Abstract. Side-channel and fault injection analyses 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 inefficient. For two decades embedded cryptography for payment, pay tv, identity areas have relied a lot on secure elements. Nowadays more alternative solutions on mobile phones appear with the aim to offer software-based security 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 often 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 whitebox cryptography becomes a strong alternative to secure element storage for assets. We explain in this paper how directly from the binary or with the extracted source code we can perform statistical and fault analyses using similar techniques as those used in hardware-based security. This concerns particularly side-channel or fault injections techniques. 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 no longer restricted by any physical limitations imposing a leakage model (and additional noise) or making fault injection tied with physical limitations. Hence statistical and fault attacks can go potentially further in software-based implementation compared to hardware-based devices. As a consequence, complex techniques like high order, collision and horizontal statistical attacks become 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, 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 [22] and 1998 [23]. From these initial publications, during two decades a myriad of derivated and newly-derivated 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 [6] and linear regression analysis technique [14]. The second security gap was initiated by the Bellcore fault attacks on RSA [4] and the differential fault attack on DES [3]. Fault attacks have become today very powerful attacks taking advantage of powerful laser and electromagnetic test benches 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) defeating 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 since 2013 (Kit Kat release) 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 environment. That requirement is very difficult to achieve in a open platform like Android where the SIM is not used to host the payment application. Such solutions require the sensitive 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 obfuscated and/or 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 obfuscated 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 lines of code including sometimes obfuscation and anti tampering techniques. Performing reverse engineering and targeting these cryptographic
implementations can be a considerable waste of time. A better time wisely approach 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 performing statistical analysis on the computation and injecting fault during these 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 gives 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 software data tracing 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. We illustrate our analysis with practical results. Section 5 is explaining the fault injection techniques we can perform using our simulation and virtualization environment with attack examples given. Limitations and combined attacks are introduced in section 6 and we conclude in section 7.

2 Preliminaries

2.1 Side-channel Analysis

It has been studied for years since it has been introduced by Kocher et al. [22]. Many attack paths have been published on the different cryptosystems like DES [16] and RSA [31] 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. [24]. We can for instance mention the Correlation Side-Channel Analysis (CSCA) introduced by Brier et al. [6], the Mutual Information Side-Channel Analysis (MISCA) from Gierlichs et al. [18] or the Linear Regression Side-Channel Analysis [14,32].

Physical Behavior and Secret Dependancy Side-channel analysis requires some measurements to be performed on the targeted device to extract some information 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 create 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 behaviour is dependant of this secret data. The main difficulty consists in detecting and measuring this physical information to exploit it for the secret recovery. The most critical part consists 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 to perform 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 example of such physical measurement is given in Figure.

However when this step has been performed 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 side-channel analysis** relies upon a linear leakage model in the Hamming weight of a sensitive manipulated data:

\[
W = a \cdot HW(D) + b + \epsilon
\]  

(1)  

where \( a \) and \( b \) are real values characteristic of the hardware targeted and \( \epsilon \) is a white Gaussian noise of mean 0 and standard deviation \( \sigma \). In order to measure the 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 \( C^{(i)} \) with \( 1 \leq i \leq \ell \) a set of \( \ell \) 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 $\Theta_0 = \{C^1(t), \ldots, C^\ell(t)\}$. We denote $S^{(i)}$ with $1 \leq i \leq \ell$ a set of $\ell$ 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 $\Theta_0$ and $\Theta_1$ by using the linear correlation factor $\rho_{\Theta_0, \Theta_1}$.

$$
\rho_{\Theta_0, \Theta_1} = \frac{\text{Cov}(\Theta_0, \Theta_1)}{\sigma_{\Theta_0} \sigma_{\Theta_1}} = \frac{\ell \sum (C^{(i)}(t) \cdot S^{(i)}) - \sum C^{(i)}(t) \sum S^{(i)}}{\sqrt{\ell \sum (C^{(i)}(t))^2} - (\sum C^{(i)}(t))^2 \sqrt{\ell \sum (S^{(i)})^2} - (\sum S^{(i)})^2},
$$

