Embedded Systems Security for FPGA

Benoit Badrignans, Florian Devic, Lionel Torres, Gilles Sassatelli, Pascal Benoit

▶ **To cite this version:**

Benoit Badrignans, Florian Devic, Lionel Torres, Gilles Sassatelli, Pascal Benoit. Embedded Systems Security for FPGA. Security Trends for FPGAS From Secured to Secure Reconfigurable Systems, pp.137-187, 2011, 978-94-007-1337-6. 10.1007/978-94-007-1338-3_6 . lirmm-00818934

HAL Id: lirmm-00818934

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00818934v1

Submitted on 18 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Chapter 6
Embedded Systems Security for FPGA

B. Badrignans, F. Devic, L. Torres, G. Sassatelli, and P. Benoit

Abstract The main goal of this chapter is to study FPGA devices in the field of secured applications. We mainly address data protection based on a well defined threat model. When dealing with FPGAs at the system level, two kinds of data are of paramount importance: bitstream and external memory. To cover these topics, we first review state of the art FPGA security mechanisms and good practices, followed by performance analysis achievable using hardware implementation of cryptographic algorithms in current FPGAs. We then tackle external memory protection and how FPGAs can provide an efficient solution. Next, we highlight security issues specific to FPGAs, bitstream replay attacks, for example, and suggest solutions to improve bitstream management security, focusing on secure remote updating of FPGA bitstreams. Finally we give the results of a concrete case, i.e., a platform based on an FPGA device. This last section provides both a practical and an industrial point of view that will enable readers to evaluate the pertinence of the solutions proposed.

6.1 Introduction and Objectives

Motivations to employ FPGAs (Field-Programmable Gate Array) in secure systems are multiple: hardware configuration can be updated all along system life-cycle, FPGAs can be finely configured to implement cryptographic functions efficiently, and security applications generally generate low sales volumes making FPGAs more attractive than ASICs (Application Specific Integrated Circuits). However ASICs often contain special features that are not available in all FPGAs. For instance, most current FPGAs do not include non-volatile memories that are useful in security applications (e.g., to store cryptographic keys). Moreover designers including FPGA devices in their design must not only consider in its threat analysis the applicability of attacks generally targeting ASICs and general purpose processors (e.g., memory tampering) but must also explore threats specific to FPGAs. For instance, an adversary tampering with the FPGA configuration file, a.k.a. the bitstream, can modify

B. Badrignans (✉)
Netheos, Montpellier, France
e-mail: b.badrignans@netheos.net

functions implemented inside the FPGA user logic and thereby impact the behavior of the FPGA-based system. The main goal of this chapter is to explore threats potentially impacting FPGA-based systems. We address data protection based on a well defined threat model where the adversary has physical access to the FPGA-based system.

We mainly address data/code protection since two kinds of data—bitstream and external memory—are of paramount importance when dealing with FPGAs at the system level. In order to highlight the importance of the threats discussed in this chapter, we identified three security-sensitive applications potentially including FPGAs in their designs.

Physically Inaccessible Systems Some systems like satellite or space craft are intrinsically protected against physical attacks since inaccessible during their deployment in space. Devices can also be made inaccessible using secure rooms or strong-box. Due to their peculiar location we can consider that they are protected against any attack requiring physical access to the device. However the cost of this system as well as their potential strategic importance (e.g., in military and communication satellites) emphasis the requirement to make them highly secure against remote attacks.

Personal Hardware Security Modules—HSM Hardware security modules are devices dedicated to provide a high level of security which generally cannot be achieved using general purpose computers. They often offer ways to physically protect cryptographic keys or sensitive data. These systems are used in highly secure applications such as in-line banking applications, ATMs transactions, and companies network infrastructures. Since the appearance of smart card for payment and money withdrawal, most individual owns an increasing number of security modules (e.g., biometric passport, mobile phone SIM card, French health insurance card).

Smart cards do not embed programmable logic yet (even though the growing market of multi-applications cards might encourage such features in a near future [25]) but they are not the only personal secure devices available. For instance the smart drive provided by the French company Bull embeds an FPGA device in charge of cryptographic and key management operations. This device, called *Globull*, is a secure USB hard disk drive protected by a user PIN (Personal Identification Number). It provides also cryptographic services like smart cards, such as on-board RSA key generation, RSA signature and encryption. Therefore we can reasonably imagine that in coming years most individuals traveling with sensitive data will have advanced personal security devices embedding FPGA devices. Applications can be: mobile disk encryption, e-mail protection, secure key management, VPN access.

Therefore we can reasonably assume that in the near future most individuals who travel with sensitive data will own advanced personal security devices that can take embedded FPGA devices. Some applications are mobile disk encryption, e-mail protection, secure key management, or VPN access. Whatever the application, personal security devices have their own constraints. Firstly, most devices assume that attackers will never access them when they are unlocked by the user. For instance,

laptop disk encryption is ineffective if someone steals it when its disk is unlocked. In addition, the devices can reasonably trust their owner, after all the information that these devices protect belong to the user.

Set-Top Boxes/Video Game Console Set-top boxes are devices generally rented out to customers by ISP (Internet Service Provider) or pay-TV providers. Since these systems provide access to protected multimedia content they must embed some security features to enforce digital right management. Their environments are dramatically different from previous examples of applications. First such systems are not shielded or physically protected and the user is potentially the adversary. Therefore, the adversary has physical access and is not limited in time to carry out his/her attack. Moreover he often benefits from a community of attackers or security researchers [42] sharing their technical knowledge and discoveries online.

In order to address security concerns related to these possible scenarios, this chapter is divided into three main parts each of which addresses one of the three key points involved in dealing with security and FPGAs. The first point is related to protecting data and code processed by FPGAs. We present a widely recognized threat model and existing efforts to protect a system against these threats, and then several possible solutions. These approaches leverage applications and FPGA characteristics to reduce the cost of security. The benefit is a strong optimization component in terms of memory overhead and performance. The second point is related to reconfiguration management of targeted architectures. This point is very sensitive as FPGAs offer considerable flexibility through dynamic reconfiguration. This feature can not only be used to perform upgrades and bug fixing, but also allows for reactions in the case of an attack. However, if not handled correctly, this strong advantage can become a security breach. After describing potential threats that use the reconfiguration link, we detail current efforts to counter them. Then we propose an original technique to perform remote partial reconfiguration. Based on such technology, a whole protocol and architecture were designed to perform secure reconfiguration. The third point is defining a secure platform. We thoroughly analyze what is required to obtain a fully secure FPGA based on previous solutions. A prototype of this platform has been designed and offers interesting features to target, including gigabit network encryption, SSL accelerator or offload engine, VPN accelerator, Hardware Security Module with high performances.

6.2 Definitions—Glossary

In this chapter, the following vocabulary is used to distinguish the different internal logic types in FPGA devices:

- **static logic** is the static part of the FPGA; the designer cannot modify its architecture. FPGA devices often embed general purpose processor in the static logic.
- **user logic** is the configurable logic embedded in FPGA devices. The user logic may include LUTs, RAM blocks, hardware multipliers, DSP blocks or switch matrix for routing signals.

- **configuration logic** is the part of the static logic in charge of loading the bitstream into the user logic, it is typically composed of a JTAG chain or any configuration port and of the bitstream decryption engine (if any).

In the following section we distinguish four stakeholders involved in an FPGA-based system development life cycle:

- The **FPGA vendors** are the companies designing, producing and providing FPGA chips (e.g., Xilinx, Altera or Actel).
- **IP designers** provide reusable units of hardware units often delivered as netlist or Hardware Description Language (HDL) codes. IP cores may be encryption engines, general purpose processors or memory interfaces.
- The **System Designer** (SD) assembles IPs provided by different IP designers to produce the final bitstream which configures the FPGA device. In case of complex FPGA-based systems, we assume in this book that the SD is also responsible to produce the platform potentially composed of different types of circuit (e.g., General purpose processors, ASICs or FPGAs).
- The **system owner** is the end user who exploits the system, the SDs and IP designers do not necessarily trust owners.

We distinguish three types of FPGA chips with respect to their configurability properties:

Current FPGAs can be classified in three different types:

- **Anti-fuse** FPGAs are historically the first non-volatile FPGAs. Each configuration point is controlled by an anti-fuse element. Configuration is fixed and cannot be changed after programming. Actel is the leader in this market.
- **Flash-based** FPGA configuration sites are controlled by a Flash transistor, they are reconfigurable but non-volatile.
- The last type is **SRAM based** FPGAs or volatile FPGAs are composed of SRAM memory cells and, therefore, cannot keep their configuration when power is down. An external non-volatile memory is generally required to store the bitstream. Altera and Xilinx are the leading companies in the market of SRAM FPGAs, offering low-cost as well as high-performance SRAM-based FPGA devices.

FPGAs can be configured in two main ways:

Static Reconfiguration is the classical ability of FPGA devices to be reconfigured when powered down. This feature is useful for cryptography and security-sensitive applications. Cryptographic algorithms have a limited lifetime, the system designer can provide new algorithms during the exploitation of the system. New attacks can also be discovered and again the system designer can provide a more robust version of the system.

Dynamic Reconfiguration is the ability of FPGAs to be reconfigured at runtime. Xilinx has been supplying tools and mechanisms for this purpose for many years. This feature allows the system to swap logic blocks thus allowing many applications. However, the process also introduces security issues since portions of the bitstream

need to be secured. The system designer may encrypt and authenticate these pieces of hardware in order to avoid a malicious bitstream being loaded. This has to be done inside the user logic since the SD is in charge of partial reconfiguration through an internal port located in the user logic, called ICAP for Internal Configuration Access Port. Partial reconfiguration security is not addressed in this book since it is not applicable on all FPGAs, and also because the concepts developed to secure classical bitstreams can be used for partial reconfiguration.

6.3 Confidentiality and Integrity of Data Processed by Reconfigurable Platforms

Data processed by general purpose processors embedded in the static logic of FPGAs or by IPs implemented in the user logic are commonly stored in off-chip memories. In applications where physical adversaries are considered (e.g., set top box), an attacker can retrieve data transiting on the bus, thereby challenging data confidentiality, or tamper with them, thereby challenging data integrity. In this section we first describe the passive and active attacks allowing an adversary to retrieve or tamper with data transiting between the FPGA chip and the memory. Then we describe potential countermeasures to this security issue and solutions we recently proposed.

6.3.1 Threat Analysis

Confidentiality and integrity of data stored in off-chip memory is usually a security concern when attackers with physical access to the device are considered (e.g., in application like set-top box). However, the main trust assumption made is that the processor chip is resistant to all physical attacks and is thus trusted. Moreover, the cryptographic engine required for encryption and authentication are assumed resistant to side channel attacks. We consider the adversary has full control of the data stored in memory and transiting on the bus between the FPGA chip and the memory. We consider two kinds of physical attacks an adversary can carry out: passive and active attacks. Passive attacks consist in probing the bus to retrieve the memory content. Such attacks challenge the confidentiality of data in memory.

In an active attack, the adversary corrupts the data residing in memory or transiting over the bus; this corruption may be considered as data injection since a new value is created. Figure 6.1 gives the example of an attacked device where an adversary connects his own (malicious) memory to the targeted platform via the off chip bus.

We distinguish between three classes of active attacks, defined with respect to how the adversary chooses the inserted data. Figure 6.2 depicts the three active attacks; below, we provide a detailed description of each one based on the attack framework in Fig. 6.1:

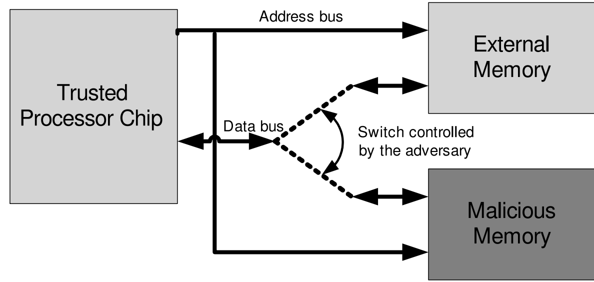


Fig. 6.1 An example of a framework of attack targeting the external memory of a computing platform

1. **Spoofing attacks**: the adversary exchanges an existing memory block with an arbitrary fake one (Fig. 6.2-a, the block defined by the adversary is stored in the malicious memory, the adversary activates the switch command when he wants to force the processor chip to use the spoofed memory block).
2. **Splicing or relocation attacks**: the attacker replaces a memory block at address A with a block at address B, where $A \neq B$. Such an attack can be considered

External Memory seen by the Processor Chip ...

...before spoofing | ...after spoofing | ...before splicing | ...after splicing

Spoofing | Splicing | Replay

...before replay at time t1 | ...after replay at time t2

Data (@1;t1) A genuine block stored at address 1 at time t1

A malicious memory block

Fig. 6.2 Three kinds of active attacks: (**a**) spoofing, (**b**) splicing and (**c**) replay

as a spatial permutation of memory blocks (Fig. 6.2-b: the adversary stores the content of the block at address 1 in the genuine memory at address 5 in the malicious memory. When the processor requests the data at address 5, the adversary activates the switch command so the processor reads the malicious memory. As a result, the processor reads the data at address 1).

3. **Replay attacks**: a memory block located at a given address is recorded and inserted at the same address at a later point in time; by doing so, the value of the current block is replaced by an older one. Such an attack can be considered as a temporal permutation of a memory block, for a specific memory location (Fig. 6.2-c: at time t1, the adversary stores the content of the block at address 6 in the genuine memory at address 6 in the malicious memory. At time t2, the memory location at address 6 has been updated in the genuine memory but the adversary does not perform this update in the malicious memory. The adversary activates the malicious memory when the processor requests the data at address 6, thus forcing it to read the old value stored at address 6).

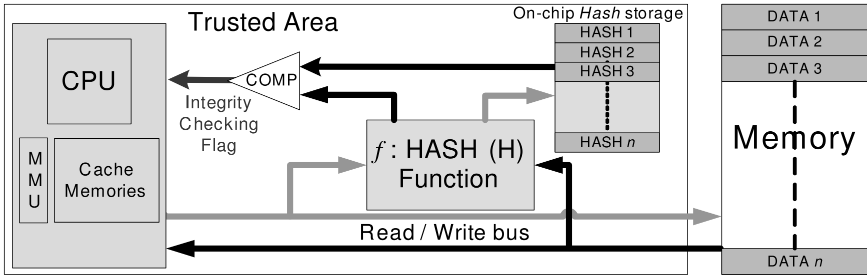
6.3.2 State of the Art

Two distinct strategies are described in the state of the art to thwart the active attacks described in our threat model. Each strategy is based on different authentication primitives, namely a cryptographic hash function and a message authentication code (MAC) function. In this section, we first describe how these primitives allow for memory authentication and how they should be integrated in tree structures in order to avoid excessive overheads in on-chip memory.

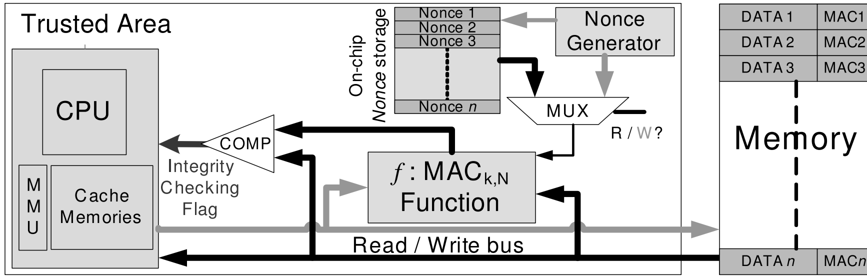
6.3.2.1 Authentication Primitives for Memory Authentication

Hash Functions The first strategy (Fig. 6.3-a) that enables memory authentication consists in storing on-chip a hash value for each memory block stored off-chip (write operations). The integrity of read operations is checked by re-computing a hash over the loaded block and by then comparing the resulting hash with the on-chip hash fingerprinting the off-chip memory location. The on-chip hash is stored in the tamper resistant area, i.e., the processor chip, and is thus inaccessible to adversaries. Therefore, spoofing, splicing and replay are detected if a mismatch occurs in the hash comparison. However, this solution may have an unaffordable on-chip memory cost: by considering the common strategy [17, 22, 38] of computing a fingerprint per cache line and assuming 128-bit hashes and 512-bit cache lines, the overhead will be 25% of the memory space to be protect.

