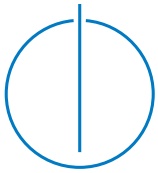
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Side-channel attacks against white-box cryptography implementations on Android

Michael Eder



DEPARTMENT OF INFORMATICS

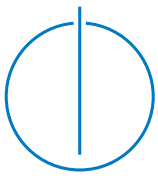
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Side-channel attacks against white-box cryptography implementations on Android

Seitenkanalangriffe auf white-box Kryptographie unter Android

Author:	Michael Eder
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	Dr. Julian Schütte, Dennis Titze
Submission Date:	15.6.2016



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.6.2016 Michael Eder

Acknowledgments

I would like to thank Prof. Dr. Claudia Eckert for giving me the opportunity to write this thesis at her chair in cooperation with Fraunhofer AISEC. This work would not have been possible without Dr. Julian Schütte who suggested the topic and, together with Dennis Titze, gave me a lot of valuable feedback from the very beginning until the end and pointed me into right directions. Andreas Zankl and Florian Unterstein were great contact persons for questions regarding Hardware Security Modules, side-channel attacks, and related topics. Philippe Teuwen gave me hints and valuable feedback and his ongoing work to the SCAMarvels toolchain as well as his wiki provide good resources for future work on this topic. I also like to thank all proof-readers and the authors of the free and open source software I used for their great work.

Last but not least I like to thank my family, especially my grandparents, for giving me the opportunity to study computer science at TUM and supporting me in everything I am doing.

Abstract

White-box cryptography aims to protect keys of cryptographic primitives in hostile environments. All academic schemes have been broken by cryptanalysts. In 2015, a side-channel attack called *Differential Computation Analysis (DCA)* was presented. It allows to break nearly all publicly available white-box cryptography implementations without the need for further analyzing or reverse engineering them.

This work validates these results and further we try to adapt the attack to a new platform, the Android operating system. Although the toolchain is not completely usable on Android, we feel certain that an adaption of the attack against native libraries is feasible with some additional improvements to the toolchain. For regular Java, we show problems of the approach using the current toolchain.

Additionally, we create a Docker container image in order to ease further development, automation and analysis of white-box binaries. Further, it is possible to speed up attacks by running multiple containers in parallel.

Contents

1. Introduction

White-box cryptography is a paradigm at the intersection of cryptography and code obfuscation. While white-box cryptography might not be theoretically sound, it is regarded as a feasible protection mechanism in some scenarios. However, lately Bos et al. [7] and Sanfelix et al. [36] published attacks against publicly available white boxes without reverse engineering or breaking the mathematical foundations of them. The side-channel attacks used to recover the keys are closely related to side-channel attacks also used against *Hardware Security Modules (HSM)*. The goals of this thesis are to validate the results of Bos et al. as well as to port the attacks to the Android platform. Further, we evaluate whether efficient attacks against white boxes shipped by Android applications are a realistic threat.

White-box cryptography is different from traditional cryptography. Concerning types of keys, cryptography can be divided into symmetric and asymmetric cryptography. The first uses the same key for encryption and decryption whereas the latter uses a public key for encryption and a private key for decryption. Symmetric cryptography has usually less mandatory mathematical requirements and is therefore considered to be faster whereas asymmetric cryptography solves problems regarding key exchange and can often be used to create digital signatures.

Game-based attacker models are frequently used to evaluate security properties of cryptographic algorithms, for example *Ciphertext Indistinguishability under Chosen Plaintext Attack (IND-CPA)* [20]. While cryptographic algorithms may be secure under game-based attacker models, these models make assumptions that do often not hold entirely. For example, it is assumed that keys are kept perfectly secret and are only known to the owner which only holds under other assumptions, e.g. hardness of factorization or $P \neq NP$.

In practice the security of a cryptographic system depends to a great extent on its robustness against attacks at system and software level, aiming at retrieving information about secret keys. For example, real-world attacks like Heartbleed [46] took encrypted communication of millions of users at risk because it was possible to read cryptographic key material directly from memory. Another example were keys must be protected but

might be exposed to attackers are payment or multimedia applications. First require that transaction data is kept secure whereas second often involve copyrighted media which needs to be protected against bootleggers. Such applications require solutions that even withstand attackers that are able to read and modify the entire execution environment at any point of execution.

This is where white-box cryptography comes in. In contrast to the black-box model, the white-box model assumes that an attacker has complete control over the environment and white-box cryptography aims to protect keys against this powerful attacker. Currently, all publicly available white-box cryptography schemes are broken. Additionally, attacks were published that enable to recover keys from white boxes without actually reverse engineering or analyzing the structure of the white boxes. Because white-box cryptography is used to protect sensitive information, first of all we tried to validate previous work and we were able to reproduce the attacks and the resulting key recoveries. Also we tried to port the attacks to the Android mobile operating system, focussing both on native libraries as well as white boxes implemented in Java. Our results are that attacks are easy in theory, but in practice we discovered some problems like the lack of appropriate tools on Android. A further result of our work is a Linux container holding the tools written by the authors of [7].

The remainder of this work is structured as followed: In Chapter 2 we describe related work. Chapter 3 gives an overview of the inner workings of white boxes based on public academic schemes for AES white boxes. Chapter 4 introduces *Differential Computation Analysis* as side-channel attack against white boxes and describes how the attack works. It also describes the toolchain used as well as the Docker environment we created in order to ease setup and automation of trace acquisition. Chapter 5 describes our attempts to attack white boxes written in Java as well as native libraries on Android. In Chapter 6 we draw conclusions of our work.

2. Related work

This chapter describes important concepts and technologies that enabled this work.

2.1. Obfuscation

Obfuscation describes the transformation of a program or parts of a program into a semantically equivalent program that is harder to reverse engineer. This is desired in order to protect intellectual property or to hide certain behaviour from analysts, e.g. a malicious behaviour which should not be detected. There are several obfuscation strategies, for example renaming identifiers, encrypting strings and decrypting them on demand, changing control flow when possible, replacing statements with semantically equivalent statements or inserting useless code. There has been a lot of work on this topic with some interesting results, for example that there are unobfuscatable function families regarding the definition of an obfuscator given by Barak et al. [3].

The field of obfuscation is related to *white-box cryptography (WBC)* because both try to protect information against a powerful attacker which is able to analyze and run code as often as desired and has complete insight at every execution step. White-box cryptography may be confused with an obfuscated version of black-box cryptography, but Saxena et al. point out that "obfuscation and WBC are two different concepts and should not be confused with each other" [38]. White-box cryptography has to be additionally secure under a predefined security notion, e.g. *ciphertext indistinguishability under chosen plaintext attack (IND-CPA)*, which is important for cryptographic purposes but not required for obfuscators in general. Other authors do not make this distinction and see white-box cryptography as kind of obfuscation, for example Chow et al.'s first approach to construct a white box [12] is called "obfuscated AES implementation" by Xiao et al. [49].

2.2. Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) is a technique to modify and analyze binaries at runtime. This has several use cases, for example DBI can be used to find memory leaks or other information leakage and help to improve security when developing applications. Also, runtime behaviour such as cache usage may be observed, resulting in information on hot spots in program execution and optimization capabilities. For analysts and attackers DBI is often used to get information on application internals, to gather information on internal data flow or to find bugs by fuzzing the application. Due to the capability to add code on demand, it is even possible to alter the execution, for example to add logging features or remove security features like TLS, time and ownership checks. Popular DBI frameworks include Valgrind, PIN, DynamoRIO and Frida. The relevance for this work is that DBI frameworks were used to observe program execution in order to attack white-box cryptography. This may have been possible using debuggers or virtual machines, too, but DBI frameworks are a good trade off between the possibilities of a virtual machine and debuggers. Virtual machines such as *qemu* allow complete insight into binary execution but are producing a lot of noise, meaning that big parts of collected information is unrelated to the binary trace. Also, VMs are slow, whereas debuggers like *gdb* are fast and scriptable when targeting specific information. A reason to not directly using debuggers is that binaries are often protected against debugging and gaining required information may be harder with debuggers because they are simply built to serve a different purpose. Also, due to overhead resulting from host communication, debuggers can in some cases also be quite slow, especially when creating full execution traces of binaries. A comprehensive introduction to DBI can be found in [34].

2.3. Side-channel attacks

The term *side-channel attack* refers to a class of attacks against cryptographic implementations, meaning that not the cipher itself, but a specific implementation is vulnerable to the attack. The basic idea is to observe the behaviour of an implementation regarding different inputs and outputs and to learn something about the key using this information. For example timing attacks try to identify key information based on knowledge that some operations take more time than others depending on the actual key, input and output data. Other side channels than timing include electromagnetic radiation, power consumption, acoustic noise or memory access. Side channels are not limited

to passive attacks only observing cryptographic operations. For example, *Differential Fault Analysis (DFA)* modifies intermediate states and analyzes error propagation in order to learn information about the key. Side-channel attacks work against software and hardware implementations and are popular especially for attacking HSMs such as TPMs or smart cards. A more extensive introduction to the topic can be found in [42] and [29].

2.4. Container-based virtualization

Contrary to "classical" *hypervisor-based virtualization* emulating complete hardware, *container-based virtualization* uses capabilities of the Linux kernel such as namespaces, control groups and mandatory access control to implement resource limiting and process separation. Because most hardware is not emulated[1] and the host kernel is reused, container-based virtualization has a more lightweight resource footprint. There are several tools like *Docker* [15] or *rkt* to manage creation and container maintenance. We used Dockers capabilities to build ready to use container images containing the toolchain used across this paper. Containers basing on these images can be used to analyze binaries and automate attacks. It is also possible to spawn multiple instances of the containers, so attacks can be run distributed and scalable. An overview about container-based virtualization can be found in [17].

[1] Some systems like network are still virtual in order to create stronger separation and ease network management

3. White box Overview

Complying to Kerckhoffs' open design principle is considered good practice for crypto systems [39, pp. 40,41]. This means that the security of the system must only depend on the strength of the (secret) key. If the specification of a crypto system is not publicly available, this is against Kerckhoffs' principle and called *security by obscurity* and is generally discouraged. In the past, ciphers that were not disclosed publicly were distributed to millions of users. Reverse engineering and analyzing them showed that they are weak and easy to break compared to modern, state of the art cryptography that has passed several reviews by experts. For example, the A5/1 cipher was used for several years to protect GSM communication. Not disclosing how the cipher works did not prevent Biryukov et al. [6] from breaking it.

