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

Joppe W. Bos[1(✉)], Charles Hubain[2], Wil Michiels[1,3], and Philippe Teuwen[2]

[1] NXP Semiconductors, Leuven, Belgium
{joppe.bos,wil.michiels}@nxp.com
[2] Quarkslab, Paris, France
{chubain,pteuwen}@quarkslab.com
[3] Technische Universiteit Eindhoven, Eindhoven, The Netherlands

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, we present a new approach to assess the security of white-box implementations 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. To illustrate its effectiveness, we show how DCA can extract the secret key from numerous publicly (non-commercial) available white-box programs implementing standardized cryptography by analyzing these traces to identify secret-key dependent correlations. This approach allows 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 manner.

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. [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 other end. A popular approach which attempts to hide a cryptographic key inside a software program is known as a *white-box implementation*.

Ch. Hubain and Ph. Teuwen—This work was performed while the second and fourth author were an intern and employee in the Innovation Center Crypto & Security at NXP Semiconductors, respectively.

B. Gierlichs and A.Y. Poschmann (Eds.): CHES 2016, LNCS 9813, pp. 215–236, 2016.
DOI: 10.1007/978-3-662-53140-2_11

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 et al. [32] 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 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 [16,17].

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 platforms which store 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 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 [19] this means that such a white-box implementation should resist all existing and future side-channel attacks.

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 code obfuscation [5,36] and control-flow obfuscation [25] although these are typically applied to white-box implementations as an additional line of defense. Although it is conjectured 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 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 [61]. The other published white-box

techniques exclusively focus on symmetric cryptography. However, all such published approaches have been theoretically broken (see Sect. 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 Sect. 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 Contribution. 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 contribution is an attack that works in an automated way, and it is therefore a major threat for the claimed security level of the offered solutions compared to the ones that are already known.

In this paper 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 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 [46] and Pin [37], 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 [52].

We have developed plugins for both Valgrind and Pin to obtain *software traces*[1]: a trace which records the read and write accesses made to memory. These software traces are used to deduce information about the secret embedded in a white-box implementation by correlating key guesses to intermediate results. For this we introduce *differential computation analysis* (DCA), which can be seen as the software counterpart of the differential power analysis (DPA) [32] 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 Sect. 4.

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. We apply DCA to the publicly available white-box challenges

[1] The entire software toolchain ranging from the plugins, to the GUI, to the individual scrips to target the white-box challenges as described in this paper is released as open-source software: see https://github.com/SideChannelMarvels.

of standardized cryptographic algorithms we could find; concretely this means extracting the secret key from four white-box implementations of the symmetric cryptographic algorithms AES and DES. In contrast to the current cryptanalytic methods to attack white-box implementations, this technique does 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 Sect. 6).

The main reason why DCA works is related to the choice of (non-) linear encodings which are used inside the white-box implementation (cf. Sect. 2). These encodings do not sufficiently hide correlations when the correct key is used and enables one to run side-channel attacks (just as in gray-box attack model). Sasdrich et al. looked into this in detail [50] and used the Walsh 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 explain why first-order attacks like DPA are successful in this scenario.

Independently, and after this paper appeared online, Sanfelix, de Haas and Mune also presented attacks on white-box implementations [49]. On the one hand they confirmed our findings and on the other hand they considered software fault attacks which is of independent interest.

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 [16] 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 [3], 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. [2,10,22]). 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* [27]: 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 provide additional functionality, such as putting a fingerprint on a cryptographic key to enable traitor tracing or hardening software against tampering [42]. 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 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 $G \circ \mathrm{DES} \circ F^{-1}$).

General Idea. The general approach to implement a white-box program is presented in [16]. 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 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 bijections to hide the secret key usage in the various look-up tables.

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 [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 [26] 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 [35]. However, in 2007, two differential cryptanalytic attacks [6] were presented which can extract the secret key from this type of white-box [23,59]. 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 [16]. In 2004, the authors of [8] present 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 [12]. This approach was broken [45] 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} [44].