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

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 example 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 desynchronize the curves and prevents 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 [1,7,11,12]. Such countermeasures being sensitive to higher order side-channel attack [27] it has been necessary to develop also stronger countermeasures as for instance [30].
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 [4] for CRT RSA or in [3] and [29,20] for respectively DES and AES. Some interesting articles [ANSSI] showed that a secret AES key could be exhibited without even 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 [?] 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 [19]. 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 control 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 environment 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 Execution Tracing for Collecting Data 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 software tracing acquisition technique introduces a new approach that allows collecting 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 those 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. These traces 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, tamper resistance, 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 could 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, which might be used in the statistical analysis like:

- 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.
- ...
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 memories 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 our 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.

![Binary code related](image)

**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 [17]. This simulator was dedicated at this time to characterize the resistance 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 a 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 examples were given in [17]: 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.

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.
Fig. 7. Unprotected AES-128 trace

Fig. 8. ARM RSA execution trace

Fig. 9. x86 RSA execution trace
Hence, it has been decided to adapt the simulator from [17] 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 been 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.

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 directly 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 [5] with practical results. It is important to notice this solution is different from the ones 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 [5] 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 dependency 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 directly address the data itself without any model leakage, hence without 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 analysed 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.
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 [25].

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 [23] hardware attacks can be used but it is well known to be not optimal and the same observation applies here. Several statistical tests for data set dependency could be used (Spearman rank, Goodman Kruskal Gamma, HoeffdingD, Kendall Tau . . . ) but it is not necessary in practice.

4.2 Practical Results on DES and AES Obfuscated 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. These implementations were also enforced with obfuscation mechanisms. 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 desynchronisation 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.

**Statistical Analysis on AES Library 1** This first target used to illustrate results in practice was an unprotected AES binary library that was enforced with basic obfuscation mechanism. Both the DBI and virtualizer tools have been used to generate the data execution traces. Correlation analysis has been applied. It has been observed that the secret key has been recovered using less than 100 traces.

It is important to observe that most of the time only few traces are necessary to perform a successful attack. It is due to the absence of any noise as we are not limited to a physical measurement.

**Statistical Analysis on AES Library 2** This second target is a strongly obfuscated cryptographic AES binary library that would have required several weeks to be reverse engineered in order to be cryptanalyzed. We apply the same technique as previously to generate the data execution traces from the binary and performed statistical analysis to recover the secret AES key.

When generating the computational data traces on strongly obfuscated libraries it is worth to notice the important (huge) size of these traces. Thanks to the countermeasures and because the tracing generates a big amount of data it is important to process these data. Another option consists in identifying (as in side-channel) first the moments where the input is manipulated to reduce the scope of the analysis as shown in figure 13. It becomes then faster to perform the attack on reduced area of the traces and recover the secret key as shown in figure 14.

In both cases we have also tested the other statistical test with success for these traces. Depending on the implementation and the way the data tracing is implementing it could appear the data traces are misaligned. It could then require to improve the data tracing method and/or to perform post processing on the data traces to make the attack more efficient.

4.3  **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. Researchers use several well-known techniques for 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 and it requires using only few data execution traces. The main advantage compared to Hamming weight model leakage is that knowing the leaking point for the computational data targeted a single trace is enough to recover the whole secret key in a symmetric implementation for instance. That is a huge difference compare to hardware side channel attacks. Indeed in the case of the AES when attacking the output of ByteSub or the output of the $T_e$ 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.

### 4.4 Enhancing Computational Statistical Analysis

For years statistical side-channel have been performed from the classical first order attacks (i.e. DPA [23]) to more complex like the high order statistical attacks introduced by Messerges et al.[27]. Such complex attacks can be performed with
success on hardware devices but they require a much more consequent amount of traces compare to the ones needed for a first order attack. The noise, the misalignment of the traces collected and the Hamming weight leakage model render these techniques much more difficult to perform when combining several moments on the traces. Hence it decreases considerably the efficiency of the attacks compared to the efficiency of the first order attacks.