Crypto systems are often used to protect information in untrusted environments such as machines controlled by adversaries. In this case, users in control of such a machine have the chance to observe keys in memory and are able to bypass the crypto system and access the unprotected information. A real world example where a similar incident happened and millions of Internet users where taken at risk was the Heartbleed [46] vulnerability in the widely used OpenSSL *Transport Layer Security (TLS)* implementation. Due to a incorrect size check an attacker was able to modify requests in order to receive some data from the server's memory. It was possible to receive secret key information of the server which is used to encrypt and authenticate TLS sessions. An attacker owning this key material could read and modify any encrypted traffic without being noticed by the victims.

Secret keys need to be protected. A common approach is to keep keys on *Hardware Security Modules (HSMs)* such as *Trusted Platform Modules (TPMs)* or smart cards. Those modules act like a black box from the users perspective, they are keeping the keys secret and providing cryptographic operations with those keys via an interface to legitimate users. An attacker trying to get the keys often needs physical access to the hardware to attack it. Figure 3.1 visualizes the concept of a black box. Taking a closer look, HSMs can only be considered grey boxes, meaning that they are protecting keys and internals but may be leaking other data that can be used to start side-channel attacks to find

Figure 3.1.: An adversary given the black box can only access the plaintext and ciphertext, the internals of the black box remain unknown.

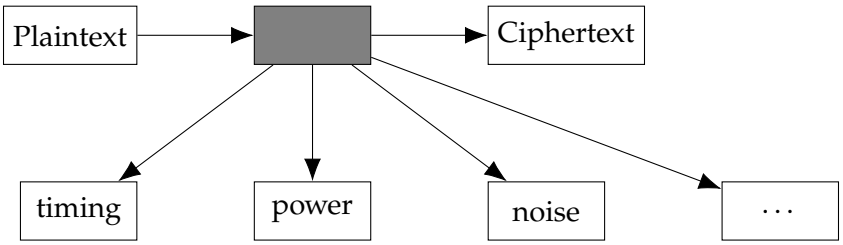


Figure 3.2.: An adversary given the grey box can access the plaintext and ciphertext. Besides, he is able to gain additional information by observing the grey box.

the keys. Side-channel attacks are used to collect information such as timing, power consumption, or electromagnetic radiation in order to gain knowledge of the key or at least parts of the key. This means that side-channel attacks are not used to attack the crypto system itself, but a specific implementation of the crypto system that is not side-channel resistant. The general idea is shown schematically in Figure 3.2.

The biggest disadvantage of HSMs is that they are often expensive and harder to deploy and to use compared to cryptography implemented in software. They are also usually far less flexible than software implementations. Cryptography implementations in software aiming to achieve the same security guarantees while being run in an environment controlled completely by an attacker are therefore called *white-box implementations*. From now on, the term *white box* refers to such an implementation aiming to be resistant against an attacker completely controlling the execution environment. Figure 3.4 shows a scheme of a white box consisting of lookup tables.

There are three areas of application for white boxes. First, white boxes as pseudo-asymmetric cryptography. If, for some reason, asymmetric cryptography is not possible, a white box may be used as public key. One of the reasons may for example be the higher performance of symmetric cryptography compared to asymmetric algorithms such as RSA. Second, to protect information like DRM content on an adversary machine. This makes it hard to steal the content while consuming is still possible. If

using asymmetric cryptography instead of a white box for this use case, the user still needs the key to view the media and the same key may be used to remove protection. White boxes aim to make key extraction difficult to impossible and are usually protected against misuse, so it should not be possible to use it to permanently remove the DRM protection. The third use case of white boxes is to protect used keys in memory. Applications performing cryptographic operations of course need the key which is usually stored unencrypted in memory and an attacker scraping memory for keys may be able to find them. For example, a hardware security module would have mitigated the Heartbleed bug and using a HSM on a web server is absolutely possible and only a matter of configuration time and acquisition costs. A white box might be a viable cheaper alternative. Securing these keys would definitely mitigate some attacks that try to read keys directly from memory and improve overall security of several applications.

Especially for mobile platforms such as smart phones, it is usual that Secure Elements [19] are available. Unfortunately it is often impossible for developers to access them since they are owned by another stakeholder, e.g. the network operator or the device vendor. So, economic factors may be another reason making the use of HSMs impossible in applications.

3.1. White boxes in practice

Today, there are academic schemes on how to construct white boxes from AES and DES ciphers. The basic design of the scheme from Chow et al. [12] and the improvements by Karroumi [26], Bringer et al. [8] and Xiao et al. [49] are presented later in this chapter. Additionally, there are white boxes available from *Capture the Flag (CTF)* and other reverse engineering contests. These are often constructed without having a mathematical background model and are mostly supposed to be cracked via reverse engineering or similar approaches by small teams within a few days. Table 3.1 gives an overview of all implementations that are publicly available and we are aware of. Table A.8 contains references to those white boxes. There are also companies selling their proprietary white-box technologies. Apart from marketing white papers, there is few information about the mathematical foundation of these white boxes available. Usually they are part of bigger software components and are additionally protected by conventional methods like obfuscation, checksums and anti-tampering measures. Companies known to sell such commercial white boxes include Gemalto,

Arxan, Whitecryption, Microsemi, Metaforic, and others. Table 3.2 lists all commercial solutions we found.

It seems that despite public research has little interest in white boxes for asymmetric ciphers, companies selling white boxes usually allow to use a lot of well-known ciphers such as RSA and some Elliptic Curve ciphers. Some vendors also implement white-box SHA or proprietary ciphers which are not explicitly listed. It remains unclear why a hashing algorithm like SHA is white-boxed, maybe salt values are directly integrated into the cipher or the hash function is used to calculate HMACs.

name	type	cipher	year
wbDES	Challenge	DES	2007
hack.lu2009	Challenge	AES-128	2009
SSTIC2012	Challenge	DES	2012
Klinec AES (C++)	Proof of Concept	AES-128	2013
Klinec AES (Java)	Proof of Concept	AES-128	2013
NoSuchCon2013	Challenge	AES-128	2013
PlaidCTF2013	Challenge	AES-128	2013
CHES2015	Challenge	AES-128	2015
OpenWhitebox AES	Proof of Concept	AES-128	2015
CHES2016	Challenge	AES-128	2016

Table 3.1.: Publicly available white-box implementations. *Challenge* means that the white box was supposed to be broken by reverse engineering. *Proof of Concept* means that the white-box implementation tries to be compliant to a publicly available white-box scheme

3.2. Inner workings of a white box

There are a few public academic white-box schemes. They describe the generation of white boxes that implement a specific cipher, usually AES or DES. This chapter gives an overview of the Chow et al. scheme which was the first public academic scheme. Furthermore, the Karroumi, Bringer et al. and Xiao et al. schemes improve the Chow scheme and the ideas behind them will be outlined, too.

name	company	cipher
Sentinel	SafeNet / Gemalto	unknown
TransformIT	Arxan	AES, RSA, DES, 3DES, ECC and more
Cryptanium Secure Key Box	Whitecryption	AES, RSA, DES, 3DES, ECC and more
WhiteboxCRYPTION	Microsemi	AES, RSA, DES, 3DES, ECC and more
WhiteBox	Metaforic	AES, RSA, ECC
Cloakware	Irdeto	AES, RSA, DES, 3DES, ECC and more

Table 3.2.: Commercial white-box implementations

The simplest version of a white box is a large lookup table mapping plaintexts to ciphertexts and back. This lookup table allows to do arbitrary cryptographic operations while being completely transparent to an attacker without leaking key information. Unfortunately, such a table is going to be extremely large, a cipher with a block size of 16 bytes (such as AES) requires more than $5.4 \cdot 10^{30}$ gigabytes of memory which is unfeasible nowadays and likely to be unfeasible in the future. Figure 3.3 visualizes the idea. The lookup table from the figure maps one byte plaintexts to ciphertexts and has already 256 entries. This big lookup table protects the key but is still vulnerable to attacks like code lifting and inversion that are described later in this chapter.

In order to shrink the size of this lookup table one possible solution is to split it into multiple reusable parts and encryption/decryption consists of following a path through these lookup tables. Figure 3.4 shows a scheme how such a white box may look like. The actual size of this white box actually depends on how many tables were generated which in turn depends completely on the scheme that is used.

3.2.1. Chow et al. scheme

In 2002, Chow et al. [12] published schemes in order to generate white boxes from AES-128 and DES. This chapter gives a general overview of the AES scheme, since DES itself is not considered secure anymore and AES is nowadays the standard symmetric encryption algorithm. Since the maths behind the creation of the structures forming the white box is out of scope of this work, the kind reader may directly refer to the original publication by Chow et al. [12]. Muir [31] describes the construction in a more

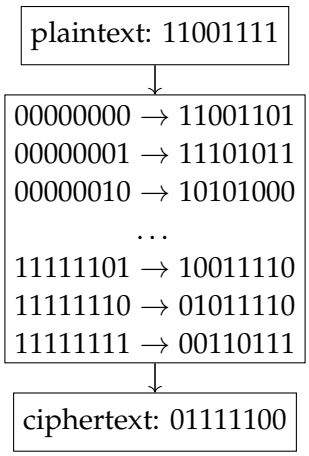


Figure 3.3.: Scheme of a white box realized with one big lookup table

detailed and illustrative way. Figure 3.5, taken from [31], shows a scheme of an inner AES round according to Chow et al. scheme.