Another interesting approach is based on using the different algebraic structure for the same instance of an iterative block cipher (as proposed originally in [7]). This approach [28] 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 [33] in 2013. Moreover, the authors of [33] built upon a 2012 result [57] 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 [33] as well.

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 (1) know the type of encodings that are applied on *the intermediate results*, and (2) 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 Sect. 5. The differential computational analysis approach we outline in Sect. 4 does not need to remove the obfuscation layers 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 [32]. 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 in the computation, 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., [39]) 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_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 implementation using a fixed unknown key, and the value t_i shows the power consumption over the time of the 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 [32] (see [31] 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 $I(p_i, k)$. 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 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, among others, higher order attacks [41], correlation power analyses [11] and template attacks [15].

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 Sect. 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 performing transformations on the code while preserving the original computational effects. DBI frameworks, like Pin [37] and Valgrind [46], 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 Sect. 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 Sect. 5, 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 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 [43]. Figure 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,

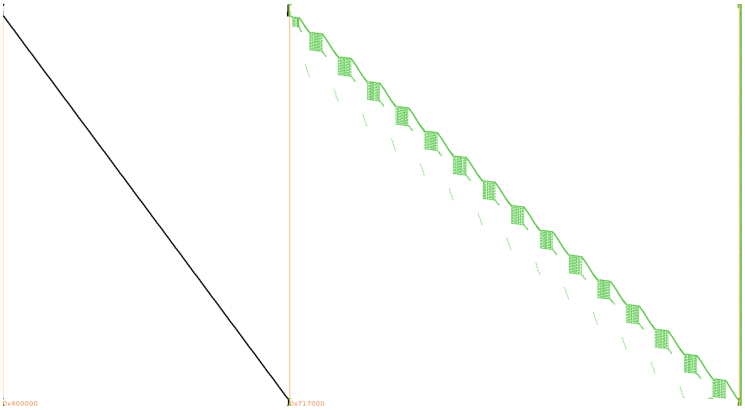


Fig. 1. Visualization of a software execution trace of a white-box DES implementation. (Color figure online)

as explained in Sect. 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 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 Sect. 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.

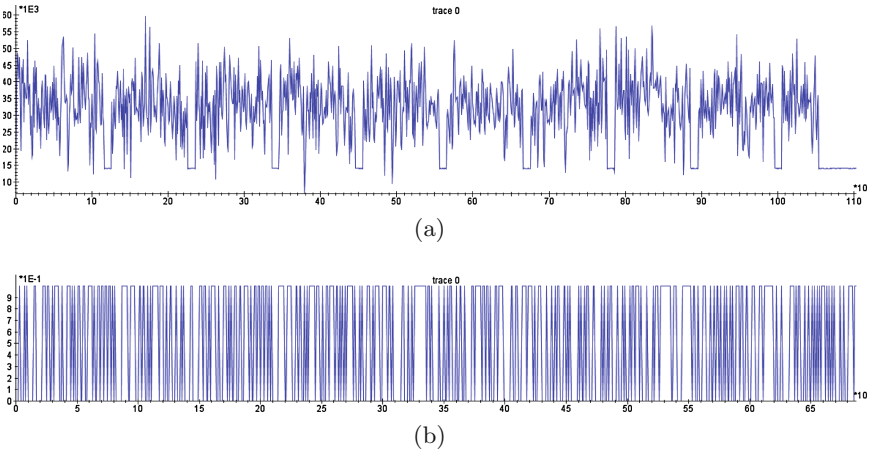


Fig. 2. Figure (a) is a typical example of a (hardware) power trace of an unprotected AES-128 implementation (one can observe the ten rounds). (b) is a typical example of a portion of a serialized software trace of stack writes in an AES-128 white-box, with only two possible values: zero or one.

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. Figure 2a and b 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 to extract the key. 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 Sect. 2.2: (1) Be able to run several times (a few dozens to a few thousands) the binary in a controlled environment. (2) having knowledge of the plaintexts (before their encoding, if any), or of the ciphertexts (after their decoding, if any).

