making it

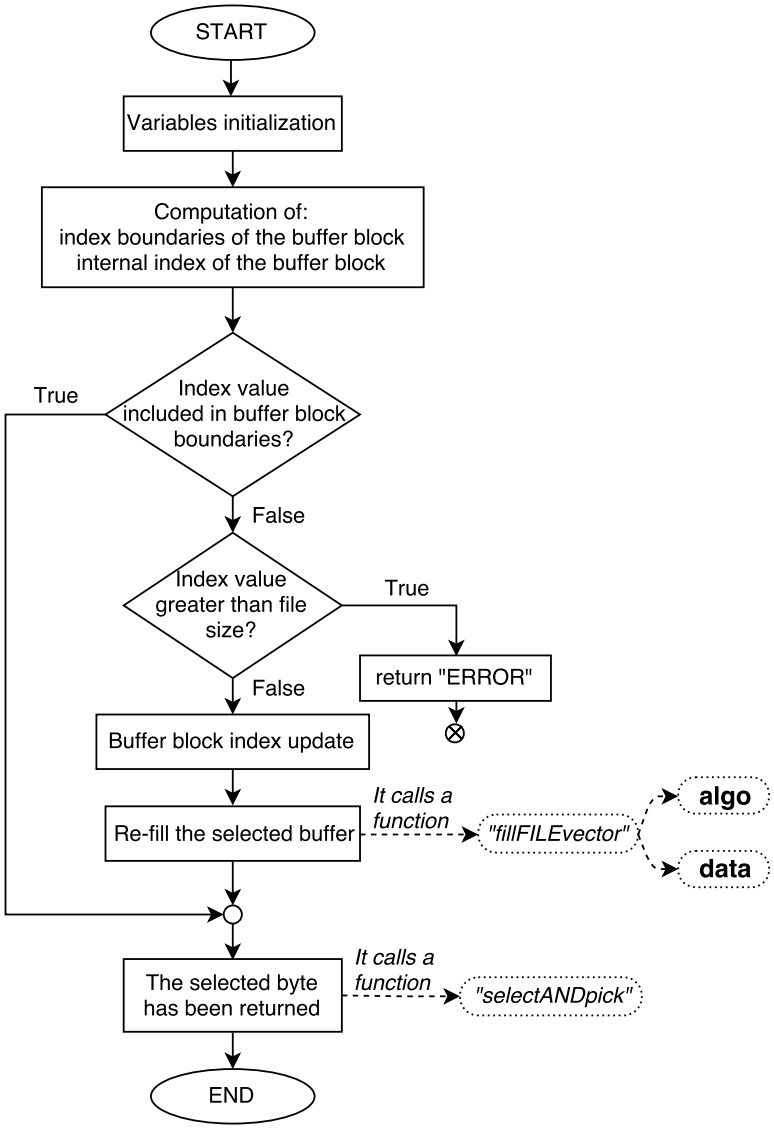


Figure 6.9. *"GetByte"* function block diagram

independent from the platform (Fig. 6.11). It is compatible with the Windows FAT version and gives various configuration options to personalize it.

Create STM32F4 firmware using STM32CubeMX

To add the FatFs to the project, a new one has been created. In this way all the *SEcube*^{TM} APIs created before are lost. The idea is to start from a newer and

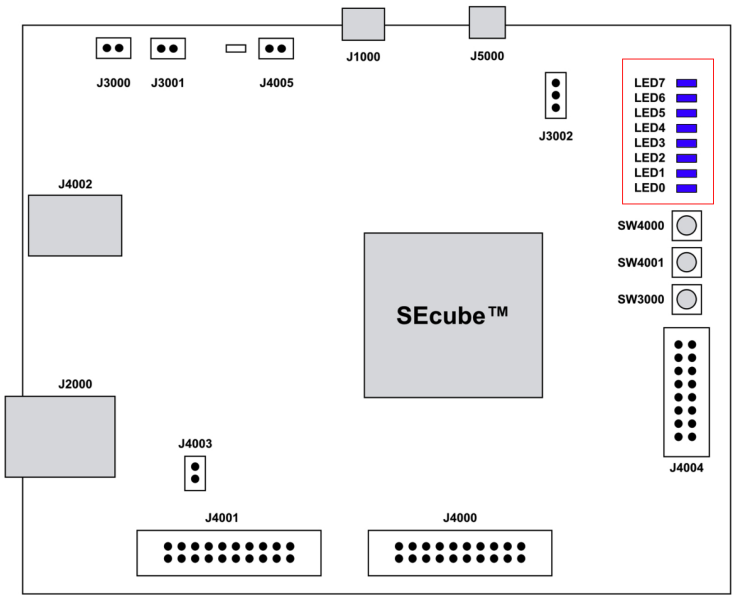


Figure 6.10. *SEcube*^{TM} DevKit board, highlighted LEDs [59]

clean firmware version, on which the FatFs libraries and the *SEcube*^{TM} APIs will be added.
To build the firmware, a specific software designed by STMicroelectronics, called STM32CubeMX has been employed. Its graphical environment allows to generate the C initialization code for STM32 microcontrollers and an easy way to personalize the code and systems properties. For example, the FatFs file system can be personalized directly by mean of STM32CubeMX environment, changing its characteristics without acting directly on the FatFs C files but setting them on a comfortable condition [67].

Test FatFs libraries

To use the SD card, with FatFs, an SD file system object has been created and then it has been linked to a compatible I/O driver (the low level device control). Then two file objects (due to the two bitstream files assumption) have been allocated and later opened with read access only. Once they've been initialized, they can be accessed in reading mode by mean of the function *f_read()* in which, one of its parameters is the number of bytes to read; this is the key point to fill the bitstream buffers.
Another useful function is the *f_lseek()*, it moves the read, or write, file pointer to

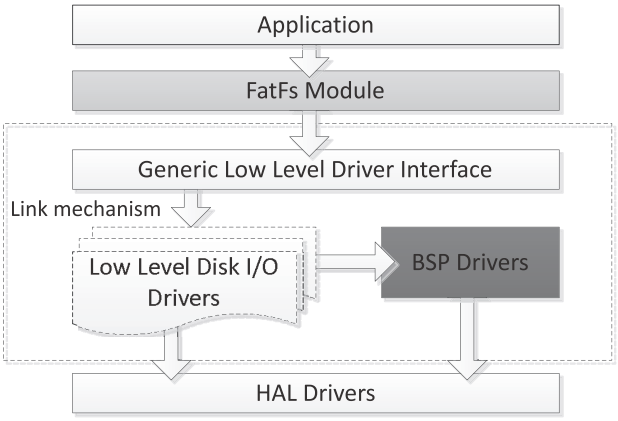


Figure 6.11. *FatFs* middleware module architecture [66]

- *BSP*: board support package;
- *HAL*: hardware abstraction layer.

the given (as parameter) offset value.

Programming FPGA using bitstream from SD card

In the previous section, all bitstreams (algorithm and data) were stored into the microcontroller. However this is not a realistic case, because may happen that the internal memory can be occupied by other software or its capability isn't big enough to store them all.
The solution employed is to keep this data into an external memory (on the SD card) and to store a portion of these files in a buffer inside the microcontroller (on an internal memory). This choice has been taken in order to avoid an high number of accesses to the files which would increase the time required to program the FPGA. This is true only if the buffer size is large enough (at least more than a single byte because the programming FPGA function gets 1 byte per time to operate).
The total available internal memory space is about 256+4 kB (SRAM) including 64-kB of CCM. The file sizes are:

- $g\_iAlgoSize = 129857$ byte
- $g\_iDataSize = 191421$ byte

Then 321278 byte (roughly 314 kB) are needed to store both bitstream in memory but they're clearly oversized for the internal memory (RAM) availability. For this reason two different buffers are added to the code: one for the algorithm bitstream (*__fpga_alg[]*) and the other for data bitstream(*__fpga_data[]*), both of them with 16384 bytes (16 kB, called *FPGA_BUFFERSIZE*) allocated, for a total amount of RAM usage of 32 kB, dedicated only to bitstream storage. To reduce as much as possible the number of modifications caused to the original *SEcube*^{TM} software, two new files have been created and added to the project: the source file *"secure_FPGA.c"* and its header file *"secure_FPGA.h"*[5].
Their main roles are:

- to initialize, manage and terminate the file system to gain access to the SD card;
- to initialize and terminate both *"algo.vme"* and *"data.vme"* files (*f_open*, *f_close*);
- to manage the access to these files (*f_read*, *f_lseek*);
- to fill the buffers (*__fpga_alg[]*, *__fpga_data[]*) under specific request (by through *fillFILEvector* function, Fig. 6.13);

Some functions, in the already existing *"FPGA.c"* source file, have been modified in order to get access to the *"secure_FPGA"* files:

- in *B5_FPGA_Programming* have been added the functions to initialize and terminate the SD file system session;
- in *GetByte* have been added all the checks required to find the actual *Index* (of an array) position, to decide if its needed to re-fill the buffer, reading bytes from one of the two files by mean of the *fillFILEvector* function; this code is essential to verify if the data required from the program is inside the buffer or in another logic block of the file; there are two different possibilities:
  - the data is in the actual block stored in the buffer: it is simply returned to the *GetByte* function;
  - the data is in another block, different from the one actually stored in the buffer: a request is sent to the system asking to read from the file the right block of bytes and to store them into the buffer.