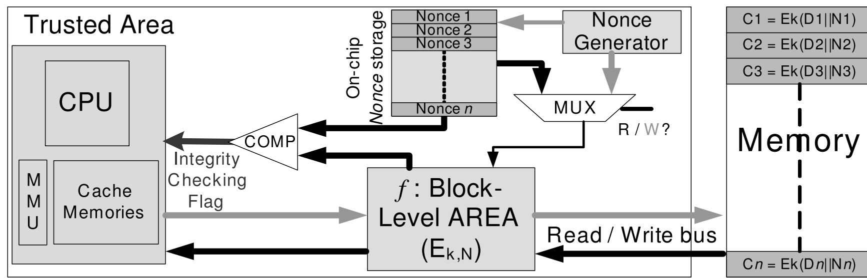
MAC Functions In the second approach (Fig. 6.3-b), the authentication engine embedded on-chip computes a MAC for every data block it writes in the physical memory. The key used in the MAC computation is securely stored on the trusted



(a) Hash functions: Hash$n$ = H(DATA n)



(b) MAC functions: MACn = MAC$_k$(DATA n)



(c) Block-Level AREA: Cn = E$_k$(Dn//Nn)

Write Operation Signals

Read Operation Signals

|| : Concatenation Operator

E$_{k,N}$: Block Encryption under key K and using a Nonce N ($E_{k,N}(D) = E_k(D||N)$)

MAC$_{k,N}$: Message Authentication Code Function under key K and using a Nonce N ($MAC_{k,N}(D) = MAC_k(D||N)$)

H : Hash Function **D** : Data

C : Ciphertext **N** : Nonce

Fig. 6.3 Authentication primitives for memory integrity checking

processor chip such that only the on-chip authentication engine itself is able to compute valid MACs. As a result, the MACs can be stored in untrusted memory because the attacker is unable to compute a valid MAC over a corrupted data block. In addition to the data contained by the block, the pre-image of the MAC function contains a nonce. This enables protection against splicing and replay attacks.

The nonce precludes an attacker from passing off a data block at address A, along with the associated MAC, as a valid (data block, MAC) pair for address B, where $A \neq B$. It also prevents the replay of a (data block, MAC) pair by distinguishing two pairs related to the same address, but written in memory at different points in time. In read operations, the processor loads the data to be read and its corresponding MAC from the physical memory. It checks the integrity of the loaded block by first re-computing a MAC over this block and a copy of the nonce used in the write operation and by then comparing the result with the fetched MAC. However, to ensure the resistance to replay and splicing, the nonce used for MAC re-computation must be genuine. A naive solution to assure this requirement is to store them in the trusted and tamper-evident area, the processor chip. The related on-chip memory overhead is 12.5% in the case of computing a MAC per 512-bit cache line using 64-bit nonces.

6.3.2.2 Integrity Trees

In the previous section, we presented two authentication primitives that can prevent the active attacks described in our threat model. These primitives require storage of reference values—i.e., hashes or nonces—on-chip to thwart replay attacks. They do provide memory authentication but only at a high cost in terms of on-chip memory. If we consider a realistic case of 1 GB of RAM memory, the hash and MAC (with nonce) solutions require respectively at least 256 MB and 128 MB of on-chip memory. These on-chip memory requirements are clearly not affordable even for high end processors. It is thus necessary to "securely" store these reference values off-chip. By securely, we mean that we must be able to ensure their integrity to preclude attacks on the reference values themselves. Several authors suggest applying the authentication primitives recursively on the references. By doing so, a tree structure is formed and only the root of the tree (the reference value obtained in the last iteration of the recursion) needs to be stored on the processor chip, the trusted area. There are two existing tree techniques (in addition to those described in this work):

1. Merkle Tree [10] uses hash functions and is historically the first integrity tree. It was originally introduced by Merkle [30] to authenticate digital signatures and adapted for integrity checking of memory content by Blum et al. [10].
2. PAT (parallelizable authentication tree) [24] overcomes the issue of non-parallelizability of the tree update procedure by using a MAC function.

General Model of Integrity Tree The common philosophy behind integrity trees is splitting the memory space to be protected into M equal size blocks that are the leaf nodes of the balanced A-ary integrity tree (Fig. 6.4). The remaining tree levels are created by recursively applying the authentication primitive f over A-sized groups of memory blocks, until the procedure yields a single node called the root of the tree. The arity of the constructed tree is thus defined by the number of children A a tree node has. The root reflects the current state of the entire memory space; making the root tamper-resistant thus ensures tampering with the memory

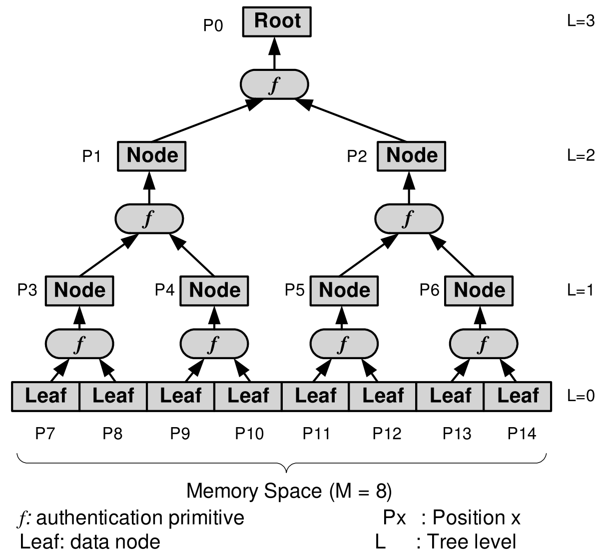


Fig. 6.4 General model of 2-ary integrity tree

space can be detected. How the root is made tamper-resistant depends on the nature of f and is detailed below. Note that the number of checks of the verification of the integrity of one leaf node required depends on the number of iterations of f and thus on the number of blocks M in the memory space. The number of checks corresponds to the number of tree levels L defined by: $L = \lg M$.

6.3.2.3 Execute-Only Memory (*XOM*)

XOM [28] is a security solution from Stanford University. The *XOM* approach, which provides memory protection, is based on a complex key management. The main *XOM* features are data ciphering, data hashing, data partitioning, interruption and context switching protection. Figures 6.5 and 6.6 provide an overview of the *XOM* architecture and mechanisms. All the security primitives are included in the trusted zone. The only security information that is not in the trusted zone are the session keys. That is why *XOM* owns a complex key management to guarantee a secure architecture.

The first version of *XOM* [28] is known to have security holes, like no protection against replay attacks. In [45], the authors extended their proposal and replaced the AES-based ciphering scheme with a system based on OTP to guarantee protection against replay attacks and also to increase the performances of the system. Concerning the global security level of the *XOM* architecture, the attack possibilities are fully dependent on the integrity checking capabilities. To succeed, attackers must be able to pass through the integrity check in order to execute their own program or use their own data. They may exploit some collisions in the hash algorithm used. For example, with MD5 the signature is 128 bits long. If attackers wish to attack the system, they need to find two inputs that will produce the same result with MD5.

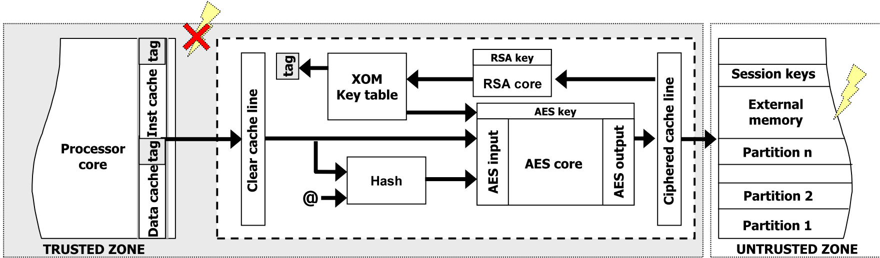


Fig. 6.5 XOM architecture for write request

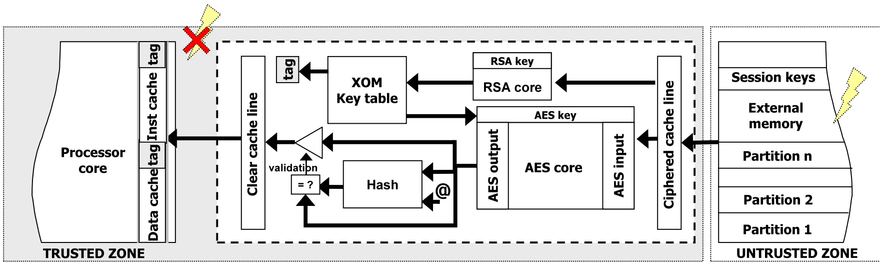


Fig. 6.6 XOM architecture for read request

So they have one chance out of 2^{128} to get the same result. The security level of the *XOM* mainly depends on the hash algorithm used, because SHA-1 could be used for integrity checking. In this case the signature would be 160 bits long and the probability of success would be one out of 2^{160}.

6.3.2.4 AEGIS

AEGIS [37, 40] is an additional memory security solution from Massachusetts Institute of Technology. The confidentiality in the AEGIS solution relies on OTP encryption [39]. This encryption method typically has a small impact on memory latency at the cost of memory space. The solution used by AEGIS for integrity checking is called cached hash tree. This hashing approach is similar to a Merkle tree [30] but to increase the efficiency of the method, some hash tree nodes are stored in a cache memory included in the secure zone. The advantage is that instead of computing all the tree nodes to the root, the system only needs to compute the values until one value from the secured memory is reached. The only weakness in this solution is architecture performance. Architecture performance is fully dependent on the size of the cache memory used to store the secured nodes. Architecture performance also depends on the algorithm used for hashing. As mentioned above, SHA-1 computation requires 80 cycles and MD5 only 64. AEGIS thus appears to be a very complete solution to protect memory and program. The overhead is high in several domains. The silicon area increased by 1.9 [40]. The CPU core is the part that is the

most affected by this overhead. Moreover, all the logic needed to control the specific mechanisms contributes to increasing the area (OTP core and hash core). The global architecture performances depend on parameters like the size of the protected memory and of the cache memory. For security concerns, like XOM, AEGIS depends on the integrity checking capabilities of the hash algorithm used for the Merkle tree. In [40], the authors use SHA-1 which leads to a 160 bit signature. This means that the likelihood of a successful attack is one out of 2^{160}.

6.3.3 *Proposed Memory Authentication Techniques*

In this section, we describe memory authentication techniques proposed recently. We first present a new authenticated encryption mode, the Block Level AREA (Added Redundancy Explicit Authentication), which provides data integrity and confidentiality and a cryptographic engine, called PE-ICE (Parallelized Encryption and Integrity Checking Engine), based on this mode. We then show that the Block Level AREA, like the authentication primitives described in Sect. 6.3.2, have to be integrated into a tree structure called TEC-Tree (Tamper-Evident Counter Tree), in order to avoid excessive overheads in on-chip memory. In the last part of this section, we present another approach named AES-TASC (Time Address Segment Cipher) that allows the use of a security policy to reduce the overhead due to security mechanisms.

6.3.3.1 Block-Level AREA and PE-ICE

Block-Level AREA Block-Level AREA [15, 17] (Fig. 6.3-c) leverages the diffusion property of block encryption to add an integrity checking capability to this type of encryption algorithm. To do so, the AREA (Added Redundancy Explicit Authentication [20]) technique is applied at the block level:

1. Redundant data (an n-bit nonce N) is concatenated to the data D we want to authenticate to form a plaintext block P (where $P = D||N$), ECB (Electronic CodeBook) encryption is performed to generate ciphertext C.
2. Integrity verification is done by the receiver who decrypts the ciphertext block C' to generate plaintext block P', and checks the n-bit redundancy in P', i.e., assuming $P' = (D'||N')$, verifies whether $N = N'$.

Thus, in a *memory write*, the on-chip authentication engine appends an n-bit nonce to the data to be written to memory, encrypts the resulting plaintext block and then writes the ciphertext to memory. The encryption is performed using a key securely stored in the processor chip. In *read operations*, the authentication engine decrypts the block it fetches from memory and checks its integrity by verifying that the last n bits of the resulting plaintext block are equal to the nonce that was inserted during encryption (in the write of the corresponding data). [15, 17] propose a system

on chip (SoC) implementation of this technique for embedded systems. They show that this engine can efficiently protect read only (RO) data of an application (e.g., its code) because RO data are not sensitive to replay attacks; therefore the address of each memory block can be efficiently used as a nonce.[1] However, for Read/Write (RW) data (e.g. stack data) the address is not sufficient to distinguish two data writes at the same address but at two different points in time. To recover the nonce in a read operation while ensuring its integrity, [15, 17] propose storing the nonce on chip. They evaluate the corresponding overhead at between 25% and 50% depending on the block encryption algorithm implemented.

PE-ICE A Parallelized Encryption and Integrity Checking Engine, PE-ICE was designed [15, 17], based on the block level AREA technique to encrypt and authenticate off-chip memory. However, to avoid re-encryption of the whole memory when the nonce reaches its limit (e.g., a counter that rolls over), we propose to replace it with the chunk address concatenated with a random number. For each memory block processed by PE-ICE, a copy of the enrolled random value is kept on chip to make it tamper resistant and secret. In the following, a *PE-ICE configuration* is defined as an implementation of PE-ICE with a given block cipher. A PE-ICE configuration is denoted PE-ICE-*bw* where *bw* is the bit width of the block processed by the underlying block cipher. In this section we first describe a PE-ICE configuration. Then we evaluate the performances of several PE-ICE configurations at runtime.

PE-ICE-160—A PE-ICE Configuration. The Rijndael algorithm is the block cipher that won the NIST contest for a new block encryption standard. The related standard is called AES [2] (Advanced Encryption Standard). AES processes 128-bit blocks and enrolls 128, 192 or 256-bit keys. However, the original Rijndael [11] block cipher supports any key and block sizes that are a multiple of 32 between 128 and 256. This leads to several configurations for PE-ICE based on this block cipher. We studied three of them, PE-ICE-128, PE-ICE-160 and PE-ICE-192, that use the Rijndael algorithm processing respectively 128-bit (AES), 160-bit (Rijn-160) and 192-bit (Rijn-192) blocks. For the sake of clarity, we only detail PE-ICE-160 configuration in this chapter; for a description of the other configurations, the reader is referred to [15].

PE-ICE shifts the physical address by inserting tags between payloads. This shift must be transparent for the CPU, thus PE-ICE handles the address translation (Fig. 6.7).

Memory Consumption The amount of memory consumed by PE-ICE depends on the tag storage of the off chip memory and on the storage of the reference random values for the on chip memory. The off chip memory overhead is defined by the ratio between the tag and the payload bit widths. For PE-ICE-160, the off chip memory overhead is 25%. The on chip memory overhead is defined by the ratio of the bit-length of a random value used to protect an RW chunk against replay to

[1] Note that the choice of the data address as nonce also prevents spoofing and splicing attacks on RO data when MAC functions are used as authentication primitives.