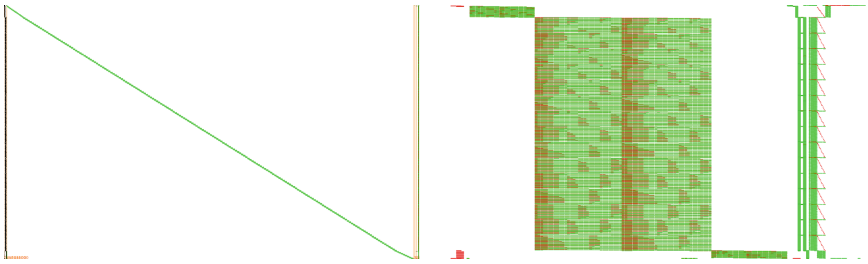
5 Analyzing Publicly Available White-Box Implementations

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[2] 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 [17,35] (see Sect. 2.1) plus "some personal improvements". The interaction with the program is straight-forward: 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 [54] and used differential cryptanalysis (similar to [23,59]) 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 Fig. 3a does not immediately look familiar to DES. However, if one takes a closer look to the address space which represents

[2] See http://whiteboxcrypto.com/challenges.php.



(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 visible. (Color figure online)

the stack (on the far right) then the 16 rounds of DES can be clearly distinguished. This zoomed view is outlined in Fig. 3b where the y-axis is unaltered (from Fig. 3a) 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 Sect. 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 [4] 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 [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 [1], an open source compatibility layer to run Windows applications under GNU/Linux. Due to the configuration of this challenge we had full control on the input to the white-box.

Visualizing the traces using our software framework clearly shows 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 [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 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 [40] was to validate a key with a Python bytecode "check.pyc": i.e. a marshalled object[3]. Internally this bytecode generates a random plaintext, forwards this 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 [40]. A number of solutions identified the implementation as a WB-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.

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[4]. The white-

[3] https://docs.python.org/2/library/marshal.html.
[4] https://docs.python.org/2/library/pickle.html.

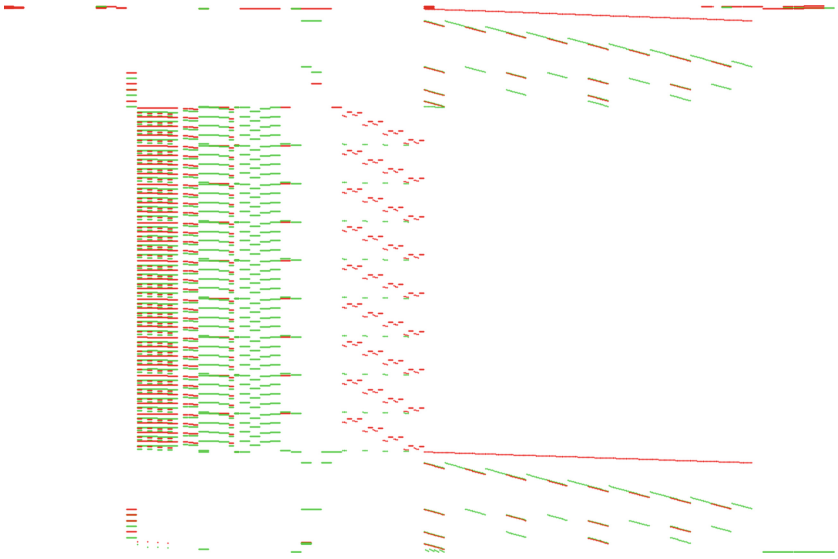


Fig. 4. Visualization of the stack reads and writes in the software execution trace portion limited to the core of the Karroumi WB-AES.

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 Sect. 5.2), 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.

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 [28] is part of the Master thesis by Dušan Klinec [30][5]. As explained in Sect. 2.1, 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[6] libraries to dynamically load and deserialize the white-box tables from a file. An initial software trace when running this white-box AES binary executable shows that the white-box code itself constitutes only a fraction of the total instructions (most of the instructions are from initializing the Boost libraries). From the stack trace (see Fig. 4) one can recognize the nine *MixColumns* operating on the four columns. Therefore we used instruction address filtering to focus on the white-box core and skip all the Boost C++ operations.