[5]At this point of the project both of this files don't contain any security functions

- *GetByte* function gets the data passing through another function, called *selectANDpick*, which simplify the access to the right buffer (among *__fpga_alg[]* and *__fpga_data[]*)

After these updates, the microcontroller FLASH memory has been programmed. The software has been tested to be ensure that the FPGA programming, using the two bitstream files on the SD card, works correctly.

6.3.3 Data decryption

Up to this point the two bitstream files have been considered as plaintext; but for this final project step they will be assumed to be ciphered. Therefore a decryption procedure needs to be added to the previous created files (*"secure_FPGA.c"*, *"secure_FPGA.h"*) in order to get the plaintext which will be used to program the FPGA.
It's very important, in order to preserve the security of the bitstream, not to save any part of the plaintext on the external memory. On the other hand, the *SEcube*^{TM} grants a safe environment ensuring a secure communication on the interconnections between microcontroller and FPGA.

Encrypted data type

The ciphered version of the bitstream files have the following properties:

- **algo.vme.enc**
  - *key*: b'0123456789ABCDEF'
  - *key length*: 16 byte (128 bit)
  - *IV*[6]: x'a230a27f051232312227d4b4cd8cf5a7'
  - *IV length*: 16 byte (128 bit)
  - *plaintext bitstream length*: 129857 byte
  - *ciphered bitstream length*: 129888 byte
- **data.vme.enc**
  - *key*: b'0123456789ABCDEF'
  - *key length*: 16 byte (128 bit)

[6] *Initialization Vector*

– *IV*[7]: x'8d02439e358dc738346a1761b132fb82'
– *IV length*: 16 byte (128 bit)
– *plaintext bitstream length*: 191421 byte
– *ciphered bitstream length*: 191456 byte

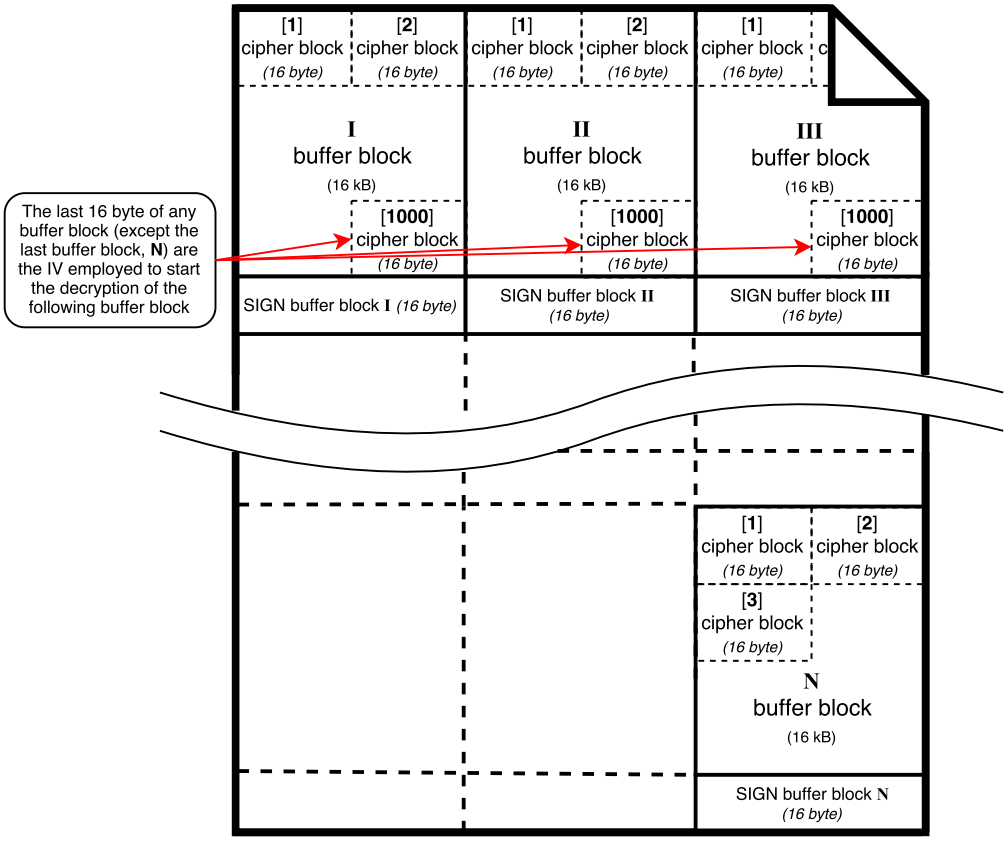


Figure 6.12. Bitstream file partitioning

Both these files have been encrypted using the *AES* algorithm in *CBC* mode (Chapter 7.2, Fig. 7.2, 7.3) . This is a block cipher mode of operation, it means that the plaintext and the ciphertext are split in blocks of the same size.
In this project, to program the FPGA, the *CBC* is used only in decryption mode: to decrypt the generic block i the algorithm needs:

[7] *Initialization Vector*

- ciphertext block i
- ciphertext block *i-1* (only the latest 16 byte are needed because this is the IV)
- the key

The case *i=1* requires a special treatment because it hasn't a *i-1* block (a block before itself), but it needs the initialization vector (IV).

To use this algorithm, the ciphertext decrypted has the same IV length. This means that the buffer size must be, in order to avoid code drawbacks, a multiple value of the IV length: for this reason, in the previous chapter the *FPGA_BUFFERSIZE* has been fixed at

$$16kB = 2^{10} \cdot 16\ byte = 1024 \cdot 16\ byte = 16384\ byte$$

Bitstream decryption

The *SEcube*^{TM} software already provides some libraries that contain encryption and decryption algorithms.
A block of encrypted data has been read from the ciphered bitstream file, it has been decrypted and then the buffer has been filled with the plaintext bitstream data. To do that, two functions have been added to *"secure_FPGA.c"* file:

- *fpga_programming_cryptoinit*: initializes the crypto environment and the procedure type (decryption in this case), set the algorithm type, it requires the decryption key;
- *fpga_programming_decrypt*: decrypts a specified amount of data passed to it; this requires the initialization vector.

They use the primitive security functions given by the *SEcube*^{TM} software.
The **fillFILEvector** function (Fig. 6.13), after reading the data from a cipher bitstream file, it recalls these two functions to start the decryption procedure to get the plaintext bitstream version and then to fill the buffer.
The decrypt function recalls the AES *B5_Aes256_Update* function which requires the number of ciphertext blocks to decrypt. In order to respect the bitstream partitioning, in accord to the *FPGA_BUFFERSIZE*, every buffer contains a number of blocks (ciphertext to be decrypted) equal to:

$$\frac{FPGA\_BUFFERSIZE}{IV\ length} = \frac{16384\ byte}{16\ byte} = 1024\ \frac{blocks}{buffer}$$