When considering more secure implementations of data obfuscation enhanced methods or whitebox cryptography such higher complexity attacks can become very efficient on data execution traces collected with the methods we described previously. It becomes more efficient because the model has been changed to a data model. In that case the important loss of efficiency observed on second order attacks in hardware devices (i.e. Hamming Weight power consumption model) is not present anymore. Higher order attacks remains as efficient as the first order ones in the data leakage model.

**Higher-Order Computational Statistical Analysis** We gives in this section first order and second order correlation analysis results for the Hamming Weight and the Computation Data Leakage model. We perform the attack on an obfuscated AES (said unmasked) and a masked version (where random values are used to hide the correct values with random data). In case of the second order attack we are able to identify the random value (or its Hamming Weight) in order to combine this value with the masked one.

In the following we randomly generated the following Hamming Weight power leakage:

\[ C = W_a f(x) + b + N(u, \sigma) \]

with \( W_a = 1.709 \) and \( W_b = 0.792 \) and \( \text{Noise} \sigma = -0.157 \). Then we perform the first order attack at the output of the substitution at the first round. For the second order attack we combine the random value used for masking and the output of the recomputed substitution table masked (with this random value). The first order attack is applied to verify the AES implementation is properly protected with masking and the second attack is performed using the absolute difference and the normalized multiplication method [15,34].

We obtain the results given in Table 1 for 200 traces and illustrations are given in Figures 15 and 16 for the first order attack results and figures 17 and 18 for the second order attacks. This corresponds to usual results on simulated traces for a hardware device. It is clear the practical measurement results on hardware devices require much more traces in practice. However these results allow us to compare with the results given for the computational statistical attack using the data leakage model. In this latest case the result are given in table 2.
Fig. 15. First Order Correlation and Convergence Traces Results on SubByte with 200 Traces

Fig. 16. First Order Correlation and Convergence Traces Results on MASKED SubByte with 200 Traces

Fig. 17. Second Order Correlation and Convergence Traces Results on masked AES with 200 Traces using Absolute Difference

Fig. 18. Second Order Correlation and Convergence Traces Results on masked AES with 200 Traces using Normalized Multiplication
Fig. 19. First Order Correlation and Convergence Traces Results on SubByte with 50 Computational Data Traces

Fig. 20. First Order Correlation and Convergence Traces Results on MASKED SubByte with 50 Computational Data Traces

Fig. 21. Second Order Correlation and Convergence Traces Results on masked AES with 50 Computational Data Traces
Collision Attacks A specific approach for Side-Channel Analysis uses information leakages to detect collisions between data manipulated during the executions of the algorithms. A side-channel collision attacks against a block cipher was first proposed by Schramm et al. in 2003 [33]. More recently Moradi et al. [28] proposed to use a correlation distinguisher to detect collisions in AES. The main advantage of this approach is that it is not necessary to define a leakage model as points of traces are directly correlated with other points of traces. Later, Clavier et al. [9] presented two collision-correlation techniques defeating different first order protected AES implementations. The same year, Witteman et al. [35] applied collision correlation to public key implementation. They describe an efficient attack on RSA using square-and-multiply-always exponentiation and message blinding. All these techniques require many side-channel execution traces.

In the same vein these attacks can be performed more efficiently using computational data execution traces thanks to the data leakage model additional properties.

Horizontal Attacks Correlation analysis on a single atomic exponentiation side-channel trace has been published in [10] where the message is known to the attacker but the exponent is blinded. This attack called horizontal correlation...
analysis requires only one exponentiation trace to recover the full RSA private exponent. It can also be used against block ciphers.

In the same vein these attacks can be performed more efficiently using computational data execution traces thanks to the data leakage model additional properties.

5 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.

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.

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. It would then allow also to perform efficient fault injection on obfuscated cipher in order to render more efficient and automated attacks as presented in [21].