[5] The code be found at https://github.com/ph4r05/Whitebox-crypto-AES.
[6] 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.

target bit	key byte															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
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	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓

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 Sect. 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.

It is interesting to observe in Table 1 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 256^{th} 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 2000 traces takes about 800s on a regular laptop (dual-core i7-4600U CPU at 2.10 GHz). This results in 3328 kbits (416 kB) of traces when limited to the execution of the first round. Running the attack requires less than 60 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.

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[7]. 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. [38,53]) 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.

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 Sect. 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 [56]. This was achieved after a painful de-obfuscation of the binary (almost completely performed by previous write-ups [38,53]), a step needed to fulfill the prerequisites for such attacks as described in Sect. 2.2.

The same white-box is found among the CHES 2015 challenges[8] in a Game-Boy ROM and the same algebraic attack is used successfully as explained in [55] once the tables got extracted.

6 Countermeasures Against DCA

In hardware, counter-measures against DPA typically rely on a random source. The output can be used to mask intermediate results, to re-order instructions, or to add delays (see e.g. [14,24,51]). 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?

[7] See http://www.nosuchcon.org/2013/.

[8] https://ches15challenge.com/static/CHES15Challenge.zip, preserved at https://archive.org/details/CHES15Challenge

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 [18] it is proposed to use *variable* encodings when accessing the look-up tables based on the affine equivalences for bijective S-boxes (cf. [9] 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 run-time) 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 [47]. A threshold implementation is a masking scheme based on secret sharing and multi-party 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.

In practice, one might resort to methods to make the job of the adversary more difficult. Typical software counter-measures 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 [21,34], DBI becomes stealthier too [29].

7 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 differential computation analysis (DCA) technique 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] or the challenge related to elliptic curve cryptography [13].

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.

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.

References

1. Amstadt, B., Johnson, M.K.: Wine. Linux J. **1994**(4) (1994). http://dl.acm.org/citation.cfm?id=324681.324684, ISSN: 1075-3583
2. Barak, B., Garg, S., Kalai, Y.T., Paneth, O., Sahai, A.: Protecting obfuscation against algebraic attacks. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 221–238. Springer, Heidelberg (2014)
3. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., Yang, K.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001)
4. Bédrune, J.-B.: Hack.lu 2009 reverse challenge 1 (2009). http://2009.hack.lu/index.php/ReverseChallenge
5. Bhatkar, S., DuVarney, D.C., Sekar, R.: 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)
6. Biham, E., Shamir, A.: Differential cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 156–171. Springer, Heidelberg (1992)
7. Billet, O., Gilbert, H.: A traceable block cipher. In: Laih, C.-S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 331–346. Springer, Heidelberg (2003)
8. Billet, O., Gilbert, H., Ech-Chatbi, C.: Cryptanalysis of a white box AES implementation. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 227–240. Springer, Heidelberg (2004)
9. Biryukov, A., Canniére, C., Braeken, A., Preneel, B.: A toolbox for cryptanalysis: linear and affine equivalence algorithms. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 33–50. Springer, Heidelberg (2003)
10. Brakerski, Z., Rothblum, G.N.: Virtual black-box obfuscation for all circuits via generic graded encoding. In: Lindell, Y. (ed.) TCC 2014. LNCS, vol. 8349, pp. 1–25. Springer, Heidelberg (2014)
11. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
12. Bringer, J., Chabanne, H., Dottax, E.: White box cryptography: another attempt. Cryptology ePrint Archive, Report 2006/468 (2006). http://eprint.iacr.org/2006/468
13. Certicom: The certicom ECC challenge. https://www.certicom.com/index.php/the-certicom-ecc-challenge
14. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999)
15. Chari, S., Rao, J.R., Rohatgi, P.: Template attack. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 13–28. Springer, Heidelberg (2003)

