Differential Computation Analysis: Hiding your White-Box Designs is Not Enough

Joppe W. Bos[1], Charles Hubain[2]*, Wil Michiels[1], and Philippe Teuwen[1]

[1] NXP Semiconductors
[2] Université Libre de Bruxelles

Abstract. Although all current scientific white-box approaches of standardized cryptographic primitives are broken, there is still a large number of companies which sell "secure" white-box products. In this paper a new approach to assess the security of white-box implementations is presented which requires *neither* knowledge about the look-up tables used *nor* any reverse engineering effort. This differential computation analysis (DCA) attack is the software counterpart of the differential power analysis attack as applied by the cryptographic hardware community. We developed plugins to widely available dynamic binary instrumentation frameworks to produce *software execution traces* which contain information about the memory addresses being accessed. We show how DCA can extract the secret key from all publicly (non-commercial) available white-box programs implementing standardized cryptography by analyzing these traces to identify secret-key dependent correlations.

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 (e.g., see [48] 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 higher range. 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 [28] 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 gray-box attack model. Since our 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 [17,16].

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, among 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 devices

---

[*] This work was performed while the second author was an intern in the Innovation Center Crypto & Security at NXP Semiconductors.

which contain private cryptographic keys of third-parties. White-box implementations 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 Host Card Emulation (HCE) and the protection of credentials for an authentication to the cloud.

As stated in [16], “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 source code obfuscation [7,31] and control-flow obfuscation [22] although these are typically applied to white-box implementations as well as an additional line of defense. Although it is conjectured in one of the seminal papers of Chow et al. that no long-term defense against attacks on white-box implementations exist [16], there are still a significant number of companies selling secure white-box solutions. It should be noted that there are no known published results on how to turn any of the standardized public-key algorithms into a white-box implementation. All the published white-box techniques exclusively focus on symmetric cryptography. However, all such published approaches have been theoretically broken (see Section 2 for an overview).

A disadvantage of these published attacks is that it requires 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). What is currently lacking is a family of attacks that, instead of attacking a design, exploit the information leakage in a white-box implementation. Our contribution is the first example in this direction.

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 [19] this means that such a white-box implementation should resist all existing and future side-channel attacks. In this paper we explore exactly this idea and check if we can deduce information about the secret embedded in a white-box implementation by correlating key guesses to intermediate results obtained from *software traces*.

Dynamic binary analysis (DBA) is often used to improve and inspect the quality of software implementations. Dynamic binary instrumentation (DBI) is a well-studied approach to implement DBA. The idea is that additional analysis code is added to the original code of the client program at run-time in order to aid memory debugging, memory leak detection, and profiling. The most advanced DBI tools, such as Valgrind [45] and Pin [33], allow one to monitor, modify and insert instructions in a binary executable. Hence, these tools are extremely useful when assessing the security of real white-box implementations (and have shown their potential for behavioral analysis of obfuscated code [52]).

We wrote plugins for both Valgrind and Pin such that it records memory addresses (read and write access). These software traces are used to check if the memory accesses made by the white-box implementations can be used to deduce information about the secret key embedded in this implementation. For this we introduce *differential computation analysis* (DCA), which can be seen as the software counterpart of the differential power analysis [28] techniques as applied by the cryptographic hardware community.

In this paper we show that DCA can be used to efficiently extract the secret key from white-box implementations. We apply DCA to all publicly available, as far as we are aware, 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.

The remainder of the paper is organized as follows. In Section 2 we recall the techniques on how to turn a regular cryptographic symmetric algorithm into a white-box implementation. Next, we give a literature survey of the attempts to build white-box implementations and the successful attacks. Section 3 recalls the basic ideas behind the differential power analysis attacks. In Section 4 we introduce the rationale behind software traces, how to obtain such traces and how to use these in a differential computational analysis attack. Section 5 summarizes how we successfully applied the differential computational analysis attack to several publicly available white-box challenges. Section 6 concludes this paper with some future research directions.