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.
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 paths 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 several 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. Depending on the software implementation, such a technique opens the door to more complex attacks as the hardware limitation is no longer an issue.

5.1 Multiple Fault Injection

A classical protection against DFA resides in adding checking after the computation with the aim to detect any inconsistency before releasing the outcome of the cryptographic operation. Typically the operation can be doubled and checked. This latter has the drawback to expose the implementation to a double
injection targeting twice the same fault. A better practice, when it is possible, executes a second operation different from the first one and performs the adequate control afterwards. By doing so, it becomes difficult for an attacker to create a consistent pair of fault leading to exploitable faulty operations without any detection. Typically for a CRT RSA, signature verification turns out to be a valid option. For symmetric algorithms, such as TDES or AES, it could be interesting to consider the inverse operations and checking the integrity of the output with the initial message. Number of techniques can be considered targeting partial operations or even recombination of data. One aspect remains the control. Indeed, one could attempt to perform a double fault injection and induce a first fault during the cryptogram computation followed by another fault compromising the verification. Such a double fault capability is a very powerful way to defeat number of implementations. However it represents a real complexity to conduct, particularly when software countermeasures are implemented to randomise a code execution with obfuscation.

5.2 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 the MixColumn operation. A fault injection of an AES 128 software implementation could target the mitigation (decreasing or increasing its value) of the number of rounds as presented in [8,13]. 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:

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.)

### 5.3 Differential Fault Analysis on AES Library 2

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.[29]. 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 case, 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.

5.4 Double Fault Injection on AES

With a proper tooling, we were able to run such an attack and to defeat a secure AES against fault injection. First of all, it required a minimum of reverse engineering of the binary to identify the sequence of code, more particularly the operation, its inverse and the control. By doing so, it was possible to locate, even roughly, where it could be valuable to fault. Once this work done, providing some scripting, we were able to run a campaign covering a range of fault. As a result, it was possible to induce two faults. A first meaningful modification led to the generation of an exploitable faulty cryptogram. A second fault resulted in the release of the cryptogram by defeating the final comparison. It showed that the complexity could be compensated by a right tooling to instrument a fault campaign in a proper way. Secondly, it demonstrated that a blind attack would have been very unlikely to succeed. An adequate expertise is necessary to run such an attack.

6 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 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 hash values are present in the executed code.

6.1 Combined Attacks

In the hardware field attacks combining fault injection and side-channel analysis were presented first by Amiel et al. in [2]. The objective is to perform a fault injection that would remove some statistical attack countermeasure rendering a statistical attack successful. It is however difficult to reproduce so easily with a high frequency such kind of attacks on hardware devices. In the software area we focus in this paper, such combined attacks can be performed faster and more efficiently in products with the tools we presented previously.
7 Conclusion

Chicken or The Egg? Through the observation made in this paper, we showed that Computational Data Attacks and Software Fault injection can be applied on Software-based solution providing some tooling and a strong expertise. Thanks to a context without the same restrictions compared to practical hardware-based attacks, it provides ways to go further into the analysis of the implementations. Indeed, the leakage and the fault model are under the control of the analyst and consequently can be chosen to the best benefit of the attack success. And even to apply sophisticated attacks that have been considered unrealistic so far from a practical point of view.

Moreover this paper aims at showing that software-based implementations can be successfully stressed using computational data execution or meaningful fault injection techniques. This concerns a wide range of embedded software implementing cryptography, including obfuscated and some whitebox cryptography solutions. We could observe that these techniques can be very powerful and may defeat number countermeasures using well-known first or higher order statistical attacks. Similarly, fault injections can be applied with the right tools and may turn out to be very effective even in complex scenarios.

Finally it confirms that attacks on cryptographic algorithms shall be taken into account regardless their implementation as they may lead to secret key exposure with a minimum of equipment. Particularly in the software-based environment where attacks can be cost effective and whose exploitation can be simple and very short.

References


