Deliverable D1.1.1

Specification of the tool, first release

Editor Jean-Louis Lanet
Authors by alphabetical order:
Damien Couroussé, CEA
Jean-Louis Lanet, XLIM
Hassan Noura, CEA
Bruno Robisson, ENSMSE

Version Revision: 78
Date Fri, 23 May 2014 15:23:59 +0200
CEA ref. V13DACLE014 – 14-0342

Contents

List of Figures

List of Tables

1 Executive Summary

This report focuses on the use of runtime code generation techniques in the context of secure devices. Its objective is to provide preliminary elements in order to support the core objective of the COGITO project: demonstrate the applicability of runtime code generation techniques for security purposes.

In the first part of this report, we describe various attacks targeting smart cards, both in terms of hardware based attacks, logical attacks, and mixed attacks (section 2). In mixed attacks, the fault injection is an enabler for logical attacks. Then, we present most of the countermeasures related to the scope of the project against these attacks and we bring to the fore their limits (section 3). We propose some clues for improving the resistance to some attacks that are less covered by current countermeasures, in particular code polymorphism.

In the second part of the document, we introduce `deGoal` as an enabling technology to improve the security of the embedded systems. The original purpose of this technology is to bring performance improvements in terms of execution time or energy consumption for computing systems. Considering that `deGoal` is well suited for embedded systems, especially systems with small memory resources and low computing capabilities such as the systems usually used in secured applications, we assume that it is possible to retarget this tool for security purposes. In this report we present the functionalities of `deGoal` (section 4), and the modifications targeted in the context of COGITO to reach the objectives of the project (section 5).

This deliverable is intended as the first release of a study report, due to the end of the COGITO projet, entitled "final specification of the tool and guidelines for future implementations (D1.1.3)".

2 Threats against Smart Cards

A smart card usually contains a microprocessor and various types of memories: RAM (for runtime data and OS stacks), ROM (in which the operating system and the *romized* applications are stored), and EEPROM (to store the persistent data). However, due to significant size constraints of the chip, the amount of memory is small. Most smart cards on the market today have at most 5 KB of RAM, 256 KB of ROM, and 256 KB of EEPROM. A smart card can be viewed as a secure data container, since it securely stores data and it is securely used during short transactions. Maintaining a sufficient level of security relies first on the underlying hardware. To resist probing an internal bus, all components (memory, CPU, crypto-processor, *etc.*) are on the same chip which is embedded with sensors covered by a resin. Such sensors (light sensors, heat sensors, voltage sensors, *etc.*) are used to disable the card when it is physically attacked. The software is the second security barrier. The embedded programs are usually designed neither for returning nor modifying sensitive information without guaranty that the operation is authorized.

Smart cards are devices prone to attacks in order to gain access to services or assets stored by the card. Several means have been used to retrieve these valuable information and recently fault injection appears to be the most efficient. Thus, smart card manufacturers try to design countermeasures to embed in their operating system to prevent such attacks. Often solutions are based on dedicated code at the applicative level.

We present in the following sections several attacks that target the software only and other attacks based on hardware. In this latter we can disgusting between passive attacks which acquire information by monitoring the processor activities through current or electromagnetic probes and active attacks that modify the behaviour of the processor or memories by injecting energy into the silicon.

2.1 Hardware Attacks

Among the security threats, a very important one is certainly due to vulnerabilities of the integrated circuits that implement cryptographic algorithms (described above). With the access to one of these circuits, the attacker tries either to reconstruct the functionality of the circuit (reverse engineering) or to recover cryptographic materials when the cryptographic algorithm is known (physical or hardware cryptanalysis). Both threats share a set of techniques. The first one consists in getting information about the chip design by direct inspection of its structure. The second one, called side channel attacks, consists in observing some physical characteristics which are modified during the circuit's computation. The third technique, called fault attacks, consists in disrupting the circuit's behaviour.

2.1.1 Inspection

This inspection may be performed by using any kind of imaging techniques or by using destructive means such as abrasion, chemical etching or focused ion beam. Figure 1 represents an integrated circuit whose package has been opened by using chemical etching. On this Figure, several logical blocks such as memories, numeric and analogic parts may be easily distinguished.

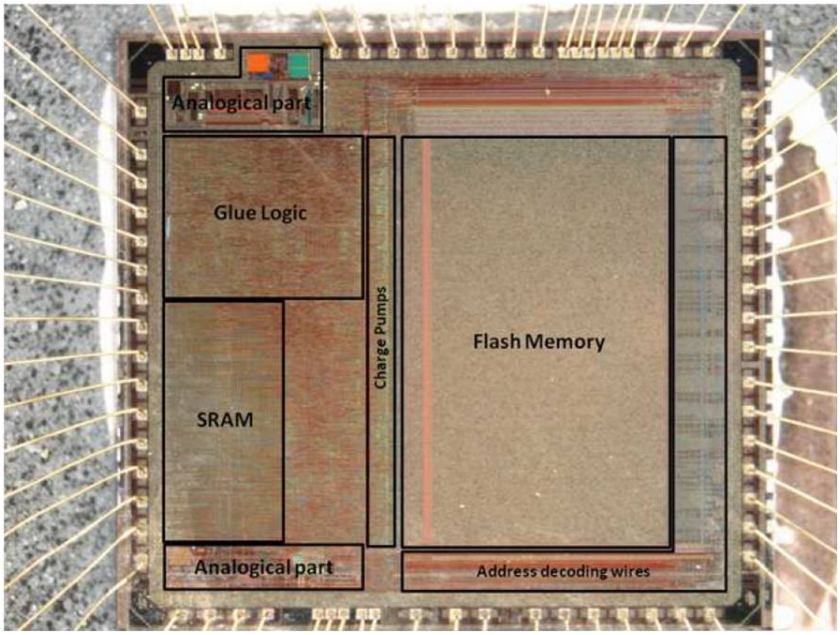

Figure 1: Inspection of a microcontroler

2.1.2 Side Channel Attacks

Side channel analysis exploits the fact that some physical values or 'side channels' (such as power consumption, electromagnetic radiation, computation time or even light emission) of a circuit depend on the key value [Koc96, KJJ99, SMP07]. A rougher technique consists in measuring directly some key-related internal computations by using micro-probing techniques. Figure 2 represents several micro-probes which are used to measure the electromagnetic field created by an integrated circuit.

There are two subcategories of key recovering techniques based on side channel measurements. The first one, called DPA-like, consists in building a set of mathematical models (i.e. mathematical formulæ) from a priori knowledge about the circuit. Each model is associated with an hypothesis on the value of the key. Then, the models are compared with measurements. The model which matches the best with measurements is generally associated with the right key hypothesis. The second kind of side channel attacks needs a profiling step on another circuit. This circuit is supposed to be identical to the target and the attacker is supposed to be able to set the key value. In this case, the profiling step is used either to improve the model a priori (stochastic attacks) or to build a statistical model only based on measurements (template attacks).

2.1.3 Fault Attacks

In this subsection, we will give an overview over actual physical methods to induce faults. This will show that there are numerous ways to induce faults into physical devices. Fault attack is an old research field. Research in avionics or space travel brought to the fore that cosmic rays can flip single bits in the memory of an electronic device [ZCM+96]. Such faults are still an issue until now for such devices. In the smart card field, researches focused on power spikes, clock glitches and optical attacks. A smart card is a portable device without any own power supply neither clock and thus requires a smart card reader providing power and clock sources in order to work. The reader can be replaced by an adversary with laboratory equipment, able of tampering with the power supply.

Figure 2: Electromagnetic micro-probes (courtesy of the Micropacks platform at GARDANNE, France)

With short variations of the power supply, which are called spikes, one can induce errors into the computation of the smart card. Spikes allow to induce memory faults but also faults in the execution of a program. This latter, which aims at confusing the program counter, can cause conditionals to work improperly, loop counters to be decreased and arbitrary instructions to be executed. The reader may provide the card with a clock signal, which incorporates short deviations from the standard signal, which are beyond the required tolerance bounds. Such signals are called glitches. Glitches can be defined by a range of different parameters and they can be used to induce memory faults as well as to cause a faulty execution behaviour. Hence, the possible effects are the same as for spike attacks. If the chip is unpacked, such that the silicon layer is visible, it is possible to use a laser to induce perturbation in the memory cells (Figure 3 represents a laser bench which is used to create faults in an integrated circuit. This equipment is mainly constituted of laser sources, of a microscope with different lenses, an X-Y table and an oscilloscope). These memory cells, *i.e.*, EEPROM memory and semiconductor transistors, have been found to be sensitive to light. This happens if the photon energy of the applied light is transformed in electron in the semiconductor. Modern green or red lasers can be focused on relatively small regions of a chip, such that faults can be targeted fairly well. The last method use changes in the external electrical field and has been considered as a possible method for inducing faults into smart cards. Here, faults are sought to be induced by placing the device in an electromagnetic field, which may influence the transistors and memory cells. A rougher technique consists in modifying the circuit's operation by modifying internal computation through micro-probes, or even modifying the circuit itself by using focused ion beam.

There are three subcategories of key recovering techniques that use the results of faults attacks. Algorithm modifications consist either in reducing the ciphering complexity of the cryptographic algorithm [CT05, DMN$^+$12] or in bypassing hardware or software protections. Differential Fault Attack (DFA), originally described in [BDL97, BS97] and enhanced in [PQ03, MSS06, Gir07, AMT12], consists in retrieving the key by comparing the correct ciphertexts with faulty ones. A detailed comparison of DFA schemes against AES, for example, is given in [SLOI12]. To perform the third kind of fault attacks, called safe-error attacks [JMR07, RM07, LSG$^+$10], the attacker does not necessarily need pairs of correct and faulty ciphertexts but only some information about the chip's behavior. The protections developped against these attacks are based on fault models. Different models have been

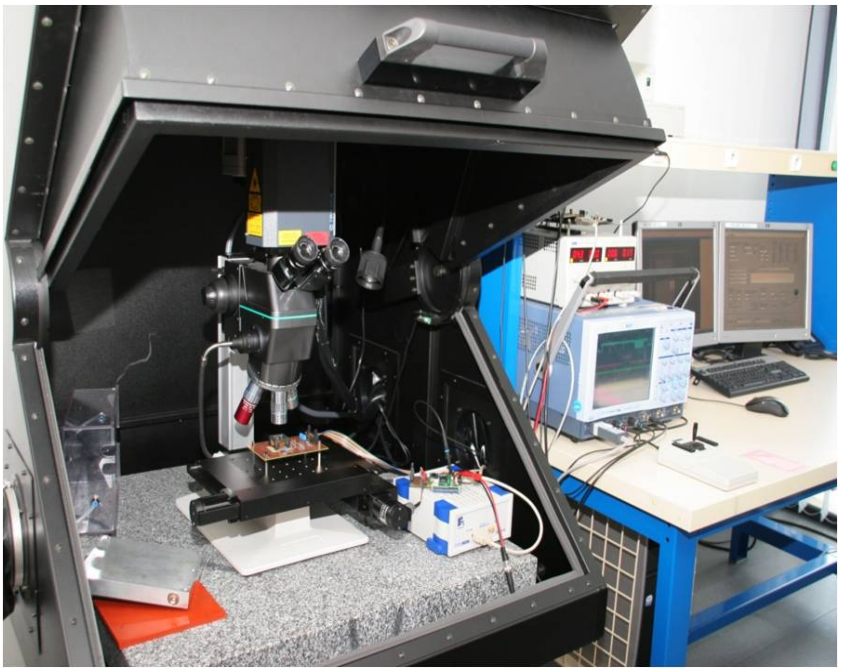

Figure 3: Laser bench (courtesy of the Micropacks platform at GARDANNE, France)

proposed in the litterature.

2.1.4 Fault Model

To prevent a fault attack from happening, we need to know its effects on the smart card. Fault models have already been discussed in details [BOS03, Wag04]. We describe in Table 1 the fault models in descending order in terms of attacker power, considering that an attacker can change one byte at a time. Several authors discuss in [SA02, ADM+10] an attack using the precise bit error model. But it is not realistic on current smart cards, because modern components implement hardware security on memories like error correction and detection code or memory encryption.

In real life, an attacker physically injects energy in a memory cell to change its state. Thus and up to the underlying technology, the memory physically takes the value `0x00` or `0xFF`. If memories are encrypted, the physical value becomes a random value (more precisely a value which depends on the data, the address, and an encryption key). To be as close as possible to the reality, we choose the precise byte error that is the most realistic fault model. Thus, we have assumed that an attacker can:

- make a fault injection at a precise clock cycle (he can target any operation he wants),
- only set or reset a byte to `0x00` or to `0xFF` up to the underlying technology (bsr[1] fault type), or he can change this byte to a random value beyond his control (random fault type),
- target any memory cell he wishes (he can target a specific variable or register).

We have defined the hypothesis concerning the attacker, then we present the countermeasures embedded in most modern smart cards in order to detect the induced fault.

[1] bit set or reset

Table 1: Existing Fault Model

Fault Model	Precision	Location	Timing	Fault Type	Difficulty
Precise bit error	bit	total control	total control	bsr, random	++
Precise bit error	byte	total control	total control	bsr, random	+
Precise bit error	byte	loose control	total control	bsr, random	-
Precise bit error	variable	no control	no control	random	–

2.2 Logical Attacks

Logical attacks, or software attacks, aim at injecting code inside the smart card and retrieving information illegally. At a first sight, it seems that logical attacks can not be mitigated with runtime code generation. But one of the patented countermeasures presented by Barbu [Bar12] and improved by [TR12] at the Java Card level consists in modifying dynamically the syntax of the byte code. The main idea is to avoid the execution of shell code. In this section we present briefly a state of the art of the logical attacks.

2.2.1 Ambiguity in the specification: type confusion

Erik Poll made a presentation at CARDIS'08 about attacks on smart cards. In his paper [HP04], he did a quick overview of the classical attacks available on smart cards and gave some counter-measures. He explained the different kinds of attacks and the associated counter-measures. He described four methods (1) CAP file manipulation, (2) Fault injection, (3) Shareable interfaces mechanisms abuse and (4) Transaction Mechanisms abuse.

CAP file manipulation and Fault injection The goal of CAP file manipulation (1) is to modify the CAP file after the compilation step to bypass the Byte Code Verifier (BCV). This is an easy attack, simple to set up but it can be mitigated using a code analysis step (byte code verification, rules checker, etc.). It has been demonstrated by Barbu in his thesis [Bar12] that passing a BCV verification is not enough. What you execute is not always what you verify. The verification is done during the load and the code stored inside the card can be latter modified using a perturbation. This comes from the solution (2), the fault injection, which sort of attack is efficient but quite difficult and expensive.

Shareable interfaces mechanisms abuse This attack is based on separate compilation unit. The idea to abuse shareable interfaces is really interesting and can lead tricking the virtual machine. The main goal is to have type confusion without the need to modify CAP files. To do that, the authors of [HP04] demonstrate this possibility by creating two applets which communicate using the shareable interface mechanism. To create a type confusion, each of the applets use a different type of array to exchange data. The first applet is compiled proposing a service with an array of bytes as parameter. The second applet is compiled using an array of shorts as parameter. Then the second will access more byte than expected by the first one. During compilation or at load time, there is no way for the BCV to detect such a problem.

Transaction Mechanisms abuse The purpose of transaction is to provide groups of atomic operations. Of course, it is a widely used concept, like in databases, but still hard to implement. In addition to atomicity, by definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction, and should void references to such objects. However, Erik Poll found some cases where the card keeps the references of objects allocated during the transaction even after a rollback, showing that the confidentiality and the integrity of the data can be breached.

2.2.2 The specification is correct but the constraints provide implementation choices: EMAN1 and EMAN2

Previous attacks aims to obtain illegally data. In this category the attacks aim at gaining control of the program and to mutate the code, *i.e.* allowing to execute arbitrary code. The executed code is different from the loaded code.

EMAN1: A self-modify mutant application. J. Iguchi-Cartigny *et al.* explained in [ICL10] the way to execute some malicious byte codes. The idea of this attack is to abuse the firewall mechanism with the unchecked instructions on static operations (as `getstatic`, `putstatic` and `invokestatic`) to call a malicious byte code contained in a Java Card applet. The attack is split it in three steps: (1) First, they try to obtain an address of an array which will contain latter a shell code and the `this` reference address (in order to get access to the instructions array). For that, a Java Card function is modified in order to push, on the top of the Java Card stack, the reference of the array which is given in the function parameter. When this reference is pushed, they *noped* each instructions between *the pushed reference* and the function that returns the value pushed on the top of the stack. (2) Next, to read and write in the memory, the `getstatic` and `putstatic` instructions are used. The Java Card firewall does not check their parameters. (3) Finally, with the modification of its instruction, in a malicious CAP file, the parameter of `invokestatic` instruction may redirect the control flow to the array that contain the shell code.

EMAN2: A Java Card Stack Overflow At CARDIS 2011, G. Bouffard *et al.* described, in [BICL11], two methods to change the Java Card control flow graph. The first one, EMAN2, modifies the return address of the current function stored in the Java Card stack header. Most of the JCVM implentations store the return address between the locals area and the stack area. There is two possibilities to modify the return address by an underflow (from the stack) or an overflow (from the locals). Changing this address by an array address leads the possibility to execute any arbitrary shell code. To modify this address with an overflow, they changed the parameter of the `sstore` instruction *i.e.* they used a non defined local index that refer to the return address.

Recently, Faugeron [Fau13] presented a way to fool the Java Card runtime based on the `dup_x` instruction. This instruction duplicates the top of operands stack words and inserts them below. This instruction duplicates the top k of the operand stack n elements down the stack. This instruction takes two parameters encoded on 1 byte where k is the denotes the high nibble, describing the number of words to duplicate, and n denotes the low nibble describing where the duplicated words are inserted. Since the Java Card operands stack does not contain enough elements, the runtime uses the system data as words for the `dup_x` instruction. Thus, an attacker can shift the value of the frame header by a custom words pushed on the stack.

Another implementation of this attack from the top of stack can exploit the `swap_x` instruction. This instruction swaps words on the top of the operand stack. The `swap_x` takes as parameters the value `m,n` and swaps `m` words with `n` words. If the stack contains less than `m+n` words, the `swap_x` instruction will trigger a stack underflow. With the appropriate values, the frame header can be overwritten. As one can see in [Fau13], this attack only works on a card which supports the integer type.

2.2.3 Combined attacks

Logical attacks can be leveraged with fault attacks. This latter can be used to activate a logical attack. The next section explain this sort of hardware attack we just introduce here the main idea. Faults can be injected by some physical attacks by exposing the device to some sort of physical stress [SA02], but researchers [ZCM+96] also highlighted the fact that cosmic rays can flip single bits in the memory of an electronic device.

Fault attacks aim at modifying the behaviour of the device, for example by changing values in memory cells, transmitting different signals through bus lines, or damaging the structural elements. These errors can generate different versions of a program by changing some instructions, interpreting operands as instructions, branching to other (or invalid) labels and so on. These perturbations can have various effects on the chip registers (program counter, stack pointer), or on the memories (variables and code can change). Mainly, it would enable an attacker to execute an operation beyond his rights, or to access secret data in the smart card. Fault attacks is an old research field mainly in avionics or space domains.

Barbu *et al.* proposed [BTG10] an attack that uses a precise model of byte errors. An applet gets installed on the card after it has been checked by an embedded BCV. It is then considered as a structurally valid applet. The aim of the attack is to create a type confusion to forge a reference of an object. The authors also explained the principle of instance confusion, similar to the idea of type confusion where the objective is to confuse an instance of object A to an object B by dynamically inducing a fault using a laser beam during the `checkcast` instruction. In the case of this study, the attack was simplified as compared to a real case, considering the fact that the authors also designed the target platform, which provided them a complete knowledge of the JCVM internals.

Following the idea of Barbu *et al.*, but with a black box approach, Bouffard *et al.* [BICL11] designed a valid applet that contains a malicious function. Their attack is entitled EMAN4. After the building step by the Java Card toolchain, a valid byte code is obtained. The `goto_w` instruction provides the jump to the beginning of the loop. Here, `0xFF19` is a signed number used to define the destination offset of the `goto_w` instruction. A laser beam may set or reset the most significant byte of the `goto_w` offset. The authors succeeded to shift the most significant byte of the `goto_w` parameter in order to jump outside the method and change the execution flow by executing another fragment of code.

3 Current countermeasures embedded into smart cards

3.1 Hardware countermeasures

Many hardware protections have therefore been proposed to counter hardware attacks. Some protections (hereafter referred as 'sensors') give information about the state of the system either by measuring the light [DPBP+13], the voltage [LZ14], the frequency or the temperature of the chip or by detecting errors during computations. This detection is generally based on spatial redundancy (i.e. making the same computation several times simultaneously), temporal redundancy (i.e. doing the same computation several times) or information redundancy (i.e. doing a computation with more bits than required) [MKRM11, JMR07, NRAD11]. Several mechanisms are also proposed to detect a modification of the execution flow of a software. In some cases, an additional hardware block is dedicated to this task [ARRJ05, MW10]. To reduce sensitivity to side channel attacks, 'noise' has been added to the power consumption, for example, by using an internal clock, by randomizing the order of the instructions, by adding dummy operations or by masking the internal computations that can be predicted by the attacker [CG01, HOM06, RPD09, MDF+09, CK10]. Another way to reduce sensitivity to side channel attacks consists in reducing the correlation between physical values (such as power consumption or electromagnetic radiation) and the data processed, for example, by using balanced data encoding and balanced place and route [GSF+10, DVDF+11], by using power filters or electromagnetic shields [Sha00]. At last, some countermeasures modify the functional behavior of the circuit in case of attacks. Such reactions may consist, for example, in temporarily stopping the communication with the reader (the card 'mutes') and/or resetting parts of the running software. The ultimate reaction consists in permanently destroying (*i.e.* killing) all the data (including sensitive information) stored in the chip.

Using only hardware countermeasures has two drawbacks. Highly reliable countermeasures are very expensive and low cost countermeasures only detect specific attacks. Since new fault attacks are being developed frequently these days, detecting only currently known forms of physical tampering is not sufficient and for long term applications (an e-passport must be valid for 10 years) it is definitely not sufficient.

3.2 Software countermeasures

Software countermeasures are introduced at different stages of the development process; their purpose is to strengthen the application code against fault injection attacks. Current approaches for software countermeasures include checksums, randomization, masking, variable redundancy, and counters. Software countermeasures can be classified by their end purpose:

- *Cryptographic countermeasures*: better implementation of the cryptographic algorithm like RSA (which is the most frequently used public key algorithm in smart cards), AES, and hash functions (e.g. the family of SHA functions).
- *Applicative countermeasures* only modify the application with the objective to provide resistance to fault injection. Generally, this class produces application with a greater size. Because beside the functional code (the code that process data), we have the security code and the data structure for enforcing the security mechanism embedded in the application. Java is an interpreted language therefore it is slower to execute than a native language (like C or assembler),

so this category of countermeasures suffers of bad execution time and increases the complexity of application development.

- *System countermeasures* harden the system by checking that applications are executing in a safe environment. The main advantage is that the system and the protections are stored in the ROM, which is a less critical resource than the EEPROM and cannot be attacked thanks to checksum mechanisms that allow to identify modification of data that are stored in the ROM. Thus, it is easier to deal with integration of the security data structures and code in the system. Another thing that must be considered is the CPU overhead, if we add some treatments to the functional code.

COGITO aims to improve system countermeasures. We develop below the known countermeasures embedded in some smart cards.

3.2.1 Code obfuscation

We consider code obfuscation as a general protection. We use the term *general* to express the fact that, whatever the attack, one key parameter of the JIL quotation [JIL06] is the knowledge of the attacker about the hardware and the software implementation of security primitives: the more the attacker needs such a knowledge to mount an effective attack, the higher the overall security level is. Thus, code obfuscation (i.e. making code execution harder to interpret and so, knowledge about the circuit implementation harder to recover) is also an effective means to secure the component.

In order to protect the confidentiality of programs, numerous techniques for program obfuscation and for the rewriting of binary code have been developed: random jumps in the program instructions, function pointers, concatenation of procedures, etc. [CTL97]. At the theoretical level, it was demonstrated by Barak et al. that there is no general obfuscation technique that applies to any code function; in other words, there is no "universal obfuscating tool" [BGI+01]. However, nothing in this theory forbids the existence of ad hoc obfuscation techniques able to target the mostly used cryptographic algorithms such as DES, AES, RSA, ECC. As a matter of fact, many obfuscating methods have recently been published for cryptography, but all of them revealed being weak. For example, an obfuscation method for AES has been proposed by Chow et al. [CEJO03b], but it was cryptanalysed only two years later [BGEC04]. Similarly, an obfuscated implementation was proposed for DES [CEJO03a] and cryptanalysed in 2007 [WMGP07].

3.2.2 Dynamic Syntax Interpretation

In his PhD thesis, G. Barbu [Bar12] proposed a countermeasure that prevents the execution of malicious bytecode. His idea is to mask each instruction during the installation step. To achieve this, each Java Card instruction `ins` performs a xor with the K_{xor} key. The hidden instructions (and their parameters) perform the following operation:

$$ins_{hidden} = ins \oplus K_{xor}$$

If an attack such as EMAN 2 succeeds, the attacker cannot execute her malicious byte code without the knowledge of the K_{xor} key. To find the xor key, she only has to change the CFG of the program to a `return` instruction. As defined by the Java Card specification [Ora11], the associated opcode is

`0x7A`. With a 1-byte xor key, this instruction may have 256 possibles values. A brute force attack offers the way to find the xor key.

To improve his countermeasure, we add the value of the Java Pointer Counter (`jpc`) to perform of the hidden instruction. This compute becomes:

$$ins_{hidden} = ins \oplus K_{xor} \oplus jpc$$

The `jpc` value depending of where each instruction is stored in the smart card memory. A modification of the CFG prevents the next instruction to be executed from a correct uncypher. Without the knowledge of where each instruction is stored in the EEPROM memory, an attacker will not have the possibility to execute some malicious byte code by the attacked JCVM.

3.2.3 Code polymorphism

In our point of view, the ultimate countermeasure for program obfuscation is the ability to modify online and dynamically the contents of a binary program. To the best of our knowledge, we found only two papers that try to implement such a protection for security and cryptography.

Amarilli et al. [AMN+11] present an implementation of self-modifying code in Scheme, which is an Lisp-based interpreted language. They describe a model for the randomisation of operations, both at block level and at instruction level. But their implementation is not applicable to processors typically used in cryptographic devices because the Scheme language lacks interpreters for such small embedded systems.

Agosta et al. [ABP12] describe a general technique that uses a modified version of the gcc compiler to produce runtime code generators and apply this technique to obfuscate the implementation of the AES. Their technique allows masking and hiding through the use of (1) instructions shuffling, (2) generation of binary code that is functionally equivalent are the level of language expressions (called *code morphing*) and (3) register renaming. They demonstrate the efficiency of their approach for an ARM926 processor, which presents very different characteristics than the typical platforms in secure elements in terms of computing power and of memory resources. The runtime code generators randomize the final binary code thanks to the embedding in the runtime engine of a lot of variants of small code fragments. The memory footprint is not detailed in the paper, but we assume it to be fairly large and maybe incompatible with the memory resources of secure elements. Furthermore, the overhead due to code generation is however weighty: in order to generate the code fragments replacing the 64 `eor` instructions in the AES kernel, the runtime code morphing executes in 90 ms ($11,970.10^6$ cycles at 133 MHz). Considering that their code morphing technique produces in average 6 instructions per morphed original instruction (4 instructions in average, plus a load/store pair to handle the clobbering of a temporary register), the cost runtime generation averages to 187.10^3 cycles per morphed instruction.

3.3 Synthesis

As seen in this section, the first threat lies in the many opportunities to execute shell code in a card, thanks to the allowance of code upload in the post issuance phase. Smart card manufacturers design

several countermeasures, each one being specific to a given attack. The second thread is related to reverse engineering of the embedded application. The reverse can be done at the native level or at the virtual processor level. To the best of our knowledge, only very few countermeasures are applicable to secure components; the only generic countermeasure targeting JavaCards is authored by Guillaume Barbu with a dynamic syntax interpretation. Nevertheless, using side channel analysis one can easily reverse the application or deduce the masking function. In fact, whatever the coding is, the execution trace will remain the same allowing the attacker to deduce the masking function. The only solution for side channel analysis would be the execution of polymorphic code because the trace could not so easily lead to reverse the code or the scrambling function.

In the scope of the COGITO project,

- the Java interpreter will be one of the components that we target for applying runtime code generation. Indeed, if the Java interpreter uses different execution functions for the same instruction, thanks to code polymorphism, this will increase the difficulty for the attacker.
- We intend to provide implementations of cryptographic primitives such as AES, using code polymorphism achieved thanks to runtime code generation.

4 deGoal: code polymorphism for embedded systems

4.1 Sketching deGoal

The CEA is developing a technology for runtime code generation around a tool called deGoal, which initial motivation comes from the observation that program performance may be strongly correlated to the characteristics of the data to process. In this section, we sketch the characteristics of deGoal and then elaborate about its use in the context of security. For further reading about the motivation and the use of deGoal in other contexts or application domains, refer in particular to the tutorial in [CLC13], and [AC13, CCL+14].

Before presenting in more details the possibilities for runtime code generation that this tool can bring in secure devices, we introduce the initial design of this tool in order to clarify the design choices and the scientific positioning in deGoal with regards to the state of the art in dynamic compilation.

Most of the times, a static compiler (such as gcc) is used to generate binary code from the original implementation in a programming language. Because compilation is performed *before* the program is run, the execution context and run-time data are not known at the time of code generation. This means that, in order to take advantage of such information in code optimisations, one has either (1) to assume about the characteristics of the execution context (and to provide verification mechanisms), (2) to add extra instructions to adapt the program behavior depending on runtime data, which is known as code specialisation, or (3) to generate the program's machine code at run-time, after the execution context is known. This latter solution is the core motivation for the development of deGoal: to provide application developers with the ability to implement application kernels tunable at run-time depending on the execution context, on the characteristics on the target processor, and furthermore on the *data to process* [Cha12].

In classical frameworks for runtime code generation, such as interpreters and dynamic compilers, the aim is to provide a generic infrastructure for code generation, bounded by the syntactic and semantic definition of a programming language. The generality of such solutions comes at the expense of an important overhead in code generation, both in terms of memory footprint and computing power. In deGoal, we chose a different approach: application kernels are tuned at runtime by *ad hoc* run-time code generators embedded in the application (one code generator for each kernel to optimise). Each code generator is specialised to produce the machine code of one application kernel. Syntactic and semantic analyses are performed at the time of static compilation, and the dynamic code generators embed only the processing intelligence that is necessary to adapt the code to generate to the properties of the data to process. As a consequence, compilettes offer very fast code generation (10 to 100 times faster than typical frameworks for runtime code interpretation or dynamic compilation), present a very low memory footprint, can run on very small microcontroller architectures such as 8/16-bit microcontrollers with less than 1 kB of RAM [AC13], and are portable.

4.2 Components for code generation

The two categories of software components around which our code generation technique is organised are called *kernels* and *compilettes*.

(a) dynamic compiler (b) compilette

Figure 4: Simplified principles of operation of a dynamic compiler and of a compilette

Kernel A kernel is a small portion of code, which is part of a larger application, and which is the target of our runtime code generation setup. Our technique focuses on the optimisation at run-time of these small parts of a larger application in order to improve the kernel's performance. In the context of the typical use of `deGoal`, good performance is understood as low execution time, low memory footprint and low energy consumption. In the context of this project, good performance could be understood as the capability to obtain a very large number of different versions of the binary code while preserving the same functionality.

Compilette A compilette is designed to generate the code of *one* kernel at run-time. A compilette can be understood as an *ad hoc* small code generator that is executed during application runtime. We use the term *compilette* to underline the fact that, in order to achieve very fast code generation, this small run-time generator does not embed all the optimisation techniques usually carried out by a static compiler.

A compilette is not a dynamic compiler. Figure 4 presents an overly simplified sketch of the working principle of compilettes, and how it compares with dynamic compilation:

- A dynamic compiler is a *generic* code generator (Figure 4a). It takes the description of a program as an input, and produces machine code. Dynamic compilers target a specific programming language, but we describe them as generic because they are able to process whatever program description that is compliant with the specification of the targeted programming language. The input program is most of the time represented in a simplified form called *bytecode*. Bytecode is a pre-processed (or pre-compiled) program representation that can be translated efficiently in machine instructions (interpretation), but that also contains enough information about the input program so that is it possible to produce optimized code by dynamically compiling it at runtime. To the best of our knowledge, dynamic compilers are very unlikely to perform runtime optimizations based on the knowledge of the execution context, and of the data to process.

- On the contrary, a compilette (Figure 4b) is a *specialized* code generator able to produce machine code for a class of kernels known before runtime. As a consequence, it does not take the representation of a program as input. It is hence capable of more lightweight and faster runtime code generation than dynamic compilers. A compilette is instead designed for data-dependent code optimizations at runtime: the main input of a compilette is data, either related to the execution context or to the data to process.

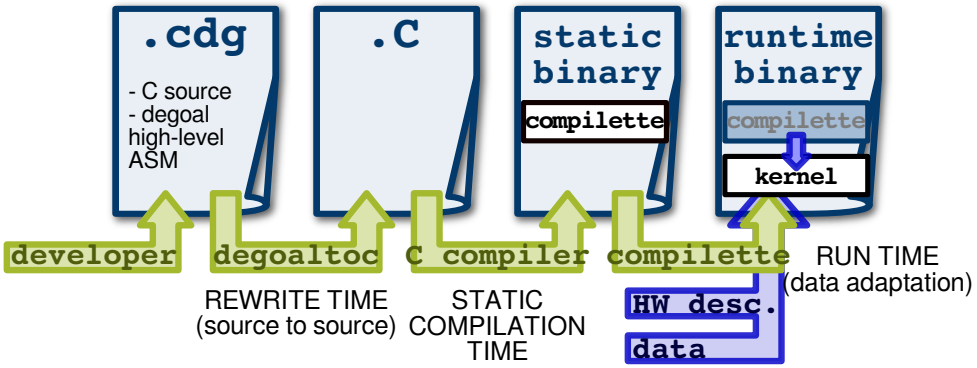

Figure 5: Illustration of `deGoal` workflow

4.3 Application building, runtime code generation and execution

The building and the execution of an application using `deGoal` consists of the following steps as illustrated in Figure 5: writing the source code; compiling the binary code of the application and the binary code of compilettes using static tools; generating the binary code of kernels by compilettes; running the kernels. These steps are explained below:

Writing the source code (application development time): This task is handled by the application developer, and/or by high-level tools. The source code of compilettes is written in specialised `.cdg` files, while the rest of the application software components are written using a standard programming language, such as C.

Generation of C source files (rewrite time): This step consists in a source-to-source transformation: the `.cdg` source files mixing high-level ASM instructions and standard C are translated into standard `C` source files. At this phase architecture-dependent features can be introduced in the C source files generated, for example pre-processing of register allocation and vectorization support.

Compilation of the application (static compilation time): The source code of the application now consists in a set of standard C source files, including the source code of the compilettes. The binary code of the application is produced by a standard C compiler. This step is the same as in the development of a standard C application.

Generation of kernel's binary code (run-time): At run-time, the compilette generates binary code for the kernel(s) to optimise. This task can be executed on a processor that is different to the processor that will later run the kernel. Furthermore, the processor executing the compilette and the processor targeted for the execution of the kernel do not necessarily need to have the same architecture. In the typical use of `deGoal`, a compilette can be run several times, for example as soon as the characteristics of the data to process have changed, in order to re-optimize the binary code of the processing kernel.

Kernel execution (run-time): The binary code of the kernel is executed on the target processor (not shown in Figure 5).

In COGITO, we will re-target the original use of `deGoal` in order to focus on security aspects: our objective is to exploit the flexibility brought by `deGoal` for runtime code generation to introduce dynamic variability in the binary code of applications.

4.4 The `Cdg` language

4.4.1 Introduction

`Cdg` is an assembly-like DSL language. It is designed for the software implementation of compilettes: it describes the instructions that will be generated at run-time. From the programmer's perspective, this represents a major paradigm shift: programming languages usually describe in a more or less straightforward manner the instructions to be *executed* by the target processor, whereas `Cdg` describes instructions to be *generated* or written into program memory during the execution of a compilette.

Compilettes are implemented by using a mix of ANSI `C` and `Cdg` instructions [Cha12, CLC13]. The `C` language is used to describe the control part of the compilette that will drive code generation, while `Cdg` instructions perform code generation.

The `Cdg` instruction set includes:

A variable length register set The instruction set uses vectorial registers with variable width and a variable number of elements. For example, it is possible to define

```
Type floatvect float 64 8
```

in order to use any register of type `floatvect` as a vector of 8 elements of 64 bit floating point values.

Classical arithmetic instructions `add`, `sub`, `mul`, `div`, but also instructions specific to the multimedia domain such as `sad` (sum of absolute differences), `mma` (matrix multiply and add) and FFT butterfly. These instructions can work on registers of variable length and type.

Load and store This family of instructions supports stride description. This allows for the description of complex memory access patterns.

From this high-level instruction set, compilettes map the `Cdg` instructions to machine instructions according to according to (1) the characteristics of the data to process, (2) the characteristics of the execution context at the time of code generation, (3) the hardware capabilities of the target processor , (4) execution time and/or energy consumption performance criteria. In all cases, code generation is fast and produces efficient code.

The main purpose of `Cdg` being *runtime* code generation, two main features are of particular importance:

1. parametrising instructions with values known at runtime only (evaluation of runtime constants),
2. use vector variables (i.e. variables describing vectors, at the opposite of scalar variables), whose size is known at runtime only.

```
1 int multiply(int a, int b)
2 {
3     return (a*b);
4 }
```

Figure 6: Simple integer multiplication; implementation in C

```
1 0001008c <multiply_func_classical>:
2    1008c:  fb01 f000    mul.w   r0, r1, r0
3    10090:  4770         bx      lr
4    10092:  bf00         nop
```

Figure 7: Simple integer multiplication; machine code corresponding to the C version (arm-thumb ISA)

4.4.2 Evaluation of runtime constants: illustration with a minimalist example

We illustrate here the evaluation of runtime constants in the `Cdg` language through a minimalist example: we illustrate the use of a compilette to generate the code of a simple routine for integer multiplication.

A naive implementation in `C` could be as illustrated in Figure 6, and the corresponding machine code compiled for the arm-thumb ISA as illustrated in Figure 7.

To generate at runtime machine code that is functionally equivalent, we can implement a compilette as illustrated in Figure 8. The use of the `Cdg` language is explained in greater details in the tutorial section in [CLC13], but lets cover the implementation of the compilette in a few words:

- `Cdg` instructions are code sections enclosed by `#[` and `]#` (lines 5, 11)
- the `Begin` instruction (line 6) indicates where to start code generation (memory address stored in the variable `code`)
- the compilette takes as an input argument the virtual register named `input` (line 6)
- and produces the result in the virtual register `output` (line 6)
- the variable `val` is an argument for the compilette `multiply_compile` (line 3). `val` is evaluated at runtime, at the time of code generation, and considered as a constant for the code generation of the machine code corresponding to the `Cdg` instruction `mul` (line 8)

Figure 9 illustrates the code that is generated at runtime by the compilette `multiply_compile` from Figure 8. Clearly the generated code will perform worse than the original implementation in `C` on such a simple example. However, it should also be clear to the reader that it is possible to produce *at runtime* many functionally-equivalent variants of a program routine (here the multiply function). For example, considering that the variable `val` equals to a power of two, denoted 2^n, it is possible to generate the instruction `lsl.w` to execute a left shift of n bits instead of producing a multiply machine instruction `mul.w`.

```
1  /* a compilette that multiplies with a constant value */
2  typedef int (*pifi)(int);
3  pifi multiply_compile(cdg_insn_t *code, int val)
4  {
5    #[
6      Begin code Prelude result = input
7
8      mul result, input, #(val)
9      rtn
10     End
11   ]#;
12   return (pifi)code;
13 }
```

Figure 8: Simple integer multiplication; implementation in a compilette

```
0x7a001:        stmdb   sp!, {r7}
0x7a00d:        movs    r7, #42
0x7a00f:        mul.w   r0, r0, r7
0x7a017:        ldmia.w sp!, {r7}
0x7a01b:        bx      lr
```

Figure 9: Simple integer multiplication; output of the compilette multiply_compile for val=42

4.4.3 Virtual registers, vectorial registers

The variables manipulated in `Cdg` are similar to assembly registers. We however describe them as virtual and vectorial registers:

virtual the mapping to physical registers and the vector size is determined at runtime, at the time of code generation, when the use of the physical registers in the programming context is known.

vectorial The use of vectorial registers allows to write vectorized data processing in a quite natural way. The purpose is also to map `Cdg` instructions to vector machine instructions when they are available on the target processor.

`Cdg` virtual registers are typed, so that it is possible to match the assembly instructions with machine arithmetic operators according to type information.

4.5 Main properties of `deGoal` backends for code generation

Similar to the software architecture of standard compilers, `deGoal` backends for code generation are architecture-dependant. The main difference is however that the backend is aimed to be invocated *at runtime*. As a consequence, drastic architecture choices have been made so that code generation can be fast and memory lightweight.

At runtime, a compilette performs in this order:

1. register allocation
2. instruction selection
3. instruction scheduling

4.5.1 Register allocation

In dynamic compilers, register allocation is performed *after* instruction scheduling: instruction selection and instruction scheduling are performed on a SSA form. Later the SSA form is analysed and register allocation is performed. Even if not optimal as compared to graph-colouring techniques, suitable approaches for runtime allocation have been developped to provide a good compromise between code quality and computational cost of register allocation [KWM+08].

On the contrary, in a compilette register allocation is done first. The idea is to lighten the pressure on instruction selection and instruction scheduling: if register allocation is done first, it becomes possible to perform instruction scheduling without intermediate representation [2]. This comes at the expense of a potential reduced code quality: for example, compilettes do not support registers spilling.

4.5.2 Instruction selection

Instruction selection is performed at runtime once the runtime constants have been evaluated by the compilette. Instruction selection is done at the level of `Cdg` instructions: each `Cdg` instruction can be mapped to one or more machine instructions depending on the execution context of the compilette. For example, the `mul Cdg` instruction can be mapped on the left shift `lsl.w` instruction for particular operands, or to SIMD instructions when the target processor has a vector unit.

The `deGoal` framework provides a simple mechanism so that a software developer can extend the set of instruction selectors.

4.5.3 Instruction scheduling

The generated machine instructions can be either directly written in program memory in order to fasten code generation, or pushed into an intermediate instruction buffer that is processed by an instruction scheduler. The purpose of the instruction scheduler was initially to create instruction bundles for VLIW processors, but we plan to exploit this feature in the COGITO project to perform instruction shuffling during runtime code generation. A functional overview of instruction scheduling for a VLIW processor is illustrated in [CLC13].

The contents of the instruction buffer is monitored by the instruction scheduler, which regularly flushes the contents of the instruction buffer to program memory.

[2] or at least with a minimalist, much lighter intermediate representation

5 Use of `deGoal` in the context of COGITO

As explained in section 3, the state of the art protections present the following limitations:

1. *General countermeasures*, which make the code execution harder to interpret and so the knowledge about the circuit's functioning harder to recover, are a primary requirement in secure devices. However, state-of-the-art solutions present performance issues or are not applicable to secure devices because of resource constraints.
2. *Security in a device is incrementally built by adding standalone ad hoc countermeasures*. Ad hoc countermeasures are often proven to be efficient when analysed independently, but are difficult to integrate in a device since each countermeasure comes with an additional performance overhead.

Our objective is to go beyond these limitations by introducing several mechanisms in a unique framework. We list here and detail below the non exhaustive list of mechanisms that will be implemented in `deGoal`.

- Introduce alea during runtime code generation
- Execute different (but functionnaly equivalent) instances of algorithms
- Rename registers on-the-fly
- With the ability to combine with hardware countermeasures
- With limited memory consumption
- With high performances
- and that are portable to very small processors and secure elements

Figure 10 reproduces the workflow of `deGoal` already presented in Figure 5. We have highlighted here in red the changes needed to integrate these mechanisms in this workflow for the purpose of the COGITO project. The proposed mechanisms are detailed below.

Introduce alea during runtime code generation The runtime code generation will vary from one code generation to another, thanks to the use of alea as an input of the compilette (instead of information about the data to process as presented in section 4). Each new code generation would produce a kernel with a different binary code, while preserving its functionality.

Execute different (but functionnaly equivalent) instances of algorithms Each `Cdg` instruction is translated by a compilette into one or more binary code fragments that are functionally equivalent. Similarly to the work of Agosta et al. [ABP12], the writing of the database for equivalent code fragments is currently written by hand. As a future improvement of our toolchain, the code fragments could be automatically generated, similarly to what is done in the backends of static compilers. When the target architecture is composed of several arithmetic units, it becomes possible to select the arithmetic unit used, or to introduce fake computations in parallel in another arithmetic unit to scramble the power consumption.

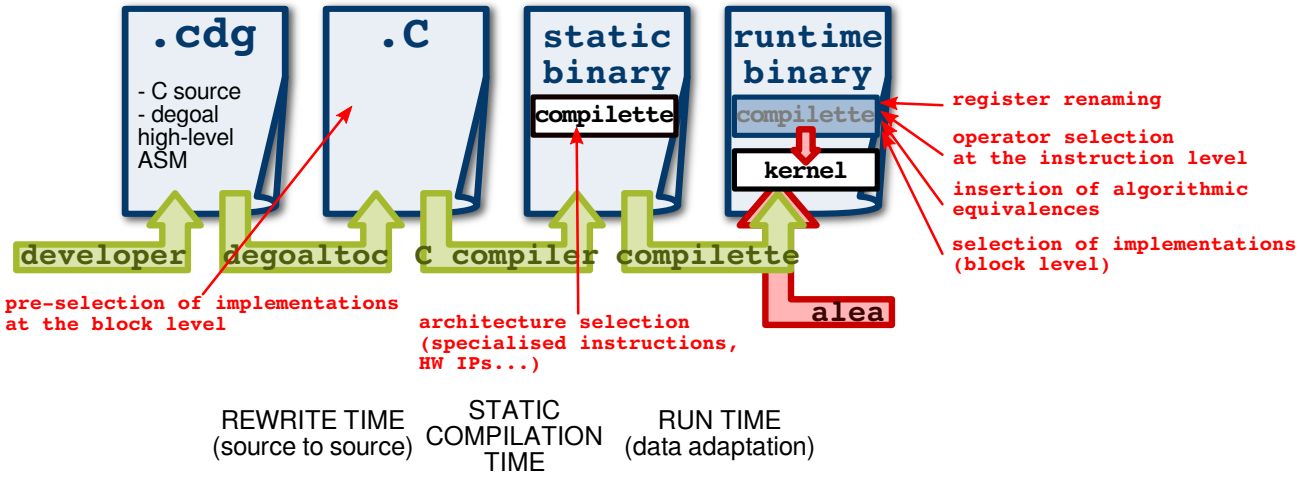

Figure 10: Update of `deGoal` workflow according to the objectives of the COGITO project

Random selection of registers at runtime `deGoal` compilettes embed a mechanism for register allocation. Our design choice was to offer a mechanism for register allocation that is simple enough to maintain a fast code generation but that provides good allocation results. In this project we will integrate simple mechanisms to randomise the register allocation at runtime.

Ability to combine with hardware countermeasures A primary concern in the design of the `Cdg` language was to ease the use of hardware accelerators and specialised instructions. Thus, our compilettes are able to achieve highly effective kernels by using intensively hardware accelerators when available. As a side-effect, it will be easy to integrate the countermeasures implemented with `deGoal` compilettes with state-of-the-art countermeasures, implemented in hardware and in software. Another side-effect is that factorizing several ad hoc implementations will help reducing the overhead incurred by ad hoc countermeasures.

Limited memory consumption The implementation that is closer to what we are able to achieve using `deGoal` is presented by Agosta et al. [ABP12]. Their technique however relies on a knowledge base, embedded in the runtime engine, that contains *all* the versions of the code fragments that can be targeted during code morphing. No figures are provided about the resulting memory consumption, but we expect it to be fairly high. On the contrary, `deGoal` compilettes embed only the necessary data to generate the instructions (and algorithmic equivalences in the case of this project) required for the target kernel.

Speed of runtime code generation Agosta et al. have demonstrated a runtime code generation technique that is close to what we are able to achieve with `deGoal` from a functional point of view. They experimented on an ARM926 processor, which is a relatively large processing platform as compared to the typical processors used in secure elements. In this approach, we estimated the speed of runtime code generation at 187.10^3 cycles per morphed instruction. In `deGoal`, the current speed of code generation ranges from 10 to 100 cycles per instruction generated in an implementation that is not targeting secure devices [CC12]. Even with the addition of extra processing to achieve the insertion of randomisation at various levels, we expect the speed of code generation to remain far above the state of the art. This will allow to randomise the binary code of critical kernels more often to raise the level of security.

Portability to very small processors and secure elements Thanks to the small footprint of compilettes and the small amount of processing they require to achieve code generation, we are able to target very small processing platforms such as secure elements. For illustrative purpose, we have ported `deGoal` on the microcontroller MSP430 from Texas Instruments [Cha12], which is a small 16-bit microcontroller with very limited memory resources: our development platform contains only 512B of RAM and 8KB of Flash memory.

6 References

[ABP12] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. A code morphing methodology to automate power analysis countermeasures. In *DAC*, pages 77–82. ACM, 2012.

[AC13] Charles Aracil and Damien Couroussé. Software acceleration of floating-point multiplication using runtime code generation. In *Energy Aware Computing Systems and Applications (ICEAC), 2013 4th Annual International Conference on*, pages 18–23, Istanbul, Turkey, December 2013.

[ADM+10] Michel Agoyan, J-M Dutertre, A-P Mirbaha, David Naccache, A-L Ribotta, and Assia Tria. Single-bit dfa using multiple-byte laser fault injection. In *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, pages 113–119. IEEE, 2010.

[AMN+11] Antoine Amarilli, Sascha Müller, David Naccache, Daniel Page, Pablo Rauzy, and Michael Tunstall. Can Code Polymorphism Limit Information Leakage? In *WISTP*, volume LNCS 6633, pages 1–21, 2011.

[AMT12] Subidh Ali, Debdeep Mukhopadhyay, and Michael Tunstall. Differential fault analysis of AES: Towards reaching its limits. *IACR Cryptology ePrint Archive*, 2012:446, 2012.

[ARRJ05] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jhaals. Secure Embedded Processing through Hardware-Assisted Run-Time Monitoring. In *Proc. Design, Automation and Test in Europe − DATE, IEEE CS*, volume 1, pages 178–183, 2005.

[Bar12] Guillaume Barbu. *On the security of Java Card platforms against hardware attacks.* PhD thesis, Grant-funded with Oberthur Technologies and Telecom ParisTech, 2012.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In W. Fumy, editor, *Advances in Cryptology - EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[BGEC04] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box aes implementation. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 227–240. Springer, 2004.

[BGI+01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology—CRYPTO 2001*, pages 1–18. Springer, 2001.

[BICL11] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In *CARDIS*, pages 283–296. 2011.

[BOS03] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. A new CRT-RSA algorithm secure against bellcore attacks. In *ACM Conference on Computer and Communications Security*, pages 311–320, 2003.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In B.S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.

[BTG10] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In *CARDIS*, pages 148–163, 2010.

[CC12] Damien Couroussé and Henri-Pierre Charles. Dynamic code generation: An experiment on matrix multiplication. In *Proceedings of the WIP Session, LCTES*, 2012.

[CCL+14] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A. Endo, and Rémy Gauguey. degoal a tool to embed dynamic code generators into applications. In *23rd International Conference on Compiler Construction (CC 2014). Proceedings of*, LNCS 8409. Springer Verlag, 2014. forthcoming.

[CEJO03a] Stanley Chow, Phil Eisen, Harold Johnson, and PaulC. Oorschot. A white-box des implementation for drm applications. In Joan Feigenbaum, editor, *Digital Rights Management*, volume 2696 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2003.

[CEJO03b] Stanley Chow, Philip Eisen, Harold Johnson, and PaulC. Oorschot. White-box cryptography and an aes implementation. In Kaisa Nyberg and Howard Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 250–270. Springer Berlin Heidelberg, 2003.

[CG01] Jean-Sébastien Coron and Louis Goubin. On Boolean and arithmetic masking against differential power analysis. In *Cryptographic Hardware and Embedded Systems*, volume 1965 of *Lecture Notes in Computer Science*, pages 231–237. Springer, 2001.

[Cha12] Henri-Pierre Charles. Basic infrastructure for dynamic code generation. In *workshop "Dynamic Compilation Everywhere", in conjunction with the 7th HiPEAC conference*, 2012.

[CK10] Jean-Sébastien Coron and Ilya Kizhvatov. Analysis and improvement of the random delay countermeasure of ches 2009. In *Proceedings of the 12th international conference on Cryptographic hardware and embedded systems*, CHES'10, pages 95–109, Berlin, Heidelberg, 2010. Springer-Verlag.

[CLC13] Damien Couroussé, Victor Lomüller, and Henri-Pierre Charles. *Introduction to Dynamic Code Generation – an Experiment with Matrix Multiplication for the STHORM Platform*, chapter 6, pages 103–124. Springer Verlag, 2013.

[CT05] Hamid Choukri and Michael Tunstall. Round reduction using faults. In *FDTC '05: Proceedings of the second Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 13–24, 2005.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[DMN+12] J-M Dutertre, A-P Mirbaha, David Naccache, A-L Ribotta, Assia Tria, and Thierry Vaschalde. Fault round modification analysis of the advanced encryption standard. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, pages 140–145. IEEE, 2012.

[DPBP+13] Jean-Max Dutertre, R Possamai Bastos, Olivier Potin, Marie-Lise Flottes, Bruno Rouzeyre, and Giorgio Di Natale. Sensitivity tuning of a bulk built-in current sensor for optimal transient-fault detection. *Microelectronics Reliability*, 53(9):1320–1324, 2013.

[DVDF+11] Marion Doulcier-Verdier, J-M Dutertre, Jacques Fournier, J-B Rigaud, Bruno Robisson, and Assia Tria. A side-channel and fault-attack resistant aes circuit working on duplicated complemented values. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 274–276. IEEE, 2011.

[Fau13] Emilie Faugeron. Manipulating the Frame Information with an Underflow Attack. In *CARDIS 2013*, November 27th-29th 2013.

[Gir07] Christophe Giraud. *Attaques de Cryptosystmes Embarqués et Contre-Mesures Associées*. PhD thesis, Université de Versailles Saint-Quentin, 2007.

[GSF+10] Sylvain Guilley, Laurent Sauvage, Florent Flament, Vinh-Nga Vong, Philippe Hoogvorst, and Renaud Pacalet. Evaluation of power constant dual-rail logics countermeasures against DPA with design time security metrics. *IEEE Trans. Computers*, 59(9):1250–1263, 2010.

[HOM06] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An aes smart card implementation resistant to power analysis attacks. In *Applied Cryptography and Network Security*, volume LNCS 3989, pages 239–252. Springer, 2006.

[HP04] E. Hubbers and E. Poll. Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. Technical report, University of Nijmegen, 2004.

[ICL10] Julien Iguchi-Cartigny and Jean-Louis Lanet. Developing a Trojan applets in a smart card. *Journal in Computer Virology*, 6(4):343–351, 2010.

[JIL06] Joint Interpretation Library – Application of Attack Potential to Smartcards. http://www.ssi.gouv.fr, 2006.

[JMR07] Marc Joye, Pascal Manet, and Jean-Baptiste Rigaud. Strengthening hardware AES implementations against fault attacks. *IET Information Security*, 1(3):106–110, September 2007.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[Koc96] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In *Advances in Cryptology - Crypto'96*, pages 104–113, New-York, 1996. Springer-Verlag.

[KWM+08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot client compiler for java 6. *ACM TACO*, 5(1):7:1–7:32, 2008.

[LSG+10] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault sensitivity analysis. In Stefan Mangard and FranÃ§ois-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 320–334. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15031-9_22.

[LZ14] K. Tobich J.-M. Dutertre P. Maurine L. Guillaume-Sage J. Clédiere A. Tria L. Zussa, A. Dehbaoui. Efficiency of a glitch detector against electromagnetic fault injection. In *Proceedings of DATE*, 2014.

[MDF+09] H. Maghrebi, J.-L. Danger, F. Flament, S. Guilley, and L. Sauvage. Evaluation of countermeasure implementations based on boolean masking to thwart side-channel attacks. In *Signals, Circuits and Systems (SCS), 2009 3rd International Conference on*, pages 1 –6, nov. 2009.

[MKRM11] M. Mozaffari-Kermani and A. Reyhani-Masoleh. A lightweight high-performance fault detection scheme for the advanced encryption standard using composite fields. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(1):85 –91, jan. 2011.

[MSS06] Amir Moradi, Mohammad T. Manzuri Shalmani, and Mahmoud Salmasizadeh. A generalized method of differential fault attack against AES cryptosystem. In *CHES*, pages 91–100, 2006.

[MW10] Shufu Mao and T. Wolf. Hardware support for secure processing in embedded systems. *Computers, IEEE Transactions on*, 59(6):847 –854, june 2010.

[NRAD11] Minh-Huu Nguyen, Bruno Robisson, Michel Agoyan, and Nathalie Drach. Low-cost recovery for the code integrity protection in secure embedded processors. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 99–104. IEEE, 2011.

[Ora11] Oracle. *Java Card 3 Platform, Virtual Machine Specification, Classic Edition*. Number 3.0.4. Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065, September 2011.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against SPN structures, with application to the AES and Khazad. In C.D. Walter, editor, *Cryptographic Hardware and Embedded Systems – CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.

[RM07] Bruno Robisson and Pascal Manet. Differential behavioral analysis. In *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 413–426. Springer Berlin Heidelberg, 2007.

[RPD09] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-order masking and shuffling for software implementations of block ciphers. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '09, pages 171–188, Berlin, Heidelberg, 2009. Springer-Verlag.

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In *CHES*, pages 2–12, 2002.

[Sha00] Adi Shamir. Protecting smart cards from passive power analysis with detached power supplies. In *Proceedings of Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, pages 71–77. Springer, 2000.

[SLOI12] K. Sakiyama, Y. Li, K. Ohta, and M. Iwamoto. Information-theoretic approach to optimal differential fault analysis. *Information Forensics and Security, IEEE Transactions on*, 7(1):109 –120, feb. 2012.

[SMP07] Elisabeth Oswald Stefan Mangard and Thomas Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer Verlag, 2007.

[TR12] B. N Thampi-J.-L. Lanet T. Razandralambo, G. Bouffard. A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks. In *SNDS*, pages 185–194, 2012.

[Wag04] David Wagner. Cryptanalysis of a provably secure CRT-RSA algorithm. In *ACM Conference on Computer and Communications Security*, pages 92–97, 2004.

[WMGP07] Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of white-box des implementations with arbitrary external encodings. In *Selected Areas in Cryptography*, pages 264–277. Springer, 2007.

[ZCM+96] James F. Ziegler, Huntington W. Curtis, Hans P. Muhlfeld, Charles J. Montrose, B. Chin, Michael Nicewicz, C. A. Russell, Wen Y. Wang, P. Hosier, L. E. LaFave, James L. Walsh, José M. Orro, G. J. Unger, John M. Ross, Timothy J. O'Gorman, B. Messina, Timothy D. Sullivan, A. J. Sykes, H. Yourke, Thomas A. Enger, Vikram R. Tolat, T. S. Scott, Allen H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus. IBM experiments in soft fails in computer electronics (1978–1994). *IBM Journal of Research and Development*, 40(1):3–18, January 1996.