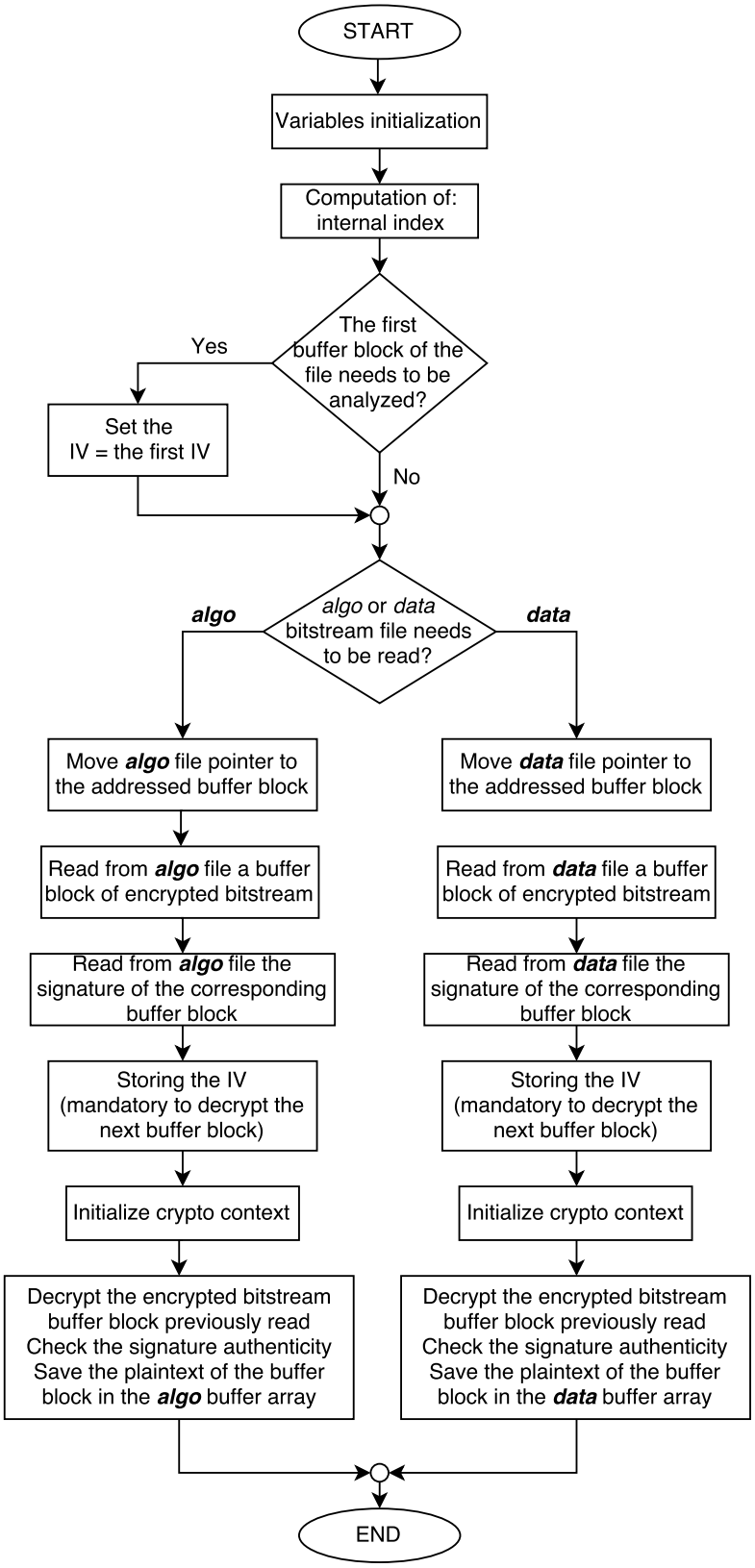


Figure 6.13. *"fillFILEvector"* function block diagram

Chapter 7

Security Analysis

7.1 Introduction

All the following security statements are assumed to be definitely secure in respect to the infeasibility hypothesis. This concerns the impossibility for an adversary to derive cryptographic algorithm secret informations or to invert a digest algorithm. The assumption can be considered valid if the cryptographic algorithms used are robust and with opportune key lengths.

7.2 Encryption mode

In first place, the objective was to implement the AES cipher algorithm in ECB mode (Fig. 7.1), but, due to its low security level, it has been changed into the CBC mode (Fig. 7.2), which grants more security paying it in terms of implementation complexity and design constraints.

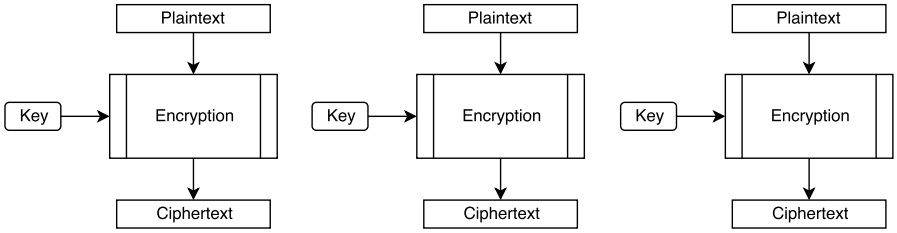


Figure 7.1. ECB mode for AES algorithms

The ECB main drawback is that, after the encryption, any resident properties of the plaintext might well show up in the ciphertext (this can be easily seen in

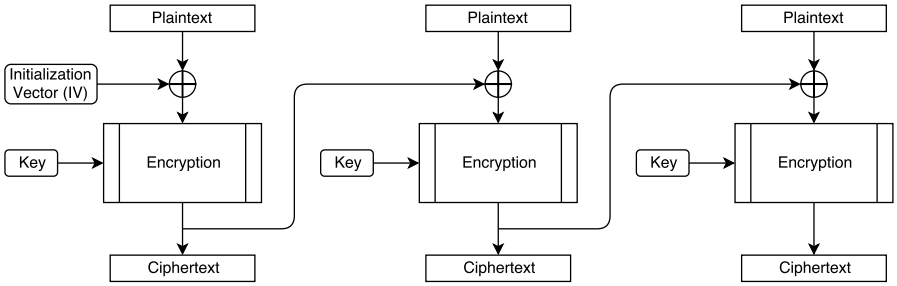


Figure 7.2. CBC encryption mode for AES algorithms

Fig. 7.4 where some informations may be comprehended in the ECB version of the encrypted logo). Obviously not as clearly, but analyzing the patterns these properties can be deduced.
On the other side ECB is faster (doesn't need to compute an EXOR operation) than the CBC mode, and its operations can be parallelized. Also the CBC operations can be parallelized but it's harder and it works mainly for the decryption mode (Fig. 7.3).
The IV, for CBC mode, doesn't need to be secret, but it must be unpredictable and its integrity preserved [68].

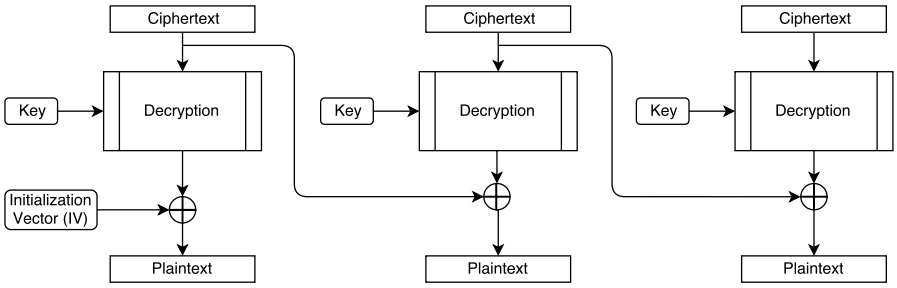
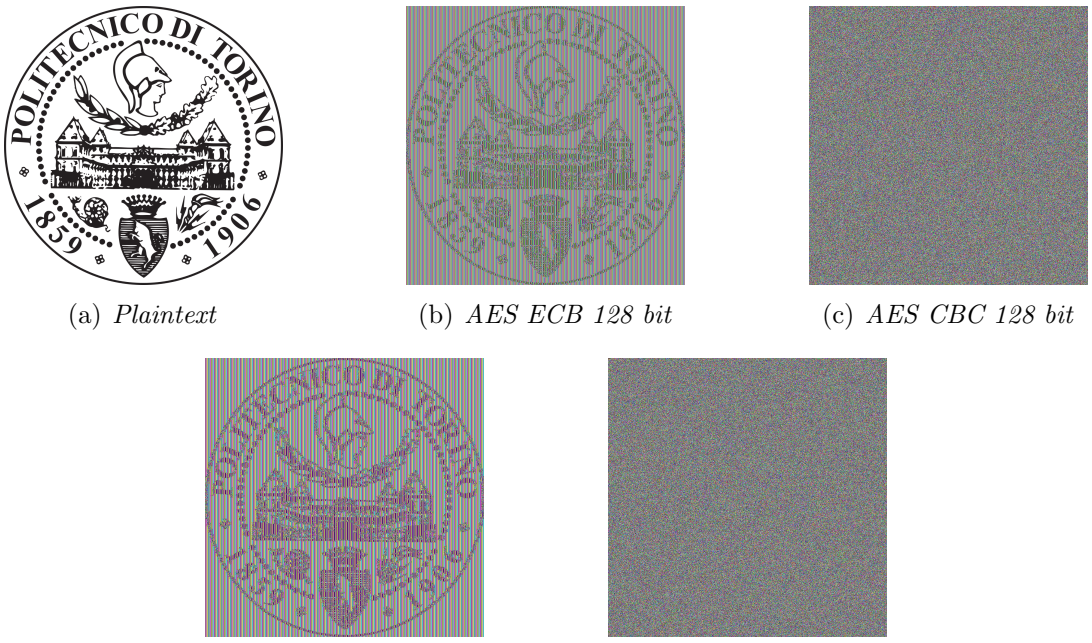


Figure 7.3. ECB decryption mode for AES algorithms

Two graphical examples have been given in order to better understand the ECB main drawback: some properties about the plaintext information can be understood even after the encryption (Fig. 7.4 and 7.5). In these examples, the key employed for the encryption is obtained by an hashing function which, given any length input string, gives a fixed length output value. This result is used as encryption key.
From a graphical point of view, properties of encrypted photographies are much more complex to be comprehended (Fig. 7.5) than the clip art ones (Fig. 7.4).

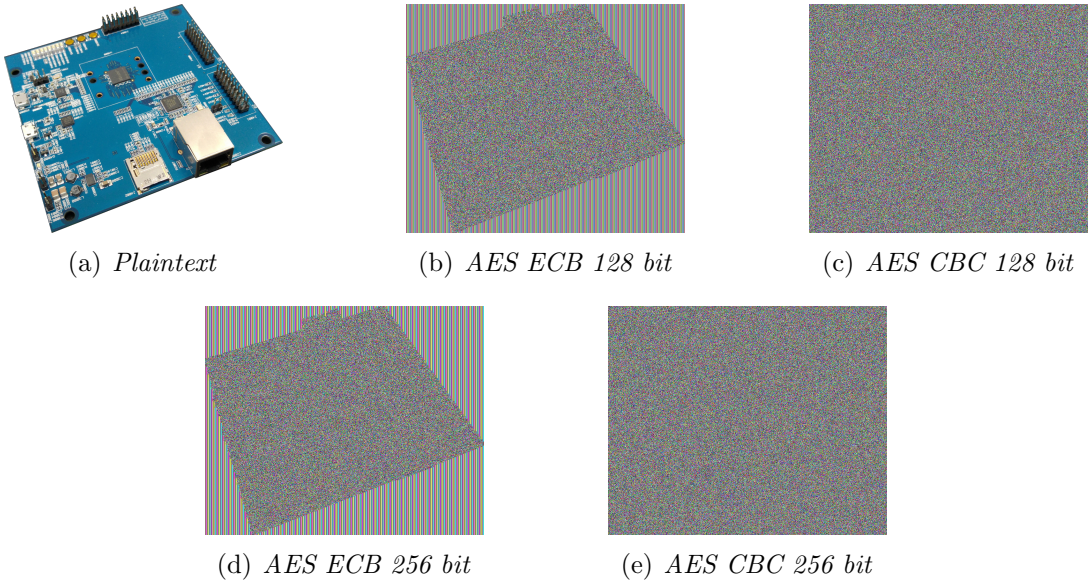
This is true because a photography has much more color informations than a simple clip art, like the one used in the example. Moreover, the absence of well defined borders, in terms of colors, between different objects in photos enhance this effect. This explains why the border of the board is well recognizable on a monochromatic background, instead, their details, like the integrated circuits, are not.