2 Scientific White-Box Cryptography Techniques: An Overview

As discussed in the Introduction, 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 [16] conjectured that that no long-term defense against attacks on white-box implementations exist. 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 [4], however, it is unknown if this result applies to a specific subset of white-box functionalities.

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 the *software aging* technique [24]: 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 makes only 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.

External encodings. Besides its primary goal to hide the key, white-box implementations can also be used to add additional functionality, such as putting a fingerprint on a cryptographic key to enable traitor tracing or harden software against tampering [38]. 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 [16]. 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) encoding are randomly chosen bijections such that the extraction of E'_k does not allow the adversary to compute E_k. 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 only consider white-box implementations without external encodings, i.e., where F and G are the identity functions. Our reasoning for this is as follows. If we assume non-trivial external encodings, then the output of a successful attack is not only the key k, but also these external encodings. As indicated, the external encodings are generally canceled locally by the calling software or remotely by the content provider. Firstly, suppose that an external encoding is canceled by the calling software. For this software, the only known methods for hiding the external encodings are generic code obfuscation techniques which are a lot weaker than white-box techniques. Hence, an attacker can obtain information from this calling software about the actual plaintext or ciphertext or about the external encoding and obtain the equivalent of a white-box implementation without external encodings. In the case of a remote content provider, part of the external encoding is handled remotely. Typically, this means E_k is a decryption algorithm and the bijection F is applied on the ciphertext by the content owner. But here again, G^{-1} is supposed to be applied locally and plaintext output made accessible to the attacker.

Secondly, suppose that the external encoding is canceled only remotely or is even not canceled at all. Therefore the content is not encrypted anymore by a standard algorithm, like AES or DES, but by a modified algorithm $G \circ$ AES or $G \circ$ DES. Hence, this scenario corresponds to the situation where the input and output encodings are the identity functions applied to a non-standardized E_k. Besides these practical arguments, a more theoretical reason to only consider "naked" versions of white-box implementations is that if one claims to have constructed an implementation of AES and DES secure in the white-box attack model then this implementation should be compliant with these algorithms. Furthermore, if an attacker is able to derive k, but not F and G, we believe this still should count as a successful attack. After all, it is this information that a white-box implementation tries to hide.

General idea. The general approach to implement a white-box program is presented in [16]. The idea is to use tables rather than individual steps and encode these 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 slower than a regular (non-white-box) implementation. 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 is in the randomly chosen bijections to hide the secret key usage in the various look-up tables.

In the remainder of this section we first briefly recall the basics of DES and AES, the two most commonly used 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 and published as a Federal Information Processing Standard (FIPS) for the United States in 1979 [57]. 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 [57].

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 [1,18] 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 straight-forward to create a byte-oriented implementation. For the scope of this paper we are interested in AES-128: AES using a key-size of 128 bits. This version uses 10 rounds to compute the encryption of the input.

2.1 White-Box Data Encryption Standard (WB-DES)

The first publication attempting to construct a WB-DES implementation dates back from 2002 [17] 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 [23] 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 [30]. However, in 2007, two differential cryptanalysis [8] attacks were presented which can extract the secret key from this type of white-box [21,59]. This latter approach has a time complexity of only 2^{14}.

2.2 White-Box Advanced Encryption Standard (WB-AES)

The first approach to realize a WB-AES implementation was proposed in 2002 [16]. In 2004, the authors of [10] presented an idea 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 [13]. This approach was broken [44] using algebraic analysis with a 2^{17} time complexity in 2010. Another WB-AES approach which resisted the previous attacks was presented in [60] in 2009 and got broken in 2012 with a work factor of 2^{32} [43].

Another interesting approach is based on using the different algebraic structure for the same instance of an iterative block cipher (as proposed originally in [9]). This approach [25] 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 [16] in [29] in 2013. Moreover, the authors of [29] built upon a 2012 result [56] 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 [29] as well.