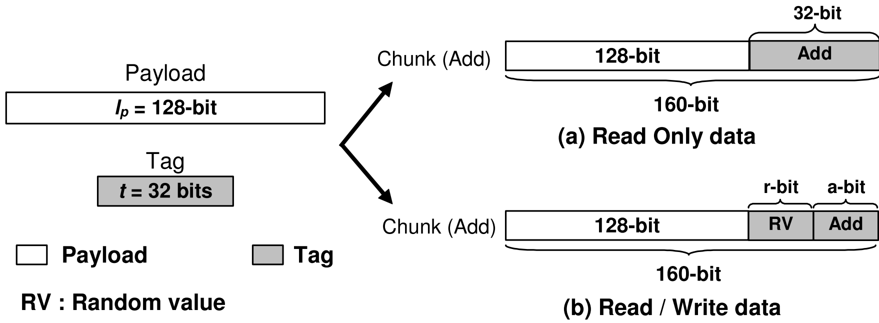


Fig. 6.7 Layout of a PE-ICE-160 chunk before encryption

the bit length of the corresponding protected payload. For PE-ICE-160, the on chip memory overhead is respectively 6.25% and 25% depending if it is 8 bits or 32 bits long. As we show in Sect. 6.3.3.2, we proposed in [15, 18] a scheme to reduce this overhead.

The cost of data authentication in PE-ICE can be evaluated by its overhead compared to AES-ECB encryption. On average this cost is 22%. This latency overhead is partially due to the increase in the intrinsic latency of the underlying block cipher. The hardware cost of PE-ICE-160 and of the AES-ECB is approximately 80 Kgates. At no additional hardware cost and with a low latency overhead, we showed that PE-ICE:

1. Strengthens AES-ECB encryption—the tag inserted before encryption prevents an adversary from detecting when the same data is transferred twice by monitoring bus transactions.
2. Provides data authentication in addition to data confidentiality.

Performance Evaluation *Results* Eight benchmarks [1] designed for embedded systems were used in this evaluation, running on an ARM processor core. The simulation results for the base platform serve as the reference and are shown in IPC (instructions per cycle) in Fig. 6.8 for two different data cache and instruction cache sizes (4 KB and 128 KB). We observed that the performance slowdown was mainly related to the data cache miss rate, see Fig. 6.9.

Figure 6.10 shows the simulation results of the platforms emulating the AES-ECB engine, PE-ICE-128, PE-ICE-160 and PE-ICE-192, in IPC normalized to the performance of the base platform. The AES-ECB engine chart clearly shows that the overhead of PE-ICE is mainly due to encryption; in the worst case it is 50% (CJPEG–4 KB) and 31.5% and of 14.3% on average for a 4 KB and 128 KB data cache, respectively. This quite significant timing performance cost can be dramatically reduced by using a wider processor-memory bus (e.g. 64 bits) and by running the encryption algorithm at its maximum frequency. We evaluated the implementation of PE-ICE with several block ciphers and showed that it provides data integrity in addition to data confidentiality with negligible hardware cost and performance

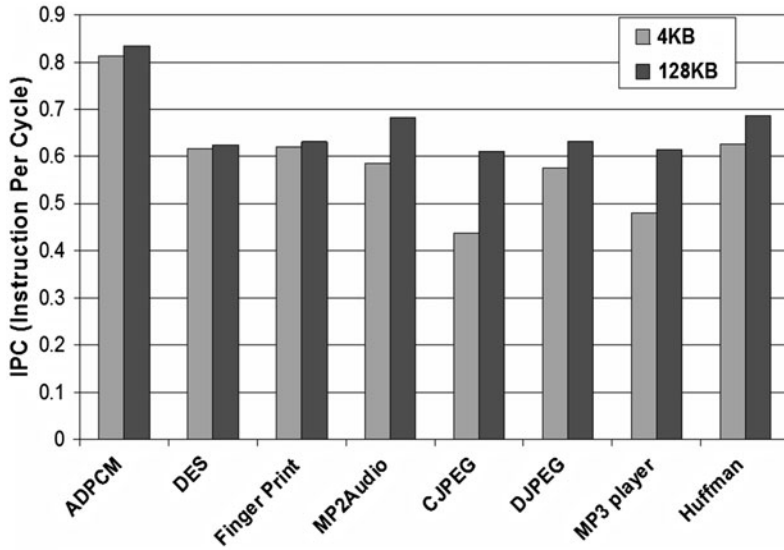


Fig. 6.8 Simulation results of the base platform for two different data cache sizes (4 KB and 128 KB) and two different instruction cache sizes (4 KB and 128 KB)

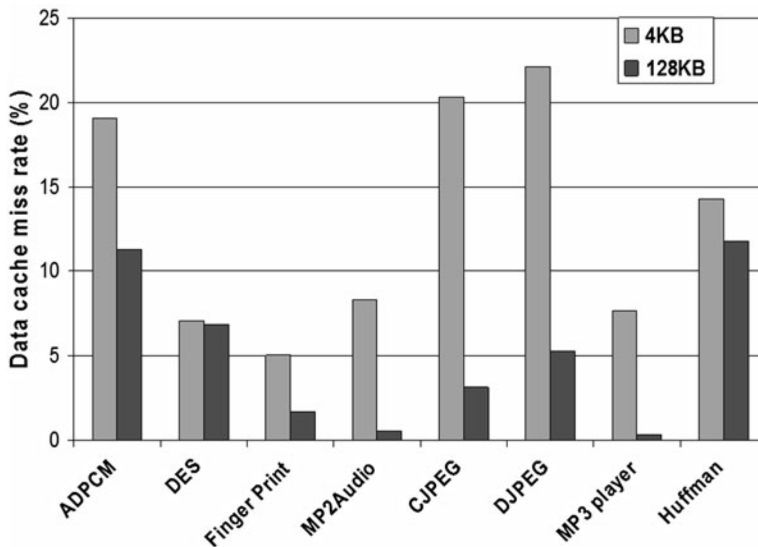
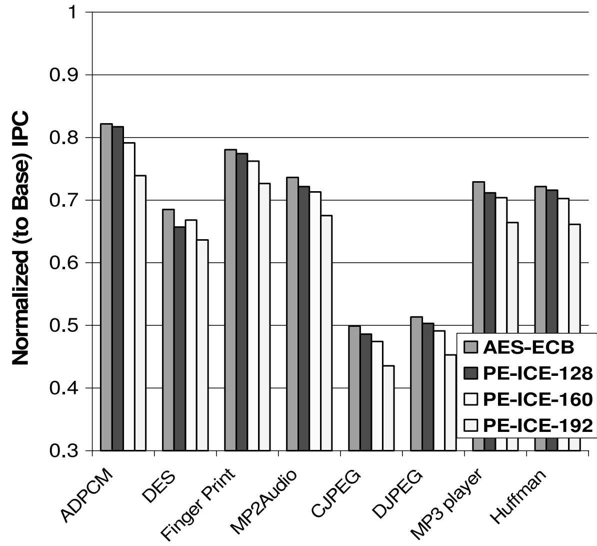


Fig. 6.9 Data cache miss rate for the set of benchmarks used for performance evaluation

overhead compared to standard encryption. In [15, 16], we also showed that PE-ICE is more efficient than the conventional approach in ensuring data confidentiality and integrity. The conventional approach is called generic composition and consists in



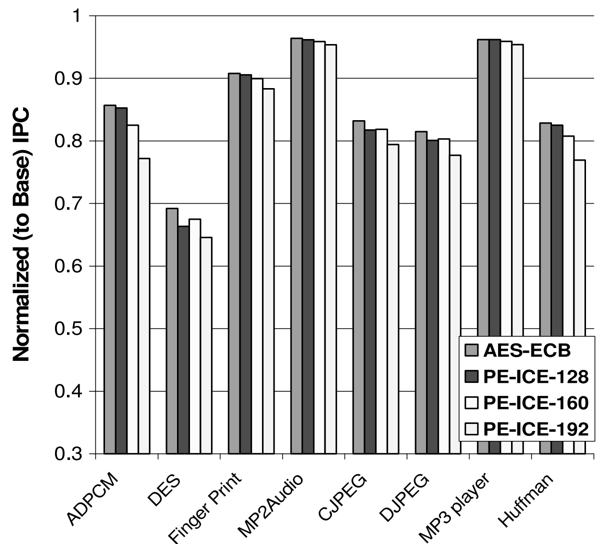


Fig. 6.10 Run time overhead of AES-ECB encryption and of PE-ICE configurations for two data cache sizes (4 KB–128 KB)

chaining encryption with authentication performed with a message authentication code algorithm. We showed that a generic composition scheme can require 50% more hardware resources than PE-ICE and has an 18% overhead compared to an encryption only scheme. In order to decrease the on chip memory overhead and be sure to prevent replay attacks, we propose to build a tree that uses the block level

AREA as authentication primitive. In the next section, we describes the TEC-Tree (Tamper-Evident Counter Tree) [18].[2]

6.3.3.2 The Tamper-Evident Counter Tree (TEC-Tree)

In the TEC-Tree [18] the authentication primitive f is the Block-level AREA (Fig. 6.4). Thus, the authentication primitive tags its input with a nonce N before ciphering it with a block encryption algorithm in ECB mode and a secret key K kept on-chip. The block level AREA is first applied to the memory blocks to be stored off chip, and then recursively over A-sized groups of nonces used in the last iteration of the recursion. The resulting ciphered blocks are stored in external memory and the nonce used in the ciphering of the last block created—i.e., the root of the TEC-Tree—is kept on chip, making the root tamper resistant. Indeed, without the key, an adversary cannot create a tree node and without the on chip root nonce he cannot replay the tree root. During verification of a data block D, D's branch is brought on-chip and decrypted. The integrity of D is validated if:

- Each decrypted node bears a tag equal to the nonce found in the payload of the node in the tree level immediately above;
- The nonce obtained by decrypting the highest level node matches the on chip nonce.

The tree update procedure consists in:

- Loading D's branch decrypting nodes,
- Updating nonces,
- Re-encrypting nodes.

TEC-Tree authentication and update procedures are both parallelizable because f operates on independently generated inputs: the nonces. The distinctive characteristic of TEC-Tree is that it allows for data confidentiality. Indeed, as its authentication primitive is based on a block encryption function, the application of this primitive on the leaf nodes (data) encrypts them. The memory overhead[3] MO_{TEC} of TEC-Tree [18] is:

$$MO_{TEC} = \frac{2}{A-1}.$$

[2]TEC-Tree uses nonce in its design as redundancy for the block level AREA techniques. In [15], we first proposed to build a tree—called PRV-Tree, for PE-ICE protected Random Value Tree—similar to TEC-Tree except that it uses random numbers instead of nonces. The purpose of the PRV-Tree is to decrease the probability for an adversary of succeeding a replay by increasing the length of the random number while limiting the on chip memory overhead to the storage of a single random number (the root of PRV-Tree).

[3][18] give a different formula for their memory overhead because they consider ways to optimize it (e.g. the use of the address in the constitution of the nonce). For the sake of clarity, we give a simplified formula of the TEC-Tree memory overhead by considering that the whole nonce is made of a counter value.

Table 6.1 Architectural Parameters for Simulation

	Merkle Tree	PAT (Parallelizable Authentication Tree)	TEC-Tree (Tamper-Evident Counter Tree)
Splicing, Spoofing Replay resistance	Yes	Yes	Yes
Parallelizability	Tree Authentication only	Tree Authentication and Update	Tree Authentication and Update
Data Confidentiality	No	No	Yes
Memory Overhead	$1/(A-1)$	$3/2(A-1)$	$2/(A-1)$

Comparison with Existing Trees Table 6.1 sums up the properties of the existing integrity trees. PAT and TEC-Tree are both parallelizable for tree authentication and update procedures while preventing all the attacks described in the state of the art. TEC-Tree additionally provides data confidentiality. However, TEC-Tree and PAT also have a higher off chip memory overhead than Merkle Tree, in particular because they require storage of additional meta-data, the nonces.

6.3.3.3 AES-TASC

Memory Security Architecture The AES-TASC (Time Address Segment Cipher) approach, shown in Fig. 6.11, relies on a hardware security core (HSC) fashioned from FPGA logic and embedded memory that is able to manage different security levels according to the data address received from the processor. A small lookup table (the security memory map or SMM) is included in the core to store the security level of memory segments accessed by tasks. Three security levels are possible for each memory segment: confidentiality only, confidentiality and integrity, or no security. The implementation of the security policy in the SMM is independent of the processor and associated operating system. The configuration of the SMM and the rest of the core is contained in the encrypted FPGA bitstream. The isolation of the SMM makes it secure against software modifications at the expense of software level flexibility. New multi-task applications require a new FPGA bitstream to achieve a new memory security protocol.

Security Level Management The increased use of soft and hard-core processors in FPGAs has facilitated the use of operating systems in FPGA based systems. The use of an OS provides a natural partitioning of the application code and data. In Fig. 6.11, the application instructions and stack data of Task 1 have different security levels. In this case, the application designer may wish to keep task processor data secure to prevent copying. The application code may be less sensitive, consequently eliminating the need for security. Our approach is designed to be used in conjunction with a memory management unit (MMU). This unit ensures that a task will not read or write memory segments that are not associated with it, which

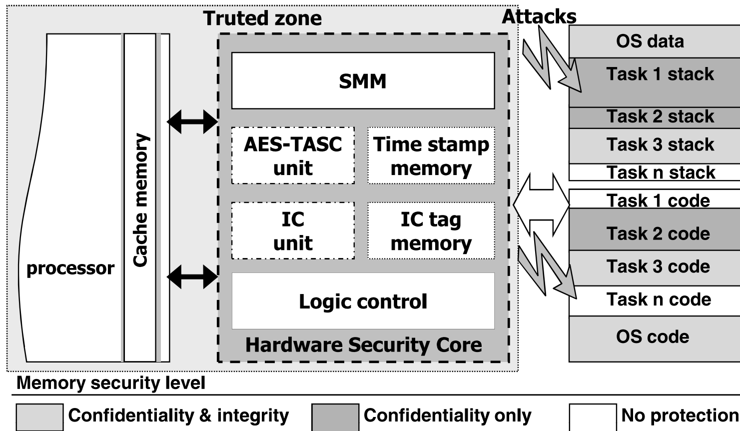


Fig. 6.11 Overview of the memory security system

creates a security risk if the security levels differ. The availability of configurable security levels has an advantage over requiring all memory to perform at the highest security level of confidentiality and integrity checking. The amount of on chip memory required to store tags for integrity checking can be reduced if only external memory that requires security is protected. In addition, the latency and dynamic power of unprotected memory accesses is minimized since unneeded security processing is avoided. FPGA reconfigurability enables optimization of the required on chip storage and modification via a new bitstream.