(a) *Plaintext* (b) *AES ECB 128 bit* (c) *AES CBC 128 bit*

(d) *AES ECB 256 bit* (e) *AES CBC 256 bit*

Figure 7.4. Graphical examples of AES encryption using ECB and CBC modes, with two different key length: 128 bit and 256 bit. The source image used is the Polytechnic of Turin logo (a clip art). The encryption key employed by the algorithm is obtained by the input string: "CARLO".



(a) *Plaintext* (b) *AES ECB 128 bit* (c) *AES CBC 128 bit*

(d) *AES ECB 256 bit* (e) *AES CBC 256 bit*

Figure 7.5. Graphical examples of AES encryption using ECB and CBC modes, with two different key length: 128 bit and 256 bit. The source image used is a photo. The encryption key employed by the algorithm is obtained by the input string: "CARLO".

7.3 Types of attackers

For the scenario presented in chapter 5.2 two different types of attackers has been considered and they need to be countered:

- MITM - Man in the middle
- MATE - Man at the end

7.3.1 MITM - Man in the middle

The MITM is an attacker that can intercept messages or impersonate one of the endpoints. If it happen the adversary may not only read the messages, but also modify or delete some of these. This is the standard security problem in computer networks and lots of different solution already exist to protect against them. A strong peer authentication mechanism avoids the impersonation over the communication channel; a key exchange allow to agree on the symmetric key system to be used to protect the data which flow over the channel using faster symmetric algorithms; then, with the symmetric agreed algorithms, ensure the data confidentiality (e.g., AES), integrity and authentication (e.g., keyed digest, HMAC using SHA256).

7.3.2 MATE - Man at the end

The MITM implicitly assumes trusted endpoints, but this is not acceptable if also MATE attackers are assumed to be interested in obtaining the bitstream and their attacks are more difficult to be addressed.
A MATE attacker has physical access to the device, it allows him to employ tools and techniques to reverse engineering the system and to tamper it with the software (e.g., debuggers, emulators). He has different ways to get the bitstream: using the I/O subsystem, the memory management subsystem, the communication channel used by the FPGA, and many others. He also has full control on the platform hardware. It means that the adversary can read and write every memory location, processor registers included. In the same way he has free access to the storage medium.
The only platform part of the user's environment, that can be considered secure is the FPGA; its stored informations and executed routines respect the confidentiality criterion.

7.4 Simple scenario security analysis

The simple scenario introduced in chapter 5.2 has been analyzed under the security point of view. Two communication peers are able to encrypt and decrypt messages by mean of a symmetric key. When one of the end users receives an encrypted message, which can be decrypted with the symmetric key, he suppose the message has been delivered by the other end user. The id_{FPGA} has been sent from the client to the hardware vendor passing through the store and the software provider. In every communication step a key is introduced to secure the id_{FPGA}. In this way, if strong encryption algorithms have been used, a MITM attacker can't read the id_{FPGA}. Moreover, the peers performing symmetric authentication avoid the impersonation and using data authentication and integrity mechanism avoid modifications which can lead to DoS attacks. Ad hoc security mechanism aren't required. The actual channel protection implementations already accomplish the security requirements.

The bitstream can be read in clear only by the software provider and the hardware vendor. All the time it is shared across the channel it is encrypted: when the SP need to send it to the HWV, it encrypts the bitstream using the shared key K; when the HWV receive the bitstream, it decrypts it using the key K and encrypt it with the key K_{FPGA}, which correspond to the physical devices specified by the id_{FPGA}, and sends it back to the client passing through the SP and the store.

The encrypted bitstream, which has been downloaded into the client device, is stored in an external memory device. There it is read and then decrypted by mean of the key K_{FPGA} obtaining the plaintext version of the bitstream (which is not saved into external memory storage devices). The process to program the FPGA starts and when it finishes this clear version of the bitstream is deleted.

Therefore, MITM and MATE attacks that aim at reading the bitstream are avoided.

Chapter 8

Conclusions

The project main goal is to allow a secure bitstream exchange among multiple independent and unrelated parties. A fully working solution of a secure FPGA programming process has been realized. The *SEcube*^{TM} platform has been adopted as physical environment and it has demonstrated to be a good choice, thanks to its architecture with good resources against physical attacks.

The security requirements are mandatory to avoid both unwanted bitstream files disclosure and malicious injection. To achieve that confidentiality, integrity and authenticity requisites, different security algorithms have been adopted:

- *Confidentiality* is grant by mean of the AES encryption block algorithm that works in CBC mode with a key-length of 256 bit;
- *Integrity* is grant by a digest algorithm which has been employed by mean of a SHA 256 function;
- *Authenticity* is grant by a signature algorithm which has been employed by mean of a SHA 256 function.

A module able to verify the correctness of the received bitstream and to decrypt it has been provided. Moreover it has been integrated with the remainder of the open-source firmware available on the prototyping platform. The designed structure is able to program the internal FPGA with a protected bitstream remotely downloaded. The integration of this module allowed to realize a possible future scenario for mobile applications deployment where reconfigurable devices assist the main processing unit with the computation.

The adopted solution is not the only possible one. For example, a single buffer can be used, filling it with the *"data"* or *"algo"* file content according to the FPGA programming algorithm request. Or maybe a different buffer length (*FPGA_BUFFERSIZE*) can be set, always respecting the design constraint of

$$FPGA\_BUFFERSIZE = N \cdot cipher\ block\ size$$

where N is an integer number and, according to this thesis assumptions, the *cipher block size* is equal to 16 byte.

Some possible future project steps could be:

- Reducing the API functions given by FatFs, acting on the C code ore directly on the STM32CubeMX environment, to easily modify some parameters like: *FS_READONLY*, *FS_MINIMIZE*, *VOLUMES*. These parameters let choose to the developer what functions he doesn't need and to remove them.
- employing different security structures;
- find the better tradeoff between memory occupation and programming speed;
- enhance the complex scenario proposed and test it;

Assuming an unlimited internal memory (RAM) availability, the best solution would have been to store all data from the files and to close them as soon as possible:

- start the FatFs session;
- open the ciphered bitstream files;
- read and decrypt *all* the file content;
- store the decrypted data into two arrays;
- close both files;
- terminate the FatFs session;
- access the clear version of the bitstream, every time is required, accessing directly to the arrays.

In this way, the number of access to the files (it is a slow operation) is reduce at the minimum. In other words, the total amount of time required to program the FPGA is reduced.

The current solution keeps the CPU busy until the FPGA is programmed; this isn't obviously the best solution. A possible enhancement could be to use a multicore system in which one core (or more, it depends on the core number) might program the FPGA while the others remain available for other operations.

Appendix A

Datasheet

A.1 Datasheet connection scheme

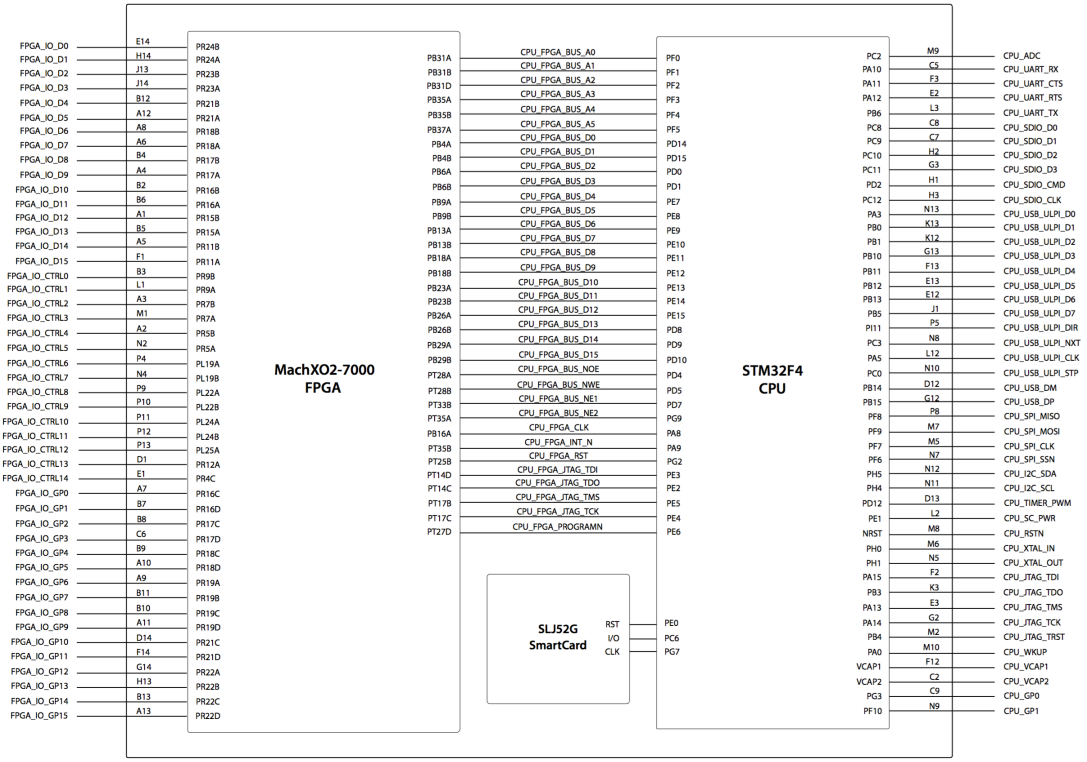


Figure A.1. *SEcube*^{TM} internal main blocks connections [62]

Appendix B

Programming code: C code

Some parts of the following codes are omitted ("[...]") on purpose to let the reader focus on the main contribution of the thesis project.