The Chow et al. scheme transforms the four AES operations *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey* into equivalent bijections. The only place where the user-defined key is added are modified S-Boxes, called *T-Boxes*. Regular AES uses the S-Boxes in the SubBytes transformation. Muir shows that it is possible to rearrange the AES transformations in a way such that "we are able to combine `AddRoundKey`, `SubBytes` and part of `MixColumns` into a series of table look-ups " [31]. These lookups happen in the aforementioned T-Boxes. There are 160 8×8 T-Boxes and they are used to implement the combined AddRoundKey and SubBytes transformations. They are derived from the AES S-Box and are key-dependent. After the application of the T-Boxes, the MixColumns step is performed by applying T_{y_i} tables. These map 8-bit input to 32-bit output and it is possible to use them in order to perform a MixColumns transformation on 4 bytes by doing three XORs of four table lookups. There are four different T_{y_i} tables in each round and because of the ciphers block size of 16 bytes, each round requires four copies of each table. This makes 16 tables per AES round and 144 tables for a complete iteration since there are only nine rounds performing a MixColumns operation. The fact that data flows from T-Boxes into T_{y_i} tables directly every time allows to compose them and integrate T-Boxes into T_{y_i} tables directly, saving a little bit space and table lookups in total. Further it is possible to implement the XOR operations by using lookup tables. In general, one lookup table would be sufficient,

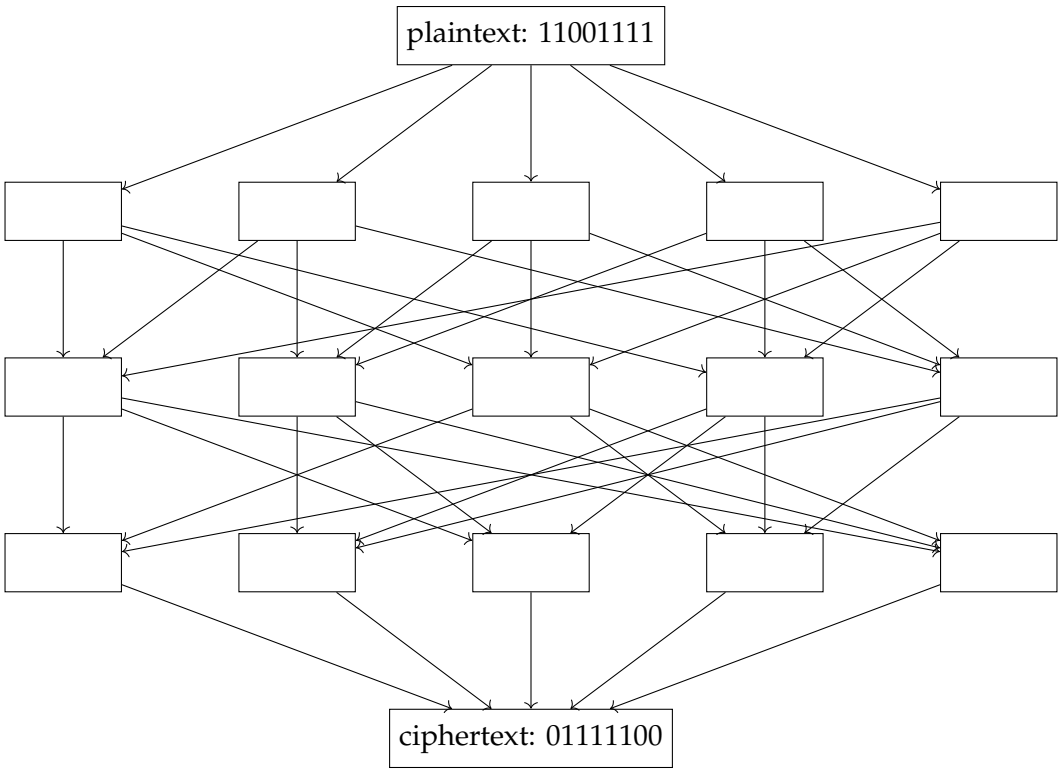


Figure 3.4.: Scheme of a white box consisting of many small lookup tables

but in order to protect the implementation in a subsequent step it is required to create multiple lookup tables. These map two 4-bit inputs to a 4-bit output. Each round requires twelve 32-bit XORs to perform MixColumns on 16 bytes since MixColumns on 4 bytes requires 3 32-bit XORs. This means that it is required to construct 96 8×4 bit XOR tables for the MixColumns step of a single round and 864 in total for a complete iteration.

This basic construction allows to perform operations equivalent to AES-128 with a given key. Unfortunately, an attacker in the white-box context can obtain information about the tables and easily recover the key used, therefore additional actions need to be taken in order to protect the key. *Internal encodings* are supposed to protect the lookup tables. These internal encodings are randomly generated bijections and their sole use is to obfuscate the lookup tables in order to make it harder for attackers to recover the key. Simply spoken, instead of doing a regular lookup X, it is obfuscated via X' where

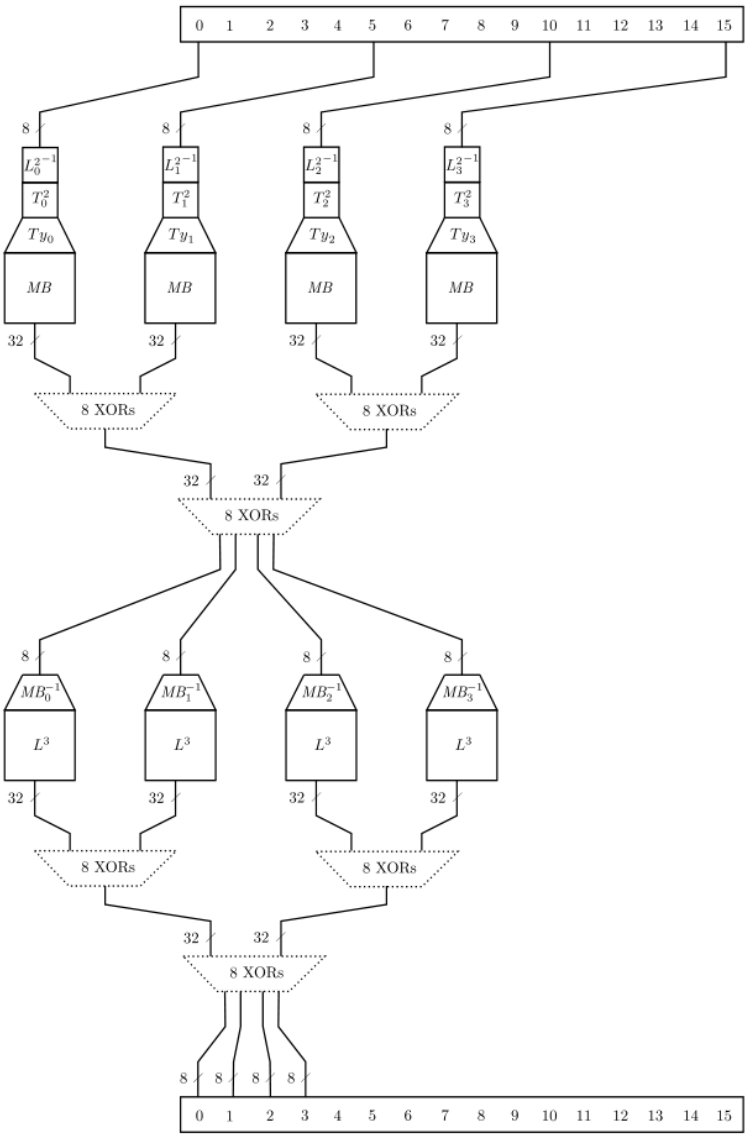


Figure 3.5.: Scheme of an AES round according to Chow et al.'s proposal, taken from Muir [31, p. 14]. The figure shows how rounds 2–9 look like.

$X' = F \circ X \circ G$ and the functions F and G are created randomly. Because X is never accessed directly, an attacker can not observe intermediate values directly. Additionally to internal encodings which provide additional *confusion*, so called *mixing bijections* are introduced in order to provide additional *diffusion* of data while processing. These bijections are also created randomly and need to have a few mathematical properties. They are applied at the beginning and end of round 2–9, the first round starts applying mixing bijections before passing data to the second round and the last round does not apply mixing bijections to the output. Generation and application of internal encodings and mixing bijections requires some preconditions, a detailed overview can be found in [12] and [31]. In order to further harden the white box, an external encoding is applied. It is generated randomly, too, and ensures that an attacker never accesses raw plaintext or raw ciphertext, making attacks harder. It is important that external encoding does not mean that it is necessarily applied outside of the attackers environment. Chow et al. even state that the external encoding removes "the manipulation which is performed elsewhere in the program" [12] and is therefore inside of the environment controlled by an attacker. Because the external encoding is no remote encoding and only adds further protection against reverse engineering, performing side-channel attacks still works.

3.2.2. Improvements to Chow et al. scheme

The original proposal by Chow et al. has been analyzed by several authors and some weaknesses were found and improvements proposed. Xiao et al. [49] propose a scheme with modified internal linear encodings. The modifications avoid weaknesses used by Billet et al. [4] for their attack against Chow et al.'s scheme. Xiao et al.'s improved scheme was broken by De Mulder et al. [13]. Bringer et al.'s [8] improvements using random perturbations was broken in [14]. Karroumi tries to strengthen the white box by introducing dual ciphers [26]. AES is an algorithm working on a finite Galois field and works per definition with some given constants. The idea behind the dual ciphers approach is to change these constants to get a cipher that works completely like AES but is based on different constants. Mapping inputs and outputs between those dual ciphers is possible because there exists a linear transformation accomplishing this. Building on top of this, a white box based on the scheme by Chow et al. is presented using different AES representations internally. De Mulder et al. show that "Karroumi's white-box AES implementation is vulnerable to the attack it was designed to resist" [32]. There are some other attacks discussed later. Currently, all academic schemes are broken.

3.3. Problems of white boxes

The following sections list current problems of white boxes. They need to be addressed in order to create secure white boxes that are easily usable.

