White-Box Cryptography: Don't Forget About Grey-Box Attacks

Estuardo Alpirez Bock[1], Joppe W. Bos[2], Chris Brzuska[1], Charles Hubain[3], Wil Michiels[2,4], Cristofaro Mune[5], Eloi Sanfelix Gonzalez[5], Philippe Teuwen[3], and Alexander Treff[6]

[1] Aalto University
[2] NXP Semiconductors
[3] Quarkslab
[4] Technische Universiteit Eindhoven
[5] Riscure
[6] Hamburg University of Technology

Abstract. Despite the fact that all current scientific white-box approaches of standardized cryptographic primitives have been publicly broken, these attacks require knowledge of the internal data representation used by the implementation. In practice, the level of implementation knowledge required is only attainable through significant reverse engineering efforts.

In this paper we describe new approaches to assess the security of white-box implementations which require *neither* knowledge about the look-up tables used *nor* expensive reverse engineering efforts. We introduce the *differential computation analysis* (DCA) attack which is the software counterpart of the differential power analysis attack as applied by the cryptographic hardware community. Similarly, the *differential fault analysis* (DFA) attack is the software counterpart of fault-injection attacks on cryptographic hardware.

For DCA, we developed plugins to widely available dynamic binary instrumentation (DBI) frameworks to produce *software execution traces* which contain information about the memory addresses being accessed. For the DFA attack, we developed modified emulators and plugins for DBI frameworks that allow injecting faults at selected moments within the execution of the encryption or decryption process as well as a framework to automate static fault injection.

To illustrate the effectiveness, we show how DCA and DFA can extract the secret key from numerous publicly available non-commercial white-box implementations of standardized cryptographic algorithms. These approaches allow one to extract the secret key material from white-box implementations significantly faster and without specific knowledge of the white-box design in an automated or semi-automated manner.

This is an extended version of the article published by Springer-Verlag available at 10.1007/978-3-662-53140-2_11 and includes mathematical foundations, a generalization of the DCA attack and the treatment of differential fault attacks.

Keywords: White-box cryptography, Software execution traces, Differential computation analysis, Differential power analysis, Differential fault analysis

1 Introduction

The widespread use of mobile "smart" devices enables users to access a large variety of ubiquitous services. This makes such platforms a valuable target (cf. [72] for a survey on security for mobile devices). There are a number of techniques to protect the cryptographic keys residing on these mobile platforms. The solutions range from unprotected software implementations on the lower range of the security spectrum, to tamper-resistant hardware implementations on the other end. A popular approach which attempts to hide a cryptographic key inside a software program is known as a *white-box implementation*.

Traditionally, people used to work with a security model where implementations of cryptographic primitives are modeled as "black boxes". In this black box model the internal design is trusted and only the in- and output are considered in a security evaluation. As pointed out by Kocher, Jaffe, and Jun [48] in the late 1990s, this assumption turned out to be false in many scenarios. This black box may leak some meta-information: e.g., in terms of timing or power consumption. This side-channel analysis gave rise to the grey-box attack model. Since the usage of (and access to) cryptographic keys changed, so did this security model. In two seminal papers from 2002, Chow, Eisen, Johnson and van Oorschot introduce the white-box model and show implementation techniques which attempt to realize a white-box implementation of symmetric ciphers [28,27].

The idea behind the white-box attack model is that the adversary can be the owner of the device running the software implementation. Hence, it is assumed that the adversary has full control over the execution environment. This enables the adversary to, amongst other things, perform static analysis on the software, inspect and alter the memory used, and even alter intermediate results (similar to hardware fault injections). This white-box attack model, where the adversary is assumed to have such advanced abilities, is realistic on many mobile platforms which store private cryptographic keys of third-parties. White-box cryptography can be used to protect which applications can be installed on a mobile device (from an application store). Other use cases include the protection of digital assets (including media, software and devices) in the setting of digital rights management, the protection of payment credentials in Host Card Emulation (HCE) environments and the protection of credentials for authentication to the cloud. If one has access to a "perfect" white-box implementation of a cryptographic algorithm, then this implies one should not be able to deduce any information about the secret key material used by inspecting the internals of this implementation. This is equivalent to a setting where one has only black-box access to the implementation. As observed by [31] this means that such a white-box implementation should resist all existing and future side-channel and fault injection attacks.

As stated in [27], "*when the attacker has internal information about a cryptographic implementation, choice of implementation is the sole remaining line of defense.*" This is exactly what is being pursued in a white-box implementation: the idea is to embed the secret key in the implementation of the cryptographic operations such that it becomes difficult for an attacker to extract information about this secret key even when the source code of the implementation is provided.

Note that this approach is different from anti-reverse-engineering mechanisms such as code obfuscation [12,53] and control-flow obfuscation [38] although these are typically applied to white-box implementations as an additional line of defense. On top of these, protocol level countermeasures can also be used to mitigate risks in cases where the application is network-connected. In this work we focus exclusively on the robustness of *naked* white-boxed cipher implementations, without any additional mode of operation or protocol.

Although it is conjectured that no long-term defense against attacks on white-box implementations exists [27], there are still a significant number of companies selling white-box solutions. It should be noted that there are almost no known published results on how to turn any of the standardized public-key algorithms into a white-box implementation, besides a patent by Zhou and Chow proposed in 2002 [88]. The other published white-box techniques exclusively focus on symmetric cryptography. However, all such published approaches of standardized cryptographic schemes have been theoretically broken (see Section 2 for an overview). A disadvantage of these published attacks is that they require detailed information on how the white-box implementation is constructed. For instance, knowledge about the exact location of the S-boxes or the round transitions might be required together with the format of the applied encodings to the look-up tables (see Section 2 on how white-box implementations are generally designed). Vendors of white-box implementations try to avoid such attacks by ignoring Kerckhoffs's principle and keeping the details of their design secret (and change the design once it is broken).

Our Contributions. All current cryptanalytic approaches require detailed knowledge about the white-box design used: e.g. the location and order of the S-boxes applied and how and where the encodings are used. This preprocessing effort required for performing an attack is an important aspect of the value attributed to commercial white-box solutions. Vendors are aware that their solutions do not offer a long-term defense, but compensate for this by, for instance, regular software updates. Our contributions are attacks that work in an automated way, and are therefore a major threat for the claimed security level of the offered solutions compared to the ones that are already known. For some of the attacks, we use dynamic binary analysis (DBA), a technique often used to improve and inspect the quality of software implementations, to access and control the intermediate state of the white-box implementation.

One approach to implement DBA is called dynamic binary instrumentation (DBI). The idea is that additional analysis code is injected into the original code of the client program at runtime in order to aid memory debugging, memory leak detection, and profiling. The most advanced DBI tools, such as Valgrind [68] and Pin [54], allow one to monitor, modify and insert instructions in a binary executable. These tools have already demonstrated their potential for behavioral analysis of obfuscated code [77].

We have developed plugins for both Valgrind and Pin to obtain *software traces*[1]: traces which record the read and write accesses made to memory. Additionally, we developed plugins and modified emulators to introduce faults into software, as well as a framework to automate static fault injection[2]. We introduce two new attack vectors that use these techniques in order to retrieve the secret key from a white-box implementation:

– *differential computation analysis* (DCA), which can be seen as the software counterpart of the differential power analysis (DPA) [48] techniques as applied by the cryptographic hardware community. There are, however, some important differences between the usage of the software and hardware traces as we outline in Section 3.2.
– *differential fault analysis* (DFA), which is equivalent to fault-injection [20,15] attacks as applied by the cryptographic hardware community, but uses software means in order to inject faults, which allows for dynamic but also static fault injection.

We demonstrate that DCA can be used to efficiently extract the secret key from white-box implementations which apply at most a single remotely handled external encoding. Similarly, we show that DFA can be applied to white-box implementations that do not apply external encoding to the output of the encryption or decryption process. In this paper, we apply DFA and DCA techniques to publicly available white-box challenges of standardized cryptographic algorithms; concretely this means extracting the secret key from four white-box implementations of the symmetric cryptographic algorithms AES and DES. More examples are available in the repository of our open-source software toolchain.

In contrast to the current cryptanalytic methods to attack white-box implementations, these techniques do not require any knowledge about the implementation strategy used, can be mounted without much technical cryptographic knowledge in an automated way, and extract the key significantly faster. Besides this cryptanalytic framework we discuss techniques which could be used as countermeasures against DCA (see Section 6.1) and DFA (see Section 7).

The main reason why DCA works is related to the choice of (non-) linear encodings which are used inside the white-box implementation (cf. Section 2). These encodings do not sufficiently hide correlations when the correct key is used and enable one to run side-channel attacks (just as in grey-box attack model). Sasdrich, Moradi, and Güneysu looked into this in detail [74] and used the Walsh

[1] The entire software toolchain ranging from the plugins, to the GUI, to the individual scripts to target the white-box challenges, to the tool to analyze the collected traces is released as open-source software: see `https://github.com/SideChannelMarvels`.

[2] The static fault injection framework, the individual scripts and the tool to analyze the collected outputs are released in the same project.

transform (a measure to investigate if a function is a balanced correlation immune function of a certain order) of both the linear and non-linear encodings applied in their white-box implementation of AES. Their results show extreme unbalance where the correct key is used and this explains why first-order attacks like DPA are successful in this scenario.

In this work, we take a step further and analyze *how* the presence of internal encodings on white-box implementations affects the effectiveness of the DCA attack. Thereby, we focus on the encodings suggested by Chow et. al. [28,27], which are a combination of linear and non-linear transformations. We start by studying the effects of a linear transformation on a single key dependent look-up table. We derive a sufficient and necessary condition for the DCA attack to successfully extract the key from a linearly encoded look-up table. Namely, if the outputs of a key-dependent look-up table are encoded via an invertible matrix that contains at least one row with Hamming weight (HW) $= 1$, then the DCA will be successful, since one output bit of the look-up table will not be encoded. Next, we consider the effect that non-linear nibble encodings have on the outputs of key-dependent look-up tables and prove that the use of nibble encodings firstly provides conditions so that the DCA attack succeeds. Namely, when we attack a key dependent look-up table encoded via non-linear nibble encodings, we always obtain DCA results corresponding to a precise set of values for the correct key guess. Therefore, it becomes easy to identify the correct key candidate through the presence of these results. The results obtained from these analyzes help us determine why the DCA attack also works in the presence of both linear and non-linear nibble encodings.

Throughout the paper, we also present experimental results of the DCA attack when performed on single key-dependent look-up tables and on complete white-box implementations. In all cases, the experimental results align with the theoretical observations.

We conclude our investigations on the DCA attack by proposing a generalized DCA attack that successfully extracts the key from a linearly encoded look-up table, no matter the Hamming weights of the invertible matrix rows used to encode the outputs of a key-dependent look-up table. As for the original DCA attack, non-linear encodings can't protect efficiently against the generalized DCA attack either. In this same line, our generalized DCA attack can successfully extract the key from a look-up table (and a complete white-box implementation) that makes use of both linear and non-linear encodings.

2 Overview of White-Box Cryptography Techniques

The white-box attack model allows the adversary to take full control over the cryptographic implementation and the execution environment. It is not surprising that, given such powerful capabilities of the adversary, the authors of the original white-box paper [27] conjectured that no long-term defense against attacks on white-box implementations exists. This conjecture should be understood in the context of code obfuscation, since hiding the cryptographic key inside an implementation is a form of code obfuscation. It is known that obfuscation of *any* program is impossible [6], however, it is unknown if this result applies to a specific subset of white-box functionalities. Moreover, this should be understood in the light of recent developments where techniques using multilinear maps are used for obfuscation that may provide meaningful security guarantees (cf. [35,21,5]). In order to guard oneself in this security model in the medium to long run, one has to use the advantages of a software-only solution. The idea is to use the concept of *software aging* [40]: this forces, at a regular interval, updates to the white-box implementation. It is hoped that when this interval is small enough, this gives insufficient computational time to the adversary to extract the secret key from the white-box implementation. This approach only makes sense if the sensitive data is only of short-term interest, e.g. the DRM-protected broadcast of a football match. However, the practical challenges of enforcing these updates on devices with irregular internet access should be noted.

Protocol level mitigations to limit the impact or applicability of these attacks could also be implemented. Additionally, risk mitigation techniques could be deployed to counter fraud in connected applications with a back-end component such as e.g. mobile payment solutions. However these mitigation techniques are outside the scope of this paper.

External encodings. Besides their primary goal to hide the key, white-box implementations can also be used to provide additional functionality, such as putting a fingerprint on a cryptographic key to enable traitor tracing or hardening software against tampering [60]. There are, however, other security concerns besides the extraction of the cryptographic secret key from the white-box implementation. If one is able to extract (or copy) the entire white-box implementation to another device then one has copied the functionality of this white-box implementation as well, since the secret key is embedded in this program. Such an attack is known as *code lifting*. A possible solution to this problem is to use external encodings [27]. When one assumes that the cryptographic functionality E_k is part of a larger ecosystem then one could implement $E'_k = G \circ E_k \circ F^{-1}$ instead. The input (F) and output (G) encodings are randomly chosen bijections such that the extraction of E'_k does not allow the adversary to compute E_k directly. The ecosystem which makes use of E'_k must ensure that the input and output encodings are canceled. In practice, depending on the application, input or output encodings need to be performed locally by the program calling E'_k. E.g. in DRM applications, the server may take care of the input encoding remotely but the client needs to revert the output encoding to finalize the content decryption.

In this paper, we can mount successful attacks on implementations which apply *at most a single remotely handled external encoding*. When both the input is received with an external encoding applied to it remotely and the output is computed with another encoding applied to it (which is removed remotely) then the implementation is not a white-box implementation of a standard algorithm (like AES or DES) but of a modified algorithm (like $G \circ \mathrm{AES} \circ F^{-1}$ or $G \circ \mathrm{DES} \circ F^{-1}$) which is of limited practical usage.

General idea using internal encodings. The general approach to implement a white-box program is presented in [27]. The idea is to use look-up tables rather than individual computational steps to implement an algorithm and to encode these look-up tables with random bijections. The usage of a fixed secret key is embedded in these tables. Due to this extensive usage of look-up tables, white-box implementations are typically orders of magnitude larger and potentially slower than a regular (non-white-box) implementation of the same algorithm. It is common to write a program that automatically generates a random white-box implementation given the algorithm and the fixed secret key as input. The randomness resides in the randomly chosen internal encodings. Through this randomization, it becomes harder to recover secret-key information from the look-up tables.

Random bijections, i.e. non-linear encodings, are used to achieve *confusion* on the tables, i.e. hiding the relation between the content of the table and the secret key. When non-linear encodings are applied, each look-up table in the construction becomes statistically independent from the key and thus, attacks need to exploit key dependency across several look-up tables. A table T can be transformed into a table T' by using the input bijections I and output bijections O as follows:

$$T' = O \circ T \circ I^{-1}.$$

As a result, we obtain a new table T' which maps encoded inputs to encoded outputs. Note that no information is lost as the encodings are bijective. If table T' is followed by another table R', their corresponding output and input encodings can be chosen such that they cancel out each other. Considering a complex network of look-up tables of a white-box implementation, we have input- and output encodings on almost all look-up tables. The only exceptions are the very first and the very last tables of the implementation, which take the input of the algorithm and correspondingly return the output data. The first tables omit the input encodings and the last tables omit the output encodings. As the internal encodings cancel each other out, the encodings do not affect the input-output behavior of the white-box implementation.

Descriptions of uniformly random bijections (which are non-linear with overwhelming probability) are exponential in the input size of the bijection. Therefore, a uniformly random encoding of the 8-bit S-box requires a storage of 2^8 bytes. Although this may still be acceptable, the problem arises when two values with a byte encoding need to be XORed. An encoded XOR has a storage requirement of 2^{16} bytes. As we need many of them, this becomes an issue. Therefore, one usually splits large values in nibbles of 4 bits. When XORing those, we only need a lookup table of 2^8 nibbles. However, by moving to a split non-linear encoding we introduce a vulnerability since a bit in one nibble does no longer influence the encoded value of another nibble in the same encoded word. To (partly) compensate for this, Chow et al. propose to apply linear encodings whose size is merely quadratic in the input size and thus, they can be implemented on larger words.

Linear transformations, i.e. mixing bijections can be applied at the input and output of each table to achieve *diffusion* such that if one bit is changed in the input of a table, then several bits of its corresponding output are changed too. The linear encodings are invertible and selected uniformly at random. For example, we can select L and A as mixing bijections for inputs and outputs of table T respectively:

$$A \circ T \circ L^{-1}.$$

In the white-box designs of Chow et al. we have 8-bit and 32-bit mixing bijections. The former encode the 8-bit S-box inputs, while the latter obfuscate the MixColumns outputs.

In the remainder of this section we first briefly recall the basics of DES and AES, the two most common choices for white-box implementations, before summarizing the scientific literature related to white-box techniques.

Data Encryption Standard (DES). The DES is a symmetric-key algorithm published as a Federal Information Processing Standard (FIPS) for the United States in 1979 [84]. For the scope of this work it is sufficient to know that DES is an iterative cipher which consists of 16 identical rounds in a criss-crossing scheme known as a Feistel structure. One can implement DES by only working on 8-bit (a single byte) values and using mainly simple operations such as rotate, bitwise exclusive-or, and table lookups. Due to concerns of brute-force attacks on DES the usage of triple DES, which applies DES three times to each data block, has been added to a later version of the standard [84].