2.3 Miscellaneous White-Box Results

The above mentioned scientific work only relates to constructing and cryptanalyzing white-box DES and AES. White-box techniques have been studied and used in a broader context. In 2007, the authors of [39] presented a white-box technique to make code tamper resistant. In 2008, the cryptanalytic results for DES and AES were generalized to any substitution linear-transformation (SLT) cipher [40]. In turn, this work was generalized even further and a general analytic toolbox is presented in [3] which can extract the secret for a general SLT cipher.

Formal security notions for symmetric white-box schemes are discussed and introduced in [51,19]. In [11] it is shown how one can use the ASASA construction with injective S-boxes (where ASA stands for the affine-substitution-affine [47] construction) to instantiate white-box cryptography. A tutorial related to white-box AES is given in [42].

2.4 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 known 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 5. The differential computational analysis approach we outline in Section 4 does not need to remove the obfuscation layer nor requires reverse engineering of the binary executable.

3 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 [28]. Typically, one assumes access to the (hardware) implementation of a known cryptographic algorithm. With $I(p_i, k)$ we denote a target intermediate state of the algorithm with input p_i and where only a small portion of the secret key is used, denoted by k. One assumes that the power consumption of the device at state $I(p_i, k)$ is the sum of a data dependent component and some random noise, i.e. $\mathcal{L}(I(p_i, k)) + \delta$, 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., [35]) that the noise is random, independent from the intermediate state and is normally distributed with zero mean. Since the adversary has access to the (hardware) implementation he can obtain triples (t_i, p_i, c_i). Here p_i is one plaintext input chosen arbitrarily by the adversary, the c_i is the ciphertext output computed by the (hardware) implementation using a fixed unknown key, and the value t_i shows the power consumption over the time of the (hardware) implementation to compute the output ciphertext c_i. The measured power consumption $\mathcal{L}(I(p_i, k)) + \delta$ is just a small fraction of this entire power trace t_i.

The goal of an attacker is to recover the part of the key k by comparing the real power measurements t_i 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 [28] — see [27] 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 the least significant bit of the intermediate state $I(p_i, k)$. 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_i and input p_i, one enumerates all possible values for k (typically $2^8 = 256$ when attacking a key-byte), computes the intermediate value $g_i = I(p_i, k)$ for this key guess and divides the traces t_i into two sets according to this property measured at g_i. 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, among 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 attack applies 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, among others, higher order attacks [37], correlation power analyses [12] and template attacks [15]. Recently it was shown that the electromagnetic-based leakage from a complex System on a Chip (SoC)-based platform, such as the popular general purpose ARM core and on-chip co-processor, running at high clock frequencies can be used to deduce the secret key [32]. Nevertheless, many modern platforms use some form of software protection (such as white-box techniques) to protect the cryptographic master keys of a particular device.

4 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). 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 to perform transformations on the code while preserving the original computational effects. These transformations are performed at the basic block[3] level and are stored 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 [6]. DBI frameworks, like Pin [33] and Valgrind [45], 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 such plugins for both frameworks to trace execution of binary executables on x86, x86-64 and ARM 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 lookups 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. The trace gets stored in a database for easy access. 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. Even if ASLR cannot be disabled, it would just be a mere annoyance to realign the obtained traces.