3.3.1. Code Lifting

There are a few problems affecting white boxes especially or affecting white boxes more than traditional implementations. The first and most important practical problem is *code lifting*. This means that controlling a white box which is protecting a key allows an attacker to perform cryptographic operations without knowing the key, so knowledge of the key is not required anymore. The fundamental problem is that the user of a white box is not authenticated by the white box and therefore any adversary is able to use it for arbitrary cryptographic operations without further protection. The solution to this problem implemented in Hardware Security Modules (HSM) are secret PINs or passwords. If a user demands to use cryptographic capabilities of e.g. a smart card, he has to prove that he knows the secret PIN in order to proceed. This approach is difficult to implement in a white-box context because the attacker may be able to observe PIN processing. Additionally, it must be assumed that a valid user knowing the PIN might act maliciously making the protections applied by PIN usage useless. Protecting against code lifting is very difficult and requires to tie the use of the white-box implementation with other parts of the system the white box is embedded in. For example, the white box may run integrity checks against other components and interact with them in opaque ways making it harder for attackers to clone the white box and use it in different contexts. It is also possible to try to bind the use of a certain white box to a specific device. Another effective way to protect against code lifting is to use external encodings where a part of the encoding is outside of the attacker's scope. For example, encrypted data sent from a server is going to be decrypted by a white box. Instead of sending the raw encrypted data, it is mapped to other data by a random bijection as external encoding. An attacker does not know the bijection and is not able to find any coherence between plaintext and ciphertext since he is not able to access the ciphertext directly. This protection does not prevent a white box from being attacked via side-channel attacks such as *Differential Computation Analysis (DCA)* or *Differential Fault Analysis (DFA)* as seen later. After finding the key it is possible to recover the encoding, too.

3.3.2. Invertibility

As described earlier, white boxes may also be used as pseudo asymmetric cryptography. A big problem of this approach is that white boxes may be inverted by inverting the lookup tables. This means that a white box performing only encryption can be transformed in a white box performing decryption and vice versa without knowing anything about the actual cipher or key. Considering that in asymmetric cryptography, reverting the operations is supposed to be hard only given the public key, white boxes are not well suited for this purpose.

3.3.3. Slow operation

Modern cryptography is considered to be comparatively fast. For example, [21] has up to date information on protocol support and performance of TLS in modern applications and cites a Google engineer saying that "On [Google's] production front end machines, SSL/TLS accounts for less than 1% of the CPU load, less than 10 KB of memory per connection and less than 2% of network overhead". There are several reasons why cryptography is usually really fast these days, including ciphers designed to be not only secure, but also being implemented in an efficient manner. When announcing the AES competition in 1997, a required characteristic for all algorithm candidates was to implement it "securely and efficiently in a wide variety of platforms and applications (e.g., 8-bit processors, ATM networks, voice & satellite communications, HDTV, B-ISDN, etc.)" and "If one can also implement the algorithm efficiently in firmware, then this will be an advantage in the area of flexibility" [33]. Today, there are fine-tuned high speed implementations of AES and other symmetric ciphers and CPU instructions providing even more performance. Considering asymmetric cryptography having higher computation demands because of mathematical requirements, *Elliptic Curve Cryptography (ECC)* became more important over the last years because of smaller key sizes and faster processing compared to RSA. The possibility to work with other cryptographic systems like the Diffie Hellman Key Exchange on Elliptic Curves makes them even more attractive.

Regarding speed of white box implementations of ciphers, there is not much information publicly available. Chow et al. claim that their implementation has tables of size of 770048 bytes and requires 3104 lookups whereas the AES proposal's tables take only 4352 bytes and require about 300 operations [12]. The conclusion that the use of their implementation "does come at quite a substantial price" is therefore justified. Karroumi's Dual Cipher improvement to the Chow et al. scheme's "overall performance

is unchanged to previously proposed implementations“ [26]. We are not aware of any recent performance measurements of academic schemes as well as proprietary white boxes, but Cho et al. point out that “previously suggested WBC constructions are 12–55 times slower than the standard AES“ [25]. They also propose a new scheme combining a white box and a regular black box implementation and provide measurements showing that their scheme has nearly the same performance as the black-box implementation. To be fair, they assume a weakened white-box scenario where attackers do have unlimited access, but due to resource restrictions only extract a single trace using the white-box attacker model and try to extract key information from that single trace and additional black-box information collected as required.

3.3.4. Lack of public academic research

It has been more than a decade since the first publication on white-box cryptography by Chow et al. Yet, there is little public academic research on the topic and the number of academic white-box schemes is negligible. Also, most of the academic schemes build upon Chow et al.'s idea of transforming cryptographic operations into table lookups and improve the scheme partially to avoid weaknesses that have lead to attacks in the past. Only few authors use different approaches, like Biryukov et al. [5] or Bringer et al. [8] using multivariate polynomials. There are algebraic attacks against most of the academic schemes, so their security level can be considered weaker than the security level of the same ciphers in a black-box context. However, vulnerabilities such as Heartbleed [46] showed that keys in memory may be easily exposed to adversaries and the black-box assumption does not always apply. More academic work on secure white-box schemes and their implementation as well as reliable free/open source implementations is required for white-box cryptography to gain larger adoption.

3.3.5. Known attacks

The white-box scheme by Chow et al. and the Karroumi and Xiao et al. schemes based on it were already analyzed and there exists at least one specific attack against each of them. For the originally proposed scheme by Chow et al., the best known attack recovers the key in 2^{22} steps and the Karroumi scheme, intended as improvement to the Chow et al. scheme, is susceptible to this attack, too [30]. According to a summary by Cho et al., keys may be extracted from all current academic schemes within a minimum attack complexity of at least 2^{32} [25]. However, all these attacks required knowledge

of the construction and the inner workings of white boxes in order to find and attack weaknesses. It requires a lot of effort and knowledge that may be gained only by reverse engineering. We are currently not aware of any cryptanalysis of a proprietary white-box scheme.

In 2015, Bos et al. [7] showed that publicly available white-box implementations are highly vulnerable to an attack called *Differential Computation Analysis* which is a side-channel attack against memory access traces captured from executions of a white-box. This attack was the inspiration for this work and porting it to Android was one of the goals of this work. On Black Hat Europe 2015, Sanfelix et al. showed another attack [36] based upon Differential Fault Analysis. This approach seems to be more effective against white boxes using internal encodings but requires an attacker to actively modify intermediate values, whereas Differential Computation Analysis works completely passive. Chapter 4 gives further information about those side-channel attacks against white-box cryptography. Interestingly, some authors point out that they had success in breaking proprietary white-box schemes using these side-channel attacks [7, 36].

4. Differential Computation Analysis

A side-channel attack is an attack against an implementation of a cryptographic algorithm. Therefore, it does not break the cryptographic primitive itself, but information making key recovery possible is leaked by the implementation. A concrete example is the side-channel attack against GnuPG, a widely used implementation of the OpenPGP [10] standard for E-Mail encryption/signatures. Genkin et al. [18] showed that it was possible to recover a 4096-bit RSA key only by recording the acoustic noise with a mobile phone next to the computer performing the operations. The key recovery took about an hour. Side-channel attacks work also against HSMs and are therefore applicable against hardware as well as software implementations. However, while the example above may be extreme, it is required for modern cryptography implementations to be resistant against side-channel attacks.

There are different types of side-channel attacks. They often target timing or power consumption side channels, but information may leak on any channel an attacker is able to observe. Further, side-channel attacks are not bound to be passive. For example, a *Differential Fault Analysis (DFA)* attack tries to gather additional information by injecting faults like flipped bits in order to obtain information about error propagation. This information may be used to get additional information to make attacks easier in some cases.

Bos et al. [7] show that most of the public available implementations of white boxes are vulnerable to an attack they call *Differential Computation Analysis (DCA)* which uses information about accessed memory and its content during the execution of the white box. Sanfelix et al. [36] show that DFA is also successful against these implementations. DFA can be more efficient compared to DCA regarding the number of traces if the faults are injected in the right places. Finding these places may however require additional reverse engineering. DCA is highly efficient compared to regular Differential Power Analysis attacks running against hardened HSMs or software implementations used in production. These require up to millions of traces and several hours to acquire them and run the attack whereas the attacks against the white boxes required at most around 2000 traces (most white boxes required far less traces) and acquiring the traces and

running the attack takes only a few minutes. The reason for this is that DCA traces do not contain signal noise, so values can always be observed precisely. DFA can also be highly effective against regular targets if the attack is optimized, but since it is an active attack it may be required to modify hardware. Both point out that they also successfully tested the attacks against commercial white-box products. Another interesting fact was presented by Sasdrich et al. [37], showing that regular white-box implementations on reconfigurable hardware is also vulnerable to classical side-channel attacks such as *Differential Power Analysis (DPA)*. This means that the idea to build a cheap(er) HSM by simply using a FPGA carrying a white box does not necessarily work.

4.1. Required steps for DCA attacks against white boxes and our experiences

This section sums up the steps required to attack an white-box implementation. The first step may sound trivial but we found that it is a lot of work: Finding an attackable white box. For us this was especially hard on Android because due to performance problems of Valgrind we were not able to create even a trace of a white box on Android. Also, all public attacks we are aware of using side channels are targeting academic implementations, CTF challenges [7, 36, 41] or self-written implementations [37]. Those are most of the time created for demo purposes, proof of concepts or to be cracked by reverse engineering within a few days and are alone-standing executables. Real world white boxes are usually integrated into bigger applications and driving execution to the white box while efficiently tracing memory access and input as well as output can be a time-consuming task, especially because it is likely that those white boxes are protected against attackers, for example by using DBI detection methods as described in [23].

Once a way to execute the white box has been found, one needs to create a software execution trace of the application/binary performing cryptographic operations using the white box. When analyzing the white box for the first time, one trace may be sufficient to identify the white box. This can be done by visually analyzing the trace. By now there is no way around doing this by hand, knowing characteristics of the cipher such as number of rounds and identifying these characteristics in the execution trace. For example, AES-128 has ten rounds, eight of them should look almost the same whereas the first and last round might look slightly different due to key expansion at the beginning and missing `MixColumns` in the last step. Also, external encodings may be directly applied to these rounds and make them look a bit different. Identifying the

white box enables us to limit the memory addresses getting traced to the minimum which is required to recover the key. This may for example be the complete white box or only an intermediate round value. As said before, white boxes are usually part of bigger systems and this step makes traces smaller, execution faster and the attack more precise since there is less unrelated data. An interesting open question is the possibility of hiding the white boxes in these visualizations. If it was harder for humans to find the white box in this visual trace, it may be required to attack the complete trace which may fail completely or results in higher computational resources required. Automating the white box detection using image recognition techniques or directly via analyzing the trace may be one of the future steps to make these attacks easier and more efficient. Figure 4.1, 4.2 and 4.3 show the zoomed white boxes from the software execution traces of three completely different white-box AES-128 implementations. The eight inner rounds can be identified since they look completely the same, first and last round may differ because of key expansion or missing MixColumns step.