Advanced Encryption Standard (AES). In order to select a successor to DES, NIST initiated a public competition where people could submit new designs. After a roughly three-year period the Rijndael cipher was chosen as AES [1,29] in 2000: an unclassified, publicly disclosed symmetric block cipher. The operations used in AES are, as in DES, relatively simple: bitwise exclusive-or, multiplications with elements from a finite field of 2^8 elements and table lookups. Rijndael was designed to be efficient on 8-bit platforms and it is therefore straightforward to create a byte-oriented implementation. AES is available in three security levels. E.g. AES-128 is using a key size of 128 bits and 10 rounds to compute the encryption of the input.

2.1 White-Box Results

White-Box Data Encryption Standard (WB-DES). The first publication attempting to construct a WB-DES implementation dates back from 2002 [28] in which an approach to create white-box implementations of Feistel ciphers is discussed. A first attack on this scheme, which enables one to unravel the obfuscation mechanism, took place in the same year and used fault injections [39] to extract the secret key by observing how the program fails under certain errors. In 2005, an improved WB-DES design, resisting this fault attack, was presented in [52]. However, in 2007, two differential cryptanalytic attacks [14] were presented which can extract the secret key from this type of white-box [36,86]. This latter approach has a time complexity of only 2^{14}.

White-Box Advanced Encryption Standard (WB-AES). The first approach to realize a WB-AES implementation was proposed in 2002 [27]. In 2004, the authors of [17] presented how information about the encodings embedded in the look-up tables can be revealed when analyzing the lookup tables composition. This approach is known as the BGE attack and enables one to extract the key from this WB-AES with a 2^{30} time complexity. A subsequent WB-AES design introduced perturbations in the cipher in an attempt to thwart the previous attack [24]. This approach was broken [67] using algebraic analysis with a 2^{17} time complexity in 2010. Another WB-AES approach which resisted the previous attacks was presented in [87] in 2009 and got broken in 2012 with a work factor of 2^{32} [66].

Another interesting approach is based on using the different algebraic structure for the same instance of an iterative block cipher (as proposed originally in [16]). This approach [42] uses dual ciphers to modify the state and key representations in each round as well as two of the four classical AES operations. This approach was shown to be equivalent to the first WB-AES implementation [27] in [49] in 2013. Moreover, the authors of [49] built upon a 2012 result [82] which improves the most time-consuming phase of the BGE attack. This reduces the cost of the BGE attack to a time complexity of 2^{22}. An independent attack, of the same time complexity, is presented in [49] as well.

Miscellaneous White-Box Results. The above mentioned scientific work only relates to constructing and cryptanalyzing WB-DES and WB-AES. White-box techniques have been studied and used in a broader context. In 2007, the authors of [61] presented a white-box technique to make code tamper resistant. In 2008, the cryptanalytic results for WB-DES and WB-AES were generalized to any substitution linear-transformation (SLT) cipher [62]. In turn, this work was generalized even further and a general analytic toolbox is presented in [4] which can extract the secret for a general SLT cipher.

Formal security notions for symmetric white-box schemes are discussed and introduced in [75,31]. In [18] it is shown how one can use the ASASA construction with injective S-boxes (where ASA stands for the affine-substitution-affine [70] construction) to instantiate white-box cryptography. A tutorial related to white-box AES is given in [65].

2.2 Prerequisites of Existing Attacks

In order to put our results in perspective, it is good to keep in mind the exact requirements needed to apply the white-box attacks from the scientific literature. These approaches require at least a basic knowledge of the scheme which is white-boxed. More precisely, the adversary needs to

- know the type of encodings that are applied on *the intermediate results*,

- know which *cipher operations* are implemented by which *(network of) lookup tables*.

The problem with these requirements is that vendors of white-box implementations are typically reluctant in sharing any information on their white-box scheme (the so-called "security through obscurity"). If that information is not directly accessible but only a binary executable or library is at disposal, one has to invest a significant amount of time in reverse-engineering the binary manually. Removing several layers of obfuscation before retrieving the required level of knowledge about the implementations needed to mount this type of attack successfully can be cumbersome. This additional effort, which requires a high level of expertise and experience, is illustrated by the sophisticated methods used as described in the write-ups of the publicly available challenges as detailed in Section 4.

In contrast, the DCA and DFA approaches introduced in Sections 3 and 7 do not need to remove the obfuscation layers nor require significant reverse engineering of the binary executable.

3 Side Channel Analysis of White-Box Cryptographic Implementations

3.1 Differential Power Analysis

Since the late 1990s it is publicly known that the (statistical) analysis of a power trace obtained when executing a cryptographic primitive might correlate to, and hence reveal information about, the secret key material used [48]. Typically, one assumes access to the hardware implementation of a known cryptographic algorithm. With $I(p_e, k)$ we denote a target intermediate state of the algorithm with input p_e and where only a small portion of the secret key is used in the computation, denoted by k. One assumes that the power consumption of the device at state $I(p_e, k)$ is the sum of a data-dependent component and some random noise, i.e. $\mathcal{L}(I(p_e, k)) + \delta$, where the function $\mathcal{L}(s)$ returns the power consumption of the device during state s, and δ denotes some leakage noise. It is common to assume (see e.g. [56]) that the noise is random, independent from the intermediate state and is normally distributed with zero mean. Since the adversary has access to the implementation he can obtain triples (t_e, p_e, c_e). Here p_e is one plaintext input chosen arbitrarily by the adversary, the c_e is the ciphertext output computed by the implementation using a fixed unknown key, and the value t_e shows the power consumption over the time of the implementation to compute the output ciphertext c_e. The measured power consumption $\mathcal{L}(I(p_e, k)) + \delta$ is just a small fraction of this entire power trace t_e.

The goal of an attacker is to recover the part of the key k by comparing the real power measurements t_e of the device with an estimation of the power consumption under all possible hypotheses for k. The idea behind a Differential Power Analysis (DPA) attack [48] (see [47] for an introduction to this topic) is to divide the measurement traces in two distinct sets according to some property. For example, this property could be the value of one of the bits of the intermediate state $I(p_e, k)$. One assumes — and this is confirmed in practice by measurements on unprotected hardware — that the distribution of the power consumptions for these two sets is different (i.e., they have different means and standard deviations). In order to obtain information about part of the secret key k, for each trace t_e and input p_e, one enumerates all possible values for k (typically $2^8 = 256$ when attacking a key byte), computes the intermediate value $g_e = I(p_e, k)$ for this key guess and divides the traces t_e into two sets according to this property measured at g_e. If the key guess k was correct then the difference of the subsets' averages will converge to the difference of the means of the distributions. However, if the key guess is wrong then the data in the sets can be seen as a random sampling of measurements and the difference of the means should converge to zero. This allows one to observe correct key guesses if enough traces are available. The number of traces required depends, amongst other things, on the measurement noise and means of the distributions (and hence is platform specific).

While having access to output ciphertexts is helpful to validate the recovered key, it is not strictly required. Inversely, one can attack an implementation where only the output ciphertexts are accessible, by targeting intermediate values in the last round. The same attacks apply obviously to the decryption operation.

The same technique can be applied on other traces which contain other types of side-channel information such as, for instance, the electromagnetic radiations of the device. Although we focus on DPA in this paper, it should be noted that there exist more advanced and powerful attacks. This includes, amongst others, higher order attacks [59], correlation power analyses [23] and template attacks [26].

3.2 Software Execution Traces

To assess the security of a binary executable implementing a cryptographic primitive, which is designed to be secure in the white-box attack model, one can execute the binary on a CPU of the corresponding architecture and observe its power consumption to mount a differential power analysis attack (see Section 3.1). However, in the white-box model, one can do much better as the model implies that we can observe everything without any measurement noise. In practice such level of observation can be achieved by instrumenting the binary or instrumenting an emulator being in charge of the execution of the binary. We chose the first approach by using some of the available Dynamic Binary Instrumentation (DBI) frameworks. In short, DBI usually considers the binary executable to analyze as the bytecode of a virtual machine using a technique known as just-in-time compilation. This recompilation of the machine code allows performing transformations on the code while preserving the original computational effects. These transformations are performed at the basic block[3] level and are stored

[3] A basic block is a portion of code with only one entry point and only one exit point. However, due to practical technicalities, the definition of a basic block Pin

in cache to speed up the execution. For example this mechanism is used by the Quick Emulator (QEMU, an open hypervisor that performs hardware virtualization) to execute machine code from one architecture on a different architecture; in this case the transformation is the architecture translation [10]. DBI frameworks, like Pin [54] and Valgrind [68], perform another kind of transformation: they allow to add custom callbacks in between the machine code instructions by writing plugins or tools which hook into the recompilation process. These callbacks can be used to monitor the execution of the program and track specific events. The main difference between Pin and Valgrind is that Valgrind uses an architecture independent Intermediate Representation (IR) called VEX which allows to write tools compatible with any architecture supported by the IR. We developed (and released) such plugins for both frameworks to trace execution of binary executables on x86, x86-64, ARM and ARM64 platforms and record the desired information: namely, the memory addresses being accessed (for read, write or execution) and their content. It is also possible to record the content of CPU registers but this would slow down acquisition and increase the size of traces significantly; we succeeded to extract the secret key from the white-box implementations without this additional information. This is not surprising as table-based white-box implementations are mostly made of memory look-ups and make almost no use of arithmetic instructions (see Section 2 for the design rationale behind many white-box implementations). In some more complex configurations e.g. where the actual white-box is buried into a larger executable it might be desired to change the initial behavior of the executable to call directly the block cipher function or to inject a chosen plaintext in an internal application programming interface (API). This is trivial to achieve with DBI, but for the implementations presented in Section 4, we simply did not need to resort to such methods.

The following steps outline the process how to obtain software traces and mount a DPA attack on these software traces.

First step. Trace a single execution of the white-box binary with an arbitrary plaintext and record all accessed addresses and data over time. Although the tracer is able to follow execution everywhere, including external and system libraries, we reduce the scope to the main executable or to a companion library if the cryptographic operations happen to be handled there. A common computer security technique often deployed by default on modern operating systems is the Address Space Layout Randomization (ASLR) which randomly arranges the address space positions of the executable, its data, its heap, its stack and other elements such as libraries. In order to make acquisitions completely reproducible we simply disable the ASLR, as the white-box model puts us in control over the execution environment. In case ASLR cannot be disabled, it would just be a mere annoyance to realign the obtained traces.

Second step. Next, we visualize the trace to understand where the block cipher is being used and, by counting the number of repetitive patterns, determine

and Valgrind use is slightly different and may include several entry points or exit points.

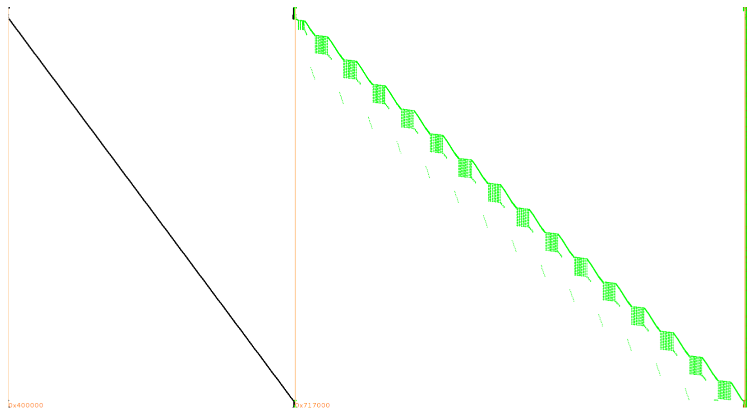


Fig. 1. Visualization of a software execution trace of a white-box DES implementation.

which (standardized) cryptographic primitive is implemented: e.g., a 10-round AES-128, a 14-round AES-256, or a 16-round DES. To visualize a trace, we decided to represent it graphically similarly to the approach presented in [64]. Fig. 1 illustrates this approach: the virtual address space is represented on the x-axis, where typically, on many modern platforms, one encounters the text segment (containing the instructions), the data segment, the uninitialized data (BSS) segment, the heap, and finally the stack, respectively. The virtual address space is extremely sparse so we display only bands of memory where there is something to show. The y-axis is a temporal axis going from top to bottom. Black represents addresses of instructions being executed, green represents addresses of memory locations being read and red when being written. In Fig. 1 one deduces that the code (in black) has been unrolled in one huge basic block, a lot of memory is accessed in reads from different tables (in green) and the stack is comparatively so small that the read and write accesses (in green and red) are barely noticeable on the far right without zooming in.

Third step. Once we have determined which algorithm we target we keep the ASLR disabled and record multiple traces with random plaintexts, optionally using some criteria e.g. in which instructions address range to record activity. This is especially useful for large binaries doing other types of operations we are not interested in (e.g., when the white-box implementation is embedded in a larger framework). If the white-box operations themselves take a lot of time then we can limit the scope of the acquisition to recording the activity around just the first or last round, depending if we mount an attack from the input or output of the cipher. Focusing on the first or last round is typical in DPA-like attacks since it limits the portion of key being attacked to one single byte at once, as explained in Section 3.1. In the example given in Fig. 1, the read accesses pattern makes it trivial to identify the DES rounds and looking at the corresponding instructions (in black) helps defining a suitable instructions address range. While recording all memory-related information in the initial trace (first step), we only record a single type of information (optionally for a limited address range) in this step. Typical examples include recordings of bytes being read from memory, or bytes written to the stack, or the least significant byte of memory addresses being accessed.

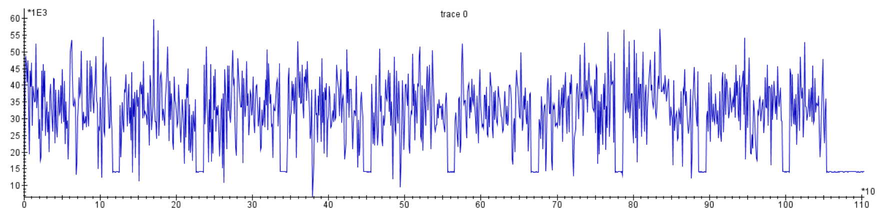
This generic approach gives us the best trade-off to mount the attack as fast as possible and minimize the storage of the software traces. If storage is not a concern, one can directly jump to the third step and record traces of the full execution, which is perfectly acceptable for executables without much overhead, as it will become apparent in several examples in Section 4. This naive approach can even lead to the creation of a fully automated acquisition and key recovery setup.

Fourth step. In step 3 we have obtained a set of software traces consisting of lists of (partial) addresses or actual data which have been recorded whenever an instruction was accessing them. To move to a representation suitable for usual DPA tools expecting power traces, we serialize those values (usually bytes) into vectors of ones and zeros. This step is essential to exploit all the information we have recorded. To understand it, we compare to a classical hardware DPA setup targeting the same type of information: memory transfers.

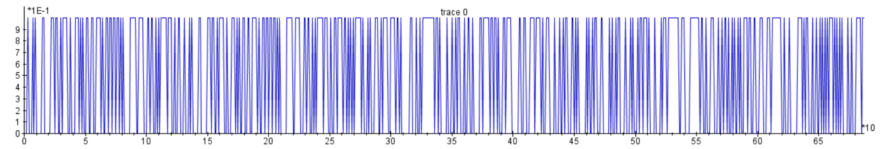
When using DPA, a typical hardware target is a CPU with one 8-bit bus to the memory and all eight lines of that bus will be switching between low and high voltage to transmit data. If a leakage can be observed in the variations of the power consumption, it will be an analog value proportional to the sum of bits equal to one in the byte being transferred on that memory bus. Therefore, in such scenarios, the most elementary leakage model is the Hamming weight of the bytes being transferred between CPU and memory. However, in our software setup, we know the exact 8-bit value and to exploit it at best, we want to attack each bit individually, and not their sum (as in the Hamming weight model). Therefore, the serialization step we perform (converting the observed values into vectors of ones and zeros) is as if in the hardware model each corresponding bus line was leaking individually one after the other.

When performing a DPA attack, a power trace typically consists of sampled analog measures. In our software setting we are working with *perfect* leakages (i.e., no measurement noise) of the individual bits that can take only two possible values: 0 or 1. Hence, our software tracing can be seen from a hardware perspective as if we were probing each individual line with a needle, something requiring heavy sample preparation such as chip decapping and Focused Ion Beam (FIB) milling and patching operations to dig through the metal layers in order to reach the bus lines without affecting the chip functionality. Something which is much more powerful and invasive than external side-channel acquisition.

When using software traces there is another important difference with traditional power traces along the time axis. In a physical side-channel trace, analog values are sampled at a fixed rate, often unrelated to the internal clock of the device under attack, and the time axis represents time linearly. With software execution traces we record information only when it is relevant, e.g. every time a byte is written on the stack if that is the property we are recording, and, more-



(a)



(b)

Fig. 2. Figure (a) is a typical example of a (hardware) power trace of an unprotected AES-128 implementation (one can observe the ten rounds).
Figure (b) is a typical example of a portion of a serialized software trace of stack writes in an AES-128 white-box, with only two possible values: zero or one.

over, bits are serialized as if they were written sequentially. One may observe that given this serialization and sampling on demand, our time axis does not represent an actual time scale. However, a DPA attack does not require a proper time axis. It only requires that when two traces are compared, corresponding events that occurred at the same point in the program execution are compared against each other. Figures 2a and 2b illustrate those differences between traces obtained for usage with DPA and DCA, respectively.