Memory Security Core Architecture Confidentiality in our system is similar to the AES-based encryption scheme called Binary Additive Stream Cipher [29]. Rather than encrypting write data directly, our approach first generates a *keystream* using AES that operates using a secret key stored in the FPGA bitstream. In our implementation, a time stamp value, the data address, and the segment ID of the write data are used as input to an AES encryption circuit to generate the keystream. These parameters are required to protect the system against spoofing, replay and reallocation attacks. This keystream is then XORed with the data to generate ciphertext that can be transferred outside the FPGA. The time stamp is incremented during each cache line write. The same segment ID is used for all cache lines belonging to a particular application segment (i.e. same level of protection). The advantage of the AES-TASC (AES in time address segment counter mode) approach over direct data encryption of the write data can be seen during data reads. Keystream generation can start immediately after the read address is known for read accesses. After the data is retrieved, a simple, fast XOR operation is needed to recover the plaintext. If direct data encryption is used, the decryption process would require many FPGA cycles after the encrypted data arrives at the FPGA. Thus, the use of AES-TASC significantly reduces the read latency of security. One limitation of this approach is the need to store the time stamp (TS) values for each data value (usually a cache

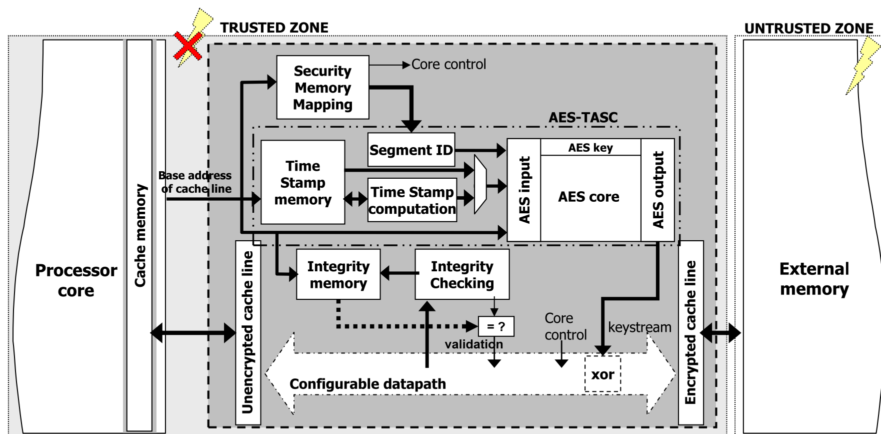


Fig. 6.12 Hardware Security Core Architecture

line) on chip so it can be used later to check data reads. A high level view of the placement of security blocks is given in Fig. 6.12.

The percentage performance loss due to our security scheme is higher for systems that include smaller caches. This is to be expected, since smaller caches are likely to have a larger number of memory accesses, increasing the average fetch latency. Performances are directly related to the designers security policy. A very conservative approach in terms of security will lead to a larger performance penalty whereas a fine tune security policy will lead to a limited reduction in performance. In practice, the use of programmable protection allows the impact on application performance to be reduced compared to uniform protection. An average of 12% performance reduction was observed for a set of applications from multimedia and communication domains. This result compares favorably with other cryptographic approaches where up to 50% performance loss can be observed. The same remarks apply to the memory overhead, which is directly impacted by time stamp and integrity tag values that consume secured on chip embedded memory and energy efficiency. Experiments that have been conducted have shown the benefit of a flexible approach to security.

6.4 Secure Bitstream Management

FPGAs are very specific silicon devices. Like microprocessors, they can be programmed and reprogrammed, but in the case of programmable logic devices the architecture of the chip is changed according to a binary file called a bitstream. In contrast, microprocessors are only programmed with instructions. If this hardware reconfiguration capability is attractive for low-volume applications and opens a wide range of opportunities for engineers or researchers, it can be a drawback in the field of security sensitive applications. Therefore in the following section, we analyze the impact of this feature on the robustness of a secure system.

6.4.1 Threat Model

The first threat to a bitstream is an attacker who succeeds in retrieving it from the system. The easiest way is to use the read-back capabilities of most FPGAs. This function, generally used for debugging, allows the bitstream to be extracted from an FPGA device at run time. Of course, this feature has to be disabled in a secure context, but this not sufficient. For low cost SRAM FPGAs, bitstream retrieval is very simple, even without a read-back mechanism. Indeed, the bitstream is stored in an external non-volatile memory, so the attacker can probe the data line between the FPGA device and this memory in order to access the bitstream. This threat does not exist for non-volatile FPGAs because configuration data are stored inside the device, so only intrusive attacks are possible. Bitstream encryption mechanisms are available for most advanced FPGAs. A secret encryption key is stored inside the programmable device, while for volatile FPGAs, an external battery is used to store key values. Thus the device accepts an encrypted bitstream and uses its dedicated decryption engine to obtain deciphered data. Attackers cannot decrypt the bitstream without the secret key. With this feature, attackers have to discover the secret key using intrusive attacks to recover bitstream data, for example. In FPGA devices with embedded configuration memory, if the designer has taken the trouble to prevent read-back, bitstream retrieval from the device is only possible using invasive attacks. However, if the design is intended to be updated during its lifetime, the bitstream will travel through insecure channels such as public networks, and the bitstream consequently needs to be protected using cryptographic mechanisms even for non-volatile FPGAs.

When an attacker retrieves a plaintext FPGA bitstream, many data are accessible and may be critical. The first threat is bitstream reverse engineering, some software projects claim to succeed in Xilinx bitstream reverse engineering [32]. Thus attackers could access all data stored in the FPGA architecture, possibly secret cryptographic keys or non-public cryptographic algorithms. Attackers can also inject their own bitstream into the programmable device thereby rendering reverse engineering unnecessary, instead they can simply create a dummy system. For instance if the FPGA chip is used to encrypt data written to a hard disk drive, the attacker could build a system that does not cipher data. Another threat is fault injection into the bitstream [7]. Small modifications can deeply modify the system without the knowledge of the architecture. The last FPGA drawback is cloning. An attacker in possession of the configuration data can *clone* the device ignoring potential intellectual property rights. This is not a real security weakness, but rather an industrial threat that is specific to programmable devices. A bitstream is the image of the FPGA underlying architecture, it is similar to the ASIC “layout”, for that reason, configuration data have to be well protected. In the following section, we provide a more detailed state of the art on these aspects.

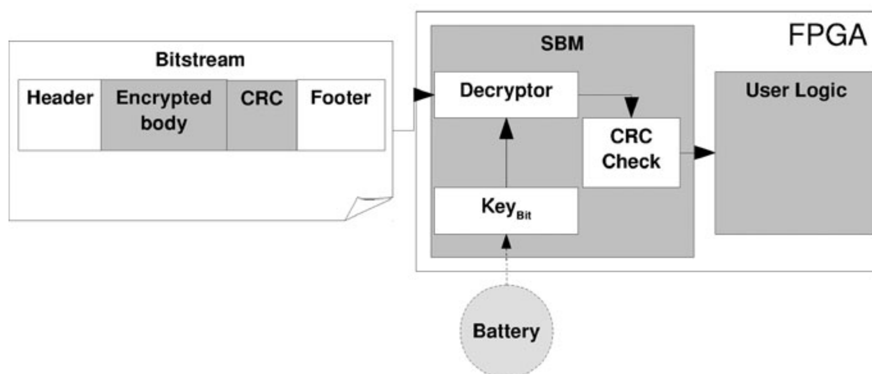


Fig. 6.13 Overview of the static logic needed to enable bitstream encryption

6.4.2 State of the Art

FPGA vendors are generally sensitive to the security issues of FPGA based designs; hence they provide tools and mechanisms that allow system designers to ensure an acceptable level of security according to their requirements. Below we review currently available security features in main FPGA devices. However, we only consider the leading FPGA vendors (Xilinx, Altera, Lattice and Actel).

6.4.2.1 Bitstream Encryption

Most often, the first security feature offered by FPGA vendors is bitstream encryption. This is very attractive even for applications that have no security concerns. Bitstream encryption was originally proposed to protect the confidentiality of any intellectual property included in the design. Without this precaution anyone gaining access to the bitstream can at least *clone* the design in another FPGA.

To allow system designers to encrypt bitstream and thus the FPGA chip to decrypt it, FPGA vendors must add to their devices a decryption engine, a non-volatile key register and control logic that manages configuration (see Fig. 6.13). During the manufacturing stage of a product, the system designer introduces a random and secret key in the non-volatile memory. Then each time the FPGA device reloads its configuration, the decryption engine decrypts the bitstream that comes from the configuration port and transmits the result to the configuration logic. Thanks to this mechanism, bitstream confidentiality is ensured. Today many FPGA devices include a decryption engine to protect the intellectual property of their customers. Currently, Xilinx and Altera provide encrypted bitstream solutions for high end FPGA chips in the Virtex and Stratix families respectively [8, 43]. On the other hand, Lattice and Actel provide solutions even for low cost devices [6, 27]. The latest Xilinx low cost devices also include bitstream encryption but only for large FPGA matrices

(Spartan 6, Virtex 6). Since volatile FPGA vendors generally choose to avoid expensive Flash process, the FPGAs require an additional external battery to store the bitstream decryption key value. However, the latest Xilinx devices include fuses that allow the user to store the cryptographic key without an external battery. Bitstream encryption is a non-negligible cost for FPGA vendors; they need to add a decryption engine that provides a reasonable throughput to be sure the configuration time is acceptable. However for Flash-based FPGAs such as Actel, the configuration time is less critical since they do not need to decrypt the bitstream at each power up. The static logic includes the decryption engine and the key memory, so the security level of the bitstream protection is determined by the FPGA vendor. For instance, if the FPGA vendor does not take side channel attacks on configuration logic into account, a lot of mechanisms that can be built on bitstream trustworthiness will be useless.

6.4.2.2 Bitstream Integrity Checking

If a message is only encrypted, nothing attests to its integrity, because classical encryption does not ensure it. Therefore, in addition to encryption, FPGA vendors provide mechanisms to check bitstream integrity at each configuration. In the absence of any integrity checking mechanisms, attackers can easily modify an encrypted bitstream. However, since they do not know the decryption key value, they cannot predict the effect of the modification. Two main effects can be obtained: (i) an incorrect bitstream loaded in the FPGA chip could damage the chip and the whole FPGA based system; (ii) the bitstream does not damage the FPGA device but modifies the behavior of a part or of the whole design in an uncontrollable way. In the latter case, attackers can target a particular area of the bitstream (and therefore of the design), for instance they could tamper with the embedded RNG to generate weak keys. So FPGA vendors have to include mechanisms that are able to detect bitstream modification. This feature has to be provided by FPGA manufacturers, not by system designers, at least during the bitstream loading time. Most FPGAs use 32- or 16-bit cyclic redundancy code (CRC) combined with AES-CBC encryption. However, the primary aim of CRC is to detect and correct errors during bitstream transmission. As this is not a cryptographically secure mechanism, the probability of loading an unauthentic bitstream is not negligible, even if the bitstream is encrypted. In the FPGA context, a corrupted bitstream can destroy the device by causing short cuts. This threat can be considered as a fault injected into the configuration logic. The latest Actel devices [6] and series 6 Xilinx FPGAs (Spartan and Virtex 6) include cryptographically secure mechanisms to ensure integrity. Actel claims to implement a real cryptographic integrity checking mechanism by using an AES block for both decryption (CBC mode) and integrity checking (AES-based MAC), a message authentication code is included in the bitstream and the configuration logic checks it before starting the design (Fig. 6.14). For Actel, configuration time is not critical since the bitstream is loaded only once. Series 6 Xilinx FPGAs [44] include an integrity checking engine in the static logic using the SHA-1 algorithm. This engine checks bitstream trustworthiness while the AES engine is decrypting it, in order to

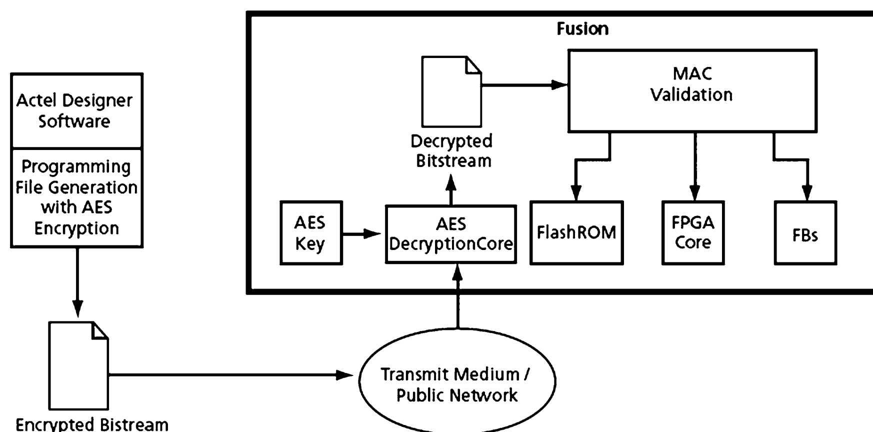


Fig. 6.14 Actel's integrity bitstream mechanism

save configuration time. This mechanism is based on an HMAC algorithm that uses SHA cryptosystem with a 256-bit key.

Cryptographic mechanisms that ensure both integrity and confidentiality already exist, they are known as Authenticated Encryption (AE) algorithms. Therefore, the academic literature proposes secure schemes that are suitable even for volatile FPGAs. These solutions can be implemented using a block cipher in a particular mode of operation, [35] proposes to use the EAX mode and provides time and area overhead which appear to be suitable for volatile FPGAs. Similarly, [12] proposes to use two AES cores and [26] suggests the AES-GCM mode.

6.4.2.3 Locking Reprogramming

In most SRAM FPGAs, there are no non-volatile elements and so the bitstream protection key memory is powered by an external battery. The corollary is that an attacker with physical access to the system can remove the battery and erase the key. Moreover, even when the key is initialized and the battery present, the FPGA chip will accept unencrypted bitstream that an attacker can easily generate. The effect on security is that attacker can load a malicious bitstream. One solution is to add a design authentication key in the bitstream used by an external trusted party to authenticate the design. The other solution, which is more convenient since it does not require an external party, is to lock the FPGA device to only accept encrypted bitstream. Another extreme solution is to prevent any further configuration.

Actel has provided this locking feature for a long time, obviously in anti-fuse based FPGAs since these are programmable only once, but also in the ProASIC 3 and Fusion families [4]. This mechanism, called Flash Lock, acts like a password, the system designer sets the key (password) in the FPGA chip then the SD can choose among different security parameters. The designer can prevent the FPGA

bitstream being read or written without the proper password, and once unlocked, the bitstream can be sent in plaintext or only encrypted depending on SD policy. If the SD wants to allow further remote updating without revealing the password key, he can configure the FPGA device to accept only encrypted bitstream, in which case the bitstream can be sent over an untrusted network; in this case FPGA chip is not locked (i.e. it accepts encrypted bitstreams without the password).

A more drastic solution provided by Actel in their ProASIC and Fusion families is to disable any further reprogramming even with a password key. Obviously any further remote update is then impossible. However this solution can be interesting for very sensitive applications that need to avoid cryptography usage when possible, mainly because secret keys can be retrieved in many different ways (such as physical attacks or even social engineering).

Finally, the latest Xilinx and Altera FPGAs include a new feature that enables system designers to lock the reprogramming of the device. To avoid the cost of Flash technology, Xilinx uses eFuse technology to implement this feature. The solution is flexible since a battery powered key can still be used depending on application requirements, if fast key erasure is needed, battery solutions are best. In order to use eFuse memory, the system designer has first to program a key in this memory. At this point he can still read and write the bitstream key in order to check the value, and if the value is valid, can lock the FPGA device by programming another eFuse register that disables any further read and write access to the key register; moreover one bit in this control register allows the SD to constrain the FPGA chip to only load bitstreams that are encrypted with the eFuse key. In this way, attackers who do not know the bitstream key cannot load a malicious design in the FPGA.

6.4.3 Remote Reconfiguration

Remote updating of hardware systems is a convenient service enabled by FPGA-based systems. This service is essential in applications like space-based FPGA systems or set top boxes. However, remote update schemes have to consider replay attacks, as described in Fig. 6.15. This attack consists in simply recording a bitstream corresponding to a version of the FPGA design, and then replaying it later to reintroduce security breaches that have been corrected by the system designer in the latest bitstream version. According to current bitstream encryption mechanisms, the attack is possible even if the bitstream is encrypted.