After detecting the white box, it is required to create a lot of traces, if possible only focusing on the white box. The difficulty to automate this is depending on how good driving application execution to specific points is possible. This step may be time consuming, too, but it can be automated and scaled by parallel executions once implemented. After that, a side-channel attack on the crafted traces can be started. The idea of such an attack is to find a correlation between known input, output, accessed memory and the unknown key.

For example, one can target the outcome of the first encryption round. Because the structure of the cipher is known and information about e.g. input and the outcome of the first round are available, it is possible to assign every possible key byte a possibility of being the correct candidate. Repeating this step for multiple different inputs, some key bytes are more likely to be correct key bytes. Ideally, one key candidate gets a high probability assigned while the other candidates are less likely according to this model. Repeating this step for all key bytes leads to key recovery in the best case. Of course, this is only one example and there are different approaches targeting other information or recovering keys based on different assumptions. [42] and [29] give more comprehensive introductions to side-channel attacks. In the case of attacking white boxes, the side-channel attack may work for the first attempt or may not work at all, it is hard to know before actually trying it. It is also required to be automated because it may be a lot of computation work computers do better than humans.

If DCA is not successful, DFA may be worth a try. Sanfelix et al. [36] show that DFA requires less traces in some circumstances than DCA.

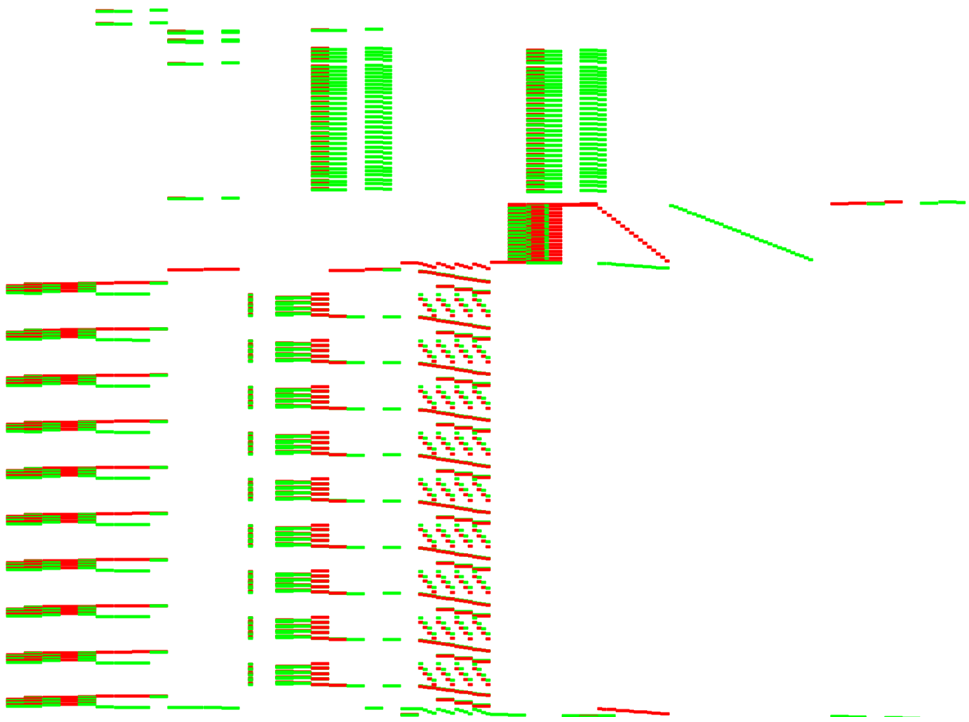


Figure 4.1.: Zoom to the memory trace of the of the hacklu2009 challenge white box, taken from the SCAMarvels Deadpool project

4.2. SCAMarvels toolchain

The toolchain used for the attacks against white box cryptography is called *Side Channel Marvels* [41] and consists of several standalone tools as well as DBI Framework plugins. The tools are written by the authors of [7] which introduced DCA against white boxes. The complete toolchain is released under a free licence (GPLv3), but some of the dependencies like Intel PIN are proprietary software. The Side Channel Marvels project consists currently of five parts: Tracer, Daredevil, Deadpool, JeanGrey and Orka. Orka is our contribution to the project.

Tracer contains plugins for DBI frameworks in order to generate traces from binary executions. Currently, there are plugins for Intel PIN and Valgrind. The PIN tool of Tracer has more options to fine tune the trace acquirement. There is also a simple

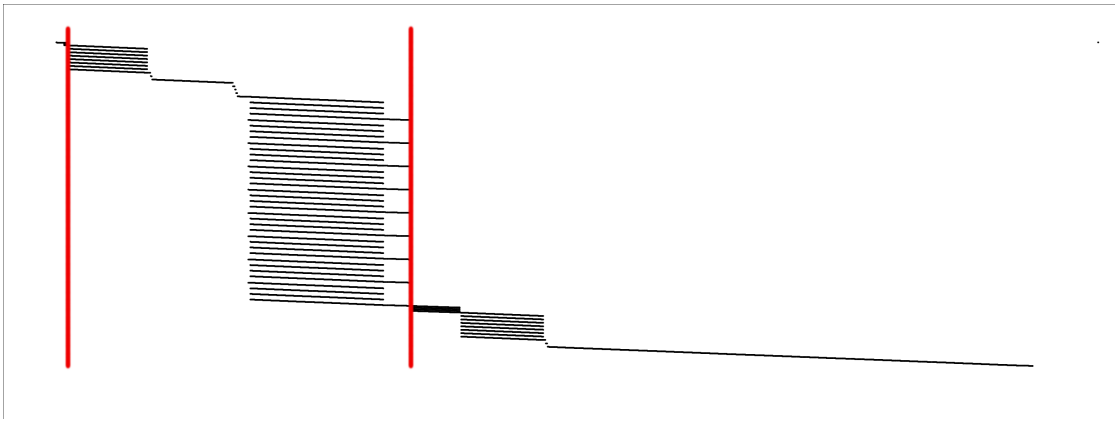


Figure 4.2.: Zoom to the memory trace of the Klinec C++ implementation white box, taken from the SCAMarvels Deadpool project

visualization tool called *tracegraph* which can be used to visualize the trace and find white boxes by visually identifying them. By now, there is no way to automatically detect white boxes so this is a step that needs to be done by hand. This step is necessary for an attack.

Daredevil is the tool used to perform the correlation power analysis side-channel attack. It is not limited to attacks against white boxes, so it can also be used to perform attacks against "classical" targets such as HSMs. For CHES2016 CTF, it is actually used to attack a hardware trace [11]. Currently there is support for DES and AES-128 and the tool is completely controlled via a configuration file.

Deadpool contains publicly available white boxes and attacks against them. This is mainly a proof of concept and for reference on how to use the tools. There are also some white boxes that are not broken via DCA but by reverse engineering or different approaches. It contains additional information and references to write ups, so the repository is worth a read when interested in practical attacks on white boxes. This is the biggest collection of white boxes on the Internet we are aware of.

JeanGrey is a tool performing Differential Fault Analysis attacks. Currently, AES is the only supported cipher and since the tool is quite young, there are only a few examples on how to use it.

Last but not least *Orka* are our Docker images. One result of this thesis is an environment for analyzing and attacking white boxes using the SCAMarvels toolchain. Since the SCAMarvels tools are independent programs written in different program-

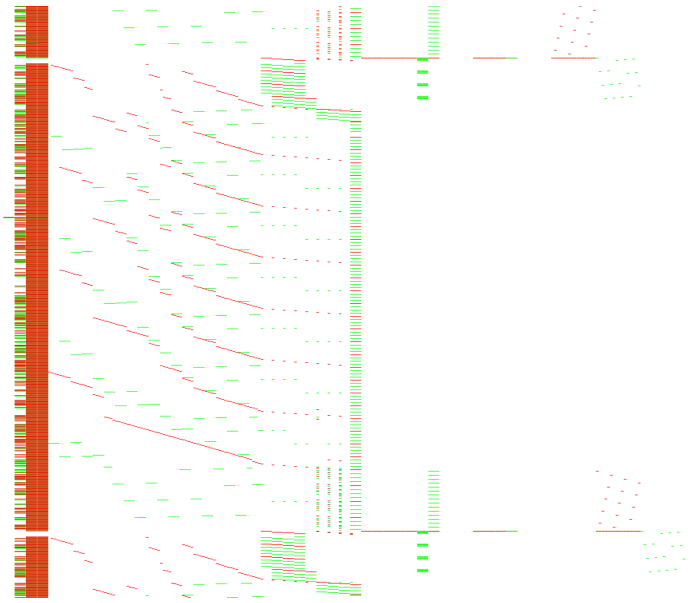


Figure 4.3.: Zoom to the memory trace of the plaidctf2013 challenge white box, taken from the SCAMarvels Deadpool project

ming languages and written and tested primarily on x86_64 Debian GNU/Linux, we created two Docker images which contain the tools. The reason for multiple Docker images is that there is a clear split between the dependencies of the tools, located in the *marvelsbase* image and the tools themselves, located in the *marvelslatest* image. In order to update the tools to their latest version, only the second image has to be rebuilt. This saves several hundreds of megabytes to download because dependencies such as Valgrind, PIN, compilers etc. are contained in the base image which need not be rebuilt and is much faster because tools like Valgrind are not recompiled every time. Caching the builds of compilers and DBI frameworks is also possible by using Docker's caching capabilities, but they do not allow to rebuild only the SCAMarvels tools without modifying the Docker file every time. Our solution allows to rebuild the SCAMarvels toolchain by running a single command. Originally the Docker images were created holding our own tools. When SCAMarvels got released in the course of this thesis, we modified our images to contain the SCAMarvels toolchain and proposed to the authors to contribute our work. This proposal was very well received and Orka is now officially part of the SCAMarvels project [43].