B.1 main.c

```
/**
  ******************************************************************
  * File Name          : main.c
  * Description        : Main program body
  ******************************************************************
  *
  * COPYRIGHT(c) 2017 STMicroelectronics
  *
  * Redistribution and use in source and binary forms, with or
    without modification,
  * are permitted provided that the following conditions are met:
  *   1. Redistributions of source code must retain the above
    copyright notice,
  *      this list of conditions and the following disclaimer.
  *   2. Redistributions in binary form must reproduce the above
    copyright notice,
  *      this list of conditions and the following disclaimer in the
    documentation
  *      and/or other materials provided with the distribution.
  *   3. Neither the name of STMicroelectronics nor the names of its
    contributors
  *      may be used to endorse or promote products derived from this
     software
  *      without specific prior written permission.
  *
  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
    CONTRIBUTORS "AS IS"
```

```
21   * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
       TO, THE
22   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
       PARTICULAR PURPOSE ARE
23   * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
       CONTRIBUTORS BE LIABLE
24   * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
       CONSEQUENTIAL
25   * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
       GOODS OR
26   * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION
       ) HOWEVER
27   * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
       STRICT LIABILITY,
28   * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
       OUT OF THE USE
29   * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
       DAMAGE.
30   *
31   ******************************************************************
32   */
33 /* Includes ------------------------------------------------------------*/
34 //   [...]
35 #include "FPGA.h"
36
37
38 /* Private function prototypes -----------------------------------------*/
39 //   [...]
40
41 int main(void)
42 {
43   /* MCU Configuration--------------------------------------------------------*/
44
45   /* Reset of all peripherals, Initializes the Flash interface and
       the Systick. */
46 //   [...]
47
48   /* Configure the system clock */
49 //   [...]
50
51   /* Initialize all configured peripherals */
52 //   [...]
53
54   SystemClock_Config();
55   MX_FATFS_Init();
56
57   device_init();
58
59   B5_FPGA_Programming();
```

```
60
61   /* Infinite loop */
62   device_loop();
63 }
64
65 /* Functions-----------------------------------------------------------*/
66 //   [...]
67
68 /************************ (C) COPYRIGHT STMicroelectronics *****END
     OF FILE ****/
```

B.2 FPGA.h

```
1 /*
2  * FPGA.h
3  */
4 #ifndef APPLICATION_SRC_FPGA_H_
5 #define APPLICATION_SRC_FPGA_H_
6
7 #include <secure_FPGA.h>
8
9
10 //   FUNCTIONS DECLARATION
11 //   [...]
12 int32_t      B5_FPGA_Programming (void);
13
14 #endif /* APPLICATION_SRC_FPGA_H_ */
```

B.3 FPGA.c

```
1 //   INCLUDES
2 //   [...]
3 #include "FPGA.h"
4
5 //   DEFINES
6 //   [...]
7
8 //   GLOBAL VARIABLES
9 //   [...]
10 uint32_t g_iAlgoSize = 129857;
11 uint32_t g_iDataSize = 191421;
12
13 //   FUNCTIONS DECLARATION
14 //   [...]
15 uint8_t selectANDpick (int32_t SubIndex, uint8_t algoORdata);
16 uint8_t GetByte(int32_t Index, uint8_t algoORdata);
17
18 //   FUNCTIONS
```

```
19 //  [...]
20 /*************************************************************
21 *                                                            *
22 * GETBYTE                                                    *
23 *                                                            *
24 * INPUT:                                                     *
25 *     Index: the current index to access.                    *
26 *                                                            *
27 *     algoORdata:1 if the return byte is to be retrieved from*
28 *     the algorithm array, 0 if the byte is to be retrieved  *
29 *     from the data array.                                   *
30 *                                                            *
31 * RETURN:                                                    *
32 *     This function returns a byte of data from either the   *
33 *     algorithm or data array.  It returns -1 if out of      *
34 *     bounds.                                                *
35 *                                                            *
36 *************************************************************/
37
38 uint8_t GetByte(int32_t Index, uint8_t algoORdata)
39 {
40 //  VARIABLEs  DECLARATION  - DEFINITION
41   uint8_t outData = 0;
42   uint8_t tmp_BufferIndex = 0;
43   uint8_t fillFILEvectorSTEPS_counter = 0;
44   uint32_t SubIndexMin = 0;
45   uint32_t SubIndexMax = 0;
46   uint32_t SubIndex;
47   uint32_t MaxVectorSize[2] = {g_iDataSize, g_iAlgoSize};
48
49   SubIndexMin = BufferIndex[algoORdata] * FPGA_BUFFERSIZE;
50   SubIndexMax = (BufferIndex[algoORdata] + 1) * FPGA_BUFFERSIZE - 1;
51   SubIndex = Index % FPGA_BUFFERSIZE;
52   reinit_iv = 1;
53
54 //  BODY
55   if ( !((Index >= SubIndexMin) && (Index <= SubIndexMax)) )
56   {
57     if ( Index >= MaxVectorSize[algoORdata] )
58     {
59       return (unsigned char) 0xFF;  //  ERROR: over limit value
60     }
61     else
62     {
63       tmp_BufferIndex = BufferIndex[algoORdata];
64       BufferIndex[algoORdata] = Index / FPGA_BUFFERSIZE;
65       fillFILEvector(algoORdata);
66     }
67   }
```

```
68   outData = selectANDpick(SubIndex, algoORdata);
69   return outData;
70 }
71
72 /*********************************************************
73 *                                                        *
74 * SELECTANDPICK                                          *
75 *                                                        *
76 * INPUT:                                                 *
77 *   SubIndex: index value of the current block           *
78 *                                                        *
79 *     algoORdata:1 if the return byte is to be retrieved from*
80 *     the algorithm array, 0 if the byte is to be retrieved  *
81 *     from the data array.                               *
82 *                                                        *
83 * RETURN:                                                *
84 *     A byte info from che correct data array            *
85 *                                                        *
86 *********************************************************/
87
88 uint8_t selectANDpick (int32_t SubIndex, uint8_t algoORdata)
89 {
90   if (algoORdata)
91   {
92     return __fpga_alg[SubIndex];
93   }
94   else
95   {
96     return __fpga_data[SubIndex];
97   }
98   return 0xFF;          //  ERROR
99 }
100
101
102
103
104 /*********************************************************
105 *                                                        *
106 * B5_FPGA_PROGRAMMING                                    *
107 *                                                        *
108 * INPUT:                                                 *
109 *     None                                               *
110 *                                                        *
111 * RETURN:                                                *
112 *     The return value will be a negative number if an error *
113 *     occurred, or 0 if everything was successful        *
114 *                                                        *
115 * DESCRIPTION:                                           *
116 *     This function opens the file pointers to the algo and  *
```

```
117 *     data file.  It intializes global variables to their    *
118 *     default values and enters the processor.               *
119 *                                                            *
120 ************************************************************/
121 int32_t B5_FPGA_Programming()
122 {
123   char szFileVersion[ 9 ] = { 0 };
124   int16_t siRetCode     = 0;
125   int16_t iIndex        = 0;
126   int16_t cVersionIndex = 0;
127 
128   /************************************************************
129   *                                                           *
130   * VARIABLES INITIALIZATION                                  *
131   *                                                           *
132   ************************************************************/
133   g_usDataType       = 0;
134   g_iMovingAlgoIndex = 0;
135   g_iMovingDataIndex = 0;
136 
137   sd_file_init();       /* creates SD-FATFS session and fill both
      buffers */
138 
139   if ( GetByte( g_iMovingDataIndex++, 0 ) ) {
140     g_usDataType |= COMPRESS;
141   }
142 
143   /************************************************************
144   *
145   * Read and store the version of the VME file.
146   *
147   ************************************************************/
148   for ( iIndex = 0; iIndex < 8; iIndex++ ) {
149     szFileVersion[ iIndex ] = GetByte( g_iMovingAlgoIndex++, 1 );
150   }
151 
152   /************************************************************
153   *
154   * Compare the VME file version against the supported version.
155   *
156   ************************************************************/
157   for ( cVersionIndex = 0; g_szSupportedVersions[ cVersionIndex ] !=
      0; cVersionIndex++ ) {
158     for ( iIndex = 0; iIndex < 8; iIndex++ ) {
159       if ( szFileVersion[ iIndex ] != g_szSupportedVersions[
      cVersionIndex ][ iIndex ] ) {
160         siRetCode = ERR_WRONG_VERSION;
161         break;
162       }
```

```
163       siRetCode = 0;
164     }
165     if ( siRetCode == 0 ) {
166
167       /*
    ************************************************************
168       *                                    *
169       * Found matching version, break.                 *
170       *                                    *
171       ************************************************************
    */
172       break;
173     }
174   }
175
176   if ( siRetCode < 0 ) {
177     /*************************************************************
178     *                                                            *
179     * Close SD-FATFS session.                                    *
180     *                                                            *
181     *************************************************************/
182       sd_file_finit();
183     /*************************************************************
184     *                                    *
185     * VME file version failed to match the supported versions.     *
186     *                                    *
187     *************************************************************/
188     return ERR_WRONG_VERSION;
189   }
190
191   /*************************************************************
192   *                                                            *
193   * Start the hardware.                                        *
194   *                                                            *
195   *************************************************************/
196     EnableHardware();
197
198   /*************************************************************
199   *                                                            *
200   * Begin processing algorithm and data file.                  *
201   *                                                            *
202   *************************************************************/
203   siRetCode = ispProcessVME();
204
205   /*************************************************************
206   *                                                            *
207   * Stop the hardware.                                         *
208   *                                                            *
209   *************************************************************/
```

```
210     DisableHardware();
211
212   /*************************************************************
213   *                                                            *
214   * Close SD-FATFS session.                                    *
215   *                                                            *
216   *************************************************************/
217     sd_file_finit();
218
219   /*************************************************************
220   *                                                            *
221   * Return the return code.                                    *
222   *                                                            *
223   *************************************************************/
224   return ( siRetCode );
225 }
```