The lack of measurement noise in the traces is an opportunity to apply techniques to reduce the size of these traces and select automatically only a fraction of the samples, which in turn speed up significantly the analysis step. These techniques are detailed by Breunesse, Kizhvatov, Muijrers, and Spruyt in [22].

Fifth step. Once the software execution traces have been acquired and shaped, we can use regular DPA tools to extract the key (see Section 3.3 for a step-by-step graphical presentation of the analysis steps of the DCA). We show in the next sections what the outcome of DPA tools look like, besides the recovery of the key.

Optional step. If required, one can identify the exact points in the execution where useful information leaks. With the help of *known-key correlation* analysis one can locate the exact "faulty" instruction and the corresponding source code line, if available. This can be useful as support for the white-box designer.

To conclude this section, here is a summary of the prerequisites of our differential computation analysis, in opposition to the previous white-box attacks' prerequisites which were detailed in Section 2.2:

- Be able to run several times (a few dozens to a few thousands) the binary in a controlled environment.
- Having knowledge of the plaintexts (before their encoding, if any), or of the ciphertexts (after their decoding, if any).

3.3 Analysis Steps of the DCA

In this section we provide a detailed description of one statistical method to analyze the software execution traces, namely the *difference of means* method. Our description is done for the case when the DCA is performed on implementations of AES-128 (see Section 2). A description for this test when performed on traces acquired from a DES implementation follows analogously. The goal of the attack is to determine the first-round key of AES as it allows to recover the entire key. The first-round key of AES is 128-bit long and the attack aims to recover it byte by byte. For the remainder of this section, we focus on recovering the first byte of the first-round key, as the recovery attack for the other bytes of the first round key proceeds analogously. For the first key byte, the attacker tries out all possible 256 key byte hypotheses k^h, with $1 \leq h \leq 256$, uses the traces to test how good a key byte hypothesis is, and eventually returns the key hypothesis that performs best according to a metric that we specify shortly. For sake of exposition, we focus on one particular key-byte hypothesis k^h.

The adversary starts by collecting memory access traces s_e which are associated with some plaintext p_e. To test the key byte hypothesis k^h, the adversary first specifies a selection function (detailed in Step **2** below) `Sel` that calculates one state byte depending on the plaintext p and the key byte hypothesis k^h. `Sel` returns only the j^{th} bit of the state byte, which we denote as b. For each pair (s_e, p_e), the adversary groups the trace s_e in a set A_b, where $b = \texttt{Sel}(p_e, k^h, j) \in \{0, 1\}$. The adversary then performs the difference of means test (explained from Step **4** on) which, essentially, measures correlations between a bit of the memory access and the bit $b = \texttt{Sel}(p_e, k^h, j)$. If those correlations are strong, then the attack algorithm considers the key byte hypothesis k^h good. We now explain the analysis steps performed in the DCA attack.

1. Collecting Traces: We first execute the white-box program n times, each time using a different plaintext p_e, $1 \leq e \leq n$ as input. For each execution, one software trace s_e is recorded during the first round of AES. Fig. 3 shows a single software trace consisting of 300 samples. Each sample corresponds to one bit of the memory addresses accessed during execution.

2. Selection Function: We define a selection function for calculating an intermediate state byte of the calculation process of AES. More precisely, we calculate a state byte which depends on the key byte we are analyzing in the actual iteration of the attack. For the sake of simplicity, we refer to this state byte as z. The selection function returns only one bit of z, which we refer to as our *target bit*. The value of our target bit will be used as a distinguisher in the following steps.

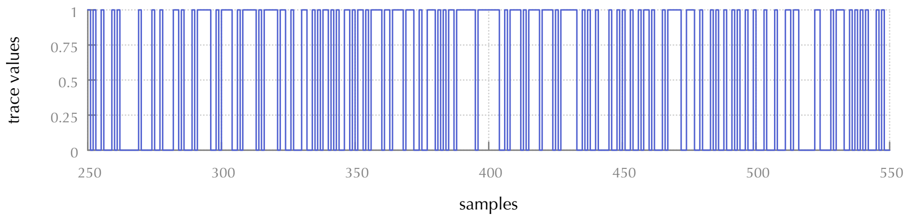


Fig. 3. Single software trace consisting of 300 samples.

In this work, we define our selection function the same way as defined in [48]. Our selection function $\texttt{Sel}(p_e, k^h, j)$ calculates thus the state z after the $\texttt{SBox}$ substitution in the first round. The index j indicates *which* bit of z is returned, with $1 \leq j \leq 8$.

$$\texttt{Sel}(p_e, k^h, j) := \texttt{SBox}(p_e \oplus k^h)[j] = b \in \{0, 1\}. \tag{1}$$

Depending on the white-box implementation being analyzed, it may be the case that strong correlations between b and the software traces are only observable for some bits of z, i.e. depending on which j we choose to focus on. Therefore, we perform the following Steps 3, 4 and 5 for each bit j of z.

3. Sorting of Traces: We sort each trace s_e into one of the two sets A_0 or A_1 according to the value of $\texttt{Sel}(p_e, k^h, j) = b$:

$$\text{For } b \in \{0, 1\}\ A_b := \{s_e | 1 \leq e \leq n,\ \texttt{Sel}(p_e, k^h, j) = b\}. \tag{2}$$

4. Mean Trace: We now take the two sets of traces obtained in the previous step and calculate a *mean trace* for each set. We add all traces of one set sample wise and divide them by the total number of traces in the set. For $b \in \{0, 1\}$, we define

$$\bar{A}_b := \frac{\sum_{s \in A_b} s}{|A_b|}. \tag{3}$$

For each of the two sets, we obtain a mean trace such as the one shown in Fig. 4.

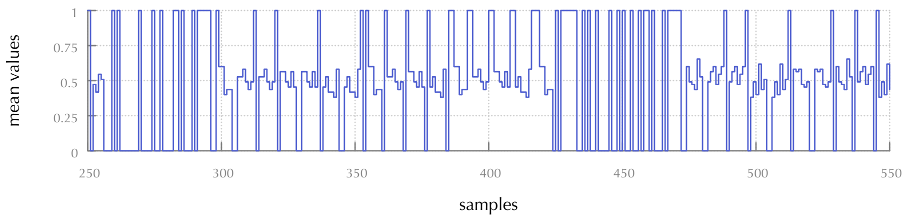


Fig. 4. Mean trace for the set A_0

5. Difference of Means: We now calculate the difference between the two previously obtained mean traces sample wise. Fig. 5 shows the resulting difference of means trace:

$$\varDelta = |\bar{A}_0 - \bar{A}_1|. \tag{4}$$

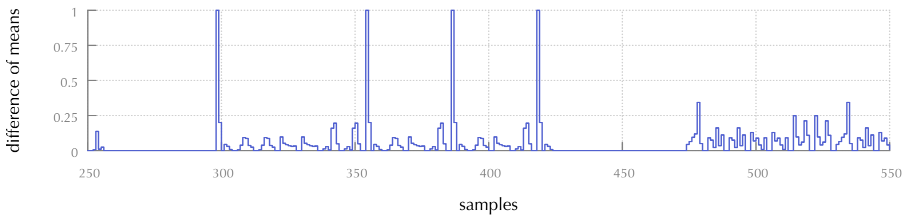


Fig. 5. Difference of means trace for correct guess

6. Best target bit: We now compare the difference of means traces obtained for all target bits j for a given key hypothesis k^h. Let $\varDelta^j$ be the difference of means trace obtained for target bit j, and let $H(\varDelta^j)$ be the highest peak in the trace $\varDelta^j$. Then, we select $\varDelta^j$ as the best difference of means trace for k^h, such that $H(\varDelta^j)$ is maximal amongst the highest peaks of all other difference of means traces, i.e.

$$\forall\ 1 \leq j' \leq 8,\ H(\varDelta^{j'}) \leq H(\varDelta^j).$$

In other words, we look for the highest peak obtained from any difference of means trace. The difference of means trace with the highest peak $H(\varDelta^j)$ is assigned as the difference of means obtained for the key hypothesis k^h analyzed in the actual iteration of the attack, such that $\varDelta^h := \varDelta^j$. We explain this reasoning in the analysis provided after Step 7.

7. Best Key Byte Hypothesis: Let $\varDelta^h$ be the difference of means trace for key hypothesis h, and let $H(\varDelta^h)$ be the highest peak in the trace $\varDelta^h$. Then, we select k^h such that $H(\varDelta^h)$ is maximal amongst all other difference of means traces $\varDelta^h$, i.e.

$$\forall\ 1 \leq h' \leq 256,\ H(\varDelta^{h'}) \leq H(\varDelta^h).$$

The higher $H(\varDelta^h)$, the more likely it is that this key hypothesis is the correct one, which can be explained as follows. The attack partitions the traces in sets A_0 and A_1 based on whether a bit in z is set to 0 or 1. First, suppose that the key hypothesis is correct and consider a region R in the traces where (an encoded version of) z is processed. Then, we expect that the memory accesses in R for A_0 are slightly different from A_1. After all, if they would be the same, the computations would be the same too. We know that the computations are different because the value of the target bit is different. Hence, it may be expected that this difference is reflected in the mean traces for A_0 and A_1, which results in a peak in the difference of means trace. Next, suppose that the key hypothesis was not correct. Then, the sets A_0 and A_1 can rather be seen as a random partition of the traces, which implies that z can take any arbitrary value in both A_0 and A_1. Hence, we do not expect big differences between the executions traces from A_0 and A_1 in region R, which results in a rather flat difference of means trace.

To illustrate this, consider the difference of means trace depicted in Fig. 5. This difference of means trace corresponds to the analysis performed on a white-box implementation obtained from the hack.lu challenge [9]. This is a public table-based implementation of AES-128, which does not make any use of internal encodings. For analyzing it, a total of 100 traces were recorded. The trace in Fig. 5 shows four spikes which reach the maximum value of 1 (note that the sample points have a value of either 0 or 1). Let ℓ be one of the four sample points in which we have a spike. Then, having a maximum value of 1 means that for all traces in A_0, the bit of the memory address considered in ℓ is 0 and that this bit is 1 for all traces in A_1 (or vice versa). In other words, the target bit $z[j]$ is either directly or in its negated form present in memory address accessed in the implementation. This can happen if z is used in non-encoded form as input to a lookup table or if it is only XORed with a constant mask. For sake of completeness, Fig. 6 shows a difference of means trace obtained for an incorrect key hypothesis. No sample has a value higher than 0.3.

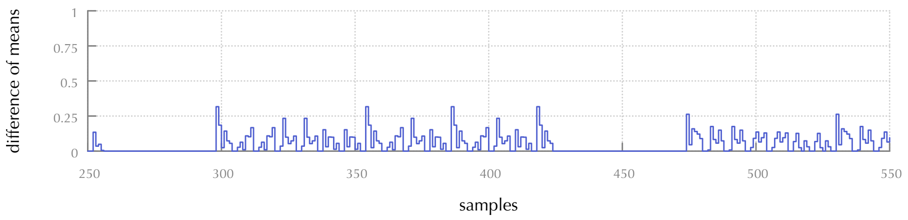


Fig. 6. Difference of means trace for incorrect guess

Successful Attack Throughout this paper, considering the implementation of a cipher, we refer to the DCA attack as being *successful for a given key k*, if this key is ranked number 1 amongst all possible keys for a large enough number of traces. It may be the case that multiple keys have this same rank. If DCA is not successful for k, then it is called *unsuccessful for key k*. Remark that in practice, an attack is usually considered successful as long as the correct key guess is ranked as one of the best key candidates. We use a stronger definition as we require the correct key guess to be ranked as the best key candidate.

Alternatively when attacking a single n-bit to n-bit key dependent look-up table, we consider the DCA attack as being *successful for a given key k*, if this key is ranked number 1 amongst all possible keys for exactly 2^n traces. Thereby, each trace is generated by giving exactly 2^n different inputs to the look-up table,

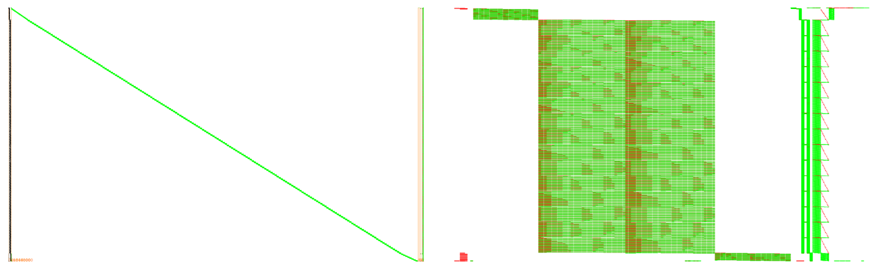


Fig. 7. (a) Visualization of a software execution trace of the binary Wyseur white-box challenge showing the entire accessed address range. (b) A zoom on the stack address space from the software trace shown in (a). The 16 rounds of the DES algorithm are clearly visible.

i.e. all possible inputs that the look-up table can obtain. To get the correlation between a look-up table output and our selection function, the correlation we obtain by evaluating all 2^n possible inputs is exactly equal to the correlation we obtain by generating a large enough number of traces for inputs chosen uniformly at random. We use this property for the experiments we perform in Section 5.

4 Analyzing Publicly Available White-Box Implementations

The Wyseur Challenge As far as we are aware, the first public white-box challenge was created by Brecht Wyseur in 2007. On his website[4] one can find a binary executable containing a white-box DES encryption operation with a fixed embedded secret key. According to the author, this WB-DES approach implements the ideas from [28,52] (see Section 2.1) plus "some personal improvements". The interaction with the program is straightforward: it takes a plaintext as input and returns a ciphertext as output to the console. The challenge was solved after five years (in 2012) independently by James Muir and "SysK". The latter provided a detailed description [79] and used differential cryptanalysis (similar to [36,86]) to extract the embedded secret key.

Figure 7a shows a full software trace of an execution of this WB-DES challenge. On the left one can see the loading of the instructions (in black), since the instructions are loaded repeatedly from the same addresses this implies that loops are used which execute the same sequence of instructions over and over again. Different data is accessed fairly linearly but with some local disturbances as indicated by the large diagonal read access pattern (in green). Even to the trained eye, the trace displayed in Figure 7a does not immediately look familiar

[4] See `http://whiteboxcrypto.com/challenges.php`.

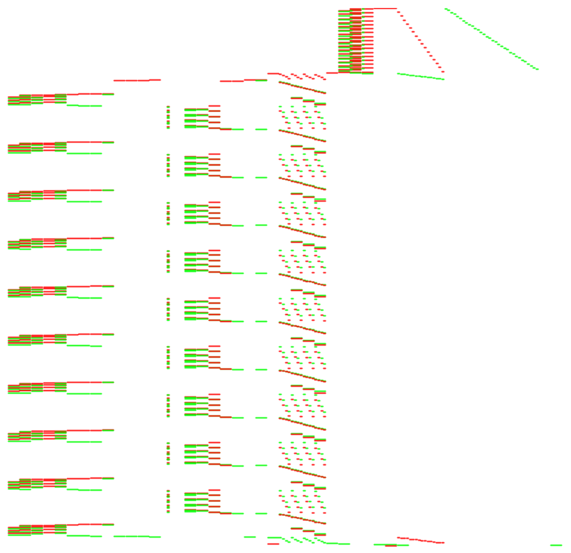


Fig. 8. Visualization of the stack reads and writes in a software execution trace of the Hack.lu 2009 challenge.

to DES. However, if one takes a closer look to the address space which represents the stack (on the far right) then the 16 rounds of DES can be clearly distinguished. This zoomed view is outlined in Figure 7b where the y-axis is unaltered (from Figure 7a) but the address range (the x-axis) is rescaled to show only the read and write accesses to the stack.

Due to the loops in the program flow, we cannot just limit the tracer to a specific memory range of instructions and target a specific round. As a trace over the full execution takes a fraction of a second, we traced the entire program without applying any filter. The traces are easily exploited with DCA: e.g., if we trace the bytes written to the stack over the full execution and we compute a DPA over this entire trace without trying to limit the scope to the first round, the key is completely recovered with as few as 65 traces when using the output of the first round as intermediate value.

The execution of the entire attack, from the download of the binary challenge to full key recovery, including obtaining and analyzing the traces, took less than an hour as its simple textual interface makes it very easy to hook it to an attack framework. Extracting keys from different white-box implementations based on this design now only takes a matter of seconds when automating the entire process as outlined in Section 3.2.

The Hack.lu 2009 Challenge As part of the Hack.lu 2009 conference, which aims to bridge ethics and security in computer science, Jean-Baptiste Bédrune released a challenge [9] which consisted of a *crackme.exe* file: an executable for the Microsoft Windows platform. When launched, it opens a GUI prompting for an input, redirects it to a white-box and compares the output with an internal reference. It was solved independently by Eloi Vanderbéken [85], who reverted the functionality of the white-box implementation from encryption to decryption, and by "SysK" [79] who managed to extract the secret key from the implementation.

Our plugins for the DBI tools have not been ported to the Windows operating system and currently only run on GNU/Linux and Android. In order to use our tools directly, we decided to trace the binary with our Valgrind variant and Wine[5] [3], an open source compatibility layer to run Windows applications under GNU/Linux. We automated the GUI, keyboard and mouse interactions using xdotool[6]. Due to the configuration of this challenge we had full control on the input to the white-box. Hence, there was no need to record the output of the white-box and no binary reverse-engineering was required at all.