6.4.3.1 Related Work

Few academic or industrial studies have addressed this attack. In [35] and [12] the keyed hash is computed only over the received bitstream. As previously mentioned, most current FPGA vendors do not provide strong bitstream integrity checking mechanisms, and none have addressed the replay attack issue. In all cases, the

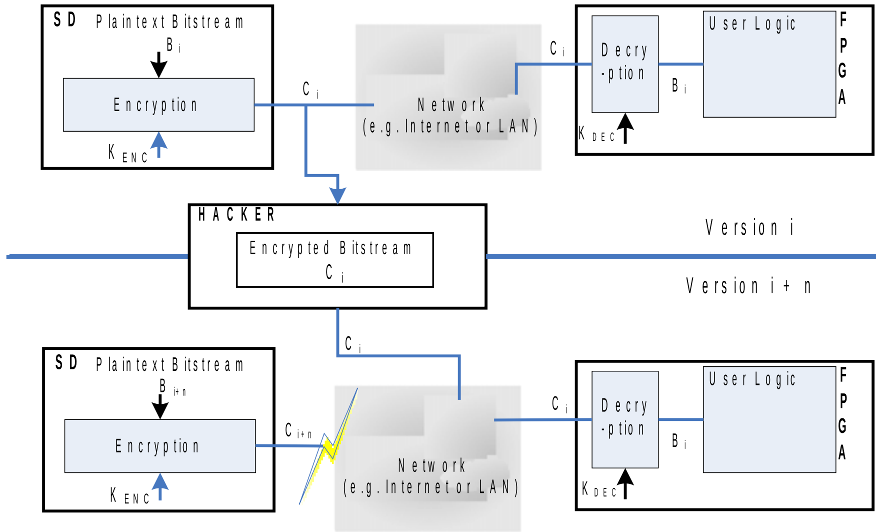


Fig. 6.15 Replay attack of an FPGA bitstream

FPGA configuration logic is unable to distinguish between different (keyed hash, bitstream) pairs legitimately generated by the SD in the past. As a result, an adversary who replays a bitstream and its keyed hash will succeed in his attacks. A replay is particularly dangerous for system security because even if bitstream encryption is enabled by the FPGA's static logic, it allows for system downgrading. The purpose of a bitstream update triggered by the SD may be to remove system vulnerabilities, so by replaying the previous FPGA configuration, an attacker can effectively preclude security-critical updates. In the following sections, we provide solutions to counter such an attack.

Existing bitstream integrity solutions prevent spoofing of the bitstream but are unable to prevent a replay attack and the bitstream is consequently exposed to system downgrade threats. In [6, 12, 35] the keyed hash is computed only over the received bitstream. For that reason, the FPGA configuration logic is unable to distinguish between different (keyed hash, bitstream) pairs legitimately generated by the SD in the past. As a result, an adversary who replays a bitstream and its keyed hash will succeed in his attack. Recent work on reconfigurable trusted computing proposes FPGA-based implementations of the Trusted Platform Module (TPM) [14]. This work leverages TPM functionalities to provide a secure update of the FPGA bitstream. In addition, [36] proposes an implementation of TPM on current FPGA technologies that does not require bitstream encryption. [36] assumes that reverse engineering of the bitstream is too difficult to achieve and relies on a trusted external non-volatile memory.

[13] introduces the issue of bitstream replay attacks. The author suggests two different avenues of research to solve the problem. The first requires the SD to implement additional security features in the user logic to send authenticated messages

to a trusted authority, who then attests to the version of the running FPGA configuration. This suggestion is explored in this section (see 6.4.3.2). While this approach fits the general reasoning of FPGA vendors, i.e., that an SD who wants a specific functionality should pay for it himself by developing it in the user logic rather than it being hardwired and supplied to everyone who buys FPGAs, we do not believe it is the best solution. Firstly, it requires the implementation of a cryptographic engine in the user logic to set up an authentication channel and a challenge-response protocol with the SD. This is not an efficient solution, since the one already provided in secure FPGAs for bitstream encryption could also be used for that purpose if the scheme was implemented in the static logic (i.e. as part of the bitstream loading logic). Secondly, the SD in need of the replay-resistant feature for his design is not necessarily a security expert; hence, custom implementation of a replay-resistant system can result in unreliable solutions. Consequently, we suggest mechanisms that require less security expertise and allow the SD to lock the FPGA based system to a particular bitstream version.

[13] also suggests a second approach that use nonces in the authentication process to ensure the freshness of the bitstream. [13] does not, however, define the architecture and protocols that would be necessary to build a replay-resistant bitstream authentication mechanism. As this solution is developed and evaluated in [9], we do not present it here and interested readers should refer to this paper for further information.

Based on these studies, it is clear that system designers lack efficient solutions to prevent bitstream downgrade and more generally to ensure bitstream security. Therefore, in the following sections we propose two possible solutions along with their advantages, drawbacks and limitations.

6.4.3.2 Contribution to Remote Configuration Security (1): FPGA Polling

Principle The first solution can be applied to any FPGA device that supports bitstream encryption and integrity checking. Bitstream encryption is used to hide a value related to the bitstream version; each new release of the bitstream will contain a new value, called TAG. In addition to this TAG, the encrypted bitstream contains a unique value that aims at identifying a particular device, each device contains its own identifier, called K_{ID}. Figure 6.16 describes this layout.

For security reasons, inner FPGA logic (user logic) cannot access the bitstream encryption key. Thus designers cannot use this key directly to perform cryptographic operations. However, designers can hide secrets in the encrypted configuration bitstream. These values may be secret or private keys stored in user logic such as look up tables or embedded RAM blocks.

As suggested by Saar Drimer in [12] these secrets can be used to authenticate the bitstream. If one bitstream decryption key is kept secret by the system designer for each FPGA chip, only the FPGA chip containing the relevant key can decrypt the authentication secret. This allows authentication of the design running on the FPGA. The key can also be used to check if the version of the secret corresponds to

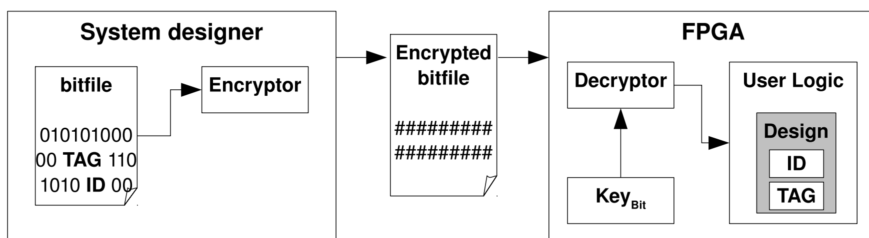


Fig. 6.16 Key equipment needed for Solution 1

the current genuine version of the design. These two values are used in the proposed protocol to ensure authentication of both the transaction and the current bitstream version. Since this solution can be applied in any FPGA chip that includes a bitstream encryption feature, we assume that like in most SRAM-based FPGAs, there is no internal non-volatile memory and that the FPGA design cannot store the current version number because it does not have an embedded trusted reference. In this solution, the SD needs to implement a cryptographic engine and glue logic able to perform authentication with an external trusted party (TP) to authenticate the bitstream version. The proposed solution consists in regular polling of the FPGA bitstream version by the external trusted party who also knows the TAG and the K_{ID} values. The trusted party could be the system designer himself but it could also be an external processor or system. The trusted party regularly sends a nonce to the FPGA chip that replies with an encrypted value. This encrypted value is the concatenation of the nonce with the TAG ciphered with K_{ID}. The trusted party can check that the bitstream version is valid by checking the TAG value, but also that the response is not a replay using the nonce. These two checks are made using K_{ID} to decipher the response; they are thus authenticated according to the FPGA device concerned.

In the following, we consider that the FPGA device and the trusted party are located on the same board, that all the considerations of this solution can be applied to a remote trusted party (access through Internet for instance) and since all the communications are tagged, authenticated and encrypted, that they are not subject to attacks. How the SD securely updates the trusted party is beyond the scope of this book, although he could a secure microprocessor able to securely store key materials, for instance, but in this case, would need to include a mechanism in the device to avoid replay attacks on trusted party updates.

Platform Initialization First, the SD initializes the platform by setting a secret key to decrypt the bitstream. To do so, he uses the appropriate tools and mechanisms provided by FPGA vendors. This key must remain secret. Next, he initializes the trusted party that is located on the FPGA based device, loads a TAG that does not need to be secret (0 for instance) and a key K_{ID} that will be used to ensure platform authentication. He can optionally load the initial bitstream in the FPGA configuration memory (which could be inside the FPGA chip in the case of non-volatile FPGAs).

Remote Update Process Detail Once the platform is initialized and the bitstream is loaded in the FPGA user logic, polling of the trusted party (TP) can begin. The goal of polling is to check that the bitstream version is genuine.

The process described here requires that an encryption engine is embedded in the FPGA device that will be used to authenticate the current version of the bitstream used. The encryption algorithm can be of any kind (i.e. stream cipher, block cipher, asymmetric) but a symmetric block cipher may be best because these algorithms are compact and easy to implement in FPGAs. A standard AES engine can be used for this purpose.

When the SD wants to provide a new version of his FPGA design, he has to remotely modify the FPGA bitstream and also the trusted party TAG value. However, he can send the bitstream securely since its integrity and confidentiality are ensured by the FPGA device. Alternatively, he can send the new TAG version to the TP using a secure network connection (for instance SSL). Once the bitstream is loaded into the FPGA chip and the TAG is changed in the TP, the polling process can (re)start.

Assuming that the AES algorithm is used, the proposed polling mechanism can be described as follows:

1. The trusted party sends a nonce to the FPGA device. This nonce will be used later to check that the FPGA response is genuine, i.e. not a replay of an old FPGA response. The method used to send the nonce is platform-dependent. For instance it could be a serial link between the TP and the FPGA, or if the TP is a remote server, it could be over Internet.
2. The FPGA chip receives the nonce and computes its response. To do so, first it concatenates the nonce and its current TAG value. Then it authenticates this concatenation using its embedded signature engine and its identity key. Since the K_{ID} value is only known by the SD, the FPGA device and the TP, this response cannot be generated by an attacker.
3. Once the answer is computed, the FPGA chip sends the value to the TP, and since the response is encrypted, an attacker has almost no chance of generating a valid response. If an attacker tries to modify this answer, the TP will reject it, which can only result in a denial of service.
4. On reception, the TP decrypts the FPGA response using K_{ID}. The decrypted value is then compared to the concatenation of its own copy of the nonce and its TAG value.
   a. If these values are correct, TP can continue its polling process as the current bitstream version is genuine.
   b. If only the TAG value is different, this means that the FPGA device is running a different bitstream version than the genuine one. TP then applies the system designer policy, which could be to turn off the whole system, or to reload a genuine version in the FPGA configuration memory.
   c. If only the nonce is different, this means an attacker is probably trying to replay an old answer given by the FPGA chip as the genuine version. To do this, he lets the FPGA device be upgraded to the genuine version, then records a set of FPGA response. After this stage, the attacker performs a downgrade of the FPGA bitstream (recorded from a previous remote update), and then

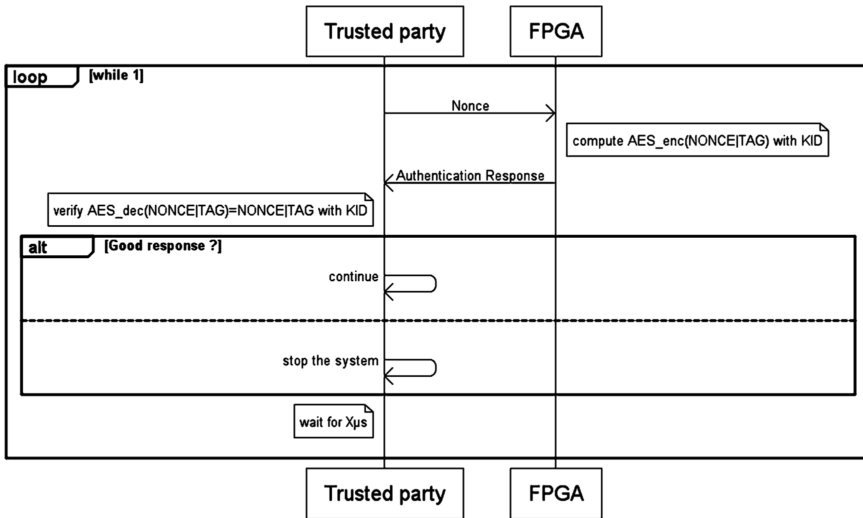


Fig. 6.17 Graphical representation of the protocol

tries to replay the old FPGA answers. The nonce ensures that this attack is not applicable. Once again the TP applies the system designer policy.

d. If both the TAG and the nonce are invalid, the cause of the error is harder to determine, maybe it is simply a transmission error due to the channel between the FPGA chip and the TP. So depending on the SD policy, the trusted party may try to poll the FPGA device one more time discarding this failure.

Figure 6.17 is a simplified graphic representation of the polling procedure.

Security Analysis The main security drawback of this solution is that the K_{ID} value has to be shared between three different entities: the SD, the FPGA chip and the trusted party. This means that attacks are more likely and that more entities have to be trustworthy. Next, the system designer has to implement a decryption engine in the user logic that is subject to physical attacks like any cryptographic algorithm implementation. However, the SD may not be a security expert and could consequently introduce weaknesses in the protocol or in the implementation. The update of the trusted entity must also be realized by the SD and can result in threats in his protocol. The polling frequency has also an impact on system security. Security does not need to be very high to counter an attack between two polls. The attacker first needs to replay an old version. This operation can take a relatively long time (a few milliseconds) depending on the FPGA bitstream loading speed. Next the attacker has to perform the attack. Finally he has to reload the genuine bitstream before the next polling of the TP. Therefore the SD should choose the pulling frequency according to requirements of his particular threat model.

Cost Evaluation The authentication engine that generates responses inside the FPGA may be symmetric or asymmetric. Both allow secure implementation of the

solution, but the symmetric solution is less convenient than the asymmetric one. With a symmetric cipher, if anyone needs to check the key, they need to have this key. The solution can be made with an AES block used as a MAC. Whereas an asymmetric cipher can be checked without compromising its secrecy. In counterpart, asymmetric ciphers are more compute intensive operations. So the choice will depend on the application.

It should be noted that an existing cryptographic engine in the user logic can be reused by the SD to perform authentication responses. The mean performance of this engine will be reduced since some operations will be reserved for the version checking protocol, meaning the application cannot use it at this time. The overhead is directly related to the polling frequency specified by the trusted party.

Assuming that the SD decides to use a dedicated cryptographic engine to implement this solution, the cost could be relatively high; especially if the FPGA device of the application has limited resources. Obviously the asymmetric approach requires much more logic and time to compute the response. But the cost of the symmetric approach is also non-negligible if the AES engine is not reused by the SD. The cost of the additional logic gates needed to implement the protocol is negligible compared to the cost of a hardware cryptographic engine.

Conclusion The proposed solution allows the TP to securely monitor the FPGA bitstream version using existing SRAM based devices. However this is not the ideal, perfectly convenient solution. First it entails a non-negligible cost for the SD. Second, the entity that wants to check the design version has to question the device. The SD must also find a polling frequency that is not too high in order to maintain reasonable performance, and not too low to avoid replay attacks between two polling sequences. Moreover, this process forces the SD to include secure key management in his design. He must manage the bitstream key, embed an identity key and an update TAG inside the encrypted bitstream. All these drawbacks are due to the fact that the FPGA device itself does not check if a bitstream replay attack is underway. Unfortunately, this is the only way to ensure update security for most current FPGAs. The ideal solution would be to lock the FPGA chip to only one bitstream version, and provide a secure solution to the SD to remotely change the current genuine bitstream version. The following section addresses this need.