---

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

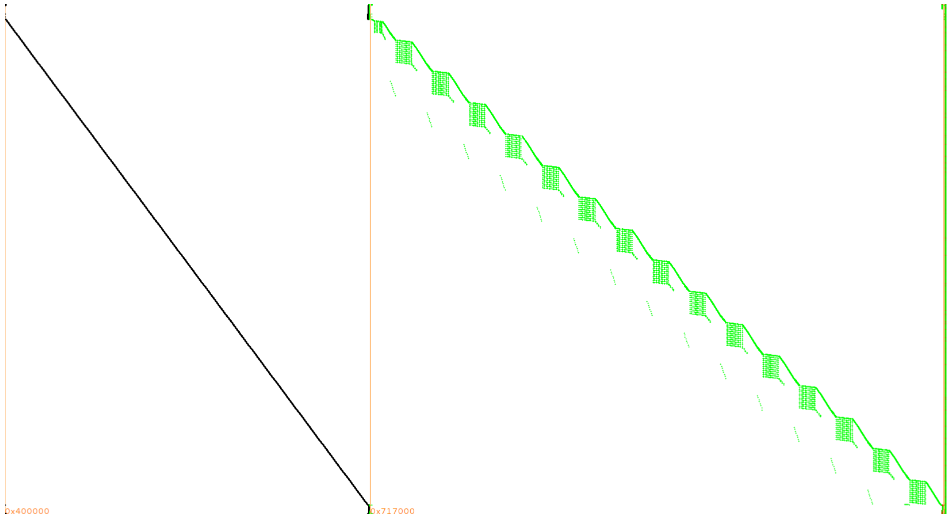
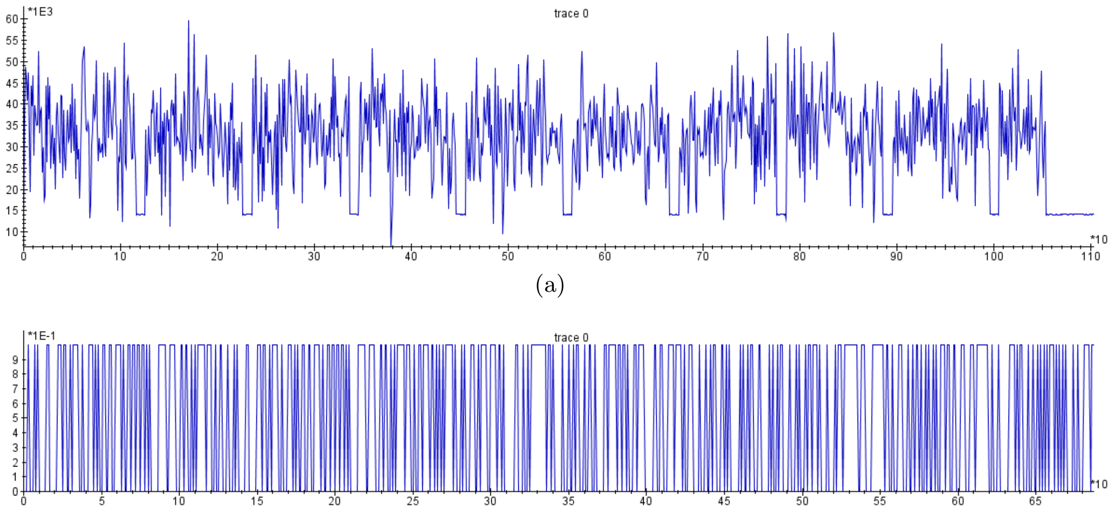


Fig. 1. Visualization of a software execution trace of a typical white-box DES implementation.

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 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 [41]. 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. In the example given in Fig. 1, the read accesses pattern make 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



(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.

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 this is not a concern, one can directly jump to the third step if one can afford to record traces of the full execution, which is perfectly acceptable for executables without much overhead and will become apparent in several examples in Section 5. 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 moreover 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.

Fifth step. Once the software execution traces have been acquired and shaped, we can use regular DPA tools such as the Riscure Inspector tool [49]: a solution we found more convenient and user-friendly than coding our own DPA tool. An alternative is to use the software part of ChipWhisperer[4] [46] even if at time of writing it supports only the Hamming weight leakage model and not yet the individual bits[5]. We show in the next section 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 2.4:

- 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).

---

[4] The software can be obtained at `http://www.newae.com/sidechannel/cwdocs/`.

[5] The Hamming weight leakage model can still be used successfully, it is simply not the ideal one and therefore will require more traces.