Fig. 8 shows the read and write accesses to the stack during a single execution of the binary. One can observe ten repetitive patterns on the left interleaved with nine others on the right. This indicates (with high probability) an AES encryption or decryption with a 128-bit key. The last round being shorter as it omits the *MixColumns* operation as per the AES specification. We captured a few dozen traces of the entire execution, without trying to limit ourselves to the first round. Due to the overhead caused by running the GUI inside Wine the acquisition ran slower than usual: obtaining a single trace took three seconds. Again, we applied our DCA technique on traces which recorded bytes written to the stack. The secret key could be completely recovered with only 16 traces when using the output of the first round *SubBytes* as intermediate value of an AES-128 encryption. As "SysK" pointed out in [79], this challenge was designed to be solvable in a couple of days and consequently did not implement any internal encoding, which means that the intermediate states can be observed directly. Therefore in our DCA the correlation between the internal states and the traced values gets the highest possible value, which explains the low number of traces required to mount a successful attack.

The SSTIC 2012 Challenge Every year for the SSTIC, *Symposium sur la sécurité des technologies de l'information et des communications* (Information technology and communication security symposium), a challenge is published which consists of solving several steps like a Matryoshka doll. In 2012, one step of the challenge [57] was to validate a key with a Python bytecode "check.pyc": i.e. a marshalled object[7]. Internally this bytecode generates a random plaintext, forwards it to a white-box (created by Axel Tillequin) *and* to a regular DES encryption using the key provided by the user and then compares both ciphertexts. Five participants managed to find the correct secret key corresponding to this challenge and their write-ups are available at [57]. A number of solutions identified the implementation as a WB-DES without encodings (naked variant)

[5] `https://www.winehq.org/`
[6] `http://www.semicomplete.com/projects/xdotool/`
[7] `https://docs.python.org/2/library/marshal.html`

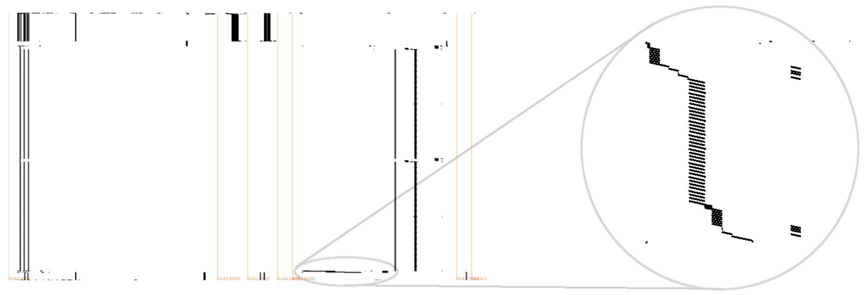


Fig. 9. Visualization of the instructions in a software execution trace of the Karroumi WB-AES implementation by Klinec, with a zoom on the core of the white-box.

as described in [28]. Some extracted the key following the approach from the literature while some performed their own algebraic attack.

Tracing the entire Python interpreter with our tool, based on either PIN or Valgrind, to obtain a software trace of the Python binary results in a significant overhead. Instead, we instrumented the Python environment directly. Actually, Python bytecode can be decompiled with little effort as shown by the write-up of Jean Sigwald. This contains a decompiled version of the "check.pyc" file where the white-box part is still left serialized as a pickled object[8]. The white-box makes use of a separate *Bits* class to handle its variables so we added some hooks to record all new instances of that particular class. This was sufficient. Again, as for the Hack.lu 2009 WB-AES challenge (see Section 4), 16 traces were enough to recover the key of this WB-DES when using the output of the first round as intermediate value. This approach works with such a low number of traces since the intermediate states are not encoded.

A White-Box Implementation of the Karroumi Approach A white-box implementation of both the original AES approach [27] and the approach based on dual ciphers by Karroumi [42] is part of the Master thesis by Dušan Klinec [46][9]. As explained in Section 2.1, this is the latest academic variant of [27]. Since there is no challenge available, we used Klinec's implementation to create two challenges: one with and one without external encodings. This implementation is written in C++ with extensive use of the Boost[10] libraries to dynamically load and deserialize the white-box tables from a file. The left part of Figure 9 shows a software trace when running this white-box AES binary executable. The white-box code itself constitutes only a fraction of the total instructions; the right part of Figure 9 shows an enlarged view of the white-box core. Here, one can recognize the nine *MixColumns* operating on the four columns. This structure can be observed even better from the stack trace of

[8] `https://docs.python.org/2/library/pickle.html`
[9] The code be found at `https://github.com/ph4r05/Whitebox-crypto-AES`.
[10] `http://www.boost.org/`

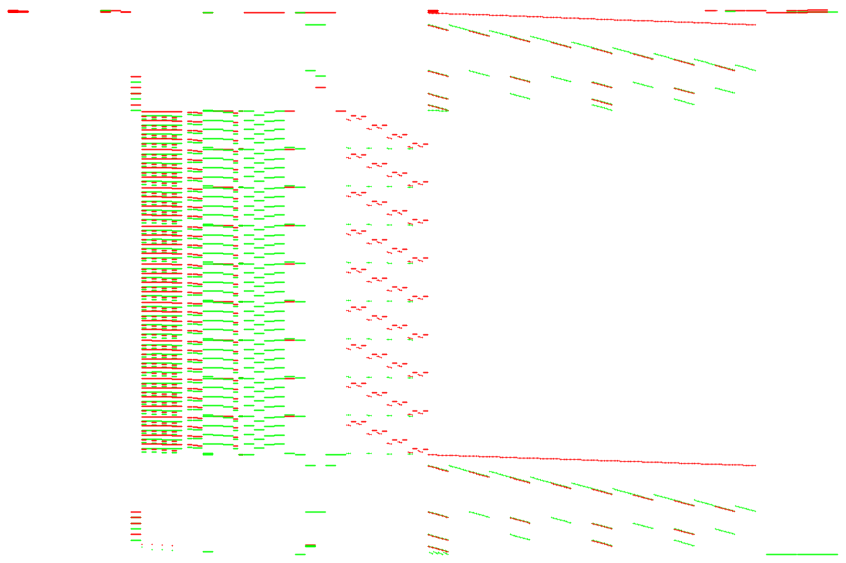


Fig. 10. Visualization of the stack reads and writes in the software execution trace portion limited to the core of the Karroumi WB-AES.

Table 1. DCA ranking for a Karroumi white-box implementation when targeting the output of the *SubBytes* step in the first round based on the least significant address byte on memory reads.

		key byte															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
target bit	**0**	1	256	255	256	255	256	253	1	256	256	239	256	1	1	1	255
	1	1	256	256	256	1	255	256	1	1	5	1	256	1	1	1	1
	2	256	1	255	256	1	256	226	256	256	256	1	256	22	1	256	256
	3	256	255	251	1	1	1	254	1	1	256	256	253	254	256	255	256
	4	256	256	74	256	256	256	255	256	254	256	256	256	1	1	256	1
	5	1	1	1	1	1	1	50	256	253	1	251	256	253	1	256	256
	6	254	1	1	256	254	256	248	256	252	256	1	14	255	256	250	1
	7	1	256	1	1	252	256	253	256	256	255	256	1	251	1	254	1
	All	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓

Figure 10. Therefore we used instruction address filtering to focus on the white-box core and skip all the Boost C++ operations.

The best results were obtained when tracing the lowest byte of the memory addresses used in read accesses (excluding stack). Initially we followed the same approach as before: we targeted the output of the *SubBytes* in the first round. But, in contrast to the other challenges considered in this work, it was not enough to immediately recover the entire key. For some of the bits of the intermediate value we observed a significant correlation peak: this is an indication that the first key candidate is very probably the correct one. Table 1 shows the ranking

Table 2. DCA ranking for a Karroumi white-box implementation when targeting the output of the multiplicative inversion inside the *SubBytes* step in the first round based on the least significant address byte on memory reads.

		key byte															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
target bit	**0**	256	256	1	1	1	256	256	256	254	1	1	1	255	256	256	1
	1	1	1	253	1	1	256	249	256	256	256	226	1	254	256	256	256
	2	256	256	1	1	255	256	256	256	251	1	255	256	1	1	254	256
	3	254	1	69	1	1	1	1	1	252	256	1	256	1	256	256	256
	4	254	1	255	256	256	1	255	256	1	1	256	256	238	256	253	256
	5	254	256	250	1	241	256	255	3	1	1	256	256	231	256	208	254
	6	256	256	256	256	233	256	1	256	1	1	256	256	1	1	241	1
	7	63	256	1	256	1	255	231	256	255	1	255	256	255	1	1	1
	All	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

of the right key byte value amongst the guesses after 2000 traces, when sorted according to the difference of means (see Section 3.1). If the key byte is ranked at position 1, this means it was properly recovered by the attack. In total, for the first challenge we constructed, 15 out of 16 key bytes were ranked at position 1 for at least one of the target bits and one key byte (key byte 6 in the table) did not show any strong candidate. However, recovering this single missing key byte is trivial using brute-force.

Since the dual ciphers approach [42] uses affine self-equivalences of the original S-box, it might also be interesting to base the guesses on another target: the multiplicative inverse in the finite field of 2^8 elements (inside the *SubBytes* step) of the first round, before any affine transformation. This second attack shows results in Table 2 similar to the first one but distributed differently. With this sole attack the 16 bytes were successfully recovered — and the number of required traces can even be reduced to about 500 — but it may vary for other generations of the white-box as the distribution of leakages in those two attacks and amongst the target bits depends on the random source used in the white-box generator. However, when combining both attacks, we could always recover the full key.

It is interesting to observe in Table 1 and 2 that when a target bit of a given key byte does not leak, such that the key byte is not ranked first, it is very often *the worst* candidate (ranked at the 256th position) rather than being at a random position. This observation, that still holds for larger numbers of traces, can also be used to recover the key.

In order to give an idea of what can be achieved with an *automated* attack against new instantiations of this white-box implementation with other keys, we provide some figures: The acquisition of 500 traces takes about 200 s on a regular laptop (dual-core i7-4600U CPU at 2.10 GHz). This results in 832 kbits (104 kB) of traces when limited to the execution of the first round. Running both attacks as described in this section requires less than 30 s. Attacking the second challenge with external encodings gave similar results. This was expected as there is no difference, from our adversary perspective, when applying external encodings or omitting them since in both cases we have knowledge of the original plaintexts before any encoding is applied.

The NoSuchCon 2013 Challenge In April 2013, a challenge designed by Eloi Vanderbéken was published for the occasion of the NoSuchCon 2013 conference[11]. The challenge consisted of a Windows binary embedding a white-box AES implementation. It was of "keygen-me" type, which means one has to provide a name and the corresponding *serial* to succeed. Internally the serial is encrypted by a white-box and compared to the MD5 hash of the provided name.

The challenge was completed by a number of participants (cf. [78,55]) but without ever recovering the key. It illustrates one more issue designers of white-box implementations have to deal with in practice. Namely for this challenge, one can convert an encryption routine into a decryption routine without actually extracting the key.

For a change, the design is not derived from Chow [27]. However, the white-box was designed with external encodings which were *not* part of the binary. Hence, the user input was considered as encoded with an unknown scheme and the encoded output is directly compared to a reference. These conditions, without any knowledge of the relationship between the real AES plaintexts or ciphertexts and the effective inputs and outputs of the white-box, make it infeasible to apply a traditional DPA attack, since, for a DPA attack, we need to construct the guesses for the intermediate values. Note that, as discussed in Section 2, this white-box implementation is *not* compliant with AES anymore but computes some variant $E'_k = G \circ E_k \circ F^{-1}$. Nevertheless we did manage to recover the key and the encodings from this white-box implementation with a new algebraic attack, as described in [81]. This was achieved after a painful de-obfuscation of the binary (almost completely performed by previous write-ups [78] and [55]), a step needed to fulfill the prerequisites for such attacks as described in Section 2.2.

The same white-box is found amongst the CHES 2015 challenges [12] in a GameBoy ROM and the same algebraic attack is used successfully as explained in [80] once the tables got extracted.

5 Explaining the Experimental Success of the DCA

We have shown in the previous section how the DCA attack can successfully attack a number of publicly available white-box implementations. Most of these white-box implementations use the encodings techniques suggested by Chow et. al. [27] to protect the contents of key dependent look-up tables inside the implementation. These types of encodings are the methods usually applied in white-box designs presented in the literature as well. In this section we analyze

[11] See `http://www.nosuchcon.org/2013/`

[12] `https://ches15challenge.com/static/CHES15Challenge.zip`

the use of such internal encodings. We discuss linear encodings, non-linear encodings and a combination of both as means to protect key dependent look-up tables in a white-box implementation. Moreover, we analyze how the presence of such encodings affects the vulnerability of a white-box implementation to the DCA attack. If intermediate values in an implementation are encoded, it becomes more difficult to recalculate such values using our selection function as defined in Step 2 of the DCA (see Section 3.3). Namely, `Sel` does not consider the transformations used to encode these intermediate values, but only calculates a value *before* it is encoded. Thus, what we calculate with `Sel` in Step 2 does not necessarily match with the actual encoded value computed by the white-box, even if the correct key hypothesis is used.

For our analyses in this section, we first build single look-up tables which map an 8-bit long input to an 8-bit long output. More precisely, these look-up tables correspond to the key addition operation merged with the S-box substitution step performed on AES (see Section 2). As common in the literature, we refer to such look-up tables as *T-boxes*. We apply the different encoding methods to the outputs of the look-up tables and obtain encoded T-boxes. Note that Chow et al. merge the T-box and the MixColumns operation into one 8-to-32 bit look-up table and encode the look-up table output via a 32-bit linear transformation. However, an 8-to-32 bit look-up table can be split into four 8-to-8 bit lookup tables, which correspond to the look-up tables used for our analyses.[13]

Following our definition for a successful DCA attack on an n-to-n look-up table given in Section 3.3, we generate exactly 256 different software traces for attacking a T-box. Our selection function is defined the same way as in Step 2 of Section 3.3 and calculates the output of the T-boxes *before* it is encoded. The output of the T-box is a typical vulnerable spot for performing the DCA on white-box implementations as this output can be calculated based on a known plaintext and a key guess. As we will see in this section, internal encodings as suggested by Chow et al. cannot effectively add a masking countermeasure to the outputs of the S-box.

5.1 Linear Encodings

The outputs of a T-box can be *linearly* encoded by applying linear transformations. To do this, we randomly generate an 8-to-8 invertible matrix such as, for

[13] It can be the case that the four lookup tables are, in isolation, not bijective. In that case, our results do not apply directly. It is left as an exercise to adapt them to this setting.

instance,

$$A = \begin{bmatrix} 1&1&1&0&1&0&0&0 \\ 0&0&1&1&0&1&0&1 \\ 1&0&1&0&0&0&1&0 \\ 0&1&1&1&0&1&0&1 \\ 0&1&1&1&0&1&1&1 \\ 1&1&0&0&0&0&1&1 \\ 0&0&1&0&0&0&1&0 \\ 1&0&0&0&0&1&1&0 \end{bmatrix} . \qquad (5)$$

For each output y of the T-box T, we perform a matrix multiplication $A \cdot y$ and obtain an encoded output m. We obtain a new look-up table lT, which maps each input x to a linearly encoded output m. Fig. 11 displays this behavior.

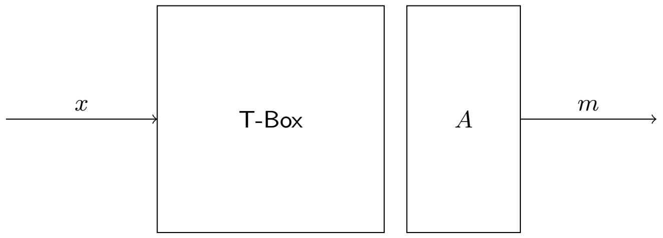


Fig. 11. An lT-box maps each input x to a linearly encoded output m.

We now compute the DCA on the outputs of lT. The difference of means analysis is performed in the same way as described in Section 3.3. Fig. 12 shows the results of the analysis when using the correct key guess. Since we are attacking only an 8×8 look-up table, the generated software traces consist only of 8 samples which correspond to the 8 output bits of the lT-box. As it can be seen, all samples in the difference of means curve have a value of zero or almost zero. No correlations can be identified and thus, the analysis is not successful if the output of the original T-box is encoded using the matrix A.

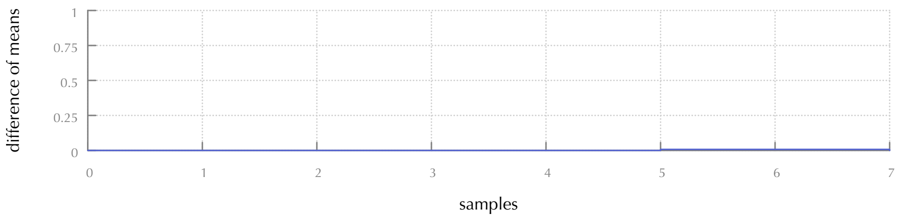


Fig. 12. Difference of means trace for the lT-box

The results shown in Fig. 12 correspond to the DCA performed on a look-up table constructed using one particular linear transformation to encode the output of one look-up table. We observe that the DCA as described in Section 3.3 is not effective in the presence of this particular transformation. However in practice, linear transformations are randomly chosen and some may not effectively hide information about a target bit, such that the DCA attack is successful. The theorem below gives a necessary and sufficient condition under which the DCA attack is successful in the presence of linear transformations.