B.4 secure_FPGA.h

```
1  /*
2   * secure_FPGA.h
3   *
4   *       Author: carlo
5   */
6
7  #ifndef APPLICATION_USER_SECURE_FPGA_H_
8  #define APPLICATION_USER_SECURE_FPGA_H_
9
10 #include "fatfs.h"
11 #include "se3_common.h"
12 #include "se3c0def.h"
13 #include "aes256.h"
14 #include "sha256.h"
15
16 /*************************************************************
17 *                                                            *
18 *              DEFINES                                       *
19 *                                                            *
20 *************************************************************/
21 #define FPGA_BUFFERSIZE 16384      /* Bytes in single paging buffer */
22 #define FPGA_NBLOCKS FPGA_BUFFERSIZE/16 /* Buffer block number */
23 #define FPGA_SIGNATURESIZE 32      /* Bytes - signature size*/
24
25 /*************************************************************
26 *                                                            *
27 *          EXTERNAL GLOBAL VARIABLES                         *
28 *                                                            *
29 *************************************************************/
```

```
30 extern uint8_t __fpga_alg[FPGA_BUFFERSIZE];    /* Algo Buffer */
31 extern uint8_t __fpga_data[FPGA_BUFFERSIZE];   /* Data Buffer */
32 extern uint8_t BufferIndex[2];
33 extern uint8_t reinit_iv;
34
35 /*************************************************************
36 *                                                            *
37 *              STRUCT                                        *
38 *                                                            *
39 *************************************************************/
40 typedef struct fpga_programming_cryptoctx_ {
41     B5_tAesCtx aesdec;
42   B5_tHmacSha256Ctx hmac;
43   uint8_t hmac_key[B5_AES_256];
44     uint8_t auth[B5_SHA256_DIGEST_SIZE];
45 } fpga_programming_cryptoctx;
46
47 /*************************************************************
48 *                                                            *
49 *              FUNCTIONS                                     *
50 *                                                            *
51 *************************************************************/
52 void sd_file_init();              /* Open SD-FATFS session */
53 void sd_file_finit();             /* Close SD-FATFS session */
54
55 void fillFILEvector(uint8_t algoORdata);  /* refill buffer */
56
57 void fpga_programming_cryptoinit(fpga_programming_cryptoctx* ctx);
    /* crypto enviroment initialization */
58 bool fpga_programming_decrypt(fpga_programming_cryptoctx* ctx, const
    uint8_t* auth, const uint8_t* iv, uint8_t* data);  /* signature
    check and decrypt */
59
60 #endif /* APPLICATION_USER_SECURE_FPGA_H_ */
```

B.5 secure_FPGA.c

```
1 /*
2  * secure_FPGA.c
3  *
4  *      Author: carlo
5  */
6 #include <secure_FPGA.h>
7 #include <stdint.h>
8
9 /*************************************************************
10 *                                                            *
11 *           GLOBAL VARIABLES                                 *
```

```
12 *                                                                    *
13 **********************************************************/
14 FATFS SDFatFs;  /* File system object for SD card logical drive */
15 FIL AlgoFile, DataFile; /* File objects */
16
17 fpga_programming_cryptoctx algo_fpga_ctx, data_fpga_ctx;  /* Crypto
       enviroment */
18
19 const uint8_t* fpga_key = se3_magic;  /* Decryption key */
20 const uint16_t fpga_flag = SE3_CMDFLAG_SIGN + SE3_CMDFLAG_ENCRYPT;
       /* 0 for not-encrypted bitstream file */
21 uint8_t InitializationVector[16] =
       {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};  /* Initial Initialization
       Vector for OFB, CBC, CTR */
22 uint8_t* IV_next[2];  // IV used to decrypt the following block (in
       sequential procedure)
23
24 uint8_t BufferIndex[2] = {0, 0};
25
26 uint8_t __fpga_alg[FPGA_BUFFERSIZE];  /* Algo Buffer */
27 uint8_t __fpga_data[FPGA_BUFFERSIZE]; /* Data Buffer */
28
29 /**********************************************************
30 *                                                         *
31 *           SD  FUNCTIONS                          *
32 *                                                         *
33 **********************************************************/
34 void sd_file_init()
35 {
36     /*********************************************************
37     *                                                        *
38     * FatFS: Link the SD disk I/O driver.                    *
39     *                                                        *
40     *********************************************************/
41     if(retSD == 0)
42       {
43     /*********************************************************
44     *                                                        *
45     * Register the file system object to the FatFs module.   *
46     *                                                        *
47     *********************************************************/
48         if(f_mount(&SDFatFs, (TCHAR const*)SD_Path, 0) != FR_OK)
49           {
50             while(1);
51           }
52           else
53           {
54     /*********************************************************
55     *                                                        *
```

```
56      * Open the text file objects with read access.                *
57      *                                                              *
58      ***************************************************************/
59              if((f_open(&AlgoFile, "algo.vme", FA_OPEN_EXISTING |
    FA_READ) != FR_OK) || (f_open(&DataFile, "data.vme",
    FA_OPEN_EXISTING | FA_READ) != FR_OK))
60                {
61                  while(1);
62                }
63      /***************************************************************
64      *                                                              *
65      * Initialize vectors: __fpga_alg[] and __fpga_data[].          *
66      *                                                              *
67      ***************************************************************/
68          fillFILEvector(1);
69          fillFILEvector(0);
70            }
71        }
72  }
73
74  void sd_file_finit()
75  {
76      /***************************************************************
77      *                                                              *
78      * Close both files.                                          *
79      *                                                              *
80      ***************************************************************/
81      f_close(&AlgoFile);
82      f_close(&DataFile);
83      /***************************************************************
84      *                                                              *
85      * Unlink the micro SD disk I/O driver.                         *
86      *                                                              *
87      ***************************************************************/
88      FATFS_UnLinkDriver(SD_Path);
89  }
90
91
92  /*************************************************************
93  *                                                            *
94  *        BUFFER MANAGER     FUNCTIONS                      *
95  *                                                            *
96  *************************************************************/
97
98  /*************************************************************
99  *                                                            *
100 * FILLFILEVECTOR                                             *
101 *                                                            *
102 * INPUT:                                                     *
```

```
103 *     algoORdata:1 if the return byte is to be retrieved from*
104 *     the algorithm array, 0 if the byte is to be retrieved  *
105 *     from the data array.                                    *
106 *                                                             *
107 * RETURN:                                                     *
108 *     Nothing                          *
109 *                                                             *
110 * DESCRIPTION:                                                *
111 *     The function fill the algorithm or data array           *
112 ************************************************************/
113
114 void fillFILEvector(uint8_t algoORdata)
115 {
116   uint32_t bytesread;
117   uint32_t InternalIndex;
118   uint8_t auth_sign[B5_SHA256_DIGEST_SIZE];
119   uint8_t* IV_int;
120
121   InternalIndex = BufferIndex[algoORdata] * ( FPGA_BUFFERSIZE +
      FPGA_SIGNATURESIZE ); //with SIGN
122 //  InternalIndex = BufferIndex[algoORdata] * ( FPGA_BUFFERSIZE );
       //without SIGN
123
124   if (InternalIndex == 0) // if the first block needs to be decrypted
125   {
126     IV_next[algoORdata] = InitializationVector; //set the first IV
127   }
128
129   if (algoORdata)
130   {
131     f_lseek(&AlgoFile, InternalIndex);
132     f_read(&AlgoFile, __fpga_alg, FPGA_BUFFERSIZE, (void*) &bytesread
      );
133     f_read(&AlgoFile, auth_sign, FPGA_SIGNATURESIZE, (void*) &
      bytesread);    //pick signature
134
135     IV_int = &IV_next[algoORdata];
136     IV_next[algoORdata] = __fpga_alg[FPGA_BUFFERSIZE-16];
137
138     fpga_programming_cryptoinit(&algo_fpga_ctx);
139     fpga_programming_decrypt(&algo_fpga_ctx, auth_sign, &IV_int,
      __fpga_alg);
140   }
141   else
142   {
143     f_lseek(&DataFile, InternalIndex);
144     f_read(&DataFile, __fpga_data, FPGA_BUFFERSIZE, (void*) &
      bytesread);
```

```
145     f_read(&DataFile, auth_sign, FPGA_SIGNATURESIZE, (void*) &
        bytesread);    //pick signature
146
147     IV_int = &IV_next[algoORdata];
148     IV_next[algoORdata] = __fpga_data[FPGA_BUFFERSIZE-16];
149
150     fpga_programming_cryptoinit(&data_fpga_ctx);
151     fpga_programming_decrypt(&data_fpga_ctx, auth_sign, &IV_int,
        __fpga_data);
152   }
153 }
154
155
156 /*************************************************************
157 *                                                            *
158 *            CRYPTO  FUNCTIONS                  *
159 *                                                            *
160 *************************************************************/
161 /*************************************************************
162 *                                                            *
163 * FPGA_PROGRAMMING_CRYPTOINIT                                *
164 *                                                            *
165 * INPUT:                                                     *
166 *     ctx: the crypto context                *
167 *                                                            *
168 * RETURN:                                                    *
169 *     Nothing                        *
170 *                                                            *
171 * DESCRIPTION:                                               *
172 *     The function initialize the crypto environment     *
173 *************************************************************/
174 void fpga_programming_cryptoinit(fpga_programming_cryptoctx* ctx)
175 {
176   B5_Aes256_Init(&(ctx->aesdec), fpga_key, B5_AES_256,
      B5_AES256_CBC_DEC);
177   memcpy(ctx->hmac_key, fpga_key, B5_AES_256);
178 }
179
180 /*************************************************************
181 *                                                            *
182 * FPGA_PROGRAMMING_DECRYPT                                   *
183 *                                                            *
184 * INPUT:                                                     *
185 *     ctx: the crypto context                *
186 *     auth: the sign mandatory to check the integrity and  *
187 *         authenticity                                      *
188 *     iv: initialization vector for the CBC mode             *
189 *     data: pointer to the data array that needs to be       *
190 *           decrypted                                        *
```

```
191 *                                                                    *
192 * RETURN:                                                           *
193 *     FALSE if the sign doesn't met correspondence       *
194 *                                                                    *
195 * DESCRIPTION:                                                      *
196 *     The function decrypt the data array and check the sign *
197 *********************************************************************/
198 bool fpga_programming_decrypt(fpga_programming_cryptoctx* ctx, const
        uint8_t* auth, const uint8_t* iv, uint8_t* data)
199 {
200   if ( fpga_flag & SE3_CMDFLAG_SIGN)
201     {
202       B5_HmacSha256_Init(&(ctx->hmac), ctx->hmac_key, B5_AES_256);
203       B5_HmacSha256_Update(&(ctx->hmac), iv, B5_AES_IV_SIZE);
204       B5_HmacSha256_Update(&(ctx->hmac), data, FPGA_NBLOCKS*
        B5_AES_BLK_SIZE);
205       B5_HmacSha256_Finit(&(ctx->hmac), ctx->auth);
206       if (memcmp(auth, ctx->auth, 16))
207       {
208         return false; // wrong signature
209       }
210     }
211
212   if (fpga_flag & SE3_CMDFLAG_ENCRYPT)
213     {
214       B5_Aes256_SetIV(&(ctx->aesdec), iv);
215       B5_Aes256_Update(&(ctx->aesdec), data, data, FPGA_NBLOCKS);
216     }
217
218     return true;
219 }
```