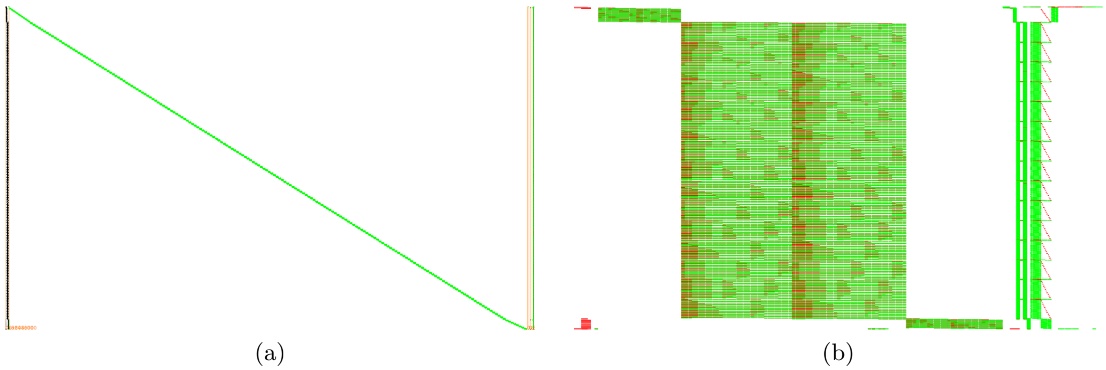

(a) (b)

Fig. 3. (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 identifiable.

5 Analyzing Publicly Available White-Box Implementations

5.1 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[6] he posted 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 [17,30] (see Section 2.1) plus "some personal improvements". The interaction with the program is straight-forward: it takes a plaintext as input parameter 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 [54] and used differential cryptanalysis (similar to [21,59]) to extract the embedded secret key.

Figure 3a 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 3a does not immediately look familiar 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 3b where the y-axis is unaltered (from Figure 3a) but the address-range (the x-axis) is rescaled to show only the read and write access to the stack.

To acquire multiple traces, 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, but we could do it on the read address range. 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

---

[6] After this challenge was solved it was removed from his webpage `www.whiteboxcrypto.com` but the binary executable can be still be found at `www.cosic.esat.kuleuven.be/sopro/wbc/wbDES.exe` (remove the ".exe" for a GNU/Linux binary).

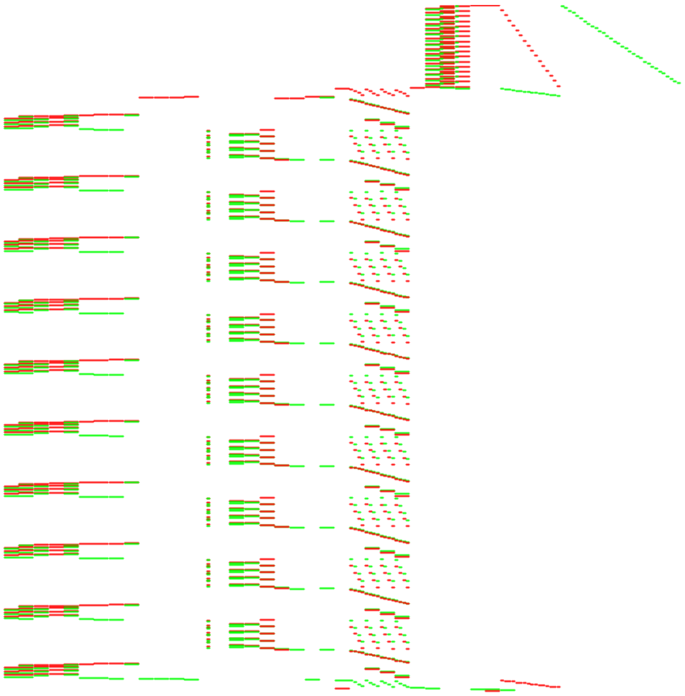


Fig. 4. Visualization of the stack reads and writes in a software execution trace of the Hack.lu 2009 challenge.

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 4.