Theorem 1. *Given a T-box encoded via an invertible matrix A. The DCA attack returns a difference of means value equal to 1 for the correct key guess if and only if the matrix A has at least one row i with Hamming weight $(HW) = 1$. Otherwise, the DCA attack returns a difference of means value equal to 0 for the correct key guess.*

Proof. For all $1 \leq j \leq 8$ let $y[j]$ be the j^{th} bit of the output y of a T-box. Let $a_{ij} \in GF(2)$ be the entries of an 8×8 matrix A, where i denotes the row and j denotes the column of the entry. We obtain each encoded bit $m[i]$ of the lT-box via

$$m[i] = \sum_{j} a_{ij} \cdot y[j] = \sum_{j:a_{ij}=1} y[j]. \tag{6}$$

Suppose that row i of A has $HW(i) = 1$. Let j be such that $a_{ij} = 1$. It follows from Equation (6) that $m[i] = y[j]$. Let k^h be the correct key hypothesis and let bit $y[j]$ be our target bit. With our selection function $\texttt{Sel}(p_e, k^h, j)$ we calculate the value for $y[j]$ and sort the corresponding trace in the set A_0 or A_1. We refer to these sets as sets consisting of encoded values m, since a software trace is a representation of the encoded values. Recall now that $y[j] = m[i]$. It follows that $m[i] = 0$ for all $m \in A_0$ and $m[i] = 1$ for all $m \in A_1$. Thus, when calculating the averages of both sets, for $\bar{A}[i]$, we obtain $\bar{A}_0[i] = 0$ and $\bar{A}_1[i] = 1$. Subsequently, we obtain a difference of means curve with $\Delta[i] = 1$, which leads us to a successful DCA attack.

What's left to prove is that if row i has $HW(i) > 1$, then the value of bit $y[j]$ is masked via the linear transformation such that the difference of means curve obtained for $\Delta[i]$ has a value converging to zero. Suppose that row i of A has $HW(i) = l > 1$. Let j be such that $a_{ij} = 1$ and let $y[j']$ denote one bit of y, such that $a_{ij'} = 1$. It follows from Equation (6) that the value of $m[i]$ is equal to the sum of at least two bits $y[j]$ and $y[j']$. Let k^h be the correct key hypothesis and let $y[j']$ be our target bit. Let $\vec{v}$ be a vector consisting of the bits of y, for which $a_{ij} = 1$, excluding bit $y[j']$. Since row i has $HW(i) = l$, vector $\vec{v}$ consists of $l - 1$ bits. This means that $\vec{v}$ can have up to 2^{l-1} possible values. Recall that each non-encoded T-box output value y occurs with an equal probability of $1/256$ over the inputs of the T-box. Thus, all 2^{l-1} possible values of $\vec{v}$ occur with the same probability over the inputs of the T-box. The sum of the $l - 1$ bits in $\vec{v}$ is equal to 0 or 1 with a probability of 50%, independently of the value of $y[j']$. Therefore, our target bit $y[j']$ is masked via $\sum_{j:a_{ij}=1,j\neq j'} y[j]$ and our calculations obtained with $\texttt{Sel}(p_e, k^h, j')$ only match 50% of the time with the value of $m[i]$. Each set A_b consists thus of an equal number of values $m[i] = 0$ and $m[i] = 1$. The difference between the averages of both sets is thus equal to zero and the DCA is unsuccessful for k^h. □

One could be tempted to believe that using a matrix which does not have any identity row serves as a good countermeasure against the DCA attack. However, we could easily adapt the DCA attack such that it is also successful in the presence of a matrix without any identity row. In Step 2, we just need to define our selection function such that, after calculating an 8-bit long output state z, we calculate all possible linear combinations LC of the bits in z. Thereby, in Step 3 we sort according to the result obtained for an LC. This means that we perform Steps 3 to 5 for each possible LC ($2^8 = 256$ times per key guess). For at least one of those cases, we will obtain a difference of means curve with peak values equal to 1 for the correct key guess as our LC will be equal to the LC defined by row i of matrix A. Our selection function calculates thus a value equal to the encoded value $m[i]$ and we obtain perfect correlations. In Section 6 we explain how we use this attack idea in order to modify our DCA attack, such that its success rate increases.

Note that Theorem 1 also applies in the presence of affine encodings. In case we add a 0 to a target bit, traces $\bar{A}_0$ and $\bar{A}_1$ do not change and in case we add a 1 the entries in $\bar{A}_0$ and $\bar{A}_1$ that relate to the target bit change to 1 minus their value. In both cases, the difference of means value does not change.

5.2 Non-Linear Encodings

Next, we consider the effect that non-linear encodings have on the outputs of a T-box. For this purpose, we randomly generate bijections, which map each output value y of the T-box to a different value f and thus obtain a non-linearly encoded T-box, which we call OT-box. Recall that a T-box is a bijective function. If we encode each possible output of a T-box T with a randomly generated byte function O and obtain the OT-box OT, then OT does not leak any information about T. Namely, given OT, *any* other T-box T' could be a candidate for constructing the same OT-box OT, since there always exists a corresponding function O' which could give us $O'T'$ such that $O'T' = OT$. Chow et. al. refer to this property as *local security* [28]. Based on this property, we could expect resistance against the DCA attack for a non-linearly encoded T-box. For practical implementations, unfortunately, using an 8-to-8 bit encoding for each key dependent look-up table is not realistic in terms of code size (see Section 4.1 of [65] for more details). Therefore, non-linear *nibble encodings* are typically used to encode the outputs of a T-box. The output of a T-box is 8-bit long, each half of the output is encoded by a different 4-to-4 bit transformation and both results are concatenated. Fig. 13 displays the behavior of an OT-box constructed using two nibble encodings.

Encoding the outputs of a T-box via non-linear nibble encodings does not hide correlations between the secret key of the T-box and its output bits as

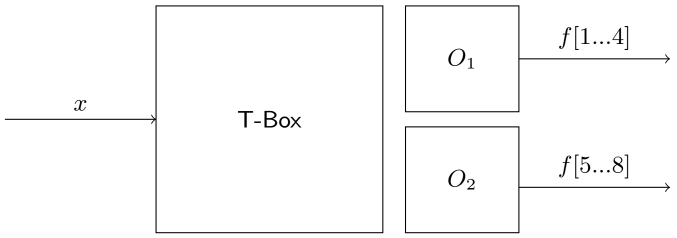


Fig. 13. Non-linear encodings of the T-Box outputs

proved in the theorem below. When collecting the 256 traces of an OT-box to perform a DCA using the correct key hypothesis, each (encoded) nibble value is returned a total of 16 times. Thereby, all encoded nibbles that have the same value are always grouped under the same set A_b in Step 3. Therefore, we always obtain a difference of means curve which consists of only 5 possible correlation values.

Theorem 2. *Given an OT-box which makes use of nibble encodings, the difference of means curve obtained for the correct key hypothesis k^h consists only of values equal to 0, 0.25, 0.5, 0.75 or 1.*

Proof. We first prove that the mean value of the set A_0 is always a fraction of 8 when we sort the sets according to the correct key hypothesis. The same applies for the set A_1 and the proof is analogous. For all $1 \leq j \leq 8$ let $y_d[j]$ be the j^{th} bit of the output y of a T-box, where $d \in \{1, 2\}$ refers to the nibble of y where bit j is located. Let k^h be the correct key hypothesis. With our selection function $\texttt{Sel}(p_e, k^h, j)$ we calculate a total of 128 nibble values y_d, for which $y_d[j] = 0$. As there exist only 8 possible nibble values y_d for which $y_d[j] = 0$ holds, we obtain each value y_d a total of 16 times. Each time we obtain a value y_d, we group its corresponding encoded value f_d under the set A_0. Recall that an OT-box uses one bijective function to encode each nibble y_d. Thus, when we calculate the mean trace $\bar{A}_0$ and focus on its region corresponding to f_d, we do the following:

$$\begin{aligned}\bar{A}_0[f_d] &= \frac{16 f_d}{128} + \cdots + \frac{16 f'_d}{128} \\ &= \frac{f_d}{8} + \cdots + \frac{f'_d}{8},\end{aligned}$$

with $f_d \neq f'_d$. We now prove that the difference between the means of sets A_0 and A_1 is always equal to the values 0, 0.25, 0.5, 0.75 or 1. Let $f_d[j]$ be one bit of an encoded nibble f_d.

- If $f_d[j] = 0$ is true for all nibbles in set A_0, then this implies that $f_d[j] = 1$ is true for all nibbles in set A_1, that is $\bar{A}_0[j] = \frac{8}{8}$ and $\bar{A}_1[j] = \frac{0}{8}$. The difference between the means of both sets is thus $\Delta[j] = |\frac{0}{8} - \frac{8}{8}| = |0 - 1| = 1$.

– If $f_d[j] = 1$ is true for 1 nibble in set A_0, then $f_d[j] = 1$ is true for 7 nibbles in set A_1, that is, the difference between both means is $\Delta[j] = |\frac{1}{8} - \frac{7}{8}| = |\frac{6}{8}| = 0.75$.
– If $f_d[j] = 1$ is true for 2 nibbles in set A_0, then $f_d[j] = 1$ is true for 6 nibbles in set A_1, that is, the difference between both means is $\Delta[j] = |\frac{2}{8} - \frac{6}{8}| = |\frac{4}{8}| = 0.5$.
– If $f_d[j] = 1$ is true for 3 nibbles in set A_0, then $f_d[j] = 1$ is true for 5 nibbles in set A_1, that is, the difference between both means is $\Delta[j] = |\frac{3}{8} - \frac{5}{8}| = |\frac{2}{8}| = 0.25$.
– If $f_d[j] = 1$ is true for 4 nibbles in set A_0, then $f_d[j] = 1$ is true for 4 nibbles in set A_1, that is, the difference between both means is $\Delta[j] = |\frac{4}{8} - \frac{4}{8}| = |\frac{0}{8}| = 0$.

The remaining 4 cases follow analogously and thus, all difference of means traces consist of only the values 0, 0.25, 0.5, 0.75 or 1. □

Seeing these values in a difference of means trace can help us recognize if our key hypothesis is correct. Moreover, a peak value of 0.5, 0.75 or 1 is significantly high, such that its corresponding key candidate will very likely be ranked as the correct key.

We now argue that, when we use an incorrect key candidate, nibbles with the same value may be grouped in different sets. Therefore, we cannot say as for the correct key hypothesis, that each encoded nibble value f_d is repeated exactly 16 times in a set. If we partition according to an incorrect key hypothesis k^h, the value we calculate for $y_d[j]$ does not always match with what is really calculated by the T-box and afterwards encoded by the non-linear function. It is not the case that for each nibble value y_d for which $y_d[j] = 0$, we group its corresponding encoded value f_d in the same set. Therefore, our sets A_b consist of up to 16 different encoded nibbles, whereby each nibble value is repeated a different number of times. This applies for both sets A_0 and A_1 and therefore, both sets have similar mean values, such that the difference between both means is a value closer to zero.

To get practical results corresponding to Theorem 2, we now construct 10 000 different OT-boxes following the idea displayed in Fig. 13. Thereby, each OT-box is based on a different T-box, i.e. each one depends on a different key, and is encoded with a different pair of functions O_1 and O_2. We now perform the DCA attack on each OT-box. The DCA attack is successful on almost all of the 10 000 OT-boxes with the exception of three. In all cases, the difference of means curves obtained when using the correct key hypotheses return a highest peak value of 0.25, 0.5, 0.75 or 1. The table below summarizes how many OT-boxes return each peak value for the correct key hypotheses.

Peak value for correct key	Nr. of OT-boxes
1	55
0.75	2804
0.5	7107
0.25	34

Fig. 14 compares the results obtained for all key candidates when analyzing one particular OT-box. In this case, the correct key is key candidate 119, for which we obtain a peak value of 0.5. The peak values returned for all other key candidates are notably lower.

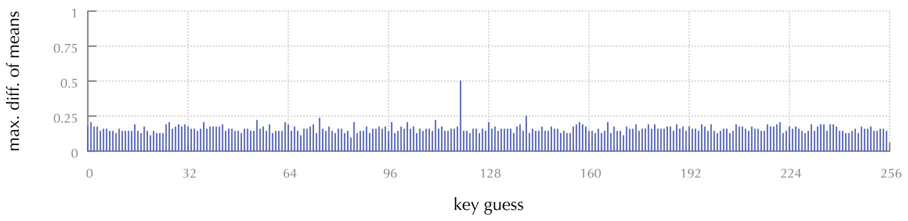


Fig. 14. Difference of means results for all key candidates when attacking one particular OT-box. Key guess 119 is the correct one.

Fig. 15 summarizes the results obtained when analyzing one of the three OT-boxes which could not be successfully attacked. In this case, the correct key is key candidate 191. The peak corresponding to this candidate has a value of 0.25 and is not the highest peak obtained. For instance, the peak for key candidate 89 has a value of 0.28. Therefore, our DCA ranks key candidate 89 as the correct one. Similarly, when analyzing the other OT-boxes which could not be successfully attacked, the peaks obtained for the correct key hypotheses have a value of 0.25 and there exists at least one other key candidate with a peak value slightly higher or with the same value of 0.25.

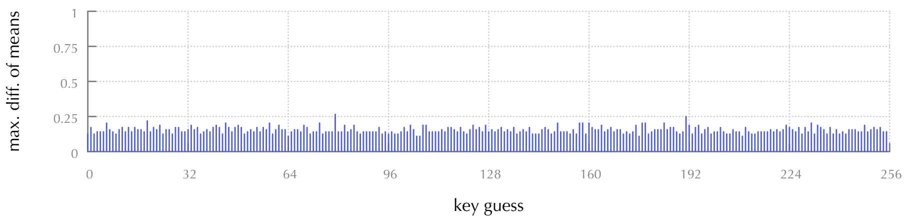


Fig. 15. Difference of means results for all key candidates when attacking one particular OT-box. Key guess 191 is the correct one.

Based on the results shown in this section we can conclude that randomly generated nibble encodings do not effectively work as a countermeasure for hiding correlations between a target bit and a selection function when performing the difference of means test. When using the correct key hypothesis, we do not always have perfect correlations such as those shown in Fig. 5, but the correlations are still high enough in order to allow a key extraction. Moreover, we learn one way to increase our success probabilities when performing the DCA attack. Namely, when ranking the key hypotheses, if no key candidate returns a difference of means curve with a peak value significantly high, we can rank the key candidates according to the convergence of their difference of means peaks to the values 0.25 or 0.

5.3 Combination of Linear and Non-Linear Encodings

We now discuss the effectiveness of the DCA when performed on white-box implementations that make use of both linear and non-linear encodings. The combination of both encodings is the approach proposed by Chow et. al. in order to protect the content of the look-up tables from reverse engineering attempts. The output of each key-dependent look-up table, such as a T-box, is first encoded by a linear transformation and afterwards by the combination of two non-linear functions as shown in Fig. 16. In the following, we refer to lOT-boxes as T-boxes that are encoded using both types of encodings.

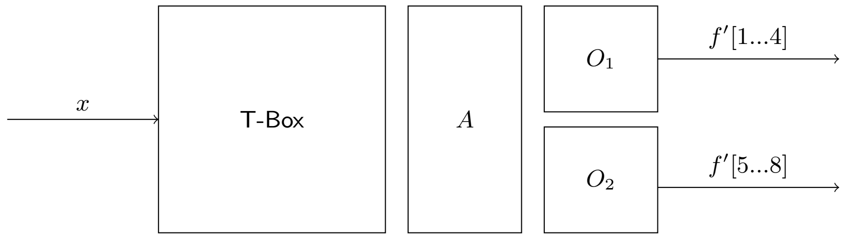


Fig. 16. Linear and non-linear encodings of the T-Box outputs

We now perform the DCA attack on the OpenWhiteBox Chow challenge.[14] This AES implementation was designed based on the work described in [27] and [65]. To perform the attack, we collect 2000 software traces. These software traces consist of values read from *and* written to the stack during the first round. We define our selection function the same way as in Section 3.3, $\texttt{Sel}(p_e, k^h, j) = z[j]$. When we use the correct key byte `0x69` with our selection function, we obtain the difference of means traces shown in Fig. 17 for the target bit $j = 2$.

Fig. 17 shows a flat trace with 7 peaks reaching a value of almost 0.5 (see samples 327, 335, 607, 615, 1007, 1023, 1047). Due to this trace, the key byte `0x69` is ranked as the best key candidate and the DCA attack is successful on this implementation. The peak values shown in Fig. 17 correspond to those described in Theorem 2. We discuss these results based on Theorems 1 and 2, a more detailed explanation of these results can be found in [58].