6.4.3.3 Contribution to Remote Configuration Security (2): Using Embedded Non-volatile Memory

Finding a solution to lock current FPGAs to a bitstream version is not easy, mainly because it requires that the FPGA is able to store a reference of the current bitstream version TAG. It is possible to implement an AES engine and the comparator, as mentioned in our first proposal above. But dealing with non-volatile storage of the TAG is more problematic. User logic is by nature volatile (even on Flash based FPGAs), and the TAG value cannot be stored inside the bitstream since, in the threat model, the attacker could replay an old bitstream. It cannot be stored in an external memory

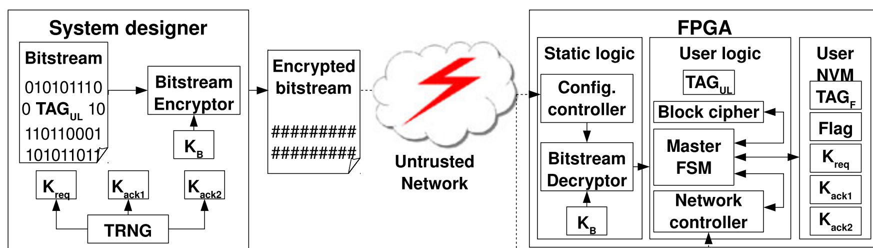


Fig. 6.18 Overview of the equipment needed for the second proposal

either since the attacker could replay the memory content. So the TAG value must be stored in an embedded non-volatile memory. This memory must be protected from external dumping and tampering, and must also be usable by user logic to enable the secure TAG update that will be performed by the FPGA design. Some Actel FPGAs [6] include a feature called Flash ROM. This non-volatile memory can be read from inner logic and can be programmed via JTAG with an encrypted bitstream. However the FROM is not writable from inner logic, so a secure TAG update mechanism cannot be used. In fact, the ideal solution would be to enable the FPGA device to perform the TAG update by itself, although such a mechanism could not use external JTAG since an attacker would be able to replay any communication performed using JTAG. For that reason, a FROM could not be used to enhance solution 1. However, FUSION FPGAs from Actel [5] include a more interesting feature: a user flash memory. This non-volatile memory is accessible from user logic for both read and write operations. In addition, using a secret enables it to be protected against dumping and tampering from external inputs and outputs such as the JTAG port. Thanks to this feature, solutions can be found to lock such an FPGA chip to a particular bitstream version and to remotely modify the version number.

Next we present a minimum but nevertheless secure solution, whose goal is to minimize hardware overhead and key management for the system designer. This is why the solution does not include complex cryptographic algorithms.

Principle The goal of this secure update mechanism is to lock the FPGA to a specific version in order to prevent replay attacks.

Generic Design Overview Figure 6.18 shows the FPGA design that enables a secure remote update of bitstream mechanism to be implemented. The FPGA is composed of three parts. The first part, static logic, is hard-wired and cannot be configured. It contains a deciphering module that protects the confidentiality and integrity of the bitstream, and its key. This key, named K, is only known by the FPGA and the SD. The second part, user logic, can be configured by the SD and contains a bitstream version verification mechanism. It is composed of a finite state machine (FSM) able to manage:

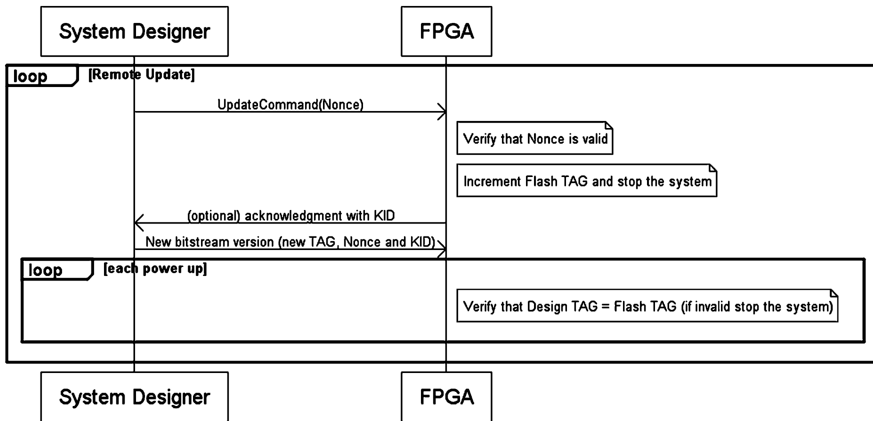


Fig. 6.19 Graphical representation of the protocol

- A network controller to enable the update to be performed remotely.
- A non-volatile memory controller to store a first power up flag, the current bitstream version number and keys shared with the SD.
- A block cipher to ensure mutual authentication of the FPGA and the SD.

The third part, user NVM, contains three keys shared with the SD. They must be unique for each FPGA and are used to encrypt the tag:

- K_{req}: for the Update command.
- K_{ack1}: for the Update command acknowledgment.
- K_{ack2}: for the new bitstream version and acknowledgment of start up on the correct device.

Since the goal is to lock the FPGA to a particular version, the NVM also contains the value indicating the current genuine version. This value, named TAG, can be only incremented by the SD. It will be compared to the tag contained in the user logic, also named TAG_{UL} (refer to Fig. 6.18). Each bitstream version contains its own TAG_{UL}. In practice, it is a constant in the design source code: version zero is tagged with a zero, version one with a one, and so on. The NVM is written the first time from outside FPGA chip in a trusted zone before being locked using the FPGA vendor mechanism [3]. After locking, the NVM can be read and written only from the user logic.

Update Process Figure 6.19 focuses on communications between the SD and the FPGA. It explains the process used to check that the current bitstream version is genuine and to securely implement this non-volatile value for a future update. The update process is described in more detail below.

Update command: This command increments TAG_F to prepare the FPGA to an update. The SD sends the update command containing the tag encrypted with the K_{req} as cipher key to the FPGA. After decryption, the FPGA compares the tag

contained in his own bitstream (TAG_{UL}) and the tag sent by the presumed SD. If they are different, the FPGA continues to work and waits for a new update command. Otherwise it implements the TAG_F and starts to encrypt the new tag with the cipher key K_{ack1}. To inform the SD that the tag increment command has been received, the FPGA sends the result of the encryption. The design is stopped.

Download a new bitstream version: The SD sends the new ciphered bitstream, with its MAC, to the FPGA. When the new design starts up, the FPGA performs the new tag encryption using K_{ack2} as cipher key and sends the result to the SD. This acknowledgment informs the SD that the new bitstream has been correctly downloaded to the right FPGA and that the design has started.

Bitstream version verification: Each time the design starts up, it checks itself that TAG_{UL} and TAG_F are the same. If they are different, a replay attack has been detected and an alarm (a signal in the design) is triggered that can be used by the SD to apply his policy. He can for instance stop or destroy the system, or enter a degraded mode.

Security Analysis This analysis focuses on bitstream replay and remote DoS attacks. Our scheme assumes that the FPGA vendors encryption and integrity verification mechanisms are secure. For instance, the Actel mechanism implemented in the static logic checks the bitstream integrity using a MAC while the device is still operating. If the MAC validates the bitstream, the device will be erased and programmed. Otherwise, the device will continue to operate uninterrupted and will not take the new bitstream into account. The tag is encrypted with three different keys to prevent replay attacks. Indeed, to avoid the attacker responding by pretending to be the device or the SD, only one-time messages (key-tag pairs) are transmitted over the untrusted network. Since, for a bitstream version, the tag is the same for all the FPGAs, K_{req}, K_{ack1} and K_{ack2} must be unique for each device.

In the step, which is to download a new bitstream version, the new TAG cannot be spoofed because its integrity has been checked thanks to a MAC. For the same reason, an attacker cannot replace this bitstream with his own. The bitstream boot up acknowledgment enables update failures to be detected.

Implementation Considerations The only requirement of this protocol is the presence of an embedded NVM in the FPGA chip and a mechanism that provides bitstream confidentiality and integrity. For instance Spartan3-AN FPGA from Xilinx has an embedded Flash memory. Xilinx also provides a DNA mechanism, but it does not protect bitstream confidentiality. Lattice also provides such services in their XP2 FPGA family. Bitstream confidentiality and integrity are provided, but NVM is used to save a RAM copy and it is less easy to store data.

Demonstration Platform Our demonstration platform is based on an Actel Fusion starter kit (FPGA: Fusion AFS600). It is a non-volatile FPGA with an embedded user flash memory and a confidentiality and integrity mechanism. Figure 6.20 describes this demonstrator. The network is emulated by the RS232 link, the bitstream is downloaded through the JTAG port, and the block cipher is a 3-DES. The fact that

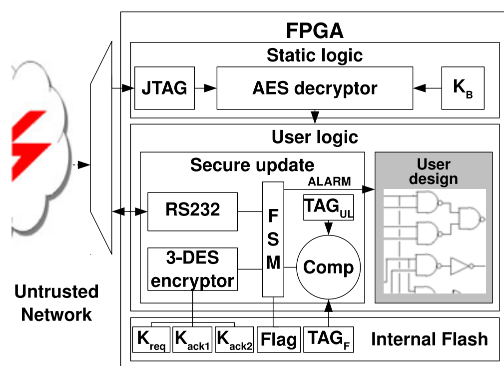


Fig. 6.20 Overview of FPGA design

K_{req}, K_{ack1} and K_{ack2} are stored in the flash memory with TAG_F allows the same bitstream file to be produced for the whole set of FPGAs: i.e. only the three keys that differentiate the device. During the initialization procedure, these three keys and TAG_F are downloaded through the JTAG port before locking.

Figure 6.21 describes the FPGA Master FSM algorithm. In order to reduce the latency, we decided to cipher TAG_{UL} with K_{req} while waiting for the update command. The two ciphered tags are then compared as soon as the SD tag is received. With this improvement, steps 1, 2 and 3 (respectively power up, first power up and authentication) are performed before receiving the SD update command.

Results Table 6.2 summarizes the overhead in terms of clock cycles and time required for each step. The design clocks at 60 MHz. Steps 2 and 3 (respectively first power-up and authentication) are performed during user design execution and do not increase the mechanism performance overhead. Step 4 is not considered here because the performance overhead is not significant compared to FPGA programming (several seconds). Considering all these elements, the performance overhead is estimated at only 54 cycles: step 1 (Power up).

Table 6.3 summarizes the overhead in terms of area. It shows the proportion of FPGA occupied by each component of this secure remote update mechanism implementation. This area overhead can be relativized considering that 3-DES, RS232 and flash memory controller can be reused by the SD. Only the master FSM cannot be reused.

The cost of flash memory is not shown because it is insignificant: 672 bits (including 32 bits for the first power up flag) on the 4 Mbits (0.016%). The SD can use the rest of this physical flash memory for his own purposes.

Conclusion Solution 2 is a communication protocol between the SD and an FPGA platform to update the FPGA configuration while preserving its confidentiality and integrity. This protocol also provides protection against replay attacks and detects update failures. In addition, we show that the corresponding area and performance overheads are negligible, thanks to the reusability of the core.

```
TAG_F : Flash memory tag
TAG_UL : User logic tag
Ek(M) : Encryption of M with K as cipher key\index{Key}
CTAGKx: Tag ciphered by K_x
```

Step 1: Power-up

```
1  Read (TAG_F)
2  if (TAG_F ≠ TAG_UL) then
3  goto 22
4  end if;
```

Step 2: First power-up

```
5  Read (flag)
6  if (flag = true) then
7  Read (K_ack2)
8  CTAGKack2 := E_K_ack2 (TAG_UL)
9  Send(CTAGKack2)
10 end if;
```

Step 3: Authentication

```
11 Read (K_req)
12 CTAGKreq := E_K_req (TAG_UL)
13 Read (K_ack1)
14 CTAGKack1 := E_K_ack1 (TAG_UL)
15 Wait for CMD
16 If (CMD = CTAGKreq) then
```

Step 4: TAG_F incrementation

```
17 Write (TAG_F+1)
18 Send(CTAGKack1)
19 Else
20 goto 15
21 end if;
22 SYSTEM SHUTDOWN
```

Fig. 6.21 Security protocol implementation on FPGA

6.4.3.4 Contribution to Remote Configuration Security (3): Security Architecture for Remote FPGA Update and Monitoring

When no non-volatile embedded memory is available, it is not possible to lock the FPGA to a particular bitstream version without an external trust party. However, one possible solution is that FPGA vendors modify their configuration logic to include

Table 6.2 Performance overhead for the AFS600 device

Step	# Cycles	Duration (μs) F = 60 MHz
1. Power-up	54	0.9
2. First power-up	187	3.1
3. Authentication	175	2.9
4. TAG_F increm.	108	1.8
Total	**524**	**8.7**

Table 6.3 Area overhead for the AFS600 device

Entity	# Tiles	% of Actel AFS600
3-DES	1305	9%
RS232	418	3%
Flash Controller	1005	7%
Master FSM	777	6%
Total	**3505**	**25%**

a version checking mechanism. A low-cost solution for FPGA vendors [9] has been proposed. The version TAG of the FPGA is kept in the configuration logic using a few flash or battery powered SRAM cells. The configuration logic uses the authenticated encryption algorithm CCM to ensure confidentiality, integrity and bitstream freshness. The logical gate cost and configuration time overhead is low regarding current implementation of bitstream encryption and integrity mechanisms. Interested readers should refer to this paper for further information.

6.4.4 FPGA Remote Update: Conclusion

From a security standpoint, remote updating of FPGA based systems is a challenge. We have shown that existing mechanisms aimed at providing bitstream confidentiality and integrity via encryption and authentication fail to prevent bitstream replay and thus system downgrade. Three different solutions were proposed to solve this problem. The first one is applicable to all encrypted FPGAs (volatile and non-volatile). The second requires a secure non-volatile memory that is currently only available in Actel Fusion FPGAs, however we hope that next generation FPGAs will be aware of this threat and will have embedded non-volatile user storage. Finally we proposed a complete solution for FPGA vendors who wish to provide a comprehensive solution to their customers at low cost.

According to Table 6.4, the proposed solutions are unique in the academic and industrial literature, since other solutions fail to take replay attack into account. Our solutions apply to both volatile and non-volatile FPGA devices, by accounting for their particularities. In addition, we propose a new complete solution [9] that pro-

Table 6.4 Secure update solution comparison

Solution	Suitable devices	Cost for FPGA vendors	Development time for system designer	Logic gates cost for system designer	Additional cost
1	All encrypted FPGAs	None	High	High	Regular polling
2	ACTEL Fusion	Low for Flash based FPGAs	Medium	Low	None
3	Currently none	Medium	Low	None	None

vides confidentiality, cryptographic integrity verification, replay attack counter measure and also provides convenient way for the system designer to remotely manage the bitstream update process. All the contributions concerning bitstream security and the study cited in this chapter about FPGA security, are used in the following section to develop a Reconfigurable Cryptographic Platform that benefits from these results. The platform can be considered as a concrete application of the concepts developed throughout this book and, thanks to its context, is close to industrial concerns.

6.5 Example of Board Integration: Toward a Secure Platform

To apply the knowledge described in this book, we use a concrete application. This approach allows us to evaluate the solutions we propose from an industrial point of view. This study was done with the help of a French company called Netheos [31] which develops applications for information security. The company aims to create products dedicated to security, based on FPGAs, to occupy low and middle volume markets. Their main targets are corporate and bank sectors, or even government infrastructures. The platform aims to be configurable to respond to many different needs, and it might be reconfigurable in order to evolve, for instance when a cryptographic algorithm or protocol is broken. The code name of the platform is RCP for Reconfigurable Cryptographic Platform. Objectives with this platform is twofold:

- To increase the security of key management compared with a software only solution. This is done by keeping cryptographic keys inside a piece of hardware where keys are only accessible to perform cryptographic computation, not for reading or writing. These keys are generated by the hardware itself with an embedded true random number generator.
- To achieve high performances for cryptographic algorithms such as RSA or AES schemes. This can be done by having a high bandwidth communication channel with the FPGA chip and by using high-speed hardware accelerators (such as RSA or AES IP cores). Depending on the application, these accelerators can be

instantiated multiple times to achieve even better performances by parallelizing computations.