5.2 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 [5] 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 (created by Axel Tillequin) and compares the output with an internal reference. It was solved independently by Eloi Vanderbéken [58], who reverted the functionality of the white-box implementation from encryption to decryption, and by “SysK” [54] 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 [2][7], an open source compatibility layer to run Windows applications under GNU/Linux. We automated the GUI, keyboard and mouse interactions using xdotool[8]. Due to the configuration of this challenge we had full

---

[7] `https://www.winehq.org/`

[8] `http://www.semicomplete.com/projects/xdotool/`

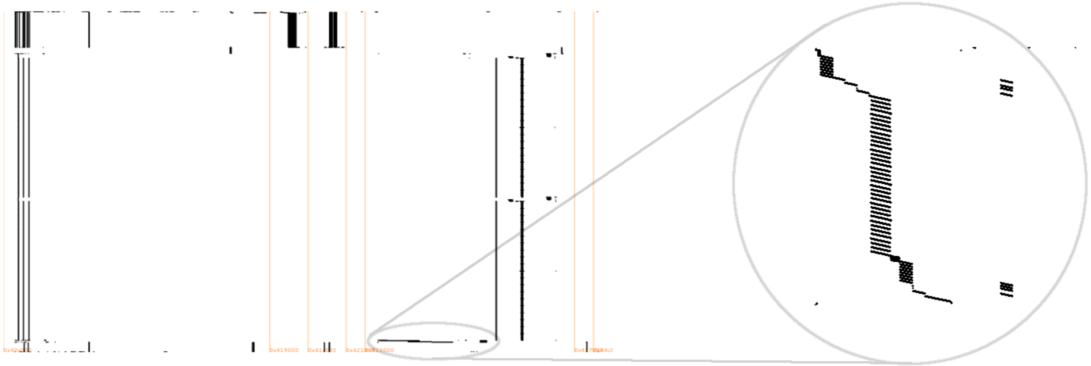


Fig. 5. Visualization of the instructions in a software execution trace of the Karroumi WB-AES implementation by Klinec, with a zoom on the core of the white-box.

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. 4 shows the read and write access 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 (with its ten rounds). 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 [54], 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 get the highest possible value, which explains the low number of traces required to mount a successful attack.

5.3 The SSTIC 2012 Challenge

Every year for the SSTIC, *Symposium sur la sécurité des technologies de l'information et des communications* (Information technologies and communications security symposium), a challenge is published which consists of solving several steps like a Matryoshka doll. In 2012, one step of the challenge [36] was to validate a key with a Python bytecode "check.pyc": i.e. a marshalled object[9]. Internally this bytecode generates a random plaintext, forwards this to the white-box *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 [36]. A number of solutions identified the implementation as a white-box DES without encodings (naked variant) as described in [17]. Some extracted the key following the approach from the literature while some performed their own algebraic attack.

---

[9] `https://docs.python.org/2/library/marshal.html`

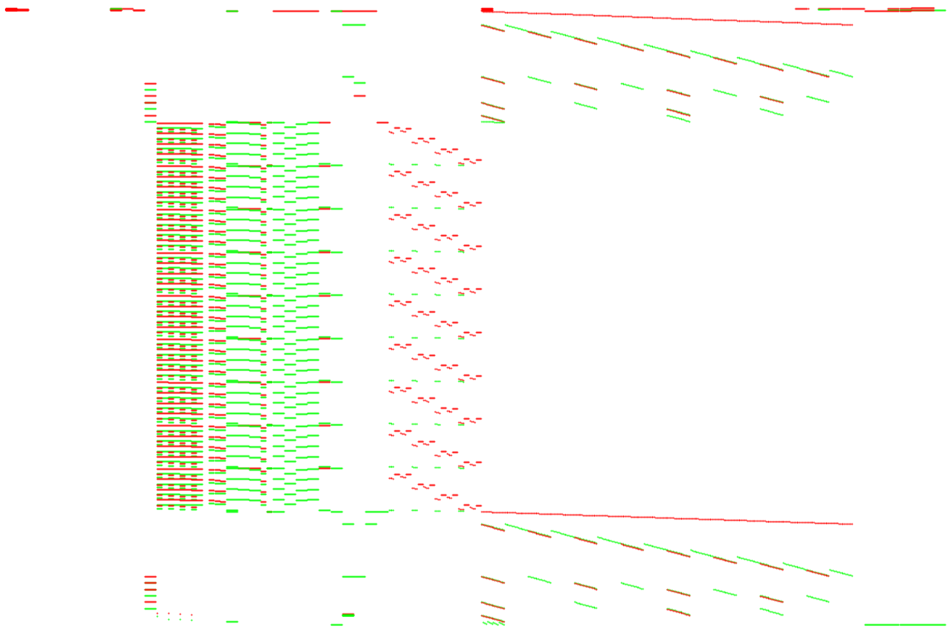


Fig. 6. Visualization of the stack reads and writes in the software execution trace portion limited to the core of the Karroumi WB-AES.