Appendix C

Others

C.1 FSM-like switch engine

This is the core of the FPGA programming process (Fig. C.1). It is the switch statement in which every case corresponds to a state in a FSM-like structure. The software, before starts running the state machine, check the internal FPGA programming code compatibility with the VME version of the bitstream file read.

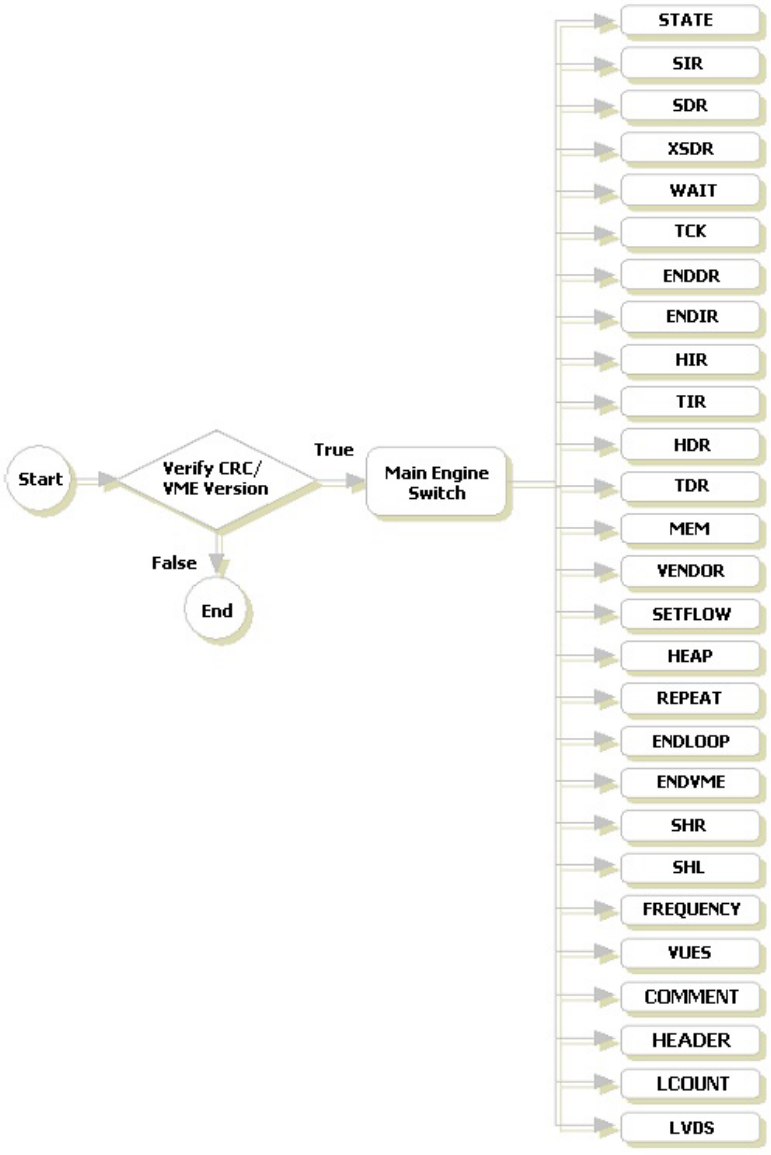


Figure C.1. Main FPGA programming process switch engine (FSM-like)

Bibliography

[1] P. Albright. (2003, October) Become a mobile apps innovator. [Online]. Available: https://www.computer.org/web/computingnow/mobile/content?g=53319&type=article&urlTitle=become-a-mobile-apps-innovator

[2] Gartner. (2011) Gartner says worldwide mobile application store revenue forecast to surpass $15 billion in 2011. [Online]. Available: https://www.gartner.com/newsroom/id/1529214

[3] R. H. Dennard, F. H. Gaensslen, H. N. Yu, V. L. Rideout, E. Bassous, and A. R. Leblanc, "Design of ion-implanted mosfet's with very small physical dimensions," *Proceedings of the IEEE*, vol. 87, no. 4, April 1999. [Online]. Available: http://www.ece.ucsb.edu/courses/ECE225/225_W07Banerjee/reference/Dennard.pdf

[4] A. McMenamin. (2013, April) The end of dennard scaling. [Online]. Available: https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/

[5] P. E. Ross. (2008, April) Why cpu frequency stalled. [Online]. Available: https://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled

[6] P. P. Mattsson. (2013, November) Why haven't cpu clock speeds increased in the last few years? [Online]. Available: https://www.comsol.com/blogs/havent-cpu-clock-speeds-increased-last-years/

[7] S. Iyer. (2010, December) Cmos power consumption. [Online]. Available: http://large.stanford.edu/courses/2010/ph240/iyer2/

[8] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26–37, 2009. [Online]. Available: http://http://ieeexplore.ieee.org/document/5230801/?part=1/courses/ECE225/225_W07Banerjee/reference/Dennard.pdf

[9] R. Iyer and D. Tullsen, "Heterogeneous computing [guest editors' introduction]," *IEEE Micro*, vol. 35, no. 4, pp. 4–5, July 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7182244/

[10] ——, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008. [Online]. Available: http://ieeexplore.ieee.org/document/4490127/

[11] J. Langguth, M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai, "Scalable heterogeneous cpu-gpu computations for unstructured tetrahedral meshes," *IEEE Micro*, vol. 35, no. 4, pp. 6–15, July 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7155461/

[12] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers, "Achieving exascale capabilities through heterogeneous computing," *IEEE Micro*, vol. 35, no. 4, pp. 26–36, July 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7155462/

[13] Xilinx. What is an fpga? [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[14] M. Bollo, A. Carelli, S. D. Carlo, and P. Prinetto, "Side-channel analysis of secube platform," *2017 IEEE East-West Design Test Symposium (EWDTS)*, pp. 1–5, Sept 2017.