16. Chow, S., Eisen, P.A., Johnson, H., van Oorschot, P.C.: White-box cryptography and an AES implementation. In: Nyberg, K., Heys, H.M. (eds.) SAC 2002. LNCS, vol. 2595, pp. 250–270. Springer, Heidelberg (2003)
17. Chow, S., Eisen, P., Johnson, H., van Oorschot, P.C.: A white-box DES implementation for DRM applications. In: Feigenbaum, J. (ed.) DRM 2002. LNCS, vol. 2696, pp. 1–15. Springer, Heidelberg (2003)
18. de Mulder, Y.: White-box cryptography: analysis of white-box AES implementations. Ph.D. thesis, KU Leuven (2014)
19. Delerablée, C., Lepoint, T., Paillier, P., Rivain, M.: White-box security notions for symmetric encryption schemes. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) SAC 2013. LNCS, vol. 8282, pp. 247–264. Springer, Heidelberg (2014)
20. EMC Corporation: The RSA factoring challenge. http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge.htm
21. Falco, F., Riva, N.: Dynamic binary instrumentation frameworks: I know you're there spying on me. In: REcon (2012). http://recon.cx/2012/schedule/events/216.en.html
22. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS, pp. 40–49. IEEE Computer Society (2013)
23. Goubin, L., Masereel, J.-M., Quisquater, M.: Cryptanalysis of white box DES implementations. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 278–295. Springer, Heidelberg (2007)
24. Goubin, L., Patarin, J.: DES and differential power analysis. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 158–172. Springer, Heidelberg (1999)
25. Huang, Y., Ho, F.S., Tsai, H., Kao, H.M.: A control flow obfuscation method to discourage malicious tampering of software codes. In: Lin, F., Lee, D., Lin, B.P., Shieh, S., Jajodia, S. (eds.) Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006, p. 362. ACM (2006)
26. Jacob, M., Boneh, D., Felten, E.W.: Attacking an obfuscated cipher by injecting faults. In: Feigenbaum, J. (ed.) DRM 2002. LNCS, vol. 2696, pp. 16–31. Springer, Heidelberg (2003)
27. Jakobsson, M., Reiter, M.K.: Discouraging software piracy using software aging. In: Sander, T. (ed.) DRM 2001. LNCS, vol. 2320, pp. 1–12. Springer, Heidelberg (2002)
28. Karroumi, M.: Protecting white-box AES with dual ciphers. In: Rhee, K.-H., Nyang, D.H. (eds.) ICISC 2010. LNCS, vol. 6829, pp. 278–291. Springer, Heidelberg (2011)
29. Kirsch, J.: Towards transparent dynamic binary instrumentation using virtual machine introspection. In: REcon (2015). https://recon.cx/2015/schedule/events/20.html
30. Klinec, D.: White-box attack resistant cryptography. Master's thesis, Masaryk University, Brno, Czech Republic (2013). https://is.muni.cz/th/325219/fi_m/
31. Kocher, P., Jaffe, J., Jun, B., Rohatgi, P.: Introduction to differential power analysis. J. Cryptogr. Eng. **1**(1), 5–27 (2011)
32. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
33. Lepoint, T., Rivain, M., De Mulder, Y., Roelse, P., Preneel, B.: Two attacks on a white-box AES implementation. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) SAC 2013. LNCS, vol. 8282, pp. 265–286. Springer, Heidelberg (2014)