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 virtual machine 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[10]. 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 white-box AES challenge (see Section 5.2), 16 traces were enough to recover the key of this white-box 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.

5.4 A White-Box Implementation of the Karroumi Approach

A white-box implementation of both the original AES approach [16] and the approach based on dual ciphers by Karroumi [25] is part of the Master thesis by Dušan Klinec [26][11]. As explained in Section 2.2, this is the latest academic variant of [16]. 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[12] libraries to dynamically load and deserialize the white-box tables from a file. The left part of Figure 5 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 5 shows an enlarged view of the white-box core. Here, one can recognize the nine

---

[10] `https://docs.python.org/2/library/pickle.html`

[11] The code be found at `https://github.com/ph4r05/Whitebox-crypto-AES`.

[12] `http://www.boost.org/`

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	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓

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	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

MixColumns operating on the four columns. This structure can be observed even better from the stack trace of Figure 6. 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 tracked 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 of the right key byte value amongst the guesses after 2000 traces, when sorted according to the difference of means (see Section 3). 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 [25] 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 Tables 1 and 2 that when a target bit of a given key byte does not leak (i.e. 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 can also be used to recover the key even quicker.

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, here are some figures: The acquisition of 500 traces takes about 200s on a regular laptop (dual-core i7-4600U CPU at 2.10GHz). 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 30s.

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.

5.5 The NoSuchCon 2013 Challenge

In April 2013, a challenge designed by Eloi Vanderbéken was published for the occasion of the NoSuchCon 2013 conference[13]. 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. [53,34]) but without ever recovering the key. It illustrates one more issue designers of white-box implementations have to deal with in practice: one can convert an encryption routine into a decryption routine without actually extracting the key. This was accomplished by reverting each round independently, with a moderate brute-force effort (of order $\mathcal{O}(2^{37.2})$ for this specific challenge).

For a change, the design is not derived from Chow [16]. 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 meaningful 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 $E'_k = G \circ E_k \circ F^{-1}$. Note, however, that we did manage to recover the key and the encodings from this white-box implementation with an algebraic attack as described in [55]. This was achieved after a painful deobfuscation of the binary (almost completely performed by previous write-ups [53] and [34]), a step needed to fulfill the prerequisites for such attacks as described in Section 2.4.

---

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

6 Conclusions and Future Work

As conjectured in the first papers introducing the white-box attack model, there is no long-term defense against attacks on white-box implementations. However, as we have shown in this work, all current publicly available white-box implementations (not using remote external encodings) do not even offer any short-term security since the differential computation analysis (DCA) technique we outlined can extract the secret key within seconds. We did not investigate the strength of commercially available white-box products since no company, as far as we are aware, made a challenge publicly available similar to, for instance, the RSA factoring challenge [20] (discontinued in 2007), the challenge related to elliptic curve cryptography [14], or the PRINCE challenge [50]. However, we do not think that it is wise, taking the results from this paper into consideration, to use white-box implementations for applications that deal with sensitive information, such as credit card processing as suggested in [61].

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 and higher order attacks. Studying other point of attack, for instance targeting the multiplicative inverse step in the first round of AES, give interesting results (see Section 5.4). Investigating other positions as a target in our DCA approach may be worth investigating as well.