From Theorem 1 we can conclude the following. If an output bit of a T-box can be transformed into any possible bit value (i.e., 0 or 1) via a linear

[14] https://github.com/OpenWhiteBox/AES/tree/master/constructions/chow

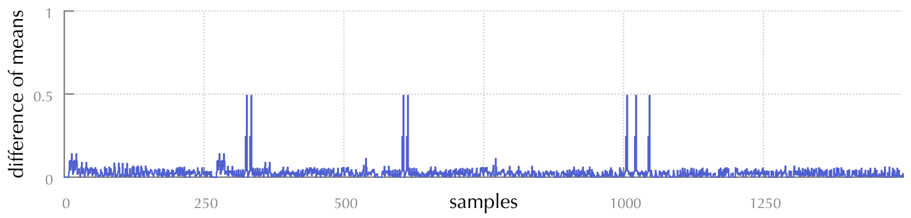


Fig. 17. Difference of means results for the OpenWhiteBox Challenge when using the correct key guess and targeting bit $z[2]$

transformation, then we can achieve resistance against the DCA attack. On the other hand, if an output bit of a T-box can only be transformed into one precise value depending on its original value (i.e. it either always keeps its original value or it is always inverted), then we know that the DCA attack will be successful when targeting that bit.

In Theorem 2, we prove that when a non-linear nibble function only takes 8 possible nibble values as input, this leads us to a successful DCA attack (see Section 5.2). Recall now that in an lO-Tbox, the outputs of a T-box are first encoded via a linear transformation, afterwards split into two nibbles and finally each nibble is encoded by a different non-linear function (see Fig. 16). Consider the case that an output nibble of a T-box can be transformed into any possible value in $GF(2^4)$ (i.e. 0, 1,...,15) via the linear transformation. This would imply that each non-linear nibble function can take any possible nibble value as input and from this we could expect robustness against the DCA. Now consider the case that an output nibble of a T-box can only be transformed into 8 possible values in $GF(2^4)$. This would imply that the non-linear nibble function which encodes that corresponding nibble will only take 8 possible nibble values as input and this would lead us to the same situation as the one described in Theorem 2 and we obtain the same difference of means results as those described in Theorem 2. This last case is the case given in the OpenWhiteBox Chow challenge and for these reasons we obtain a difference of means curve with values equal to those described in Theorem 2.

6 Generalizing the DCA Attack

Throughout this paper, we have implemented the DCA as a software counterpart of the differential power analysis (DPA). We use a selection function which calculates an unencoded intermediate value and use the bits of that value as a distinguisher in a difference of means test. We obtain 256 key candidates and rank them according to the difference of means curves obtained for each candidate, whereby the highest ranked key is the key for which we obtain the difference of means curve with the highest peaks. We have shown how this approach also works for attacking many white-box implementations.

Nevertheless, the results obtained in Sections 5.1 and 5.2 as well as the rankings obtained on the Karroumi white-box when targeting the output of the multiplicative inversion give us hints on to how our DCA attack could be modified in order to make it even more powerful. Jakub Klemsa has initially proposed a similar modification of the DCA attack in Section 3.3.5 of his thesis [45]. We now explain how the results obtained in our paper lead us to a further improvement of the DCA attack. First, in Section 5.1 we show how an invertible matrix masks each bit of an output state. Although an invertible matrix which does not have any identity row may seem like a good masking countermeasure against the traditional DCA attack, we explain that our DCA attack can be easily modified in order to succeed in the presence of any invertible matrix used for linearly encoding the outputs y of a T-box. All we need to do is to modify our selection function such that it calculates all linear combinations of the bits of y, i.e. of the output state of the T-box. Thereby, there will be one linear combination which will be equal to a linear combination defined by one of the rows of the invertible matrix used to encode y. For that case, we will obtain perfect correlations for the correct key hypothesis. Second, in Section 5.2 we prove that if the output of a T-box is encoded via non-linear nibble encodings, whenever we use the correct key hypothesis we will obtain difference of means values equal to 0, 0.25, 0.5, 0.75 or 1. We explain that when attacking a T-box encoded via non-linear nibble encodings, we can increase our success probabilities by modifying the ranking methodology for our key candidates. If none of the key candidates returns a difference of means with peaks high enough that they stand out, we can rank our key candidates according to the convergence of their difference of means peaks to the values 0.25 or 0.

Recall that in a complete white-box implementation, both linear and non-linear encodings are applied to protect intermediate state values. Thereby a state value y is first linearly encoded by an invertible matrix A into the value m. Afterwards, m is split into two nibbles and each nibble is encoded by a different non-linear function (see Fig. 16). We can now use our observations made in the previous sections in order to redefine our DCA attack and make it more powerful in the presence of both, linear and non-linear encodings. The idea is that if we manage to correctly calculate at least one bit of the *linearly* encoded value m and sort our traces according to the value of that bit, then we can assume the linear function to be an identity function and we can expect correlation values equal to those described in Theorem 2. In the following, we refer to the steps performed in Section 3.3.

1. Collecting Traces We perform this step the same way as described in Section 3.3. We remark that when attacking a complete white-box implementation, we need to generate a large enough number of traces in order to increase our success probability.

2. Selection Function We define a selection function $\texttt{Sel}(p_e, k^h, LC)$, which calculates the state z after the $\texttt{SBox}$ substitution in the first round of AES.

Afterwards `Sel` calculates a linear combination LC of the bits in z. The result from the linear combination is thus a bit value equal to 0 or 1, which we will use as a distinguisher in the following steps. LC indicates *which* linear combination we are calculating, with $1 \leq LC \leq 255$ as we can discard the null LC.

$$\texttt{Sel}(p_e, k^h, LC) := LC(\texttt{SBox}(p_e \oplus k^h)) = b \in \{0,1\}. \tag{7}$$

In order to obtain correlation values such as those mentioned in Theorem 2 for the correct key hypothesis, we need to apply a linear combination LC equal to a linear combination defined by any row i of the matrix A used for performing the linear encodings of the `SBox` output state. Therefore, we perform the following Steps 3, 4 and 5 for each linear combination LC. For each key hypothesis we obtain thus 255 difference of means traces.

3. Sorting of Traces We sort each trace s_e into one of the two sets A_0 or A_1 according to the value of $\texttt{Sel}(p_e, k^h, LC) = b$:

$$\text{For } b \in \{0,1\} \ A_b := \{s_e | 1 \leq e \leq n, \ \texttt{Sel}(p_e, k^h, LC) = b\}. \tag{8}$$

Steps 4 and 5. We calculate the mean values of each set A_0 and A_1 the same way as described in Section 3.3.

6. Best linear combination In this step we compare the difference of means traces obtained for all linear combinations LC for a given key hypothesis k^h. If there is a trace with a peak value high enough that it stands out amongst the other traces, then we select that as the best one for the key candidate we are analyzing. Otherwise, we look for a trace with values converging to 0.25 (or 0) and select that one for the key candidate we are analyzing. Let Δ^{LC} be the difference of means trace obtained for the linear combination LC, and let $H(\Delta^{LC})$ be the highest peak in the trace Δ^{LC}. Then, we select Δ^{LC} as the best difference of means trace for k^h, such that $H(\Delta^{LC})$ is maximal amongst the highest peaks of all other difference of means traces, i.e.

$$\forall \ 1 \leq LC' \leq 255, \ H(\Delta^{LC'}) \leq H(\Delta^{LC}).$$

Afterwards, we revise the height of $H(\Delta^{LC})$ before assigning Δ^{LC} to Δ^h:

- If $H(\Delta^{LC}) > 0.3$, then we select Δ^{LC} such that, $\Delta^h := \Delta^{LC}$.
- If $0.2 \leq H(\Delta^{LC}) \leq 0.3$, then we look for a trace with values converging to 0.25. The trace $\Delta^{LC'}$ with the peak values closest to 0.25 is thus selected such that $\Delta^h := \Delta^{LC'}$.
- Otherwise if $H(\Delta^{LC}) < 0.2$, then we look for a trace with values converging to 0. The trace $\Delta^{LC'}$ with the peak values closest to 0 is thus selected such that $\Delta^h := \Delta^{LC'}$.

7 Best Key Byte Hypothesis Analogous to the previous step, we perform Step 7 the same way as in Section 3.3, but if no key candidate stands out, we look for a candidate with the values closest to 0.25 or 0.

- If $H(\Delta^h) > 0.3$, then we select the key hypothesis corresponding to Δ^h as our best key candidate.
- If $0.2 \leq H(\Delta^h) \leq 0.3$, then we look for a trace with values converging to 0.25. The key hypothesis corresponding to the trace $\Delta^{h'}$ with the peak values closest to 0.25 is thus selected as our best key candidate.
- Otherwise if $H(\Delta^h) < 0.2$, then we look for a trace with values converging to 0. The key hypothesis corresponding to the trace $\Delta^{h'}$ with the peak values closest to 0 is thus selected as our best key candidate.

The complexity of the generalized DCA attack compared to the standard DCA increases by almost a factor of 32, from 8 target bits to 255 linear combinations of these bits. But in practice, each of the output bits of the targeted look-up table – possibly comprising the MixColumns operation as in Chow and outputting 32 bits – is a potential source of leakage masked by some linear combination. Therefore, one can limit itself to e.g. 34 linear combinations instead of 255 to still break successfully a key byte with a probability of 99%, making a practical generalized DCA only about four times slower than the standard DCA.

6.1 Countermeasures against DCA

In hardware, countermeasures against DPA typically rely on a random source. Its output can be used to mask intermediate results, to reorder instructions, or to add delays (see e.g. [25,37,76]). For white-box implementations, we cannot rely on a random source since in the white-box attack model such a source can simply be disabled or fixed to a constant value. Despite this lack of *dynamic* entropy, one can assume that the implementation which generates the white-box implementation has access to sufficient random data to incorporate in the generated source code and look-up tables. How to use this *static* random data embedded in the white-box implementation?

Adding (random) delays in an attempt to misalign traces is trivially defeated by using an address instruction trace beside the memory trace to realign traces automatically. In [30] it is proposed to use *variable* encodings when accessing the look-up tables based on the affine equivalences for bijective S-boxes (cf. [19] for algorithms to solve the affine equivalence problem for arbitrary permutations). As a potential countermeasure against DCA, the embedded (and possibly merged with other functionality) static random data is used to select which affine equivalence is used for the encoding when accessing a particular look-up table. This results in a variable encoding (at runtime) instead of using a fixed encoding. Such an approach can be seen as a form of masking as used to thwart classical first-order DPA.

One can also use some ideas from threshold implementations [69]. A threshold implementation is a masking scheme based on secret sharing and multiparty computation. One could also split the input in multiple shares such that not all shares belong to the same affine equivalence class. If this splitting of the shares and assignment to these (different) affine equivalence classes is done pseudo-randomly, where the randomness comes from the static embedded entropy and the input message, then this might offer some resistance against DCA-like attacks.

As mentioned in Section 5.2, the use of byte encodings to protect key dependent look-up tables provides the so-called property of local security. From this property, we can expect robustness against the DCA, but white-box implementations do not use byte encodings because of the increases in code size that they imply. Nevertheless, one could consider applying byte encodings only to the key dependent look-up tables whose outputs are analyzed for performing a DCA. Thereby, only the key dependent look-up tables corresponding to the first and the last round of the implementation could use byte encodings, while all other look-up tables in the implementation could use nibble encodings.

Another potential countermeasure against DCA is the use of external encodings. This was the primary reason why we were not able to extract the secret key from the NoSuchCon 2013 Challenge described in Section 4. However, typically the adversary can obtain knowledge related to the external encoding applied when he observes the behavior of the white-box implementation in the entire software-framework where it is used (especially when the adversary has control over the input parameters used or can observe the final decoded output). We stress that more research is needed to verify the strength of such approaches and improve the ideas presented here.

In practice, one might resort to methods to make the job of the adversary more difficult. Typical software countermeasures include obfuscation, anti-debug and integrity checks. It should be noted, however, that in order to mount a successful DCA attack one does not need to reverse engineer the binary executable. The DBI frameworks are very good at coping with those techniques and even if there are efforts to specifically detect DBI [34,50], DBI becomes stealthier too [44].

7 Differential Fault Analysis on White-box Cryptographic Implementations

Differential Fault Analysis for cryptographic hardware implementations was first introduced in [15]. DFA consists of inducing faults on an implementation and analysing how the induced faults are reflected on the outputs of the implementation. These analyses should help extract secret information from the cryptographic design, such as its secret key. Faults can be injected into hardware crypto-systems by increasing the clock frequency that coordinates the activities of the device (clock glitching). Hereby, the clock frequency is increased during a precise period of the execution, with the goal of hindering some operations from being correctly executed. If some operations are skipped or performed incorrectly during the execution process, then this should lead to an incorrect calculation of the output. The same effect can be achieved by variating the level of the power supply voltage during the execution. Other methods for inducing faults include hitting the internals of the device with light pulses (lasers) or with electromagnetic pulses (see [8] and [7] for a classification of different fault injection techniques according to their complexities and costs).

The general approach for DFA is to execute the cryptographic algorithm on a precise input and induce a fault during the calculation process. The adversary obtains thus a faulty output. The adversary repeats this process, using always the same input and inducing the fault at the same position, and collects therefore several faulty outputs. Afterwards, the adversary executes the same cryptographic algorithm on the same input, but this time without inducing any fault. Hereby, the adversary obtains a correct output. Finally, these outputs are analysed with differential cryptanalysis techniques in order to find key dependencies. DFA has been successfully performed on hardware implementations of symmetric block ciphers [73,32] and on public-key ciphers [11,13].

An attractive alternative to differential fault attacks are fault sensitivity analysis (FSA) attacks [51]. FSA does not consist of analysing the faulty outputs of a cryptographic device, but instead it analyses the *behavior* of the device whenever a fault is induced. Hereby, an adversary induces faults via clock glitching and uses the fact that the frequency increase needed for inducing a fault at a determined clock cycle varies depending on the data that is being processed. In this context, the condition where an adversary can determine that a fault has been induced is referred to as the *fault sensitivity* of the circuit being analysed. Given that the fault sensitivity of a circuit is data dependent (just like the power consumption of a circuit), the adversary can use the fault sensitivity as side channel information and use it for extracting secret information. FSA attacks are especially interesting for cases when the device under attack implements some countermeasure against DFA, such that instead of generating a faulty output, the device just outputs random data whenever a fault is induced [63].

In this section, we focus on DFA attacks on white-box cryptographic implementations. Below, we first explain how a DFA attack as explained in [15] is performed. Going more into detail, we explain how this attack is performed on DES and on AES implementations. Going forward, we explain how we inject faults in white-box cryptographic implementations and conclude this section discussing the results we obtain when performing a DFA attack on publicly available white-box implementations.

7.1 Differential Fault Analysis

We now revisit the general steps required for performing a DFA attack on a cryptographic implementation and then explain how we implement those steps for attacking implementations of DES and AES.

First step. The adversary executes the target cryptographic algorithm with the same input multiple times and records correct and faulty outputs. In order to generate faulty outputs, the adversary usually introduces faults towards the end of the execution of the algorithm (e.g. before the final round of a block cipher).

Second step. The adversary now builds a model of the induced faults, and performs an analysis of the collected outputs (correct and faulty) in order to determine the secret key used.

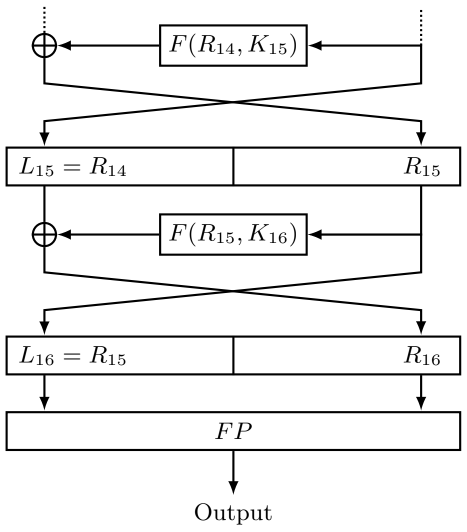


Fig. 18. The final rounds of the DES cipher.

The location within the cryptographic algorithm in which faults are injected in Step 1 and the analysis performed in Step 2 above are closely related. These depend on the cryptographic algorithm under attack. Below, we recall the differential fault attacks on triple DES and AES. Hereby, we denote as K_i the round key used in the calculation process of round i of the cipher we are describing. L_i and R_i denote the left and right half of the output of round i respectively.

DES DFA. Fig. 18 describes the final rounds of execution of a triple DES encryption (or decryption), where the *Feistel function* F is defined as shown in Fig. 19. Each round of execution uses a 48-bit round key. The attack works by recovering one round key at a time, until the complete key can be computed.