We intend to use its platform for applications such as giga-bit network encryption, SSL accelerator or off-load engines, VPN accelerators, high-performance Hardware Security Modules, but also for customer specific applications.

6.5.1 Requirements

In the following section we present the features we would like to include in the secure platform.

Scalability The platform needs to be as generic as possible regarding cost, performance and security level. It should be able to support relatively low and high security levels. In the first case, the platform needs to keep costs low and provide reasonable security. In the other cases, performance and security concerns are more important than cost. The more resources included in the FPGA, the more performances can be obtained, for instance by implementing many types of cryptographic algorithm accelerators. The selection of the FPGA model will thus determine the performances that can be achieved with the platform.

To obtain security level scalability of the platform, two types of attacks need to be distinguished:

- **Logical attacks**: that have to be countered by implementing a logically secured hardware and software design.
- **Physical attacks**: that can be prevented by implementing cryptographic algorithms with side channel and fault injection countermeasures, but also by adding a secure surface enclosure such as those proposed by [23]. This type of tamper respondent enclosure, which is widely used for Hardware Security Modules (HSM), will erase the cryptographic key when attacks are detected, thus guaranteeing protection against invasive attacks that are quasi impossible to prevent only using logical mechanisms. With such protection, this platform can hope to achieve FIPS 140-2 certification at level 3, at which key destruction is mandatory under attack.

Target Applications The board targets two applications:

- Cryptographic accelerators: the platform needs at least one high-speed interface to communicate directly over the network or through the host.
- Secure key containers (HSM): One of the most important points for a cryptographic device is secure key management in a logical and physical way. A key must never leave the device without being encrypted, or for some applications, should not leave the device even if it is encrypted.

Regarding the second application, even if confining the key to the secure device makes sense with respect to security, it reduces the usability of the solution. If the

Table 6.5 LX and SX family characteristics

FPGA Family	LXT		SXT	
FPGA Model	XC5VLX30T	XC5VLX50T	XC5VSX35T	XC5VSX50T
Slices	4800	7200	5440	8160
Logic Cells	30720	46080	34816	52224
Block RAM	36	60	84	132
DSP48E slices	32	48	192	288

secure device stops working because an irreversible failure occurs in the hardware, or maybe because an attack is detected, all the cryptographic keys become irreversibly unusable. Therefore most hardware security modules use a master intended to cypher the user keys. Encrypted user keys are stored outside the HSM. A mechanism allows master key backup on a secure medium such as a smart card. For both applications, software and hardware updating should be possible throughout the life cycle of the device, maybe to correct security vulnerabilities or to increase performances. However this feature opens the door to many threats and must be implemented carefully.

6.5.2 Platform Design

FPGA Choice The FPGA device is the most important component and determines the price and the performances of the platform. But it plays also a role in the security of the system. Based on the results of a comparison of current FPGAs, we decided to use a Virtex5 FPGA (in 2008). It provides mechanisms and features that enable the implementation of useful security countermeasures. For instance, it allows for continuous monitoring of the integrity of the loaded bitstream, an embedded thermal sensor can detect dangerous variations in the environment. Bitstream encryption allows cryptographic keys to be hidden in RAM blocks or look up tables. Internal read back of the loaded bitstream can increase the robustness of the integrity check of the configuration. Finally, the partial reconfiguration feature is attractive to implement adaptive and evolutionary design. Since the price of the FPGA chip price represents most of the cost of the board, we designed the platform to support any Virtex5 FPGA that fits an FFG-665 package, so Virtex5 LX30T, LX50T, SX30T and SX50T can be used without PCB modifications. The cheapest FPGA that can be used is Virtex5 LX30T, which has limited DSP and storage capacities. The most attractive FPGA for cryptographic applications is Virtex5 SX50T, because it includes many storage (RAM blocks) and arithmetic elements (DSP blocks) that can be used to implement high performance cryptographic cores (such as RSA, ECC or AES). LX and SX family characteristics are listed in Table 6.5.

Cryptographic Engines One of the main advantages of FPGAs over CPUs is that developers can implement specialized pieces of hardware that allow very efficient

Algorithm	Implementation	Slices	Blocks RAM	Multipliers	Throughput
AES 128 (Encryptor and decryptor. w/HWkey expansion)	Helion Tech	864	0	0	3781 Mb/s
	our	803	5	0	1024 Mb/s
SHA-256	Helion Tech	325	0	0	1722 Mb/s
	our	792	1	0	1144 Mb/s
RSA-1024 (full exponent wo/CRT)	Helion Tech	1800	1	0	20 signature/s
	our	378	2	5	24 signature/s

Fig. 6.22 Comparison of results of the implementation of cryptographic IP cores

cryptographic algorithm implementations. AES hardware implementation is a good example; it is quite easy to implement a dedicated engine that executes an AES round in one clock cycle while keeping a reasonable silicon area. However AES engines are not available on most CPU or SoC platforms.

That is why a set of cryptographic engines was developed. This work focuses on the most widely used cryptographic algorithms such as AES, RSA and SHA-2. Most optimizations were found in academic publications describing algorithmic or architectural enhancement to straightforward algorithm implementations. This work takes advantage of these optimizations and produced efficient and compact implementations. Our main contribution is the comparison of different implementation styles and the development of configurable cryptographic cores that offer large area/throughput/functionalities/security trade offs.

The details of the development of these blocks are not given in this book, but Table (Fig. 6.22) gives a brief comparison of our results using commercially available cores.

Comparing cryptographic engines is not easy [21] mainly because many implementations are available and they do not use the same FPGA family, or do not use the same core functionalities. For this reason, we used the same FPGA family for our comparison (Virtex5) and the IP cores cited have the same functionalities (for instance the AES IP core is able to perform encryption and decryption with hardware key expansion). Finding results in academic publications that allow a fair comparison was not possible, mainly because most of these publications are old and use obsolete FPGA devices (such as Virtex or Virtex2 FPGAs). The results achieved with the IP cores developed during this study are no better than the results achieved by Helion Technology (a private company). However, the goal of the present study was not to design highly efficient IP cores but to develop state of the art cryptographic engines and to use them to build a complete security platform.

Secure Microprocessor Based on the results of the FPGA bitstream security study described in previous sections, we decided to provide the board with a secure microprocessor in addition to the Virtex5 chip. The following arguments justify this choice:

- Since Virtex5 are volatile FPGAs that do not include a non-volatile storage element, it is impossible to store a value inside the chip after power down. Although it is possible to store securely values outside the chip in an encrypted flash memory, a master key has to be used to decrypt this memory, thus, to remain non-volatile, the key must be stored in the FPGA bitstream. However this solution is not applicable when the application requires volume, because each FPGA needs a different bitstream to provide each board with a unique key.
- There is no trusted hardwired TRNG in FPGA devices. Even if it is possible to implement secure TRNG in user logic [19], we decided that this represents a serious risk when high security level certification is required. For that reason, random numbers can be generated in three places on the platform: inside the user logic of the FPGA, in the secure microprocessor, or in the smart card of the board.
- The secure microprocessor we selected provides many security features such as internal and external sensor management. A battery powered key memory is automatically erased when an attack on the clock, the package of the chip, the temperature, or the power supply is detected. The package integrity sensor is very important since FPGA vendors do not provide such packages. Thanks to this feature it is possible to produce a low-cost hardware security module without surface enclosure, where the heart of the security level (the secure microprocessor) is physically protected by its package.
- The cost of such a microprocessor is very low compared to the FPGA. PCB complexity increases with this BGA device, but this cost should be compared with the non-negligible security add on.

Based on this choice, the secure microprocessor will be the heart of the system since it generates cryptographic keys and stores the master key.

FPGA Architecture The goal is not to develop a fixed architecture but a flexible one that allows the design to be scaled according to application requirements. For instance, an application that requires many RSA operations in parallel will have several embedded RSA hardware accelerators in order to achieve higher throughput. The configuration of the platform is currently performed off line, before physical synthesis of the FPGA architecture. Ultimately the goal is to provide run-time reconfiguration according to application requirements, by using dynamic reconfiguration. This goal was not addressed in this study but will the subject of future works in order to provide such functionality while keeping the same level of security. Since cryptographic operational keys are stored in the secure microprocessor of the board, the communication between the FPGA and this component must be secured. This constraint leads to the architecture described in Fig. 6.23.

Figure 6.23 shows the architecture we developed. The "core Wrapper" square refers to a slot where the platform user can instantiate any IP cores compatible with the open source wishbone bus [33], these may be cryptographic IP cores or even application specific IPs. To make the platform usable from the outside world, it is equipped with PCI Express connectivity.

A direct connection exists from the secure processor and key memories. These memories can only be written from the microprocessor, so with this FPGA architecture, the key cannot travel to the host. An AES decryption engine is placed between

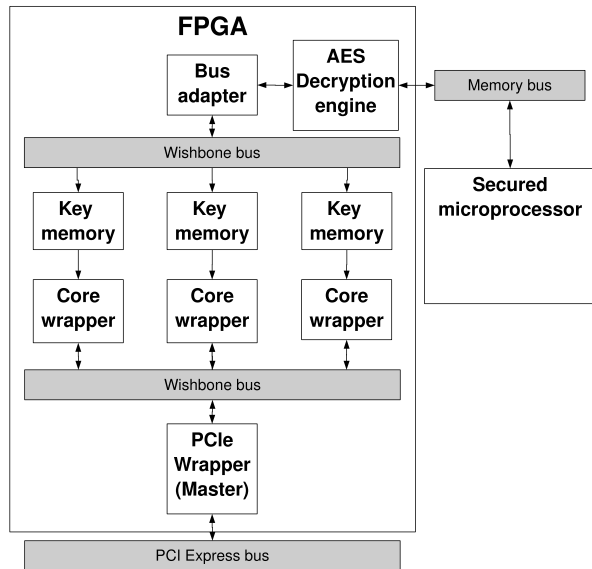


Fig. 6.23 General target architecture of the reconfigurable cryptographic processor

the external processor and the key memory to prevent board level attacks (for instance using bus probing).

High Speed Communication Interfaces To be as flexible as possible according to Virtex5 functionalities, we decided to connect the FPGA to a PCI Express bus with eight lines with a maximum throughput of 16 giga bits per second on each of the two ways (reception and transmission). We also connected the FPGA to two RJ-45 interfaces that can support giga bit Ethernet, we deliberately chose two interfaces to allow red/black architecture [41] that are commonly used for governmental or military applications.

Trusted Area A secure microprocessor allows tamper respondent mechanisms to fulfill FIPS140-2 level 3 requirements; however Virtex5 FPGA cannot comply with this standard since its package does not respond to tampering and so physical invasive attacks are feasible. To allow our secure platform to reach level 3, we designed the board's PCB to facilitate the use of a tamper resistant enclosure such as [23]. This was achieved simply by reducing the area to be protected to the minimum in order to decrease cost of enclosure. The secure area boundary is shown in Fig. 6.24, it includes:

- The FPGA where user keys are used;
- The secure microprocessor where master key is stored and user keys are generated;
- The FPGA configuration flash memory, since we have shown that bitstream security is not totally guaranteed by FPGA vendors (bitstream replay, bitstream integrity);

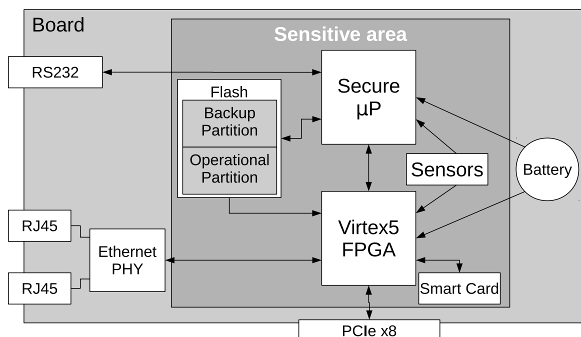


Fig. 6.24 General scheme of the NETHEOS platform

- The core voltage regulator of the FPGA, this increases the robustness of the platform regarding power analysis because an attacker can only measure current filtered by the regulator;
- A smart card used to diversify true random sources (from the smart card, the secure processor and the FPGA);
- And finally the sensors used to monitor enclosure integrity, allowing the secure microprocessor to erase the master key.

Security Scalability To achieve security level scalability of the board, two types of attacks have to be distinguished:

- Logical attacks that must be countered by implementing a logically secured hardware and software design.
- Physical attacks that can be prevented by implementing cryptographic algorithms with side channel and fault injection countermeasures, and also by adding a secure surface enclosure.

Tamper respondent enclosure is widely used for hardware security modules (HSM) that erase cryptographic keys when an attack is detected, thus ensuring protection against invasive attacks that are quasi impossible to counter only with logical mechanisms. With such protection, this platform should achieve FIPS 140-2 certification at level 3 where key destruction is required to counter attacks. Secure surface enclosure is an industrial process that consists in enclosing a security system inside a tamper respondent envelope. It is an active seal that generates an alarm when the system is tampered with; this alarm can be used by the system to erase sensitive data. These surface enclosures thus have to be powered to enable the system (which requires energy) to erase the keys. That is why most HSMs include a battery, which can be located inside or outside the secure enclosure, since if the attacker simply removes the battery, the keys will be erased as they are stored in a volatile memory. Development of a secure enclosure is a hard task, particularly if a high level of security is required. The classical approach is to enclose the system in a mesh of conductive wires. This mesh is connected to sensors that detect variations in wire conductivity and trigger alarms. Therefore, if attackers try to remove the protection, they are very likely to cut a wire. However an attacker with a high degree of

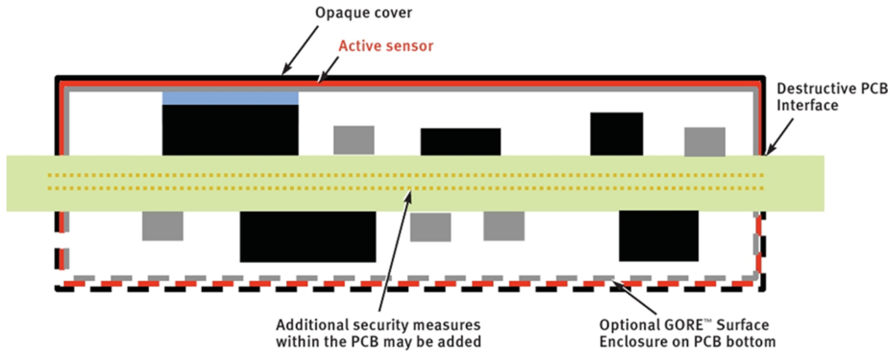


Fig. 6.25 Secure PCB surface enclosure designed by Gore

competency and sophisticated equipment may localize the mesh by using X-ray and succeed in passing between the wires without cutting them. A company called Gore proposes surface enclosures that have already been tested and validated, hence reducing the cost for a company that wants to create a physically protected device. The conductive mesh used by Gore is non-metallic to avoid X-ray attacks. The principle of the architecture of a Gore secure surface enclosure is shown in Fig. 6.25 [23]. This figure shows the PCB of the system enclosed by the Gore active sensor and an opaque cover. The cover is connected to the PCB by a destructive interface that triggers an alarm when tampered with.

The Gore tamper respondent sensor (in gray in the figure) can enclose the whole system or a part of the PCB. In the latter case, contacts between the enclosure and PCB require particular attention since the assailant may try to attack this weak part. That is why additional security mechanisms can be added inside the PCB itself. An opaque cover, generally made of metal, is added to protect the system from the environment. The board developed during the present study enables the use of such tamper respondent enclosures. It includes a battery and circuitry to allow key erasure. Moreover, since data remanence of sensitive information is obviously a concern for the final device, a mechanism that connects power to ground is also included.