[15] Xilinx. Fpga vs. asic. [Online]. Available: https://www.xilinx.com/fpga/asic.htm

[16] S. D. Carlo, P. Prinetto, and A. Scionti, "A fpga-based reconfigurable software architecture for highly dependable systems," *2009 Asian Test Symposium*, pp. 125–130, Nov 2009.

[17] A. Shan. (2006, January) Heterogeneous processing: a strategy for augmenting moore's law heterogeneous processing: a strategy for augmenting moore's law. [Online]. Available: http://www.linuxjournal.com/article/8368?page=0,0

[18] Xilinx. What is an fpga? [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[19] S. Ke-fei, "Application of fpga in aerospace remote sensing systems," *OME Information*, 2010.

[20] M. Surratt, H. Loomis, A. Ross, and R. Duren, "Challenges of remote fpga configuration for space applications," *Aerospace Conference, IEEE*, pp. 1–9, 2005.

[21] A. Ahmad, B. Krill, A. Amira, , and H. Rabah, "3d haar wavelet transform with dynamic partial reconfiguration for 3d medical image compression," *Proc. IEEE Biomedical Circuits and Systems Conf. BioCAS*, pp. 137–140, 2009.

[22] M. E. Dunham, Z. Baker, M. Stettler, M. Pigue, P. Graham, E. N. Schmierer, and J. Power, "High efficiency space-based software radio architectures: A minimum size, weight, and power teraops processor," *Proc. Int. Conf. Reconfigurable Computing and FPGAs ReConFig '09*, pp. 326–331, 2009.

[23] W. Lie and W. Feng-yan, "Dynamic partial reconfiguration in fpgas," -, vol. 2, no. 4, pp. 445–448, November 2009. [Online]. Available:

http://ieeexplore.ieee.org/document/5369525/

[24] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, "An internal partial dynamic reconfiguration implementation of the jpeg encoder for low-cost fpgasb," *Proc. IEEE Computer Society Annual Symp. VLSI ISVLSI '07*, pp. 449–450, 2007.

[25] S. D. Carlo, A. Miele, P. Prinetto, and A. Trapanese, "Microprocessor fault-tolerance via on-the-fly partial reconfiguration," *2010 15th IEEE European Test Symposium*, pp. 201–206, May 2010.

[26] S. D. Carlo, G. Gambardella, P. Prinetto, D. Rolfo, P. Trotta, and A. Vallero, "A novel methodology to increase fault tolerance in autonomous fpga-based systems," *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, pp. 87–92, July 2014.

[27] Actel. (2008) Actel proasic3 handbook. [Online]. Available: http://www.actel.com/documents/PA3_HB.pdf

[28] Altera. (2009) Design security in stratix iii devices. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01010.pdf

[29] Lattice. Xp2 family handbook. [Online]. Available: http://www.latticesemi.com/documents/H

[30] Xilinx. Lock your designs with the 4 security solution. [Online]. Available: http://www.xilinx.com/publications/xcellonline/xcell_52/xc_pdf/xc_v4security52.pdf

[31] B. Badrignans, D. Champagne, R. Elbaz, C. Gebotys, and L. Torres, "Sarfum: Security architecture for remote fpga update and monitoring," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 2, pp. 8:1–8:29, May 2010. [Online]. Available: http://doi.acm.org/10.1145/1754386.1754389

[32] S. Drimer, "Authentication of fpga bitstreams: Why and how," *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 73–84, 2007.

[33] B. Schneier, "Applied cryptography (2nd ed.): Protocols, algorithms, and source code in c," *John Wiley & Sons, Inc.*, 1995.

[34] W. Stallings, "Cryptography and network security: Principles and practice," *Prentice Hall Press*, 2010.

[35] S. Singh, "The code book: The science of secrecy from ancient egypt to quantum cryptography," *Knopf Doubleday Publishing Group*, 2011.

[36] N. R. Council, "Computers at risk: Safe computing in the information age," *The National Academies Press*, 1991.

[37] R. Shirey, "Internet security glossary, version 2," *The IETF Trust*, August 2007. [Online]. Available: https://tools.ietf.org/html/rfc4949

[38] F.-X. Standaert, "Introduction to side-channel attacks," *UCL Crypto Group*, June 2016.

[39] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Springer*, 2011.

[40] One-way function. Wolfram. [Online]. Available: http://mathworld.wolfram.com/One-WayFunction.html

[41] T. Wollinger, J. Guajardo, and C. Paar, "Securityonfpgas: State-of- the-art implementations and attacks," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 534–574, 2004.

[42] L. Bossuet, G. Gogniat, and W. Burleson, "Dynamically config- urable security for sram fpga bitstreams," *Proc. 18th Int. Parallel and Distributed Processing Symp*, 2004.

[43] A. Lesea. (2007, February) Ip security in fpgas. [Online]. Available: http://www.xilinx.com/support/documentation/whitepapers/wp261.pdf

[44] Altera. Error detection in altera fpga devices. [Online]. Available: http://www.altera.com/literature/an/an357.pdf

[45] Xilinx. Virtex-5 configuration user guide. [Online]. Available: http://www.xilinx.com/support/documentation/userguides/ug191.pdf

[46] M. Stigge, H. Platz, W. Muller, and J.-P. Redlich, "Reversing crc theory and practice," *Humboldt University Berlin, Technical Report*, 2006.

[47] M. M. Parelkar and K. Gaj, "Implementation of eax mode of operation for fpga bitstream encryption and authentication," *Proc. IEEE Int Field-Programmable Technology Conf*, pp. 335–336, 2005.

[48] M. M. Parelkar, "Authenticated encryption in hardware," *Master's thesis, George Mason University*, 2005.

[49] S. Drimer, "Authenticated of fpga bitstreams: why and how," *In Applied Reconfigurable Computing*, vol. 4419, pp. 77–84, 2007.

[50] A. S. Zeineddini and K. Gaj, "Secure partial reconfiguration of fpgas," *Proc. IEEE Int Field-Programmable Technology Conf*, pp. 155–162, 2005.

[51] Y. Hori, A. Satoh, H. Sakane, and K. Toda, "Bitstream encryption and authentication with aes-gcm in dynamically reconfigurable systems," *Proc. Int. Conf. Field Programmable Logic and Applications FPL 2008*, pp. 23–28, 2008.

[52] T. K. Tan, R. Weerakkody, M. Mrak, N. Ramzan, V. Baroncini, J.-R. Ohm, and G. J. Sullivan, "Video quality evaluation methodology and verification testing of hevc compression performance," *IEEE transactions on circuits and systems for video technology*, vol. 26, no. 1, 2016. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7254155

[53] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[54] L. S. Corporation, *Programming Tools User Guide*, Lattice Semiconductor Corporation, December 2013.

[55] R. Hassan, K. Markantonakis, and R. N. Akram, "Can you call the software

in your device be firmware?" *2016 IEEE 13th International Conference on e-Business Engineering (ICEBE)*, pp. 188–195, November 2016.

[56] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. Dray, "Announcing the advanced encryption standard (aes) - (fips pub 197)," *Federal INformation Processing Standards Publication 197 (FIPS PUBS)*, no. 197, November 2001.

[57] N. I. of Standards and T. (NIST), "Secure hash standard (sha) - (fips pub 180-1)," *Federal INformation Processing Standards Publication (FIPS PUBS)*, no. 180-1, April 1995.

[58] ——, "Secure hash standard (sha) - (fips pub 180-2)," *Federal INformation Processing Standards Publication (FIPS PUBS)*, no. 180-2, August 2002.

[59] (2017, January) Secube getting started guide. [Online]. Available: https://www.secube.eu/download/SEcube-Development-Kit-Getting-Started-PUBLIC-v1.4.pdf

[60] C. Basile, S. D. Carlo, and A. Scionti, "Fpga-based remote-code integrity verification of programs in distributed embedded systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 2, pp. 187–200, March 2012.

[61] Open hardware - software platforms for seamless trusted systems. [Online]. Available: www.blu5group.com

[62] (2015, August) Secube datasheet. [Online]. Available: https://www.secube.eu/download/SEcube-Datasheet-R7.pdf

[63] G. A. FARULLA, A. CARELLI, P. PRINETTO, G. SOMMA, and A. VARRIALE, "Secube development kit: Getting started," *SEcube*, January 2017.

[64] Y. Amir, "Operating systems 600.418 the file system," *Department of Computer Science Johns Hopkins University*, July 2016.

[65] Fatfs. [Online]. Available: http://elm-chan.org/

[66] S. Microelectronics, *Developing Applications on STM32Cube with FatFs*, ST Microelectronics, June 2014.

[67] Stm32cubemx. [Online]. Available: http://www.xilinx.com/support/documentation/userguides/ug191.pdf

[68] M. Dworkin, "Recommendation for block cipher modes of operation," *NIST Special Publication 800-38A*, 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf