Chicken or the Egg
Computational Data Attacks or Physical Attacks

Julien Allibert[1], Benoit Feix[1], Georges Gagnerot[2], Ismael Kane[1]
Hugues Thiebeauld[2], and Tiana Razafindralambo[2]

[1] UL Transaction Security, Marseille, France
firstname.lastname@ul.com
[2] eshard, Bordeaux, France
firstname.lastname@eshard.com

Abstract. Side-channel and fault injection analysis are well-known domains that have been used for years to evaluate the resistance of hardware based products. These techniques remain a threat for the secret assets embedded in products like smart cards or System On Chip. But most of these products contain nowadays several strong protections rendering side-channel and fault attacks difficult or not efficient. For two decades now embedded cryptography for payment, pay tv, identity areas have been mainly focused on secure elements. However recently, alternative solutions on mobile phones appeared to offer services including payment and security solutions as the HCE and DRM products. Cryptographic operations running in such applications are then executed most often on unprotected hardware devices. Therefore the binary code is accessible to attackers who can use static and dynamic reverse engineering techniques to extract and analyse operations including data modification as faults. Hence, hiding or obfuscating secrets and/or obfuscated or whitebox-ed cryptography becomes mainly the alternatives to secure element storage for assets. Although not proven secure, attacking such implementations in practice on a binary is another story. We explain in this paper how directly from the binary or with the extracted source code we can perform statistical and fault analysis in a manner that seems familiar with hardware side channel and fault attacks knowledge. The main difference is, using our tool and virtualization technique, an attacker can emulate and trace and modify any chosen computational data (memory or register manipulation, any machine language operation) executed in the mobile application. It means the attacker is not restricted any-more by any physical limitations as the Hamming leakage model (and additional noise) and the difficulty to fault a dedicated operation. Hence statistical and fault attacks becomes more efficient than in standard physical devices. As a consequence, complex techniques like high order, collision and horizontal statistical attacks becomes very efficient and can be easily performed on the computational data execution traces. A similar consequence applies for fault injection attacks. Hence the word statistical and fault analysis on computational data becomes more appropriate and one can wonder who has been the first between computational data or physical attack techniques? Chicken or the Egg?

Keywords: statistical, fault analysis, mobile, obfsuscated crypto, whitebox crypto, embedded cryptography, side-channel, physical attacks, computational data, DRM, HCE.

1 Introduction

Cryptographic implementations have been the heart of security for years as most of the secret assets of any security products are manipulated during their internal computations. These algorithms were embedded in hardware devices like smart cards that offered tampered resistance for years. A first security gap appeared with the first side channel attack publications from Kocher in 1996 [16] and 1998 [17]. From these initial publications, during two decades a myriad of attack paths and statistical techniques have been published. It has been concerning either new attack paths related to different cryptographic algorithms or different statistical treatment (distinguishers) to exploit the statistical dependency between the Hamming weight of the data computed by the hardware device and the physical leakage of the hardware device that is often linear in the Hamming weight of the data allowing the use of the correlation side-channel analysis technique [5] and linear regression analysis technique [9]. The second security gap was initiated by the Bellcore fault attacks on RSA [3] and the differential fault attack on DES [2]. Fault attacks have become today very powerful attacks taking advantage of powerful attack bench user laser or ElectroMagnetic means to inject the fault on the hardware device. For years payment products have been relying on hardware devices and particularly secure elements (i.e. smart cards) including several software and hardware countermeasures to defeat most of the time side-channel and fault injection attacks. This hardware security domain can be seen as a mature security area involving a lot of academic and industrial actors. However for months now a new area is emerging with the increasing use of mobile application for security services. The Host based Card Emulation (HCE in short) taking advantage of the NFC interface has become a serious market that is used for payment services. Even if this new products are connected services several sensitive assets have to be protected in confidentiality and integrity in the mobile memory. That requirement is very difficult to achieve in a open area like Android with the absence of a secure element. Such solutions requires the secret assets to be hidden as best as possible into the embedded code. It can vary from a simple code including the secret in plain to obfuscation techniques used to hide the secret or several whitebox cryptography solutions [] that embeds (hides) the secret key in the code of the algorithm itself with more or less resistance against attacks. Several techniques have been published for years but none of them has been proven secure. However recovering hidden secret in a binary or in a whitebox cryptographic implementation in real product life is very difficult to perform. Indeed for security means the secrets are hidden in a binary within thousands of code lines including sometimes obfuscation and anti tampering techniques. Performing reverse engineering and targeting this cryptographic implementa-

tions can be a considerable waste of time. A first solution would be to perform side-channel analysis on the mobile processor in order to analyse the computations that is possible but it can be quite difficult to perform in practice due to the complexity of the hardware processor of mobile phones.

We present in this paper a very efficient and easy to use solution that allows to perform statistical analysis on the computation and inject fault during these same data computations. We are able to perform statistical attacks and fault attacks in a *perfect* world having direct access to data themselves and being in control and knowledge of the execution environment. We illustrate our techniques with practical results and explain how efficient this innovative technique can be.

Roadmap. The paper is organized as follows. Section 2 reminds basics on physical side-channel and fault attacks and give the reader the necessary knowledge on embedded cryptography including whitebox cryptography for use in mobile like products. We also give the reader the necessary knowledge and background on simple side-channel and fault analysis to understand the attack techniques we are presenting. Section 3 is explaining the simulation and virtualization tools we have developed to perform our attack on the targeted embedded software products. Section 4 describes the statistical computational data analysis we can perform using such data execution traces to recover hiden embedded secrets of the product when Section 5 is explaining the fault injection techniques we can perform using our simulation and virtualization environment. Practical results are presented in section 6 and discussed in section 7. Finally we conclude in section 8.

2 Preliminaries

2.1 Side-channel Analysis

It has been studied for years since it has been introduced by Kocher et al. [16]. Many attack paths have been published on the different cryptosystems like DES [10] and RSA [21] which are widely used in the majority of the hardware embedded devices like Banking or Identity products. In the same time many statistical attack techniques have improved the original Differential Side-Channel Analysis (DSCA) (ie. Difference of Mean - DoM) from Kocher et al. [18]. We can for instance mention the Correlation Side-Channel Analysis (CSCA) introduced by Brier et al. [5], the Mutual Information Side-Channel Analysis (MISCA) from Gierlichs et al. [12] or the Linear Regression Side-Channel Analysis [9,22].

Physical Behavior and Secret Dependancy Side-channel analysis requires some measurements to be performed on the targeted device to extract some in-

formation from its physical characteristics. An electronic device such a smart card, a mobile phone or a computer is made of thousands of logical gates that switch differently depending on the complexity of the operations executed. These commutations creates power consumptions for a few nanoseconds. Hence the power consumption (or Electromagnetic emanations) are *dependant* from the operations of its several peripherals including the memory transfers and core or accelerator operations on data manipulated. When targeting the recovery of a secret data involved in computations it becomes obvious that some part of the device behavior is dependant of this secret data. The main difficulty consist in detecting and measuring this physical information to exploit it for the secret recovery. The most critical part consist then in finding the right physical source of leakage. When such attack has been performed for years on small devices like smart cards it has been proven also efficient on bigger devices like phones or computers. However the physical leakage identification and measurement is much more difficult on such complex system on chips. Such a measurement phase has become today a complex and time consuming operation when side-channel analysis has to be performed on hardware devices. An exemple of such physiscal measurement is given in Figure .

However when this step has been performes the second attack step to be performed is the statistical attack that will extract the secret from the collected physical measurement traces.

Several techniques have been published for years in the literature since the initial Differential Power Analysis (DPA) publication from Kocher et al. One of this technique is the Correlation Power Analysis (CPA) that is often considered most often as the most efficient technique for side-channel analysis. We detail this technique in the following as we are also considering later in the paper that correlation analysis is the most efficient statistical technique to be used on computational data analysis.

Correlation analysis is a statistical mathematical test allowing to define the dependancy of two different set of (same length) of values whatever the origin of the values is. It has been defined by Bravais Pearson in ...

Correlation side-channel analysis relies upon a linear leakage model in the Hamming weight of a sensitive manipulated data:

$$W = a \cdot HW(D) + b + \epsilon \tag{1}$$

where a and b are real values characteristic of the hardware targeted and ϵ is a white gaussian of mean 0 and standard deviation σ. In order to measure the

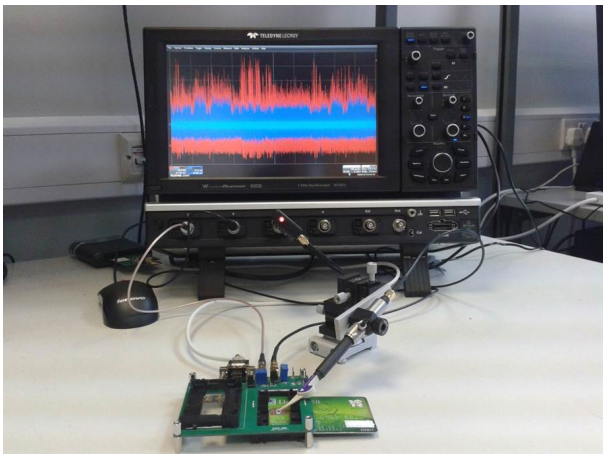

Fig. 1. Exemple of Physical Measurement Bench

dependency between the estimated value of a sensitive data and the corresponding value manipulated and represented in the physical traces measurement, the linear correlation factor from Bravais-Pearson is classically used.

Let $\mathcal{C}^{(i)}$ with $1 \leqslant i \leqslant \ell$ a set of ℓ side-channel traces captured from a device processing the targeted computations with input value $X^{(i)}$ whose processing occurs at time sample t with l the number of points acquired at time sample t. We consider $\Theta_0 = \{\mathcal{C}^1(t), \ldots, \mathcal{C}^\ell(t)\}$. We denote $S^{(i)}$ with $1 \leqslant i \leqslant \ell$ a set of ℓ guessed intermediate sensible values based on a power model, which is generally linear in the Hamming weight of the data. Let $f(X^{(i)}, \hat{K})$ be a function of the input value $X^{(i)}$ and (a part of) the targeted guessed secret $\hat{K}$. All l points in the leakage trace are equal to this value $f(X^{(i)}, \hat{K})$ for the time sample t. We then consider $\Theta_1 = \{S^{(1)}, \ldots, S^{(\ell)}\}$. The objective is to evaluate the dependency between both sets Θ_0 and Θ_1 by using the linear correlation factor ρ_{Θ_0,Θ_1}.

$$\begin{aligned}\rho_{\Theta_0,\Theta_1} &= \frac{\mathrm{Cov}(\Theta_0,\Theta_1)}{\sigma_{\Theta_0}\sigma_{\Theta_1}} \\ &= \frac{\ell \sum (\mathcal{C}^{(i)}(t) \cdot S^{(i)}) - \sum \mathcal{C}^{(i)}(t) \sum S^{(i)}}{\sqrt{\ell \sum (\mathcal{C}^{(i)}(t))^2 - (\sum \mathcal{C}^{(i)}(t))^2}\sqrt{\ell \sum (S^{(i)})^2 - (\sum S^{(i)})^2}},\end{aligned}$$

where summations are taken over $1 \leqslant i \leqslant \ell$.

The correlation value between both series is equal to 1 when the simulated model perfectly matches with the measured power traces. It then indicates

that the guess on the secret corresponds to the correct key value handled by the device in the computations.

When dealing with physical measurement of a hardware device cryptographic calculations it is then obvious that correlation is not reaching such values but the importance is that the correlation value for the right key value is higher than the others. An exemple of practical result of a side-channel correlation analysis on a hardware TDES device is given in figure 2.

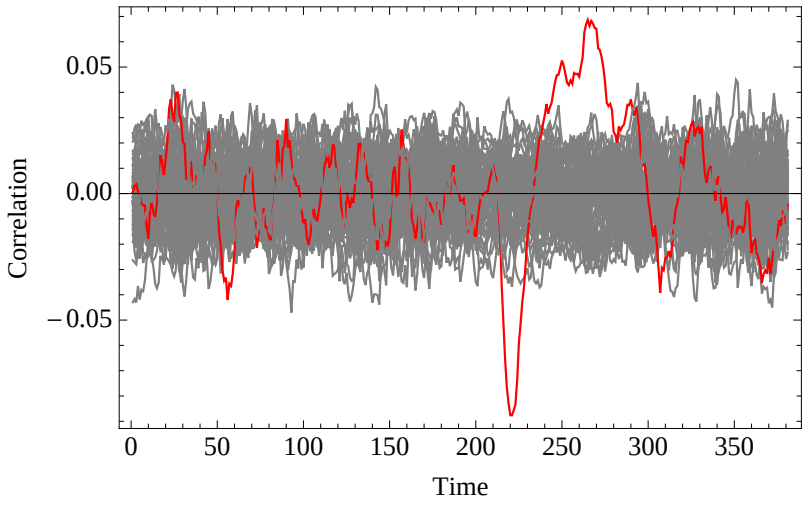

Fig. 2. Side-channel correlation attack result on hardware TDES engine measurements

Countermeasures In hardware devices efficient countermeasure can be designed and they can be classified in three categories. First countermeasures are inherently induced by the kind of application using the cryptographic algorithm. For instance the use of refreshed session keys in block ciphers or (random) padding in a RSA signature prevents the implementation from chosen message attacks; similarly the use of counter value(s) into the data sent to the card does not allow an attacker to sent twice the same data to the cryptographic algorithm. The second category targets to modify the signal either with hardware security features (noise generators, dummy cycles, clock jitters or power filtering aim at reducing the circuit leakage) or software countermeasures (*e.g.* dummy operations). The aim is to desynchronizing the curves and prevent an attacker from correctly exploiting them during their statistical treatment. The third kind of countermeasures consists in de-correlating the curves with the data related

to the algorithm's execution. The principle is to prevent attackers from predicting any intermediate value manipulated during the known algorithm execution. For instance code with constant time execution, masking and randomization techniques on input data and secret key are in this category [1,6,7,8]. Such countermeasures being sensitive to higher order side-channel attack [] it has been necessary to develop also stronger countermeasures as for instance [].

2.2 Fault Injection Analysis

Inducing a fault during a sensitive code execution may turn out to be a very effective way to downgrade the security of a program. When able to induce a fault, an adversary can take benefit of a large range of different code behaviour changes leading to malevolent benefits. As an example one could target to skip one or a set of operations with the aim to avoid some security sensitive code execution. This can be achieved by interfering directly into the runtime variables, by modifying some code offsets or even by turning some operations into other operations, like NOP.

Another way to disturb fraudulently a code execution flow targets the variables, either located in volatile memory or in non-volatile memory. Number of attack scenarios can be built from this ability. For instance one could force a part or the whole secret key to a deterministic value, as explained in [reference sorcerer apprentice], in order to guess a secret or to lose all the benefit of the secret. A second example concerns execution variables, such as outcome of sensitive comparison, resulting to severely downgrade a secure operations, such as a authentication.

In the same vein of unveiling secrets, specific techniques were developed to target cryptographic keys. Some are using differential tricks, like DFA, and were introduced in [3] for CRT RSA or in [2] and [20,14] for respectively DES and AES. Some interesting articles [ANSSI] showed that a secret AES key could be exhibited without even a knowledge of the non faulty cryptogram. Besides, the so-called safe error attacks make use of monitoring the consequences of a fault to guess the part of the secret. For a comprehensive idea of the fault injection possibilities, the best is to refer to [15] in sections 5.3 and 5.4.

An adversary with the ability to apply a meaningful fault injection has a very powerful mean to downgrade the security of a code. As a result developers need to pay a thorough attention to this threat and need to implement dedicated countermeasures. This kind of attack has been widely studied in the secure hardware execution, particularly in the secure element area [13]. As the code execution is confined in the hardware without obvious access to the runtime,

the best chance to induce a fault in a secure element is a physical disruption. This can be obtained with a laser, providing that a direct access to the silicone is possible, an electrical glitch or even an electromagnetic pulse. Whereas such a technique has shown to be efficient, the fault model remains highly uncontrollable as it is related to the physical consequences of the disruption. Indeed, when a variable or a code is modified as a consequence of a disruption, it is unlikely that the adversary has any control of the value.

Countermeasures The nature of the countermeasure relies highly on the execution environment of the sensitive code. In the secure element technology, the code execution is only accessible via its physical aspect. Therefore the best security is achieved with a balanced combination using hardware protections and software hardening. The countermeasures can follow various principles. The first one is to work at the source of the disruption. This is the case when strong filters are implemented to annihilate electrical glitches or when light sensors are spread over the die surface to trigger any laser pulses.

A second and more generic way is to implement logical controls, by adding hardware or software redundancy and by this way to detect any discrepancy during a code execution. As an example, it is common to execute the inverse of an AES or a DES computation subsequently to an execution in order to check that both computations are consistent together. Implementing such kind of control is necessarily costly and must therefore be tuned with care. Indeed, redundancy has a cost either in hardware or performance level.

To achieve a cost-effective security, a strong set of assumption about the fault must be defined in order to pick up the right level of redundancy. This needs to ascertain the adversary ability in terms of fault model. In a hardware execution typically, it is very unlikely that a physical disruption will be able to force a variable to a chosen value whatever it is. Some exception lie with 0x00 or 0xFF though, as it corresponds to a set of bits forced to 0 or 1. By doing so, a secure developer can greatly increase the difficulty of an attack making its realisation very unlikely. Such logical controls is often hardened by random executions and jitters in order to make the attack synchronisation tedious and more unlikely.

In software, this is no longer the case. Indeed, a software ability to inject code may open more opportunities for an attacker to tamper with a sensitive execution. Indeed, the fault is no longer limited by a physical disruption. In the worse case, the adversary has all control of the execution, a bit like having debug features when executing a code. For this reason, all debug features are usually taken with great care to avoid an adversary to get this control. This is typically the case when a JTAG interface is available on a secure device.

Having an open environments with full control of the code execution would dramatically change the paradigm and consequently oblige the developers to implement specific counter measures adapted to this extreme adversary model.

3 Software Simulation and Virtualization for Data Execution Traces

3.1 Objective

As it has been often seen in the literature, the evaluation of side-channel resistance can be performed by physically measuring the power consumption, or the Electromagnetic emanation. Later on, further analysis on the collected traces can then be performed. On the other hand, the virtualization technique introduces a new approach that allows to collect more precise data. It aims at directly generating the exact computational data or the Hamming weight of the data values. This step is directly performed at the runtime. An example would be the evaluation of some whitebox cryptography algorithms. Computational data are processed while those algorithms are executed. Furthermore, all the intermediate values that are stored into registers or within a stack can be gathered. It is then possible to collect any of these values at any point of the program execution. Consequently, computational data execution traces can be generated. Statistical attacks can then be applied on these traces. The latters can be seen as the software side-channel information in opposition to the physical measurements of hardware components.

As an example, in one of our practical test, we targeted a program that is executed within a mobile environment. More specifically, the Android platform has been chosen. The device embeds an ARM processor. Some software protections (obfuscation, software tampering, etc.) are in place in order to ensure the correct execution of the program in an untrusted platform. This paper does not go into further details regarding those software protections. However one should note that they can be highlighted during the analysis according to the acquisition technique model. An important property for the acquisition system is to be least invasive in order to not trigger those software protections. Concluding that the Dynamic Binary Analysis (DBA) based on the OS resource abuse and manipulation is the most powerful approach.

Multiple techniques can be used in order to acquire such computational data execution traces at runtime. Later on, they enable a post-computational data analysis like:

– Manipulation of the program during idle mode by adding debugging statements with the aim of producing relevant computational data during its execution

– Manipulation of the program at runtime hooking the OS resources i.e.: shared libraries or calls to OS API framework with the aim of producing relevant computational data during its execution
– Execution of the program in a debugger or simulator or even a tracer with the aim of producing relevant computational data handled by their virtual resources
– Execution of the program in a legacy Operating System version with the aim of producing relevant computational data without the security enhancements introduced by new version of the Operating System.
– ...

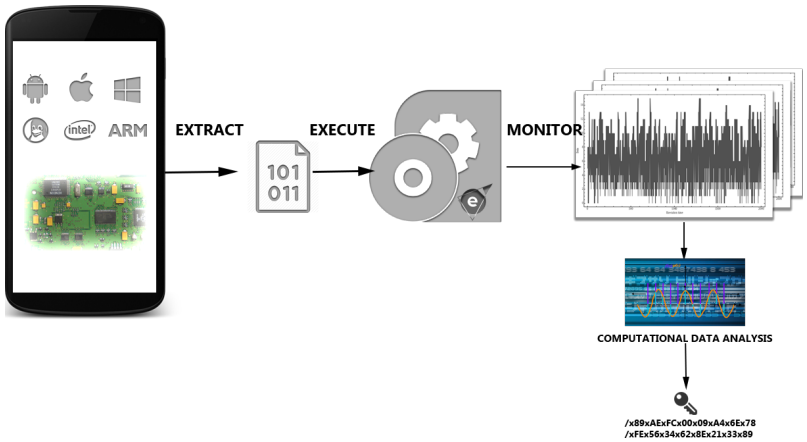

Fig. 3. In-depth runtime analysis for statistical analysis

3.2 Dynamic Binary Analyser

We have developed a simulation tool that can be used to provide data traces for performing such kind of analysis. It was then possible to trace with this method the registers and memory addressed areas and collect all these values in traces. Then as previously said traces were exploited with statistical analysis for recovering secret involved in computations.

Our DBI framework also allows an attacker to run the targeted binary, in emulated environment, instruction by instruction giving access on processes memory and CPU register, between each instruction. The programmer is able therefore to generate traces on stack, heap, CPU register in use and others data as instruction being executed. Hence it allows executing the program minimizing the effect of the current software protections and customized techniques in

order to bypass of those expected inherently from the mobile Operating system, amount others: add Loadable kernel modules (LKM) using Custom Kernel Images, bypass the Address Space Layout Randomization (ASRL) or enable the SELinux permissive mode, etcetera. Additionally UL DBI framework provides an open and extensible environment to add plugins in order to improve the post-Statistical Analysis with for example but not limited to pre-characterization of the binary combining with the result of the Static Analysis, trace alignment or ARM code instruction matching with disassembled code, etcetera.

In practice, for the example, lets go to assume the following function depicted in Figure 4 based on a trivial masking algorithm performing an exclusive or between a secret key byte and a message byte and how the code looks like in the binary depicted in figure 5. Note that the ARM assembly code shown in figure 5 has been generated with a Capstone lightweight multi-platform, multi-architecture disassembly framework.

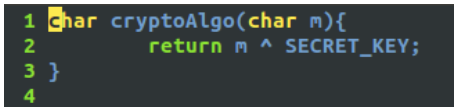

Fig. 4. Code example

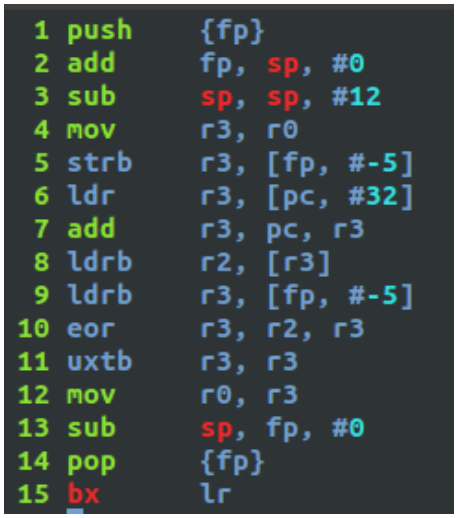

Fig. 5. Binary code related

Using the DBI framework on the binary, the output provided after running it in a verbose fashion way would be as it is depicted in Figure 6. Then it becomes simple to store these data to build the code execution data traces.

3.3 Virtualizer

A first simulator capable of tracing such data executions was presented in detail by Georges Gagnerot in 2013 [11]. This simulator was dedicated at this time

```
instruction(9)
PC : 4017CF4C
next instr addr : 4017CF50
Info : 4017cf50 : E0233002
cond : E0000000 , CPSR : 20000000 , comp : 00000001

instruction(10)
PC : 4017CF50
next instr addr : 4017CF54
 Current XOR :
Inst is not Imediate value
Am shift : 00000000
Current XOR : r3 : 00000071, r2 : 00000019, r3 : 00000071 => E0233002
In Register[r3] : 00000071
Info : 4017cf54 : E0633001
cond : E0000000 , CPSR : 20000000 , comp : 00000001

instruction(11)
PC : 4017CF54
next instr addr : 4017CF58
 After XOR :
In Register[r3] : 00000068
Info : 4017cf58 : E50B301C
cond : E0000000 , CPSR : 20000000 , comp : 00000001

instruction(12)
```

Fig. 6. DBI Output

to characterize the resitance of hardware product masks and implementations. Thus it was used for instance to validate the resistance of cryptographic implementations with regards to side-channel and fault attacks and add the following properties.

- Multi Architecture: initially dedicated to simulate side-channel traces on hardware implementations we developed a first tool version. Indeed using open source libraries we built first a simulator able to perform side channel analysis for different standard processor architecture like ARM, x86, MIPS, SPARC, SH4 ....
- Linux Compatibility: the simulator has been made compatible with standard Linux program. It can for instance use any program already compiled for a supported architecture and get nice side-channel traces out of the box. No special work is required for the compilation. It is important to note though that kernel calls were not traced by the current implementation so you only get side-channel consumption for the user space.
- Automatized Tests: software cryptographic libraries providing the good interfaces could then be tested automatically by the tool on a first step. It could be seen as a preliminary work for an security evaluator but was giving surprisingly good results close to the target.

At this time the main objective was to validate or defeat secure elements implementations and evaluate their resistance to future side-channel attack when being embedded into integrated circuits lie secure elements with different CPUs. Some exemples were given in [11]: figure 7 is giving the Hamming weight data ad code execution traces for an AES-128 encryption for an ARM 32-bit core. Same results were obtained for public key implementation like RSA as depicted by Figures 8 and 9.

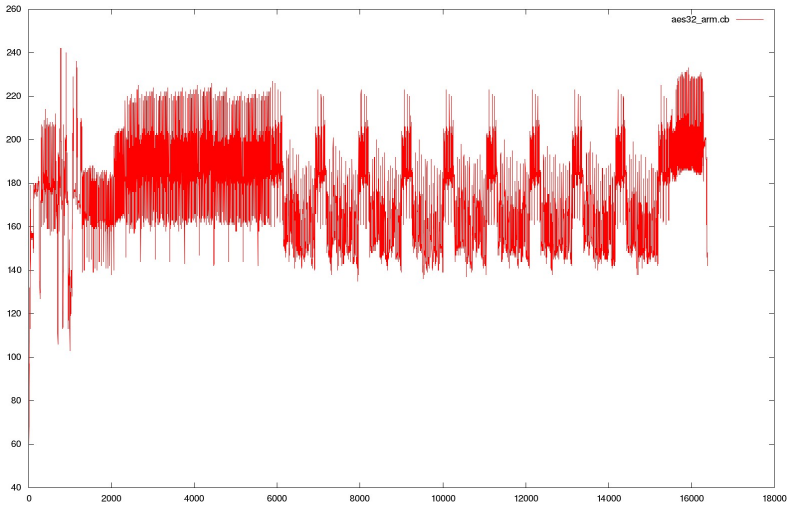

Fig. 7. Unprotected AES-128 trace

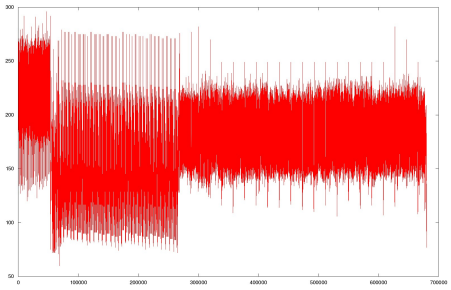

Fig. 8. ARM RSA execution trace

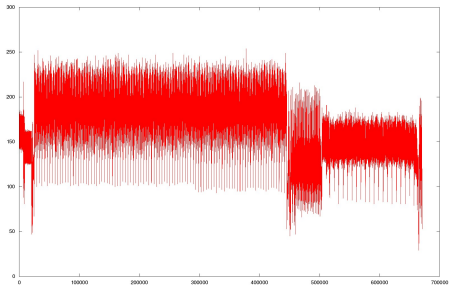

Fig. 9. x86 RSA execution trace

It has become obvious that it was also possible to improve this tool to trace the execution for our new objective and output either Hamming weight of plain data value of the executed operations into any code that can be an unprotected or a whitebox implementation.

Hence, it has been decided to adapt the simulator from [11] to address binary code of other products like mobile HCE application including sensitive secrets hidden in embedded libraries code or whitebox cryptography implementations. This initial simulator has then be improved to address these new requirements with success.

It has then become possible to trace any code instructions and data value manipulated by a binary on one of the supported architecture by the simulator. Hence it has been possible to scrutinize any code including whitebox cryptography using sour tool and then we were capable to perform very efficient statistical analysis as we explained previously.

Traces obtained from a whitebox AES execution are given in Figures 10 and 11 to illustrate this tool adaptation.

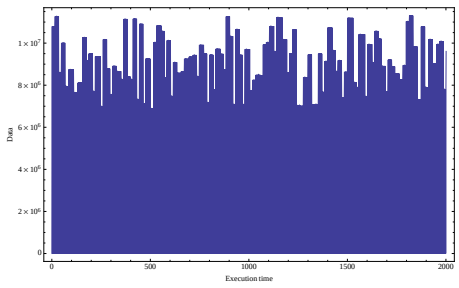

Fig. 10. Computational Data Execution Trace in whitebox AES binary

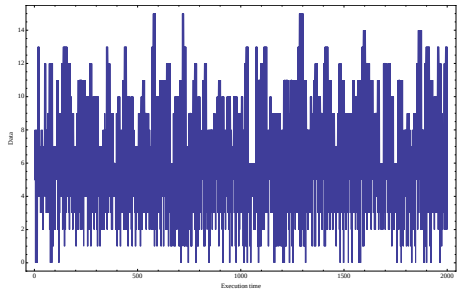

Fig. 11. Computational HammingWeight Execution trace in whitebox AES

A significant interest is that we are not relying on a particular model of information leakage. Hence the statistical analysis can directly target the real data and not a function of this data (like the Hamming weight or Hamming distance for instance in power models). Addressing directy the data and opcode values we can decide to build the execution traces we want including the data or the Hamming weight of these values. We can also get rid of any noise related to computation we do not care that is not the case in physical measurement even when dealing with EM measurements. It has then be observed that such computation data traces were allowing very powerful attacks.

It makes the analysis much more efficient than classical side-channel attacks on physical measurements.

3.4 Other Solutions

Another solution taking advantage of the Valgrind tool has been presented in [4] with practical results. It is worth to notice this solution is different from the one we present here and when the memory accesses during calls where targeted by the Valgrind solution we focus here in the registers. Indeed secret values can be hidden at different level of the code. When targeting the internal computation of a (whitebox) cryptographic algorithm to perform statistical analysis it seems us also accurate and preferable to target the registers and internal manipulations of the cryptographic internal computations. Indeed such algorithms can be written in assembly and sensitive value related to the leakage could be only present in the local memory and/or into the registers. As described by the authors in [4] it would then need to access registers even if the traces become bigger. However we think the presented solution is of strong interest and can be used as a complementary too with our methods.

4 Performing Statistical Computational Data Analysis

Side-channel analysis has only always consisted in analysing statistical dependancy between the set of intermediate computational Hamming weight or Hamming weight distance of the data guesses and the set of the real performed data in the executed operations represented by points from a measurement trace. It is important to notice that such an analysis could not be obtained if not leaking through a physical side-channel.

The objective here to reproduce a similar statistical attack with data obtained from the execution of a program (like in a debugging mode you can access any data) and the guessed values set. It is obvious but very important here to observe that computational data analysis is much more powerful than the so-named side-channel techniques used in the hardware evaluation field. Indeed as we can directy address the data itself without any model leakage, hence wihtout any restriction, most of the attacks will be more efficient .

4.1 Statistical Data Analysis

Simple Data Analysis This computational data set generation can be analyzed using graphical observation targeting information recovery on the secret. It very similar to the simple side-channel analysis but can be applied to the data itself, to its Hamming weight, to some particular bits. All kind of simple statistical test can be used here.

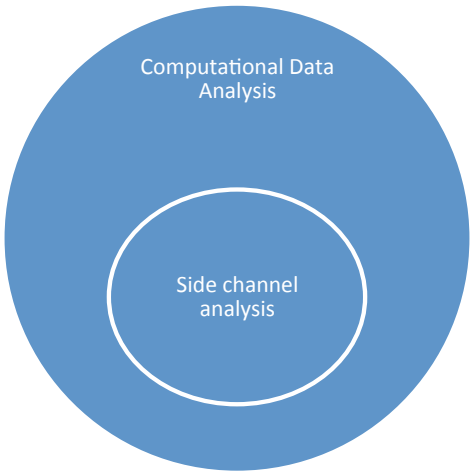

Fig. 12. Computational Data Analysis

Computational Data Correlation Analysis Computational Data Statistical analysis can be performed in many different ways as it is a subset of statistical data analysis. The important point here is that statistical analysis that can be performed using computational data execution traces which can be much more powerful than the so-named side-channel techniques used in the hardware evaluation field. Correlation analysis we explained previously can then be applied very efficiently to such computational execution data traces [19].

[On going completion – Benoit]

Other Statistical Tests It is also possible to perform successful attacks using different statistical tests than the Pearson correlation one. The basic Difference of Mean use in DPA hardware attacks can be used but it is well known to be not optimal. The Spearman rank, the Goodman Kruskal Gamma, the HoeffdingD, the Kendall Tau .... Different statistical tests could be used on data execution traces to determine and recover secret assets.

Comparing Side-Channel and Computational Data Efficiency

Computational Data Efficiency to Dynamic Reverse Engineering on a Single Execution The idea behind the dynamic analysis or also called dynamic reverse engineering is to extract program properties when the program is executed focusing on primarily on control flow or data flow. Several well-known techniques

are used by researchers by this purpose, such as: debugging, tracing, emulating, hooking and profiling. Indeed a wide variety of tools are available to support these techniques therefore the goal of this section is show how these tools and techniques have been combined by us in order to find a new use case to retrieve the program properties which allows disclosing the code secrets using Statistical Analysis.

Computational Data Correlation Analysis can defeat a non protected implementation but it requires using only few data execution traces. The main advantage compared to Hamming weight model leakage is that knowing the leaking point a single trace is enough to recover the whole secret key. Indeed in the case of the AES when attacking the output of ByteSub or the output of the Te tables of the open SSL implementation a single data value is enough to recover a key byte when knowing the input plaintext given to the AES. It is due to the bijectivity of the ByteSub operation and only data knowledge can permit such a single trace attack. Reproducing the attack once the characterisation has been done a first time would then require few seconds only using a single execution. It means even an ephemeral secret like a token with limited used could be compromised and recovered thanks to such computational data analysis. Hence, we observe here that computational data analysis and dynamic reverse engineering are joining in a combined attack technique take advantage of each of their properties.

5 Simulation and Virtualization for Fault Analysis and Reverse Engineering

We also taken into consideration, the need to perform fault attacks on a targeted binary. It has then be of strong importance to render the tools we presented previously (DBA and Virtualizer) for statistical attacks on data execution trace capable to perform fault attacks in the execution of this binary as it has been previously published in [11].

Assuming the code is available to an adversary, typically by extraction, it becomes possible to perform a sensitive code execution on an emulation platform. This can be achieved relatively easily for object-oriented code, such a Java codes, by setting up a virtual machine processing the java byte codes. This remains more complex however for native codes as the binary code is machine dependant. The virtualization, as it has been designed by us, implements a platform giving the opportunity to get a full control of the runtime execution for native codes.

Using such a tool an adversary gains the ability to monitor the execution flow at different levels of the runtime. Indeed it gives the access to the runtime

variables (program counter, intermediate registers values, the arithmetic logic, etc) as well as the memory accesses, like those in the volatile memory. Doing so, it provides the same level of control as a debugger with the opportunity to execute dynamic testing, such as analyses requiring a high number of execution occurrences.

First of all, such capability can be exploited to perform advanced reverse engineering. Indeed, multiple execution can be performed with the aim to monitor a targeted variable or memory access and consequently to collect the corresponding values for further statistical analyses. Doing so, it becomes possible to perform statistical profiling and consequently to carry out some advanced reverse engineering. This can be the case for some encoding functions relying on random representation, as it can be found in some whitebox cryptography solutions. Running statistical studies of variable representations may lead to exploitable results in case deterministic information is exhibited about the code or the variables. To perform the analysis, the tool provides capabilities to run successive executions with a given set of input data inserted at different stages of the execution.

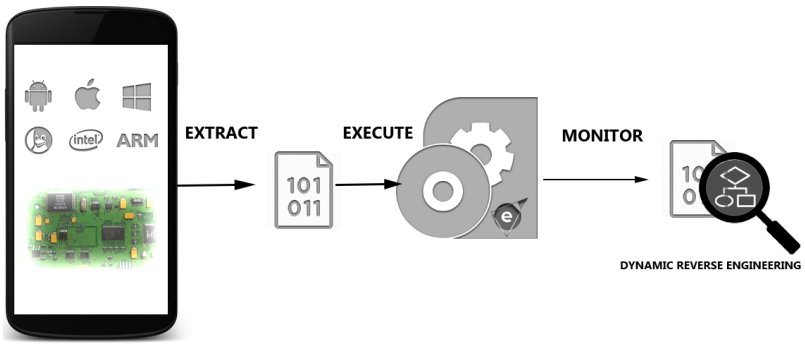

Fig. 13. In-depth runtime analysis for advanced reverse engineering

As a further step of an attack realisation, the knowledge of the code could lead to target specific mechanisms with the aim to change the code behaviour and by this way downgrade the security mechanisms. As an example, if a specific authentication is required to release some secure content, an adversary could attempt to identify how the authentication is performed and subsequently modify the execution with the aim to skip it. As software protection against code modifications, some efficient obfuscation techniques render the code execution

blurred, complex and extremely hard to interpret. With such a protection in place, it becomes extremely tedious to target when injecting the fault and the nature of the fault to induce in order to get the expected faulty behaviour.

One way to potentially circumvent the software obfuscation barrier while a weakness is there is to target in a more or less exhausting way multiple fault injection attempts by varying the time of the fault and its model, typically the variable and its value. Such an extensive campaign may turn out to be very time consuming without an appropriate tool and would remain unaffordable. Therefore a virtualisation platform represents a strong option as the runtime execution can be faulted with no restriction. By this mean an extensive fault modification campaign can be automated in order to cover a high range of different fault models at different times in an exhaustive manner.

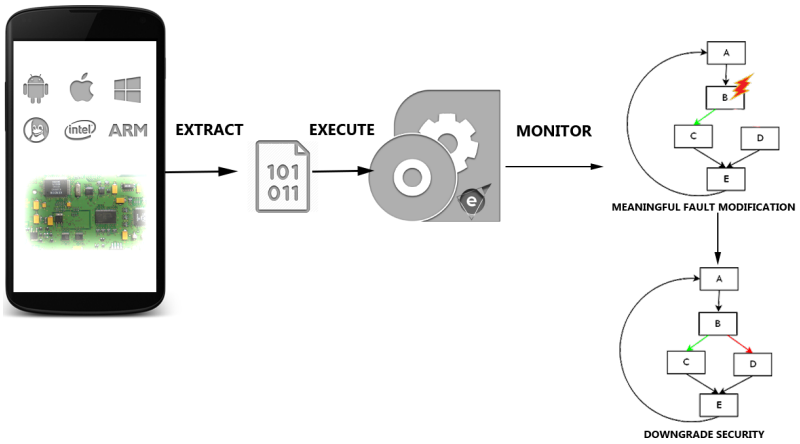

Fig. 14. Meaningful fault injection for downgrading the security

Proceeding this way, a strong coverage can be achieved in order to explore any potential flaw. By brute forcing the fault injection over a piece of execution allows an adversary to circumvent the difficulties applied by some obfuscation techniques. As an example, this can be useful when strong obfuscation methods protect a whitebox cryptography implementation. Depending on the implementation, the algorithm variable may follow some path that are hard to interpret. Implementing a sort of exhaustive fault induction may turn out to be successful by giving exploitable faulty cipher texts and subsequently applying a Differential Fault Analysis for disclosing the secret key. This can be particularly valuable

when it is possible to restrict the execution to the targeted algorithm only and therefore to avoid some further execution implementing security controls.

Furthermore attacks requiring specific fault models and a high number of faults become affordable. Initially these attacks were tuned for the hardware execution context, and turned out to be almost non-realistic as practically they were too demanding in terms of fault models and number of faults (Benoit: ref ?). Depending on the software implementation, such a technique opens the door to more complex attacks as the hardware limitation is no longer an issue.

6 Practical Results on DES and AES Implementations

We have tested our virtualisation tools on several implementations of AES and DES that were hiding the key in the operation as it has been published in several publications. This implementations were also enforced with obfuscation mechanims. When these countermeasures were rendering dynamic and static analysis harder for the attackers it was easily overpassed by using static analysis on the data execution traces that were generated by the virtualizer. Most of whitebox implementations are dealing with big lookup tables and often do not present big desynchronization among the execution traces. Hence it made the statistical analysis easy to performed and very efficient. On the other hand when desynchronisation was present it was possible to apply alignment techniques to make the attack efficient.

6.1 Statistical Analysis on AES Library 1

This first target is an unprotected AES binary library that was enforced with standard obfuscation mechanism. Both tools have been used to generate the data execution traces. Correlation analysis has been applied and the secret key has been recovered using less than 100 traces. This practical test allows to validate in practice the tools we have developed and highlight how much efficient in practice the technique we have developed can be.

6.2 Statistical Analysis on AES Library 2

This second analysis has targeted an obfuscated AES binary library that would have required several weeks/months to be reverse engineered in order to be cryptanalysed.

We have been able to recover the secret key of the AES computation using less than 100 computation data execution traces. This result illustrate how efficient the computational data analysis can be compared to usual reverse

engineering techniques. It is obvious depending on the instructions traces the size of the trace can require sometimes cost consuming computations. However only few days are generally necessary in worth case to perform the attack.

6.3 Round Reduction Fault Analysis on AES Library 1

As an example of fault that could compromise a cryptographic algorithm, a straightforward attack can severely downgrade an algorithm by reducing the number of rounds. In the case of an AES 128 encryption, the algorithm mathematical strength relies on ten successive rounds, whose last one is unique as it does not include any MixColumn as described in []. A fault injection of an AES 128 software implementation could target the mitigation of the number of round. The factor of success of the attack was to get a faulty AES 128 cipher text as outcome of a single AES round computation using the first round key. The objective being to exhibit the 128 bit-long AES secret key from these cipher texts exploiting some cryptanalysis [].

As depicted on the following picture, the practical attack was successful and the number of rounds could be turned down to one single. This could be achieved on a naive implementation by emulating the execution emulation and modifying the Program Counter in an appropriate way when executing the AES:

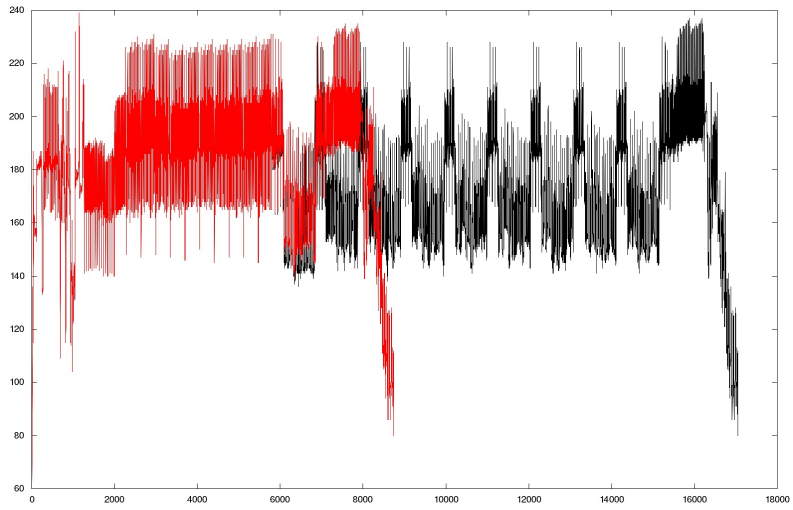

Fig. 15. Round Reduction with Fault Attack on AES Library 1

The outcome of the downgraded execution gave a faulty cipher text corresponding to a single round execution with the message given and the secret key. From this cipher text, a cryptanalysis was possible to retrieve the secret key. It is important that an identification phase may be necessary in some cases in order to interpret the faulty cipher texts and check if they are suitable. To do so, the knowledge of the secret key can make this task much easier. As the tool allowed us to run successive scripts, we ran a campaign in order to find different ways to obtain the same outcome injecting different faults (the program counter, the registers, etc).

6.4 Differential Fault Analysis on AES Library 2

The previous attack cannot be performed on the AES library 2 that With more sophisticated software implementation, the previous attack cannot be performed when the AES code is flattened and code integrity checks are integrated. Indeed, the code structure being flat, or in other words not replicating the same piece of code by successive calls, together with control flows makes the previous attack difficult or can even prevent it. Other techniques could be applied, such as a DFA attack as published by Piret et al.[20]. This article demonstrates that the most efficient way to extract the AES secret key requires two faulty injections with a specific fault models. As the fault is injected in software, it was possible to us to target the required fault model at the right time of the execution by the mean of reverse engineering. Doing this, the whole secret key could be successfully exhibited.

At some point, particularly when the obfuscation level was strong, our practical experimentations showed that it was not possible to precisely target a variable, its value and a good timing in order to get a meaningful faulty cipher text. In that cases, we ran some intensive campaigns to vary the fault over a sequence of code with the aim to find the right combination generating the expected fault cipher text. This turned out to be successful even if the code was strongly obfuscated. The success of such a campaign lies a lot in the ability to vary some parameters over the time and over a range of values in order to exhaustively parse a large range of faults.

7 Limitations and further improvements

Fault attacks are efficient and can be exploited on data but it remains more difficult on code executions if efficient anti-debug techniques have been implemented and present in the binary. For instance reducing the round number of a

TDES or an AES to a lower or higher value as published [] could be difficult as most of the time flattening techniques are used and it would require the code flow to be modified. In that case the code modification might most of the time be detected when code integrity checks or hashs are present in the executed code.

8 Conclusion

Chicken or The Egg? We can answer that Side-channel and fault analysis on hardware devices are only sub components of what we call here Computational Data Attacks. This more generic terminology lead to mathematical numerical statistical data analysis techniques and in a certain manner to cryptanalysis techniques that have been studied for decades on cryptographic algorithm. It seems we are here combining all these techniques to exploit it plainly on any kind of embedded products. We have presented a methodology based on computational data execution tracing to evaluate with statistical analysis any software embedded i.e. also obfuscated or whitebox-ed crytographic algorithms. We have observed these techniques are very powerful to apply all existing first or higher order statistical attacks on data with success as the information contained in data traces is complete.

Hence, as we have seen computational data execution trace analysis and perturbation are opening a new horizon for attacks on mobile environment where traditionally the main threat was reverse engineering. It is of strong interest to observe that all the so numerous statistical and fault attacks (and combined attacks) from the hardware products can be extended to software products with a much stronger efficiency.

Such techniques are of strong interest to evaluate in depth software technologies like DRM and HCE products.

References

1. Mehdi-Laurent Akkar and Christophe Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001.
2. E. Biham and A. 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.
3. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *EUROCRYPT*, pages 37–51, 1997.
4. J. W. Bos, C. Hubain, W. Michiels, and P. Teuwen. Differential computation analysis: Hiding your white-box designs is not enough. *IACR Cryptology ePrint Archive*, 2015:753, 2015.

5. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.
6. B. Chevallier-Mames, M. Ciet, and M. Joye. Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.
7. J-S Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 1999*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.
8. J-S Coron and L. Goubin. On boolean and arithmetic masking against differential power analysis. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 231–237. Springer, 2000.
9. J. Doget, E. Prouff, M. Rivain, and F.-X. Standaert. Univariate side channel attacks and leakage modeling. *IACR Cryptology ePrint Archive*, 2011:302, 2011.
10. Federal Information Processing Standards Publication (FIPS). Data Encryption Standard - DES, FIPS PUB 46-3, 1999.
11. Georges Gagnerot. *Study of attacks on embedded devices and associated countermeasures. Modelisation and Simulation of associated Leakages and perturbation vectors.* PhD thesis, Limoges University, 2013.
12. B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel. Mutual Information Analysis. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 426–442. Springer, 2008.
13. C. Giraud and H. Thiebeauld. A survey on fault attacks. In J-J. Quisquater, P. Paradinas, Y. Deswarte, and A. A. El Kalam, editors, *Sixth International Conference on Smart Card Research and Advanced Applications - CARDIS 2004*, pages 159–176. Kluwer, 2004.
14. Christophe Giraud. DFA on AES. In H. Dobbertin, V. Rijmen, and A. Sowa, editors, *Advanced Encryption Standard, 4th International Conference, AES 2004*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2004.
15. JIL. Jil-application-of-attack-potential-to-smartcards.
16. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
17. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
18. P.C. Kocher, J.M. Jaffe, and B.C. June. DES and Other Cryptographic Processes with Leak Minimization for Smartcards and other CryptoSystems. *US Patent 6,278,783*, 1998.
19. UL Evaluation Lab. Computation data statistical analysis. Technical report, Underwriters Laboratories, 2014.
20. G. Piret and J.-J. Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.
21. R. L. Rivest, A Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM 21*, pages 120–126, 1978.
22. W. Schindler, K. Lemke, and C. Paar. A stochastic model for differential side channel cryptanalysis. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2005.