One of the main reasons for choosing Docker is that container-based virtualization has a much smaller resource footprint compared to hypervisor-based virtualization. Since there are efforts to make Docker run native on other operating systems such as Microsoft Windows or Mac OS X, users of these platforms can use these tools which are targeted at Linux, too. Also, providing Docker images allows to automate the build and setup completely. A user does not need to read documentation to get a working environment, instead everything is done by Docker when creating the image the first time. In the future, one could imagine other Docker images based on our images that target trace generation and the side-channel attack itself. This means that recovering keys from white boxes may be done automatically and in a scalable way since running several Docker containers is possible without a big performance loss. While working with this environment, it showed that it is also really well suited for development and cross compilation of the toolchain, which is required for example to get a working Valgrind build for Android.

Apart from these tools, the repository contains a lot of information, e.g. write ups on how to recover keys from white boxes or how to create traces on Android.

4.3. Reproducing academic side-channel attacks

We started implementing the approach from [7] and were able to reproduce their memory traces using a Valgrind plugin we wrote based on the Lackey [47] Valgrind plugin. Additionally we were working on a PIN plugin, too. In order to view the traces, we built a viewer using Python, numpy and matplotlib. These tools were already built inside a Docker container in order to keep them portable over different machines.

While working on the DCA implementation, the SCAMarvels toolchain was released. While these tools were more mature than ours, they lacked the container image. We ported the tools to our image, proposed it to the authors and the proposal got a warm welcome by them and is official part of the project now.

Using the SCAMarvels toolchain, all results from Bos et al. can be reproduced. Since the project is constantly growing, DFA attacks against some white boxes are already documented. Therefore, at least some results of Sanfelix et al. [36] can already be reproduced, too.

4.4. Considerations for use in practice

The basic goal is to recover keys from white boxes without any reverse engineering or further knowledge of the white box. In practice, this is supposed to be hard. For public white boxes, this may be most of the time feasible, but protected white boxes are often integrated into bigger components and tracing them may require more efforts and resources. Also, knowing which algorithm and its mode of operation is used is required in order to start an attack. If a white box manages to hide this information from an attacker, it is harder to identify the white box and attacking the traces takes more work since the attack has to be started against different algorithm candidates. Hertle shows that it is possible in several ways to detect if a DBI framework is currently attached to a process [23, 22]. Working around tracing protections implemented by white boxes may be additional time-consuming work. Additionally, creating software execution traces requires usually root permissions. A white box refusing to run on e.g. a jailbroken iPhone needs an extra workaround for this issue and summing all of the possible protections up, trace acquirement can get annoying and time consuming.

After all, a lot of these protections are security by obscurity and an attacker in the white-box context having enough resources to attack the white box is likely to overcome most of these protections, so creating an attack-resistant implementation is still required.

5. Extension of the attack to Java and native Android white boxes

Today, Android is the market leading mobile operating system. Even if Android is not powering all the devices, there is nearly always something Linux based running on ARM because those platforms provide a lot of benefits for manufacturers, e.g. being open and modifiable while they are cheap and do not require much energy. Because people are going for mobile solutions more and more, payment providers started to bring their services to phones or built completely new services filling market niches. Companies providing flat-rate media on demand were successful in the last years, too. The possibility to consume media anytime and anywhere on demand via mobile devices is used increasingly by customers.

For both of the presented scenarios, mobile payment and digital media, white-box cryptography is becoming progressively popular to achieve certain security goals: keeping money and payment information of involved parties safe in the payment scenario, protecting intellectual properties against bootleggers in the content on demand scenario. The main reasons for the use of white boxes in these scenarios are the high monetary value of the protected information as well as technical and user acceptance shortcomings when using an additional HSM.

The motivation behind adaptation of the previously discussed attack to Java and native Android is that real world applications today are already using white-box cryptography implementations which perhaps may be susceptible to Differential Computation Analysis as well.

5.1. Native Android

Usually cryptographic primitives are implemented in a fast, low level programming language such as C or C++ and compiled to machine code in order to run them. Typically this applies also to most white-box implementations used in wild. Therefore, generating memory traces from native libraries on Android was one of our primary

goals for this work. A native library on Android is similar to an ELF library on Linux. It usually contains code written in C or C++ which is called from the application via the *Java Native Interface (JNI)*. Apart from cryptographic code, this is often used for applications relying on performant graphic processing like games or when porting existing applications. Also, we are focusing on 32-bit ARMv7 architecture, since most Android smartphones are powered by this architecture for the last few years and other architectures, for example x86 or ARM64 are not very common by now. Based on our experience, most apps available on the Google Play Store contain at least shared libraries for ARMv7 targets and therefore the ability to trace them is the most interesting to us.

5.1.1. Available dynamic binary instrumentation tools

When starting this work, there were no publicly available tools suitable for generation of memory traces. Bos et al. describe that they used PIN and Valgrind for their measurements [7], so we wrote our own plugins in order to create memory traces. For Valgrind, we developed a plugin based on the Lackey Valgrind tool to instrument binaries and to extract information on memory accesses, such as size and access modes. The memory tracer plugin we used for PIN was based on another memory tracer written by Philippe Teuwen, one of the authors of [7], for a writeup where he shows that their attack works against obfuscated cryptography, too [44].

In order to visualize the results, we wrote a simple visualization tool based on Python and matplotlib. With these tools we were able to reproduce the results from [7] according to trace generation and visualization. While we were working on the actual attack, the authors of [7] released their tools [41] which were more mature than ours, under a free licence, so we decided to switch to their toolchain which is described in greater detail in Section 4.2.

Intel's PIN is a dynamic binary instrumentation framework consisting of a proprietary core and some free tools for reference to build more sophisticated tools on top of them. For our target architecture, Android on ARMv7, PIN does not work since it is written by Intel and it seems that they are only interested in providing support for their own architectures. Valgrind is a free dynamic binary instrumentation framework using its own virtual machine and representation of instructions. It is ported to many platforms, including our target architecture. Also, writing a Valgrind plugin is not that straightforward than writing a PIN plugin, but there are more ready-to-use plugins

such as *memcheck* for finding memory leaks or *cachegrind* to analyze use of caches in order to further optimize programs.

There exist a lot more DBI frameworks, in our opinion Frida and DynamoRIO are mentionable, too. Frida is a young, free framework injecting a V8 Javascript engine to the target process. Frida runs on all platforms the V8 also runs on, making it interesting especially for dynamic binary instrumentation on Apple's iOS, but all common platforms are supported. Since the project is quite young, some APIs are not stable or even implemented and we did not further investigate in using it. Instrumentation performed by Frida is primarily based on setting hooks making it harder to acquire DCA traces, but performing DFA with Frida works fine. A proof of concept can be found in [35]. DynamoRIO is an open source binary instrumentation framework which has been around for a longer time and seems mature. Some of its tools such as *Dr.Memory* are popular and it supports Windows and Linux on Intel and ARM architectures, Mac OS X support is still in development. We did not write a memory tracer tool for DynamoRIO nor Frida because this would have exceeded the time frame for this work. Since Valgrind, does not support attaching to already running processes and we had some trouble with it on Android, a memory tracer tool for DynamoRIO or Frida may nevertheless be useful. For now, DynamoRIO seems to be a better option because it has been around for some years, but Frida is under heavy development and some of its advantages like the platform independence allowing also iOS instrumentation may pay off in the future.

5.1.2. Valgrind on Android

In order to run Valgrind on Android, it is required to obtain a binary cross-compiled for Android. Supporting Android seems not to be one of the main goals of the project, their build and installation instructions only cover Android up to version 4.1 on ARM [48]. The SCAMarvels repository containing the toolchain developed by Bos et al. contains instructions on how to get a Valgrind build for Android [45], we managed to reproduce those builds and contributed with some instructions on how to instrument Android apps and troubleshooting for errors we discovered during our work.

Once an Android build of Valgrind is available, tracing regular binaries on Android like core utils such as *ls*, *md5sum* or *grep* works the same as on Linux. We also installed a busybox on our test device and instrumenting the tools provided by the busybox worked fine. Because Valgrind runs code in its own virtual machine, it is required that an application runs completely in Valgrind from the beginning. Attaching to an

already running process is not possible with Valgrind. Unfortunately, starting an app on Android works slightly different than regular execution of a binary on Linux. On Linux, it is common that execution of a binary forks the executing process using the *fork* system call. After that, an *exec* system call copies the relevant information into memory of the process and takes care of additional steps that are required, e.g. calling linkers. So, one possible way to run a binary instrumented by a DBI framework is to let the framework fork itself and execute the binary with the injected code. On Android, this is rather hard to achieve. Android has a process called *Zygote* which takes over several functions, including starting Android applications, starting the *system_server* process and handling RPC/IPC [16, p. 41]. Zygote is holding all relevant framework files and libraries in its memory. When starting an Android application, an intent is sent to Zygote and Zygote forks itself, loads classes and other information required by the app into memory and drops the permissions of the new process by switching the user to a non-privileged user[1]. This saves loading of framework files and other libraries, improving speed, saving resources and giving Zygote control over applications. For dynamic binary instrumentation of an Android app, an instrumentation tool has to either simulate the same behaviour in order to start an app or to instrument the complete Zygote process. Fortunately, Android provides a system property which can be used in order to define a wrapper command which is executed when an app is started. This way, an Android application can be started and instrumented by Valgrind. The concrete steps to instrument an Android application are described in [45].

Our tests were primarily made with a Nexus 6 on a rooted Android 6 Stock ROM. We verified that the Valgrind builds and our method for starting apps with them work using the Nexus 6, a Nexus 4 on Android 4.4 Stock ROM and a Galaxy Nexus running CyanogenMod 11 which is based on Android 4.4 as well. When instrumenting an Android application with Valgrind, we discovered a few problems making it hard for now to instrument Android applications.