Secure Key Management The secure microprocessor stores the value of a master key in its secure battery powered key memory, and operational keys are encrypted using this master value. Since the FPGA is the device that performs high-speed cryptography using dedicated hardwired engines, the secure microprocessor must decrypt operational key and send the plaintext value to the FPGA. The communication channel between FPGA and microprocessor can be secured using encryption according to security objectives of the application.

Secure Backup/Restore Operational key backup and restoration operations are performed using a safepad and a smart card. The safepad ensures there is a trusted path to enter a PIN code (without passing through the host computer), the smart card ensures the physical security of the backup.

Communication Between FPGA and Secure Processor The secure microprocessor is the device that generates and securely stores user objects. The FPGA device is used to accelerate cryptographic computations, consequently it has to use these cryptographic keys in plaintext form. However unencrypted communications between the FPGA and the processor are not acceptable when local attacks are possible, an attacker can spy on the communication bus to retrieve cryptographic keys. Even with a tamper respondent enclosure, it is preferable to encrypt the communication channel between the two chips so that all security does not rely on the enclosure. In order to communicate safely, the two chips have to share a symmetric key; this cannot be the master key because that key must never leave the secure microprocessor, otherwise the confidentiality of this key is harder to ensure. Thus a mechanism of key sharing must be implemented. At least two methods can be used to reach this goal:

- During the bitstream and processor code generation process the system designer hard codes a symmetric key in the FPGA logic and in the flash memory of the processor. There is no need for a key sharing protocol. However this method has certain drawbacks, first, the system designer knows the symmetric key value and the customer does not necessarily trust the SD. And, in order to simplify key management, bitstream and processor code generation process, the SD should provide the same symmetric key to all his designs, therefore breaking one product enables all the products to be broken. Finally the shared key remains the same throughout the life of the product, whereas side channel attacks such as DPA can lead to key discovery; however, if the key is changed regularly, power analysis attacks become more difficult.
- The system designer hard codes a private key pair in the processor code and another key pair in the FPGA bitstream, both signed by a trusted authority to avoid man in the middle attacks. These key pairs will be used to regularly exchange symmetric keys to avoid differential side channel attacks (DPA, DEMA). However this method has the same drawbacks as the previous one, and the cost of implementation is higher since asymmetric and symmetric algorithms have to be used.

6.5.3 Performance Results

In order to measure real performance and to show the advantage of having many IPs that can operate in parallel, an openSSL engine [34] was developed. The term engine here refers to a library that is dynamically linked to openSSL and allows this application to perform cryptographic operations using hardware devices. The engine developed is quite simple, it is based on the GMP (GNU Multi Precision) engine, i.e. the software implementation of the cryptographic algorithm that uses GMP, whereas openSSL uses a library called *Big Numbers*. This option was preferred since the GMP engine code is quite simple to modify. The code that performs the modular exponent in software has been replaced by a call to the RSA IP

Key size	RCP (-multi 4)	AMD Sempron 3000+ (-multi 1)	Intel Core2 CPU 2.13 GHz (-multi 2)
512	1282 sign/s	1579 sign/s	3334 sign/s
1024	421 sign/s	360 sign/s	731 sign/s
2048	85 sign/s	66 sign/s	134 sign/s
4096	13 sign/s	10 sign/s	22 sign/s

Fig. 6.26 Practical results of openSSL engine with CRT using six RSA IP cores

cores connected to the platform. GMP uses by default the Chinese remainder theorem (CRT), which allows the RSA exponent to be split into two smaller exponents, thereby dramatically reducing modular exponent computation time. For instance a 1024-bit modular exponent is reduced to two 512-bit modular exponents. This engine measures performance and checks the results of RSA computation using a simple call to the command line:

openssl speed -multi X -engine gmp rsa

The *multi* argument of the command line allows X RSA computation to be run in parallel, for instance a call with *-multi 2* gives better results when the test runs on a dual core processor. For the platform, the best results are achieved when the multiple parallel operation number is equal to the number of RSA IP cores instantiated. For instance, results of a test on a platform that includes six RSA IP cores are given in Table 6.26.

The results of openSSL engines were compared to two different types of general purpose CPUs. A single core processor: AMD Sempron, whose best performance is obviously achieved using the argument *-multi 1* while the Intel Core 2 processor allows parallel RSA computation giving better results with *-multi 2*. The best results were achieved with the platform using *-multi 4* because the CRT algorithm is used, and consequently two RSA engines are needed to perform one complete exponent. In terms of efficiency, a good way to compare is to sum the costs of the solutions. An Intel Core 2 processor running at 2 GHz cost about US$ 70 at the time of writing, while a Virtex5 LX30T cost about US$ 350, i.e. a ratio of five to one, while performance was twice lower on our implementation than on Virtex5 LX30T. So with our RSA engine, FPGAs are not competitive in terms of the cost: performance ratio using a general purpose CPU. However when the platform is used for massive RSA computation, the host CPU usage remains low. The most important argument for our FPGA-based implementation, is that by using the platform, it is possible to enhance the physical security of RSA private keys since they can be generated on board, used inside the platform, and never travel through host memory.

6.5.4 Conclusions

According to the results of the present study, there is no perfect FPGA device currently available for security-sensitive or cryptographic applications. Even the choice

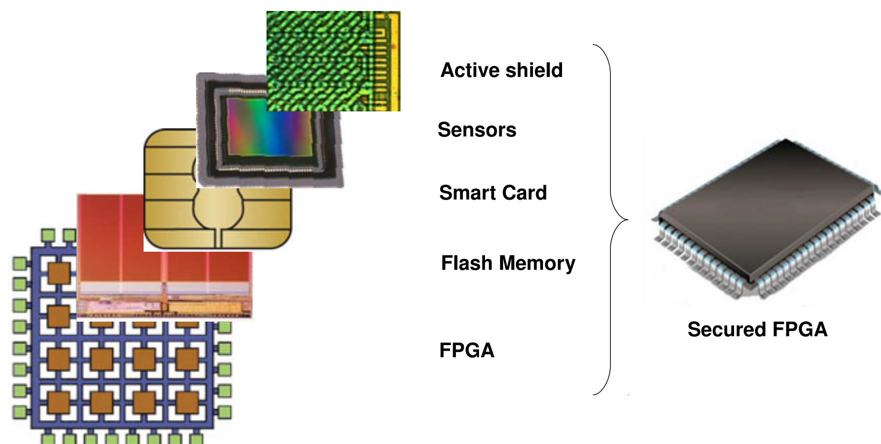


Fig. 6.27 An imaginary highly secured FPGA

between non-volatile and volatile FPGAs is not trivial and also often depends on the application. For instance, we were forced to use an external processor, which complicates the PCB design, increases vulnerabilities of the communication with the FPGA and represents an additional cost. Therefore, in this conclusion we 'imagine' a perfect FPGA device for such applications (see also Fig. 6.27):

- First it must have some embedded secure battery powered memories for highly sensitive data that can be very rapidly erased from user logic or by an external alarm.
- It must also include non-volatile memories that are useful in security, for instance to implement PIN code mechanisms or to store bitstream version TAGs to prevent bitstream replay attacks.
- FPGA vendors must provide strong mechanisms that ensure bitstream confidentiality and integrity, and that prevent bitstream replay attacks.
- The device must have an embedded mechanism allowing the system designer to force the FPGA device to accept only encrypted and authenticated bitstream, like the Actel Flash Lock mechanism or the Xilinx e-Fuse.
- Ideally it should have an embedded certified random number generator, such as a smart card. An even better solution would be to embed a smart card in the same package as the FPGA matrix.
- The FPGA device should be embedded inside a tamper proof or even better a tamper respondent package that triggers alarms when attacks are detected. For instance by using a metal mesh able to detect package opening and to trigger built in mechanisms to erase a battery powered memory. This package should also provide basic protection against electromagnetic analyses such as a Faraday cage.

Future works will focus on improving the platform by adding run time reconfiguration based on application requirements and by adding new cryptographic IP cores or by improving the performance of existing ones as well as improving communication architecture.

References

1. The Embedded Microprocessor Benchmark Consortium (EEMBC)
2. Fips Pub 197: "Advanced Encryption Standard (AES)" (2001)
3. In-System Programming (ISP) of Actel low-power flash devices using Flashpro3. Microsemi (ex Actel) corporation. Version 1.5 (August 2009)
4. Actel: Implementation of security in Actel's ProASIC and ProASICPLUS flash-based FPGAs. URL. http://www.actel.com/documents/Flash_Security_AN.pdf (2003)
5. Actel: Fusion FPGA Handbook. URL. http://www.actel.com/documents/Fusion_HB.pdf (2008)
6. Actel: ProASIC3 Handbook. URL. http://www.actel.com/documents/PA3_HB.pdf (2008)
7. Alderighi, M., D'Angelo, S., Mancini, M., Sechi, G.R.: A fault injection tool for SRAM-based FPGAs. In: IEEE International On-Line Testing Symposium, p. 129 (2003). doi:10.1109/OLT.2003.1214379
8. Altera: Design Security in Stratix III Devices. URL. http://www.altera.com/literature/wp/wp-01010.pdf (2006)
9. Badrignans, B., Champagne, D., Elbaz, R., Gebotys, C.H., Torres, L.: Sarfum: security architecture for remote FPGA update and monitoring. ACM Trans. Reconfigurable Technol. Syst. **3**(2), 8 (2010)
10. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: Annual IEEE Symposium on Foundations of Computer Science pp. 90–99 (1991). doi:10.1109/SFCS.1991.185352
11. Daemen, J., Rijmen, V.: AES proposal: Rijndael (1998)
12. Drimer, S.: Authentication of FPGA bitstreams: why and how. In: Applied Reconfigurable Computing. Lecture Notes in Computer Science, vol. 4419, pp. 73–84 (2007)
13. Drimer, S.: Volatile FPGA design security—a survey (v0.96) (April 2008)
14. Eisenbarth, T., Güneysu, T., Paar, C., Sadeghi, A.-R., Schellekens, D., Wolf, M.: Reconfigurable trusted computing in hardware. In: Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing, STC '07, pp. 15–20. ACM, New York (2007). doi:10.1145/1314354.1314360
15. Elbaz, R.: Hardware mechanisms for secured processor memory transactions in embedded systems. PhD thesis, University of Montpellier (December 2006)
16. Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M., Martinez, A.: A comparison of two approaches providing data encryption and authentication on a processor memory bus. In: Vounckx, J., Azemard, N., Maurine, P. (eds.) Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation. Lecture Notes in Computer Science, vol. 4148, pp. 267–279. Springer, Berlin (2006). doi:10.1007/11847083_26
17. Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M., Martinez, A.: A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In: Proceedings of the 43rd Annual Design Automation Conference, DAC '06, pp. 506–509. ACM, New York (2006). doi:10.1145/1146909.1147042
18. Elbaz, R., Champagne, D., Lee, R., Torres, L., Sassatelli, G., Guillemin, P.: TEC-tree: a low-cost, parallelizable tree for efficient defense against memory replay attacks. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems—CHES 2007. Lecture Notes in Computer Science, vol. 4727, pp. 289–302. Springer, Berlin (2007). doi:10.1007/978-3-540-74735-2_20

19. Fischer, V., Bernard, F., Bochard, N., Varchola, M.: Enhancing security of ring oscillator-based RNG implemented in FPGA. In: Field-Programable Logic and Applications (FPL), September, pp. 245–250 (2008)
20. Fruhwirth, C.: New methods in hard disk encryption. Technical report, Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology (2005)
21. Gaj, K., Chodowiec, P.: Fast implementation and fair comparison of the final candidates for advanced encryption standard using field programmable gate arrays (2001)
22. Gassend, B., Suh, G.E., Clarke, D., van Dijk, M., Devadas, S.: Caches and Merkle trees for efficient memory integrity verification. In: Proceedings of Ninth International Symposium on High Performance Computer Architecture, February (2003)
23. GORE: GORE tamper respondent surface enclosure. Commercial Brochure (2007)
24. Hall, W.E., Jutla, C.S.: Parallelizable authentication trees. In: Selected Areas in Cryptography, pp. 95–109 (2005)
25. Hendry, M.: Multi-application Smart Cards: Technology and Applications. Cambridge University Press, Cambridge (2007).
26. Hori, Y., Satoh, A., Sakane, H., Toda, K.: Bitstream encryption and authentication using AES-GCM in dynamically reconfigurable systems. In: Matsuura, K., Fujisaki, E. (eds.) Advances in Information and Computer Security. Lecture Notes in Computer Science, vol. 5312, pp. 261–278. Springer, Berlin (2008). doi:10.1007/978-3-540-89598-5_18
27. Lattice: LatticeXP2 Family Handbook. URL. http://www.latticesemi.com/dynamic/view_document.cfm?document_id=24315 (2008)
28. Lie, D., Thekkath, C.A., Horowitz, M.: Implementing an untrusted operating system on trusted hardware. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pp. 178–192. ACM, New York (2003). doi:10.1145/945445.945463
29. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography, 1st edn. CRC Press, Boca Raton (1996). ISBN 0849385237
30. Merkle, R.C.: Protocols for public key cryptography. In: Proceedings of IEEE Symp. on Security and Privacy, pp. 122–134 (1980)
31. Netheos: Official Netheos Website. URL. http://www.netheos.net (2010)
32. Note, J.-B., Rannaud, E.: From the bitstream to the netlist. In: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays, FPGA '08, pp. 264–264. ACM, New York (2008). doi:10.1145/1344671.1344729
33. OpenCores.org: WISHBONE system-on-chip interconnection architecture for portable IP cores specification revision B.3. URL. www.opencores.org/downloads/wbspec_b3.pdf (2002)
34. OpenSSL.org: OpenSSL cryptography and SSL/TLS toolkit, engine documentation. URL. http://www.openssl.org/docs/crypto/engine.html
35. Parelkar, M.M., Gaj, K.: Implementation of EAX mode of operation for FPGA bitstream encryption and authentication. In: Brebner, G.J., Chakraborty, S., Wong, W.-F. (eds.) FPT, pp. 335–336. IEEE Press, Singapore (2005)
36. Schellekens, D., Tuyls, P., Preneel, B.: Embedded trusted computing with authenticated non-volatile memory. In: Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing—Challenges and Applications, Trust '08, pp. 60–74. Springer, Berlin (2008). doi:10.1007/978-3-540-68979-9_5
37. Suh, G.E., O'Donnell, C.W., Devadas, S.: AEGIS: a single-chip secure processor. IEEE Des. Test **24**, 570–580 (2007). doi:10.1109/MDT.2007.179
38. Suh, G.E., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: AEGIS: architecture for tamper-evident and tamper-resistant processing. In: Proceedings of the 17th Annual International Conference on Supercomputing, ICS '03, pp. 160–171. ACM, New York (2003). doi:10.1145/782814.782838
39. Suh, G.E., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: Efficient memory integrity verification and encryption for secure processors. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, p. 339. IEEE Comput. Soc., Washington (2003)

40. Suh, G.E., O'Donnell, C.W., Sachdev, I., Devadas, S.: Design and implementation of the AEGIS single-chip secure processor using physical random functions. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture, ISCA '05, pp. 25–36. IEEE Comput. Soc., Washington (2005). doi:10.1109/ISCA.2005.22
41. Thales: Security architecture reinforced in two or three physically separated sections. Commercial Brochure (2007)
42. Unknown: Open freebox project website. URL. http://www.f-x.fr/ (2006)
43. Xilinx: Virtex-5 FPGA configuration user guide. URL. http://www.xilinx.com/support/documentation/user_guides/ug191.pdf (2008)
44. Xilinx: Xilinx Ug360 Virtex-6 FPGA configuration user guide. URL. www.xilinx.com/support/documentation/user_guides/ug360.pdf (2010)
45. Yang, J., Zhang, Y., Gao, L.: Fast secure processor for inhibiting software piracy and tampering. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, p. 351. IEEE Comput. Soc., Washington (2003)