In order to recover the last round key (K_{16}), the attacker can inject faults during the execution of round 15, i.e. during the computation of R_{15}. Considering the correct outputs, i.e. outputs collected in the cases when we do not induce any faults, we have the following equation:

$$\begin{aligned} L_{16} &= R_{15} \\ R_{16} &= F(R_{15}, K_{16}) \oplus L_{15} \\ \text{Output} &= FP(L_{16} \| R_{16}). \end{aligned} \tag{9}$$

Correspondingly, considering the outputs collected in the cases when we induce faults, we have the following equation:

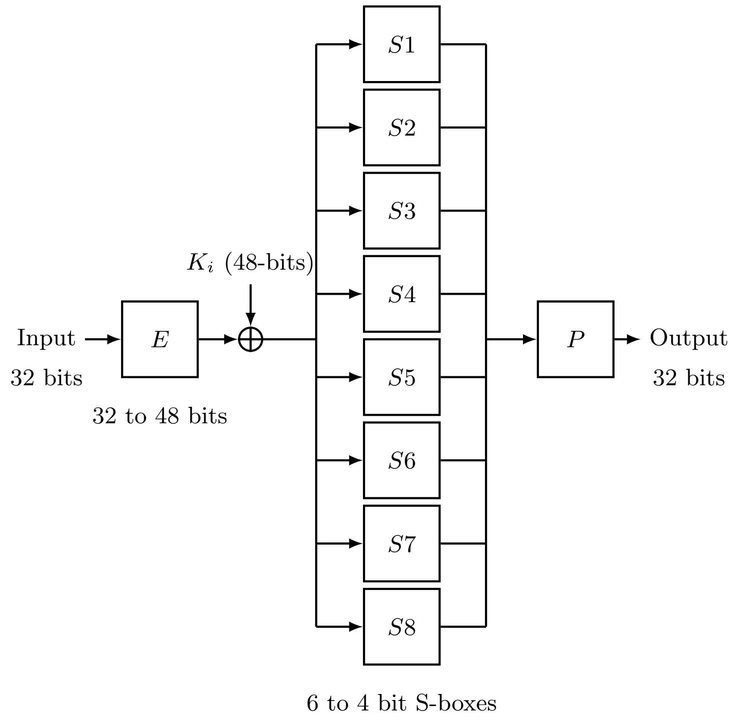


Fig. 19. The Feistel function (F) within the DES cipher

$$\begin{aligned} L'_{16} &= R'_{15} \\ R'_{16} &= F(R'_{15}, K_{16}) \oplus L_{15} \\ \text{Output}' &= FP(L'_{16} \| R'_{16}). \end{aligned} \tag{10}$$

In both expressions, the values of K_{16} and L_{15} are unknown. Combining equations 9 and 10, we obtain the following equation, in which the only unknown variable is the round key K_{16}:

$$R_{16} \oplus R'_{16} = F(R_{15}, K_{16}) \oplus F(R'_{15}, K_{16}).$$

From the F function in Fig. 19 we can see that DES uses 6 bits of the round key K_i to compute 4 output bits in each round. This makes it possible to solve the equation above in blocks of 6 bits of the round key. In particular, for each individual S-Box S_j the following equation needs to be solved:

$$(P^{-1}(R_{16} \oplus R'_{16}))_j = S_j(E(R_{15}) \oplus K_{16,j}) \oplus S_j(E(R'_{15}) \oplus K_{16,j}),$$

where the index j denotes the number of the S-Box and $K_{16,j}$ denotes the corresponding 6 bit block of K_{16} present in the input for that S-Box. E and P

represent the expansion and permutation steps of the F function, respectively. Note that this equation can be solved by exhaustive search.

Typically this results in a number of candidates for each affected sub-key per fault. Therefore, an attacker needs to iterate this process until only one candidate remains for each sub-key. However, in some cases, when the faults are not injected in the exact way as expected by the attack it may happen that a correct key gets discarded. Therefore, a counting strategy is used instead of discarding key candidates: for each fault, we compute the set of solutions to the equation above and increase the count for the respective candidates. Once all faults are analyzed, the candidate with the highest count for each sub-key is selected as the correct candidate. Once the last round key is known, the attack can be iterated to the previous round key. For this, the attacker injects faults one round earlier and computes the output of the one but last round by using the recovered last round key. If a triple DES cipher is being used, the same attack can be applied to the middle DES once the final DES key is known. Finally, if a triple DES cipher with three keys is used, the attack can be iterated to the initial DES.

AES DFA. Several DFA attacks have been published for the AES cipher. Faulting a recent smartcard is difficult, with a high risk to cause the chip to self-destruct if it detects an attack; that is why recent DFA research tries to minimize the requirements on the number of faults [43,83]. However as we explain later, in the white-box attack model faults are very easy and cheap to perform. Therefore applying the DFA attack which was introduced in [33] in 2002 is probably one of the best strategies in our white-box context. We thus describe this attack strategy in this section. We first describe the attack on AES-128 encryption, and later discuss how to extend it to AES-192 and AES-256. We focus on attacking and analysing the last two rounds of AES. Note however that as explained in [71], faults can also be induced one round earlier in case the last two rounds are protected against fault injection.

Recall now that for AES-128, the last but one round of the encryption process (round 9) consists of the following operations: **SubBytes**, **ShiftRows**, **MixColumns**, **AddRoundKey (K_9)**. Recall also that the last round (round 10) consists only of the following operations **SubBytes**, **ShiftRows**, and **AddRoundKey (K_{10})**, where these operations are defined as follows:

– **SubBytes**: Each byte in the AES state is transformed by applying the AES S-Box.
– **ShiftRows**: The rows in the AES state are shifted to the left by 0, 1, 2 or 3 cells (row 1 to 4).
– **MixColumns**: A matrix multiplication is applied, which results in applying a 32-bit transformation to each column of the state.
– **AddRoundKey**: A round key value is bytewise added to the current state by using an exclusive-or operation.

We explain now how a one-byte fault introduced before the MixColumns in round 9 propagates to the output of the cipher. Consider the matrix on the left, where each letter from A to P corresponds to one byte. If the first byte of the state is altered from A to X, we obtain the matrix on the right.

$$\begin{pmatrix} A & E & I & M \\ B & F & J & N \\ C & G & K & O \\ D & H & L & P \end{pmatrix} \longrightarrow \begin{pmatrix} X & E & I & M \\ B & F & J & N \\ C & G & K & O \\ D & H & L & P \end{pmatrix}$$

In this case, the faulty byte X will propagate to the complete first column after the MixColumns operation as we show below.

$$\begin{pmatrix} 2A+3B+C+D & \cdots & \cdots & \cdots \\ A+2B+3C+D & \cdots & \cdots & \cdots \\ A+B+2C+3D & \cdots & \cdots & \cdots \\ 3A+B+C+2D & \cdots & \cdots & \cdots \end{pmatrix} \longrightarrow \begin{pmatrix} 2X+3B+C+D & \cdots & \cdots & \cdots \\ X+2B+3C+D & \cdots & \cdots & \cdots \\ X+B+2C+3D & \cdots & \cdots & \cdots \\ 3X+B+C+2D & \cdots & \cdots & \cdots \end{pmatrix}$$

The remaining steps after the last MixColumns operations are performed byte-wise. For this reason, one faulty byte before the last MixColumns operation propagates into 4 faulty bytes at the output as we show below. Hereby, we denote as $K_{i,j}$ the 8-bit long sub-key of the round key K_i. Note that $K_{10,j}$ is used for calculating the corresponding byte j in the output state.

$$\begin{pmatrix} S(2X+3B+C+D+K_{9,0})+K_{10,0} & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & S(X+2B+3C+D+K_{9,1})+K_{10,13} \\ \cdots & \cdots & S(X+B+2C+3D+K_{9,2})+K_{10,10} & \cdots \\ \cdots & S(3X+B+C+2D+K_{9,3})+K_{10,7} & \cdots & \cdots \end{pmatrix}$$

Consider now the first byte $j = 0$ of the output state. We can write the following expressions, where O_0 corresponds to a correct, non faulty output and O'_0 corresponds to a faulty output:

$$S(2A+3B+C+D+K_{9,0})+K_{10,0} = O_0$$
$$S(2X+3B+C+D+K_{9,0})+K_{10,0} = O'_0.$$

By xor-ing both equations we obtain:

$$S(2A+3B+C+D+K_{9,0}) \oplus S(2X+3B+C+D+K_{9,0}) = O_0 \oplus O'_0$$

Similar expressions can be written for each of the faulty bytes, obtaining a set of 4 related equations. Just like in the attack for DES described above, the attack on AES involves solving these equations to obtain a subset of candidates for parts of the round key. In this case, one fault provides potential candidates for 4 bytes of the key. Repeating this process with different faults allows finding unique values for each sub-key. Repeating this process for the other columns of the AES state allows recovering candidates for all the sub-keys, and therefore leads to the last round key. For AES-128, the last round key K_{10} is enough to perform a reverse key schedule operation and retrieve the original AES key. As mentioned earlier, the detailed description of this attack can be found in [33]. A variant exists for AES decryption.

Note that for AES-192 and AES-256, two round keys need to be recovered in order to compute the full AES key. To do this, we can first recover the last round key K_n as when attacking AES-128. Afterwards, we can recover the round key used in the last but one round (round $n-1$) via the following operations. Since we know the output of the last round and the value of the last round key, we can rewind the operations performed in the last round and thus calculate the input value of the last round. The input of the last round is equal to the output of round $n-1$. Now, we can perform our analysis based on the operations performed during round $n-1$ and the output value of that round. Recall however that unlike the last round, round $n-1$ consists also of a $MixColumns$ operation. In order to simplify our analysis, we can adapt the succession of operations in round $n-1$. Hereby, our goal is to obtain a sequence of operations equal to the sequence of operations of the final round. For this purpose, we can swap the $MixColumns$ and $AddRoundKey(K_{n-1})$ steps, and invert the $MixColumns$ operation. We obtain thus a sequence of operations consisting of $SubBytes$, $ShiftRows$ and $AddRoundKey(InvMixColumns(K_{n-1}))$. Once the equivalent round key $InvMixColumns(K_{n-1})$ is recovered, we apply the $MixColumns$ operation to recover K_{n-1}.

7.2 Fault Injection for White-Box Cryptography

In order to perform DFA attacks on a white-box cryptographic implementation, one fundamental requirement needs to be satisfied: the output of the implementation needs to be available in a non-encoded form. This requirement is satisfied whenever the white-box implementation under attack does not make use of external encodings to protect the output of the algorithm implemented (see **External encodings** in Section 2).

Once the output of the white-box implementation is available in a non-encoded form, the DFA attack requires the ability to inject faults into the cryptographic process at the right locations within the algorithm. In order to locate the appropriate location in which faults need to be injected, we typically combine static and dynamic code analysis. For example, recording execution and memory access traces and visualizing them can highlight the location of the target cryptographic algorithm and its rounds, as explained in Section 3.2. In Fig. 1 for instance, we can clearly see the execution of 16 rounds, and therefore we are able to determine at which time and in which memory region faults need to be introduced.

This reverse-engineering step can even be skipped by following another strategy. Namely, we can randomly inject faults during the computation and observe the output of the cryptographic algorithm. The structure of the output value of the algorithm can thus help to determine if we have induced the fault at the correct position. For example, injecting bit faults in early rounds of a DES execution will result in a fully randomized output. On the other hand, injecting bit faults during the computation of the last round (i.e. too late for the attack described in Section 7.1) will result in changes to the left half, but not the right half of the output. Similarly, for the AES cipher attack described in Section 7.1, injecting one-byte faults anywhere between the last two *MixColumns* steps will result in 4 corrupted bytes. The location of the 4 corrupted bytes must follow a specific pattern, as indicated by the *MixColumns* and *ShiftRows* combination. Finally, injecting faults can be as simple as flipping bits of the DES or AES intermediate results during the execution of the algorithms. To this end, we can use a number of different techniques:

– If the code can be easily lifted to a high-level language (e.g. C/C++), we can introduce code to inject random faults during the target computation. However, in some situations (i.e. with highly obfuscated code) this is a very labor-intensive and error-prone task.
– If the code can be run under a DBI framework (PIN, Valgrind, etc.) we can instrument the code to inject faults and collect the output data.
– A scriptable debugger (e.g. vtrace, gdb) can also be used. To this end, we can write debugger scripts to automate the execution of the cipher and the injection of faults and finally collect the output data.
– Alternatively, emulator-based setups can be used. This allows an attacker to monitor the complete execution and alter the code or data at any desired time. For example, the white-box program code can be extracted at runtime and loaded into a standalone emulator such as the Unicorn Engine[15], or a full system emulator such as the PANDA[16] framework could be used.
– In case there is no or only weak self-integrity checks, the tables or even the binary itself could be corrupted to introduce faults statically. A divide and conquer approach can be used by faulting initially large parts of the tables to detect which parts are effectively used when processing a specific input.

Once a way to inject faults is implemented, performing the attack is just a matter of collecting enough faulty outputs, filter out the non-exploitable ones and plugging the outputs with a good fault pattern into the appropriate DFA algorithm.

Presence of internal encodings. Note that unlike as with the DCA attack, the presence of internal encodings in white-box implementations does not affect the results obtained when performing a traditional DFA attack. For instance, the model of the DFA attack on AES described in Section 7.1 assumes a one-byte fault introduced before the MixColumns in round 9. When inserting a fault in table-based white-box implementations, we are actually injecting a fault in the encoded input of a look up table. As explained in Section 5, the encodings applied to most of the white-boxes analyzed in this paper are either nibble- or byte- encodings. For this reason, we know that a faulty encoded input byte will affect exactly one byte of the internal AES state. Therefore the general DFA hypothesis described in Section 7.1 still holds and we can apply the formulas shown throughout that section for performing a DFA attack.

[15] `http://www.unicorn-engine.org/`
[16] Platform for Architecture-Neutral Dynamic Analysis, `https://github.com/panda-re/panda`

7.3 Analyzing Publicly Available White-Box Implementations

We now explain how we successfully perform a DFA attack on publicly available white-box implementations. For more details on the white-box implementations considered in this section we refer the reader to Section 4.

The Wyseur Challenge. For the Wyseur challenge, we simply load the ELF binary into a custom emulator script based on the Unicorn Engine. Our custom emulator records every read and write to memory, and allows to inject faults at selected moments. This approach is fairly trivial for this challenge because the provided binary offers very little functionality and does not interface with many system libraries. The only call to external libraries performed by the binary is used to print the plaintext and ciphertext, which we can simply patch out. We use the execution graph shown in Figure 7a in Section 4 to select the desired interval for the fault injection. We then implement the following process in our emulator:

1. We select a random number within the target interval. We call this the target index.
2. At each memory access, we increment a counter. We then flip a random bit of the first memory read that occurs after the selected target index.
3. At the end of the execution, we record the faulty ciphertext.
4. After enough faults are collected, we run the DFA algorithm described in 7.1 to recover the DES key.

Following this process,we are able to retrieve the correct key after injecting 40 faults.

The Hack.lu 2009 Challenge. For this challenge, we follow a similar approach as for the Wyseur challenge. The only difference here is that we isolate the white-box code and emulate only the AES function in order to prevent any interaction with the Windows OS. An alternative solution would be to use Intel PIN, or to inject faults in the data section of the binary since no integrity checks are performed. Once the white-box implementation is isolated, the rest of the fault injection process is identical to the Wyseur challenge. The final step is to apply an AES DFA, as described in Section 7.1. With this attack, we were able to recover the AES key after injecting 90 faults.

The SSTIC 2012 Challenge. As described in Section 4, the SSTIC challenge was provided as a compiled Python program. We used uncompyle2 to lift and recover the original Python code and inject faults into it. The DFA algorithm used was the one presented in Section 7.1, which allowed us to recover the DES key after injecting 60 faults.

A White-Box Implementation of the Karroumi Approach. Since the source code of this implementation is available, we simply injected faults in the original C++ code. We then collected the results and ran the AES DFA attack described in Section 7.1. This allows us to recover the complete AES key after injecting 80 faults.

The NoSuchCon 2013 Challenge. As explained in 4, the NSC challenge binary incorporates unknown external encodings. For this reason, the effects of the injected faults cannot be easily observed in the output of the white-box. Therefore, the DFA attack described in this paper cannot be directly applied without first recovering the output encoding applied by the implementation. If the output encoding is known or not present, it is possible to retrieve the key in a similar way to the other AES implementations.

7.4 Countermeasures against DFA

Countermeasures against DFA attacks usually involve some sort of redundant computation. For example, a device could perform the encryption process twice and compare both outputs. For this purpose devices implement concurrent error detection (CED) schemes, which compare the results obtained from an algorithm on one input with the results obtained form a *predictor* of the same algorithm on the same input. CED schemes can also be implemented to compare the outputs of single operations performed within an algorithm and thus achieve an even better fault detection coverage [2]. Note that with these countermeasures, one assumes that the adversary does not have access to the intermediate data and if a fault is detected, we can prevent the faulty value from being outputed. However, in the white-box settings this direct approach is not valid: if the attacker is able to observe the comparison of the two results, he can simply duplicate the faulty result and bypass the check. In order to protect a white-box cryptographic implementation against DFA, the following two avenues might be used:

– Carrying redundant data (e.g. error-correcting codes) along with the computation in such a way that a modification performed by an attacker can be compensated, without affecting the data in the non-encoded domain.
– Implementing the internal data encodings in such a way that faults propagate into larger parts of the cipher state. In this way, the fault models expected by the standard DFA attacks do not apply and therefore an attacker would have to develop customized attacks.

8 Conclusions and Future Work

As conjectured in the first papers introducing the white-box attack model, one cannot expect long-term defense against attacks on white-box implementations. However, as we have shown in this work, all current publicly available white-box implementations do not even offer any short-term security since the DCA and DFA techniques can extract the secret key within seconds.

Although we sketched some ideas on countermeasures, it remains an open question how to guard oneself against these types of attacks. The countermeasures against differential power analysis attacks applied in the area of high-assurance applications do not seem to carry over directly due to the ability of the adversary to disable or tamper with the random source. If medium- to long-term security is required then tamper resistant hardware solutions, like a secure element, seem like a much better alternative.