The first problem and probably the biggest issue is the bad performance of Valgrind. Since the complete app is instrumented and apps using white boxes are usually quite big and complex, some apps do not even start up completely, making them unusable. This is a problem completely related to Valgrind being poorly performing. On devices

[1] On Android, each app has its own unprivileged user in order to improve security and to improve application separation. Permissions given to apps are mapped to Unix groups when possible, the app user being member of the according groups. IPC is realized by sending Intents to another applications. Further information can be found in the Security overview page of the Android Open Source Project (AOSP) [40] or in [16].

having CPUs primarily targeted to low energy consumption this leads to really bad runtime. Unfortunately, we were not able to drive the execution to the use of a white box inside a real world app from Google's Play Store.

The second problem is a rather hypothetical one: In order to prevent code lifting attacks like described in Section 3.3.1, it is usual to implement countermeasures against attackers such as detection of rooted devices, anomaly checking and other anti-debugging tricks. [23] shows that there are several ways for a program to detect if it is currently instrumented. For example, a process instrumented by Valgrind has opened more file descriptors than a regular process, nearly all DBI frameworks load shared libraries which can be detected into the address space of a process. Also, measuring time is inaccurate which may be easily detected and time consumption of operations such as *malloc* is bigger. Demo code can be found in [22].

The third problem we discovered are Android libraries containing instructions unknown to Valgrind. In some cases, it seems that programs are doing this on purpose in order to find out which instructions are supported by a CPU [9], whereas sometimes this is indeed a lack of support by Valgrind preventing instrumented execution. We discovered different unhandled instructions when running apps (even those that do not use native code) and some command line tools such as *dex2oat*, but statically compiled binaries like busybox and other core utils tools had none of these instructions. We suspect Bionic, Android's libc implementation or Dalvik/ART runtime to be the source of these issues but we did not investigate further since these unhandled instructions mostly did not lead to an app crash, so we simply ignored them in our traces.

Since the time frame for this work was too small to start writing a new plugin for another DBI framework, collecting an execution trace of a real world Android white box was not possible. The only open source white box written in C/C++ available to us is Dušan Klinec's implementation [27]. We did not manage to get it running on Android because of its dependencies Boost and NTL. We did manage to create a Boost build for Android, but we found no way to get a running build of the NTL maths library on Android. We collected execution traces of regular binaries such as *ls* or *grep* on Android and Linux on x86_64 and compared their execution traces. Those look very similar between the architectures and we assume that a white-box implementation susceptible to a side-channel attack on x86 Linux will be also susceptible to the same attack on Android.

An alternative way to get an execution trace of a white box may be lifting the white box and create arbitrary execution traces from it. This makes sense if there is special interest in the key itself because a lifted white box can already be used for encryption or

decryption. The problem with this approach is that white boxes are usually protected against code lifting and such an attack would require massive reverse engineering. However, if such a lifted white box is available, it is easily possible to create a large number of execution traces within a short time frame which can be analyzed and probably be attacked. Because this is contentual out of scope of this work, we did not follow that path.

5.2. Overview of tested apps

Table 5.1 lists all apps we took a closer look at because we were suspecting white boxes inside. Only for a few apps we believe that white boxes are contained, most of the apps showed that they do not ship white boxes or were hiding them good enough to remain undetected.

Most of these apps were selected by hand from Google's Play store, searching for popular banking, payment or multimedia streaming applications. The apps are listed in Table 5.1. Also, we tried searching for information about customers of companies selling white boxes. If we found such information we took a look at their Android apps, if available. We also made a mass scan of a local app archive containing more than 18000 unique apps looking for shared libraries containing suspicious strings like *whitebox*, company names of white box sellers or product names of commercial white boxes. A lot of the findings where false positives because they contained amongst other things graphics libraries with method names containing the phrase *transformit* (which is also the name of a white-box solution by Arxan) or variable names defining a white-colored box called *whitebox*. Some of the apps used the same shared libraries, so most of the findings were false positives.

All manually found apps and the promising results from the mass scan where extracted with apktool [1]. We took a look at the shared libraries by hand and scanned the *smali* and other resource files for information indicating the use of a white box. We did not further analyze the applications, because the idea of the side-channel attack is to have as little knowledge as required. All reverse engineering steps were only done to justify our suspicions on apps containing white boxes and to filter out false positives.

package name	app name	version code	version name
com.coinbase.android	Coinbase	80	3.8.5
com.rtli.clipfish	Clipfish	61	3.7.10
tv.dailyme.android	dailyme TV	228	16.02.04
com.justwatch.justwatch	JustWatch	6412	0.6.41
de.maxdome.app.android	maxdome	20210006	2.2.1 (2015112416)
de.cellular.myvideo	MyVideo	13	2.0.3
de.telekom.as.mywallet	MyWallet	200114	2.1.0.114_DE
com.netflix.mediaclient	Netflix	6233	4.4.0 build 6233
com.nousguide.android.rbtv	Red Bull TV	1000003	3.7.0.1
de.sky.bw	Sky Go	69078	1.5.3
de.rtli.tvnow	TV NOW	9	1.0.1
com.wu.mobapp.de	Western Union	2	1.1
air.nn.mobile.app.main	NeuroNation	2002020	2.2.20
com.adobe.air	Adobe AIR	19020190	19.0.0.190
com.tivophone.android	TiVo	16709	3.1.0-841080
com.apple.android.music	Apple Music	194	0.9.6
deezer.android.app	Deezer Music	705	5.3.0.133
com.google.android.music	Google Play Music	2513	6.5.2513X.2681020
com.rhapsody.napster	Napster	348	5.3.1.348
com.spotify.music	Spotify	11013025	4.9.0.992

Table 5.1.: All applications that have been analyzed by hand. Only a few of them met our conditions described in 5.2 to make us believe a white box is contained. We did not further investigate on them because they were only used for testing our approach.

5.3. Java

We were also interested in Java white boxes. Java is the programming language most Android applications are written in because it is the official application programming language on Android. Java is platform independent and popular in several other fields, e.g. web services or large business applications. The language is compiled to a bytecode which can be run in the so called *Java Virtual Machine (JVM)*. There are several different implementations of JVMs, the specifications of the Java language including the JVM specification can be found at [24]. On Android, the Dalvik virtual machine was used until Android 4.4. Starting with Android 5, Google enabled the Android Runtime (ART) virtual machine as default runtime on Android. Google provides further information about both runtimes in [2]. Java is a language containing a garbage collector and is completely taking over memory management. The garbage collector is a module running periodically or on demand, e.g. requested by the application programmer to remove objects[2] or the system, e.g. on low memory. The virtual machine specification does not specify how memory management is implemented by a virtual machine, so different virtual machines may behave different regarding memory management.

The side-channel attack by Bos et al. [7] relies heavily on tracing memory accesses by white boxes and their targets were mostly written in low-level programming languages having a quite predictable behaviour regarding memory management. Java having a different memory management behaviour and being prevalent in the Android ecosystem seemed like an interesting target.

Currently we are not aware of any commercial white box implementations written in Java, but we found an open source implementation of a white box written in Java [28] on Github. If white box cryptography is getting more important in the future, there may be implementations in pure Java where this attack vector is of interest.

We took the Java white box and built an executable jar to create traces on Linux on a regular x86_64 system. We used Debian and OpenJDK 7, but gave all JVMs available on Debian a shot. Tables A.1 to A.7 give some detailed information on our results regarding the use of different JVMs. Unfortunately, we found out that tracing Java applications with our toolchain is not very effective. Java is doing memory management in a very flexible way, meaning that it is for example not possible to decide when memory is actually freed. Also, the server VM is applying on the fly optimizations. This means that code is optimized while running instead of while compiling and there are no

[2] Interestingly, the garbage collector is non-deterministic and requesting garbage collection does not necessarily mean that objects are removed from memory

guarantees of specific optimizations happening at a certain point of execution. Since we are tracing the complete JVM process, the memory trace is polluted with all of these additional information. The two resulting main problems are that traces get really big really fast and that it may be significantly harder to start an automated attack because the memory trace of the actual white box may occur at different memory locations and different moments in different traces. At this point we stopped our investigation in the topic because we believe that there are easier ways to obtain this information than DBI-Frameworks like Valgrind or PIN. Possibilities include debuggers, DBI frameworks aiming to instrument JVM code or even a modified JVM. Bos et al. point out that they had an overhead problem with a Python white box, too, but modifying the interpreter worked fine [7].

6. Conclusion

The two goals of this work were to validate the results of Bos et al. [7] and adapt the attack to the Android operating system.

We were able to reproduce the results from Bos et al. and contributed our own work, two Docker images for easy development and testing of the toolchain as well as an environment for analyzing and automate attacks against white boxes. We also added instructions and troubleshooting information for running instrumented apps on Android to the project wiki. Further, we evaluated the feasibility of the attack on Android. The current toolchain lacks capabilities required for efficiently tracing native libraries on Android running on ARMv7. However, this is something that can be overcome and our results show that traces on Android do not differ in a way that an attack would not be possible. We also investigated Java white boxes, but it clearly showed that low-level DBI is likely to be the wrong approach here. For interpreted languages or programs running in a VM like Java, modifying the interpreter/VM or working with higher-level hooking or debugging techniques may be successful.

Mobile platforms are usually protected through several mechanisms against attackers. In order to mount such an attack, system-level permissions are required and applications have a lot of possibilities to protect themselves against being traced, on the other hand an attacker investing enough resources will still be able to obtain the required information.

This leads to our conclusion that, for now, white boxes on mobile platforms are a feasible way to protect short-term keys for e.g. live-streams of sports events or data that needs only temporarily protected, e.g. keys residing in memory.