34. Li, X., Li, K.: Defeating the transparency features of dynamic binary instrumentation. In: BlackHat US (2014). https://www.blackhat.com/docs/us-14/materials/us-14-Li-Defeating-The-Transparency-Feature-Of-DBI.pdf
35. Link, H.E., Neumann, W.D.: Clarifying obfuscation: improving the security of white-box DES. In: International Symposium on Information Technology: Coding and Computing (ITCC 2005), pp. 679–684. IEEE Computer Society (2005)
36. Linn, C., Debray, S.K.: Obfuscation of executable code to improve resistance to static disassembly. In: Jajodia, S., Atluri, V., Jaeger, T. (eds.) Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, pp. 290–299. ACM (2003)
37. Luk, C., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, pp. 190–200. ACM (2005)
38. Maillet, A.: Nosuchcon 2013 challenge - write up and methodology (2013). http://kutioo.blogspot.be/2013/05/nosuchcon-2013-challenge-write-up-and.html
39. Mangard, S., Oswald, E., Standaert, F.: One for all - all for one: unifying standard differential power analysis attacks. IET Inf. Secur. **5**(2), 100–110 (2011)
40. Marceau, F., Perigaud, F., Tillequin, A.: Challenge SSTIC 2012 (2012). http://communaute.sstic.org/ChallengeSSTIC2012
41. Messerges, T.S.: Using second-order power analysis to attack DPA resistant software. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, pp. 238–251. Springer, Heidelberg (2000)
42. Michiels, W.: Opportunities in white-box cryptography. IEEE Secur. Priv. **8**(1), 64–67 (2010)
43. Mougey, C., Gabriel, F.: Désobfuscation de DRM par attaques auxiliaires. In: Symposium sur la sécurité des technologies de l'information et des communications (2014). http://www.sstic.org/2014/presentation/dsobfuscation_de_drm_par_attaques_auxiliaires
44. De Mulder, Y., Roelse, P., Preneel, B.: Cryptanalysis of the Xiao–Lai white-box AES implementation. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 34–49. Springer, Heidelberg (2013)
45. De Mulder, Y., Wyseur, B., Preneel, B.: Cryptanalysis of a perturbated white-box AES implementation. In: Gong, G., Gupta, K.C. (eds.) INDOCRYPT 2010. LNCS, vol. 6498, pp. 292–310. Springer, Heidelberg (2010)
46. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Ferrante, J., McKinley, K.S., (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pp. 89–100. ACM (2007)
47. Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 529–545. Springer, Heidelberg (2006)
48. Polla, M.L., Martinelli, F., Sgandurra, D.: A survey on security for mobile devices. IEEE Commun. Surv. Tutor. **15**(1), 446–471 (2013)
49. Sanfelix, E., de Haas, J., Mune, C.: Unboxing the white-box: practical attacks against obfuscated ciphers. In: BlackHat Europe 2015 (2015). https://www.blackhat.com/eu-15/briefings.html
50. Sasdrich, P., Moradi, A., Güneysu, T.: White-box cryptography in the gray box - a hardware implementation and its side channels. In: FSE 2016, LNCS. Springer, Heidelberg (2016, to appear)

51. Schramm, K., Paar, C.: Higher order masking of the AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 208–225. Springer, Heidelberg (2006)
52. Scrinzi, F.: 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. Souchet, A.: AES whitebox unboxing: No such problem (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. Teuwen, P.: CHES2015 writeup (2015). http://wiki.yobi.be/wiki/CHES2015_Writeup#Challenge_4
56. Teuwen, P.: NSC writeups (2015). http://wiki.yobi.be/wiki/NSC_Writeups
57. Tolhuizen, L.: Improved cryptanalysis of an AES implementation. In: Proceedings of the 33rd WIC Symposium on Information Theory. Werkgemeenschap voor Inform.-en Communicatietheorie (2012)
58. Vanderbéken, E.: Hacklu reverse challenge write-up (2009). http://baboon.rce.free.fr/index.php?post/2009/11/20/HackLu-Reverse-Challenge
59. Wyseur, B., Michiels, W., Gorissen, P., Preneel, B.: Cryptanalysis of white-box DES implementations with arbitrary external encodings. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 264–277. Springer, Heidelberg (2007)
60. Xiao, Y., Lai, X.: A secure implementation of white-box AES. In: 2nd International Conference on Computer Science and its Applications 2009, CSA 2009, pp. 1–6 (2009)
61. Zhou, Y., Chow, S.: System and method of hiding cryptographic private keys. 15 December 2009. US Patent 7,634,091