Similarly, many publicly known differential fault analysis countermeasures are based on introducing redundancy in the computation. If this is not done carefully, an attacker might still be able to inject faults while bypassing the redundancy checks. Additional research on improving the robustness of white-box implementations against fault attacks is thus required.

Another interesting research direction is to see if the more advanced and powerful techniques used in side-channel analysis from the cryptographic hardware community obtain even better results in this setting. Examples include correlation power analysis, higher order analysis and ideas related to fault sensitivity analysis attacks.

References

1. Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, Nov. 2001.
2. A. Aghaie, A. Moradi, S. Rasoolzadeh, F. Schellenberg, and T. Schneider. Impeccable circuits. Cryptology ePrint Archive, Report 2018/203, 2018. `https://eprint.iacr.org/2018/203.pdf`.
3. B. Amstadt and M. K. Johnson. Wine. *Linux Journal*, 1994(4), August 1994.
4. C. H. Baek, J. H. Cheon, and H. Hong. Analytic toolbox for white-box implementations: Limitation and perspectives. Cryptology ePrint Archive, Report 2014/688, 2014. `http://eprint.iacr.org/2014/688`.
5. B. Barak, S. Garg, Y. T. Kalai, O. Paneth, and A. Sahai. Protecting obfuscation against algebraic attacks. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 221–238, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.
6. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In J. Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18, Santa Barbara, CA, USA, Aug. 19–23, 2001. Springer, Heidelberg, Germany.
7. A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi. Injection technologies for fault attacks on microprocessors. In Joye and Tunstall [41], pages 275–293.
8. A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. In *Proceedings of the IEEE*, volume 100, pages 3056–3076. IEEE, 2012.
9. J.-B. Bédrune. Hack.lu 2009 reverse challenge 1. online, 2009. `http://2009.hack.lu/index.php/ReverseChallenge`.
10. F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
11. A. Berzati, C. Canovas-Dumas, and L. Goubin. A survey of differential fault analysis against classical RSA implementations. In Joye and Tunstall [41], pages 111–124.
12. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium.* USENIX Association, 2003.

13. I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In M. Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 131–146, Santa Barbara, CA, USA, Aug. 20–24, 2000. Springer, Heidelberg, Germany.
14. E. Biham and A. Shamir. Differential cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer. In J. Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 156–171, Santa Barbara, CA, USA, Aug. 11–15, 1992. Springer, Heidelberg, Germany.
15. E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In B. S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 513–525, Santa Barbara, CA, USA, Aug. 17–21, 1997. Springer, Heidelberg, Germany.
16. O. Billet and H. Gilbert. A traceable block cipher. In C.-S. Laih, editor, *ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 331–346. Springer, Heidelberg, Germany, 2003.
17. O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a white box AES implementation. In H. Handschuh and A. Hasan, editors, *SAC 2004*, volume 3357 of *LNCS*, pages 227–240, Waterloo, Ontario, Canada, Aug. 9–10, 2004. Springer, Heidelberg, Germany.
18. A. Biryukov, C. Bouillaguet, and D. Khovratovich. Cryptographic schemes based on the ASASA structure: Black-box, white-box, and public-key (extended abstract). In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 63–84, Kaoshiung, Taiwan, R.O.C., Dec. 7–11, 2014. Springer, Heidelberg, Germany.
19. A. Biryukov, C. De Canniére, A. Braeken, and B. Preneel. A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In E. Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 33–50, Warsaw, Poland, May 4–8, 2003. Springer, Heidelberg, Germany.
20. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 37–51, Konstanz, Germany, May 11–15, 1997. Springer, Heidelberg, Germany.
21. Z. Brakerski and G. N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In Y. Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 1–25, San Diego, CA, USA, Feb. 24–26, 2014. Springer, Heidelberg, Germany.
22. C.-B. Breunesse, I. Kizhvatov, R. Muijrers, and A. Spruyt. Towards fully automated analysis of whiteboxes: Perfect dimensionality reduction for perfect leakage. Cryptology ePrint Archive, Report 2018/095, 2018. `http://eprint.iacr.org/`.
23. E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In M. Joye and J.-J. Quisquater, editors, *CHES 2004*, volume 3156 of *LNCS*, pages 16–29, Cambridge, Massachusetts, USA, Aug. 11–13, 2004. Springer, Heidelberg, Germany.
24. J. Bringer, H. Chabanne, and E. Dottax. White box cryptography: Another attempt. Cryptology ePrint Archive, Report 2006/468, 2006. `http://eprint.iacr.org/2006/468`.
25. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412, Santa Barbara, CA, USA, Aug. 15–19, 1999. Springer, Heidelberg, Germany.

26. S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In B. S. Kaliski Jr., Çetin Kaya. Koç, and C. Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 13–28, Redwood Shores, CA, USA, Aug. 13–15, 2003. Springer, Heidelberg, Germany.
27. S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. White-box cryptography and an AES implementation. In K. Nyberg and H. M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 250–270, St. John's, Newfoundland, Canada, Aug. 15–16, 2003. Springer, Heidelberg, Germany.
28. S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. A white-box DES implementation for DRM applications. In J. Feigenbaum, editor, *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002*, volume 2696 of *LNCS*, pages 1–15. Springer, 2003.
29. J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer, 2002.
30. Y. de Mulder. *White-Box Cryptography: Analysis of White-Box AES Implementations*. PhD thesis, KU Leuven, 2014.
31. C. Delerablée, T. Lepoint, P. Paillier, and M. Rivain. White-box security notions for symmetric encryption schemes. In T. Lange, K. Lauter, and P. Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 247–264, Burnaby, BC, Canada, Aug. 14–16, 2014. Springer, Heidelberg, Germany.
32. P. Dusart, G. Letourneux, and O. Vivolo. Differential fault analysis on AES. In J. Zhou, M. Yung, and Y. Han, editors, *ACNS 03*, volume 2846 of *LNCS*, pages 293–306, Kunming, China, Oct. 16–19, 2003. Springer, Heidelberg, Germany.
33. P. Dusart, G. Letourneux, and O. Vivolo. Differential fault analysis on A.E.S. In J. Zhou, M. Yung, and Y. Han, editors, *ACNS 2003*, volume 2846 of *Lecture Notes in Computer Science*, pages 293–306. Springer, 2003.
34. F. Falco and N. Riva. Dynamic binary instrumentation frameworks: I know you're there spying on me. REcon, 2012. `http://recon.cx/2012/schedule/events/216.en.html`.
35. S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 40–49. IEEE Computer Society, 2013.
36. L. Goubin, J.-M. Masereel, and M. Quisquater. Cryptanalysis of white box DES implementations. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 278–295, Ottawa, Canada, Aug. 16–17, 2007. Springer, Heidelberg, Germany.
37. L. Goubin and J. Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya. Koç and C. Paar, editors, *CHES'99*, volume 1717 of *LNCS*, pages 158–172, Worcester, Massachusetts, USA, Aug. 12–13, 1999. Springer, Heidelberg, Germany.
38. Y. Huang, F. S. Ho, H. Tsai, and H. M. Kao. A control flow obfuscation method to discourage malicious tampering of software codes. In F. Lin, D. Lee, B. P. Lin, S. Shieh, and S. Jajodia, editors, *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006*, page 362. ACM, 2006.
39. M. Jacob, D. Boneh, and E. W. Felten. Attacking an obfuscated cipher by injecting faults. In J. Feigenbaum, editor, *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002, Washington, DC, USA, November 18, 2002, Revised Papers*, volume 2696 of *LNCS*, pages 16–31. Springer, 2003.

40. M. Jakobsson and M. K. Reiter. Discouraging software piracy using software aging. In T. Sander, editor, *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001*, volume 2320 of *LNCS*, pages 1–12. Springer, 2002.
41. M. Joye and M. Tunstall, editors. *Fault Analysis in Cryptography*. ISC. Springer, Heidelberg, Germany, 2012.
42. M. Karroumi. Protecting white-box AES with dual ciphers. In K. H. Rhee and D. Nyang, editors, *ICISC 10*, volume 6829 of *LNCS*, pages 278–291, Seoul, Korea, Dec. 1–3, 2011. Springer, Heidelberg, Germany.
43. C. H. Kim and J. Quisquater. New differential fault analysis on AES key schedule: Two faults are enough. In G. Grimaud and F. Standaert, editors, *CARDIS 2008*, volume 5189 of *Lecture Notes in Computer Science*, pages 48–60. Springer, 2008.
44. J. Kirsch. Towards transparent dynamic binary instrumentation using virtual machine introspection. REcon, 2015. `https://recon.cx/2015/schedule/events/20.html`.
45. J. Klemsa. *Side-Channel Attack Analysis of AES White-Box Schemes*. PhD thesis, Czech Technical University in Prague, 2016.
46. D. Klinec. White-box attack resistant cryptography. Master's thesis, Masaryk University, Brno, Czech Republic, 2013. `https://is.muni.cz/th/325219/fi_m/`.
47. P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.
48. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397, Santa Barbara, CA, USA, Aug. 15–19, 1999. Springer, Heidelberg, Germany.
49. T. Lepoint, M. Rivain, Y. D. Mulder, P. Roelse, and B. Preneel. Two attacks on a white-box AES implementation. In T. Lange, K. Lauter, and P. Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 265–285, Burnaby, BC, Canada, Aug. 14–16, 2014. Springer, Heidelberg, Germany.
50. X. Li and K. Li. Defeating the transparency features of dynamic binary instrumentation. BlackHat US, 2014. `https://www.blackhat.com/docs/us-14/materials/us-14-Li-Defeating-The-Transparency-Feature-Of-DBI.pdf`.
51. Y. Li, K. Sakiyama, S. Gomisawa, T. Fukunaga, J. Takahashi, and K. Ohta. Fault sensitivity analysis. In S. Mangard and F.-X. Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 320–334, Santa Barbara, CA, USA, Aug. 17–20, 2010. Springer, Heidelberg, Germany.
52. H. E. Link and W. D. Neumann. Clarifying obfuscation: Improving the security of white-box DES. In *International Symposium on Information Technology: Coding and Computing (ITCC 2005)*, pages 679–684. IEEE Computer Society, 2005.
53. C. Linn and S. K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003*, pages 290–299. ACM, 2003.
54. C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200. ACM, 2005.
55. A. Maillet. Nosuchcon 2013 challenge - write up and methodology. online, 2013. `http://kutioo.blogspot.be/2013/05/nosuchcon-2013-challenge-write-up-and.html`.

56. S. Mangard, E. Oswald, and F. Standaert. One for all - all for one: unifying standard differential power analysis attacks. *IET Information Security*, 5(2):100–110, 2011.
57. F. Marceau, F. Perigaud, and A. Tillequin. Challenge SSTIC 2012. online, 2012. `http://communaute.sstic.org/ChallengeSSTIC2012`.
58. E. Alpirez Bock, C. Brzuska, W. Michiels, and A. Treff. On the ineffectiveness of internal encodings - revisiting the dca attack on white-box cryptography. Cryptology ePrint Archive, Report 2018/301, 2018. `https://eprint.iacr.org/2018/301.pdf`.
59. T. S. Messerges. Using second-order power analysis to attack DPA resistant software. In Çetin Kaya. Koç and C. Paar, editors, *CHES 2000*, volume 1965 of *LNCS*, pages 238–251, Worcester, Massachusetts, USA, Aug. 17–18, 2000. Springer, Heidelberg, Germany.
60. W. Michiels. Opportunities in white-box cryptography. *IEEE Security & Privacy*, 8(1):64–67, 2010.
61. W. Michiels and P. Gorissen. Mechanism for software tamper resistance: an application of white-box cryptography. In M. Yung, A. Kiayias, and A. Sadeghi, editors, *Proceedings of the Seventh ACM Workshop on Digital Rights Management*, pages 82–89. ACM, 2007.
62. W. Michiels, P. Gorissen, and H. D. L. Hollmann. Cryptanalysis of a generic class of white-box implementations. In R. M. Avanzi, L. Keliher, and F. Sica, editors, *SAC 2008*, volume 5381 of *LNCS*, pages 414–428, Sackville, New Brunswick, Canada, Aug. 14–15, 2009. Springer, Heidelberg, Germany.
63. A. Moradi, O. Mischke, C. Paar, Y. Li, K. Ohta, and K. Sakiyama. On the power of fault sensitivity analysis and collision side-channel attacks in a combined setting. In B. Preneel and T. Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 292–311, Nara, Japan, Sept. 28 – Oct. 1, 2011. Springer, Heidelberg, Germany.
64. C. Mougey and F. Gabriel. Désobfuscation de DRM par attaques auxiliaires. In *Symposium sur la sécurité des technologies de l'information et des communications*, 2014. `www.sstic.org/2014/presentation/dsobfuscation_de_drm_par_attaques_auxiliaires`.
65. J. A. Muir. A tutorial on white-box AES. In E. Kranakis, editor, *Advances in Network Analysis and its Applications*, volume 18 of *Mathematics in Industry*, pages 209–229. Springer Berlin Heidelberg, 2013.
66. Y. D. Mulder, P. Roelse, and B. Preneel. Cryptanalysis of the Xiao-Lai white-box AES implementation. In L. R. Knudsen and H. Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 34–49, Windsor, Ontario, Canada, Aug. 15–16, 2013. Springer, Heidelberg, Germany.
67. Y. D. Mulder, B. Wyseur, and B. Preneel. Cryptanalysis of a perturbated white-box AES implementation. In G. Gong and K. C. Gupta, editors, *INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 292–310, Hyderabad, India, Dec. 12–15, 2010. Springer, Heidelberg, Germany.
68. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007.
69. S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against side-channel attacks and glitches. In P. Ning, S. Qing, and N. Li, editors, *Information and Communications Security, ICICS*, volume 4307 of *LNCS*, pages 529–545. Springer, 2006.

70. J. Patarin and L. Goubin. Asymmetric cryptography with S-boxes. In Y. Han, T. Okamoto, and S. Qing, editors, *ICICS 97*, volume 1334 of *LNCS*, pages 369–380, Beijing, China, Nov. 11–14, 1997. Springer, Heidelberg, Germany.
71. 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, Çetin Kaya. Koç, and C. Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, pages 77–88, Cologne, Germany, Sept. 8–10, 2003. Springer, Heidelberg, Germany.
72. M. L. Polla, F. Martinelli, and D. Sgandurra. A survey on security for mobile devices. *IEEE Communications Surveys and Tutorials*, 15(1):446–471, 2013.
73. M. Rivain. Differential fault analysis of DES. In Joye and Tunstall [41], pages 37–54.
74. P. Sasdrich, A. Moradi, and T. Güneysu. White-box cryptography in the gray box - - A hardware implementation and its side channels -. In T. Peyrin, editor, *FSE 2016*, volume 9783 of *LNCS*, pages 185–203, Bochum, Germany, Mar. 20–23, 2016. Springer, Heidelberg, Germany.
75. A. Saxena, B. Wyseur, and B. Preneel. Towards security notions for white-box cryptography. In P. Samarati, M. Yung, F. Martinelli, and C. A. Ardagna, editors, *ISC 2009*, volume 5735 of *LNCS*, pages 49–58, Pisa, Italy, Sept. 7–9, 2009. Springer, Heidelberg, Germany.
76. K. Schramm and C. Paar. Higher order masking of the AES. In D. Pointcheval, editor, *CT-RSA 2006*, volume 3860 of *LNCS*, pages 208–225, San Jose, CA, USA, Feb. 13–17, 2006. Springer, Heidelberg, Germany.
77. F. Scrinzi. Behavioral analysis of obfuscated code. Master's thesis, University of Twente, Twente, Netherlands, 2015. `http://essay.utwente.nl/67522/1/Scrinzi_MA_SCS.pdf`.
78. A. Souchet. AES whitebox unboxing: No such problem. online, 2013. `http://0vercl0k.tuxfamily.org/bl0g/?p=253`.
79. SysK. Practical cracking of white-box implementations. Phrack 68:14. `http://www.phrack.org/issues/68/8.html`.
80. P. Teuwen. CHES2015 writeup. online, 2015. `http://wiki.yobi.be/wiki/CHES2015_Writeup#Challenge_4`.
81. P. Teuwen. NSC writeups. online, 2015. `http://wiki.yobi.be/wiki/NSC_Writeups`.
82. L. Tolhuizen. Improved cryptanalysis of an AES implementation. In *Proceedings of the 33rd WIC Symposium on Information Theory*. Werkgemeenschap voor Inform.- en Communicatietheorie, 2012.
83. M. Tunstall, D. Mukhopadhyay, and S. Ali. Differential fault analysis of the advanced encryption standard using a single fault. In C. A. Ardagna and J. Zhou, editors, *WISTP 2011*, volume 6633 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2011.
84. U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. *Data Encryption Standard (DES)*.
85. E. Vanderbéken. Hacklu reverse challenge write-up. online, 2009. `http://baboon.rce.free.fr/index.php?post/2009/11/20/HackLu-Reverse-Challenge`.
86. B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 264–277, Ottawa, Canada, Aug. 16–17, 2007. Springer, Heidelberg, Germany.
87. Y. Xiao and X. Lai. A secure implementation of white-box AES. In *Computer Science and its Applications, 2009. CSA '09. 2nd International Conference on*, pages 1–6, 2009.

88. Y. Zhou and S. Chow. System and method of hiding cryptographic private keys, Dec. 15 2009. US Patent 7,634,091.