References

1. Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, Nov. 2001.
2. B. Amstadt and M. K. Johnson. Wine. *Linux Journal*, 1994(4), August 1994.
3. 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`.
4. 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, Berlin, Germany.
5. J.-B. Bédrune. Hack.lu 2009 reverse challenge 1. online, 2009. `http://2009.hack.lu/index.php/ReverseChallenge`.
6. F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
7. 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.
8. 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, Berlin, Germany.
9. O. Billet and H. Gilbert. A traceable block cipher. In C.-S. Laih, editor, *ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 331–346. Springer, Berlin, Germany, 2003.
10. 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, Berlin, Germany.
11. 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, Berlin, Germany.
12. 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, Berlin, Germany.

13. 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`.
14. Certicom. The certicom ECC challenge. Webpage. `https://www.certicom.com/index.php/the-certicom-ecc-challenge`.
15. 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, California, USA, Aug. 13–15, 2003. Springer, Berlin, Germany.
16. 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, Berlin, Germany.
17. 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.
18. J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer, 2002.
19. 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, Berlin, Germany.
20. EMC Corporation. The RSA factoring challenge. Webpage. `http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge.htm`.
21. 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, Berlin, Germany.
22. 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.
23. 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.
24. 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.
25. 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, Berlin, Germany.
26. D. Klinec. White-box attack resistant cryptography. Master's thesis, Masaryk University, Brno, Czech Republic, 2013. `https://is.muni.cz/th/325219/fi_m/`.
27. P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.
28. 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, Berlin, Germany.
29. 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, Berlin, Germany.
30. 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.
31. 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.
32. J. Longo, E. D. Mulder, D. Page, and M. Tunstall. Soc it to EM: electromagnetic side-channel attacks on a complex system-on-chip. Cryptology ePrint Archive, Report 2015/561, 2015. `http://eprint.iacr.org/`.
33. 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.

34. A. Maillet. Nosuchcon 2013 challenge - write up and methodology. online, 2013. `http://kutioo.blogspot.be/2013/05/nosuchcon-2013-challenge-write-up-and.html`.
35. 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.
36. F. Marceau, F. Perigaud, and A. Tillequin. Challenge sstic 2012. online, 2012. `http://communaute.sstic.org/ChallengeSSTIC2012`.
37. 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, Berlin, Germany.
38. W. Michiels. Opportunities in white-box cryptography. *IEEE Security & Privacy*, 8(1):64–67, 2010.
39. 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.
40. 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, Berlin, Germany.
41. 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`.
42. 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.
43. 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, Berlin, Germany.
44. 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, Berlin, Germany.
45. 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.
46. C. O'Flynn and Z. D. Chen. ChipWhisperer: An open-source platform for hardware embedded security research. In E. Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - COSADE*, volume 8622 of *LNCS*, pages 243–260. Springer, 2014.
47. 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, Berlin, Germany.
48. 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.
49. Riscure. Inspector SCA. online. `https://www.riscure.com/security-tools/inspector-sca/`.
50. Ruhr University Bochum, NXP Semiconductors, and Technical University of Denmark. The PRINCE challenge. Webpage. `https://www.emsec.rub.de/research/research_startseite/prince-challenge/`.
51. 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, Berlin, Germany.
52. 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`.
53. A. Souchet. AES whitebox unboxing: No such problem. online, 2013. `http://0vercl0k.tuxfamily.org/bl0g/?p=253`.
54. SysK. Practical cracking of white-box implementations. Phrack 68:14. `http://www.phrack.org/issues/68/8.html`.
55. P. Teuwen. NSC writeups. online, 2015. `http://wiki.yobi.be/wiki/NSC_Writeups`.
56. L. Tolhuizen. Improved cryptanalysis of an AES implementation. In *Proceedings of the 33rd WIC Symposium on Information Theory*. Werkgemeenschap voor Inform.-en Communicatietheorie, 2012.
57. U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. *Data Encryption Standard (DES)*.
58. E. Vanderbéken. Hacklu reverse challenge write-up. online, 2009. `http://baboon.rce.free.fr/index.php?post/2009/11/20/HackLu-Reverse-Challenge`.

59. 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, Berlin, Germany.
60. 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.
61. N. Yadav and S. Tanwar. Implementation of white-box cryptography in credit card processing combined with code obfuscation. *International Journal of Computer Applications*, 70(2):35–38, May 2013.