A. Appendix

VM name	`java -$vmname -version`
server	OpenJDK 64-Bit Server VM (build 24.95-b01, mixed mode)
zero	OpenJDK 64-Bit Zero VM (build 24.95-b01, interpreted mode)
jamvm	JamVM (build 2.0.0, inline-threaded interpreter)
avian	Avian (build 1.1.0, package 1.1.0-4)
dcevm	Dynamic Code Evolution 64-Bit Server VM (build 24.60-b09-dcevm-full, mixed mode)

Table A.1.: Java virtual machine versions. The Java version used for the tests was 1.7.0_101

regular execution (mm:ss)	instrumented execution (mm:ss)	trace size (bytes)
0.075	2:08.58	3490097152
0.074	2:07.27	3490027520
0.074	2:10.29	3489990656
0.074	2:09.62	3490107392
0.075	2:08.21	3490051072
0.076	2:08.00	3490053120
0.076	2:09.90	3490009088
0.075	2:07.30	3489948672
0.073	2:06.42	3490028544
0.076	2:06.90	3490012160

Table A.2.: `server` VM tested with a Hello World program without Instrumentation and with PIN Tracer attached on a Intel Core i7-4770HQ CPU.

regular execution (mm:ss)	instrumented execution (mm:ss)	trace size (bytes)
0.071	4.221	27648
0.070	3.648	27648
0.069	3.822	27648
0.069	3.720	27648
0.068	3.680	27648
0.067	3.635	27648
0.070	3.665	27648
0.069	3.760	27648
0.072	3.744	27648
0.075	3.648	27648

Table A.3.: `zero` VM tested with a Hello World program without Instrumentation and with PIN Tracer attached on a Intel Core i7-4770HQ CPU. The traces are always identical and the fact that `jamvm` had similar results leads us to the assumption that something with PIN or our tracer plugin is wrong but we did not further investigate.

regular execution (mm:ss)	instrumented execution (mm:ss)	trace size (bytes)
0.054	1.768	27648
0.049	1.727	27648
0.046	1.767	27648
0.048	1.720	27648
0.050	1.733	27648
0.048	1.696	27648
0.048	1.716	27648
0.047	1.795	27648
0.048	1.719	27648
0.047	1.715	27648

Table A.4.: `jamvm` VM tested with a Hello World program without Instrumentation and with PIN Tracer attached on a Intel Core i7-4770HQ CPU. The traces are always identical and the fact that `zero` had similar results leads us to the assumption that something with PIN or our tracer plugin is wrong but we did not further investigate.

regular execution (mm:ss)	instrumented execution (mm:ss)	trace size (bytes)
0.148	18.610	268052480
0.145	18.855	268052480
0.143	18.896	268052480
0.149	18.752	268055552
0.141	18.848	268052480
0.148	18.821	268052480
0.144	19.021	268052480
0.145	19.000	268052480
0.145	18.704	268052480
0.142	18.749	268052480

Table A.5.: `avian` VM tested with a Hello World program without Instrumentation and with PIN Tracer attached on a Intel Core i7-4770HQ CPU.

regular execution (mm:ss)	instrumented execution (mm:ss)	trace size (bytes)
0.073	2:02.28	3399974912
0.071	2:03.86	3399981056
0.074	2:04.85	3399924736
0.072	2:05.42	3400073216
0.072	2:05.85	3400027136
0.074	2:04.12	3400188928
0.074	2:05.50	3400113152
0.071	2:03.34	3400050688
0.073	2:04.97	3400006656
0.070	2:03.00	3399981056

Table A.6.: `dce` VM tested with a Hello World program without Instrumentation and with PIN Tracer attached on a Intel Core i7-4770HQ CPU.

vm name	table generation (ms)	encryption (ms)
server	1181	6.5
zero	15364	21
jamvm	7657	11
avian	crash	crash
dcevm	1176.5	6.5

Table A.7.: Average time spent on AES table generation and encryption of the AES Java white-box [28] based on 10 executions on Intel Core i7-4770HQ CPU. The numbers do not include VM startup/teardown.

name	URL
wbDES	`http://www.whiteboxcrypto.com/challenges.php`
hack.lu2009	`http://2009.hack.lu/index.php/ReverseChallenge`
SSTIC2012	`http://communaute.sstic.org/ChallengeSSTIC2012`
Klinec AES (C++)	`https://github.com/ph4r05/Whitebox-crypto-AES`
Klinec AES (Java)	`https://github.com/ph4r05/Whitebox-crypto-AES-java`
NoSuchCon2013	`http://seclists.org/fulldisclosure/2013/Apr/133`
PlaidCTF2013	`http://shell-storm.org/repo/CTF/PlaidCTF-2013/Binary/drmless-250/`
CHES2015	`https://www.cryptoexperts.com/ches2015/challenge.html`
OpenWhitebox AES	`https://github.com/OpenWhiteBox/AES`
CHES2016	`https://ctf.newae.com/flags/`

Table A.8.: References to publicly available white-boxes. Accessed on 23.5.2015

List of Figures

List of Tables

Bibliography

[1] *apktool project homepage*. URL: http://ibotpeaches.github.io/Apktool/ (visited on May 13, 2016).

[2] *ART and Dalvik | Android Open Source Project*. URL: https://source.android.com/devices/tech/dalvik/ (visited on May 13, 2016).

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. "Advances in Cryptology — CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings." In: ed. by J. Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. Chap. On the (Im)possibility of Obfuscating Programs, pp. 1–18. ISBN: 9783540446477. DOI: 10.1007/3-540-44647-8_1. URL: http://dx.doi.org/10.1007/3-540-44647-8_1.

[4] O. Billet, H. Gilbert, and C. Ech-Chatbi. "Selected Areas in Cryptography: 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers." In: ed. by H. Handschuh and M. A. Hasan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. Chap. Cryptanalysis of a White Box AES Implementation, pp. 227–240. ISBN: 9783540305644. DOI: 10.1007/978-3-540-30564-4_16. URL: http://dx.doi.org/10.1007/978-3-540-30564-4_16.

[5] A. Biryukov, C. Bouillaguet, and D. Khovratovich. *Cryptographic Schemes Based on the ASASA Structure: Black-box, White-box, and Public-key*. Cryptology ePrint Archive, Report 2014/474. http://eprint.iacr.org/. 2014.

[6] A. Biryukov, A. Shamir, and D. Wagner. "Fast Software Encryption: 7th International Workshop, FSE 2000 New York, NY, USA, April 10–12, 2000 Proceedings." In: ed. by G. Goos, J. Hartmanis, J. van Leeuwen, and B. Schneier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. Chap. Real Time Cryptanalysis of A5/1 on a PC, pp. 1–18. ISBN: 9783540447061. DOI: 10.1007/3-540-44706-7_1. URL: http://dx.doi.org/10.1007/3-540-44706-7_1.

[7] J. W. Bos, C. Hubain, W. Michiels, and P. Teuwen. *Differential Computation Analysis: Hiding your White-Box Designs is Not Enough*. Cryptology ePrint Archive, Report 2015/753. http://eprint.iacr.org/. 2015.

[8] J. Bringer, H. Chabanne, and E. Dottax. *White Box Cryptography: Another Attempt*. Cryptology ePrint Archive, Report 2006/468. http://eprint.iacr.org/. 2006.

[9] *Bug 344802 - disInstr(arm): unhandled instruction: 0xEC510F1E*. URL: https://bugs.kde.org/show_bug.cgi?id=344802 (visited on May 12, 2016).

[10] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. *OpenPGP Message Format*. RFC 4880. http://www.rfc-editor.org/rfc/rfc4880.txt. RFC Editor, Nov. 2007. URL: http://www.rfc-editor.org/rfc/rfc4880.txt.

[11] *CHES2016 CTF Wiki*. URL: https://wiki.newae.com/CHES2016_CTF#Converting_Traces_to_Binary_Format_.28used_by_Daredevil_attack.29 (visited on June 7, 2016).

[12] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot. "Selected Areas in Cryptography: 9th Annual International Workshop, SAC 2002 St. John's, Newfoundland, Canada, August 15–16, 2002 Revised Papers." In: ed. by K. Nyberg and H. Heys. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. Chap. White-Box Cryptography and an AES Implementation, pp. 250–270. ISBN: 9783540364924. DOI: 10.1007/3-540-36492-7_17. URL: http://dx.doi.org/10.1007/3-540-36492-7_17.

[13] Y. De Mulder, P. Roelse, and B. Preneel. "Selected Areas in Cryptography: 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15-16, 2012, Revised Selected Papers." In: ed. by L. R. Knudsen and H. Wu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Chap. Cryptanalysis of the Xiao – Lai White-Box AES Implementation, pp. 34–49. ISBN: 9783642359996. DOI: 10.1007/978-3-642-35999-6_3. URL: http://dx.doi.org/10.1007/978-3-642-35999-6_3.

[14] Y. De Mulder, B. Wyseur, and B. Preneel. "Progress in Cryptology - INDOCRYPT 2010: 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15, 2010. Proceedings." In: ed. by G. Gong and K. C. Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. Chap. Cryptanalysis of a Perturbated White-Box AES Implementation, pp. 292–310. ISBN: 9783642174018. DOI: 10.1007/978-3-642-17401-8_21. URL: http://dx.doi.org/10.1007/978-3-642-17401-8_21.

[15] *Docker Homepage*. URL: https://www.docker.com/ (visited on June 9, 2016).

[16] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski. *Android Hacker's Handbook*. Wiley, 2014. ISBN: 111860864X.

[17] M. Eder. *Hypervisor vs. Container-based virtualization*. To appear in: Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Winter Semester 2015/2016, Chair for Network Architectures and Services, Department of Computer Science, Technische Universität München, Munich, Germany.

[18] D. Genkin, A. Shamir, and E. Tromer. *RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis*. Cryptology ePrint Archive, Report 2013/857. `http://eprint.iacr.org/`. 2013.

[19] *GlobalPlatform made simple guide: Secure Element*. URL: `https://www.globalplatform.org/mediaguideSE.asp` (visited on June 7, 2016).

[20] S. Goldwasser and S. Micali. "Probabilistic encryption." In: *Journal of computer and system sciences* 28.2 (1984), pp. 270–299.

[21] I. Grigorik. *Is TLS fast yet?* URL: `https://istlsfastyet.com/` (visited on May 9, 2016).

[22] S. Hertle. *DBI detection*. URL: `https://github.com/svenhertle/dbi-detection` (visited on May 13, 2016).

[23] S. Hertle. *Detection of Dynamic Binary Instrumentation*. not published, Seminar Reverse Engineering, Winter Term 2015/16 TUM. 2016.

[24] *Java Language and Virtual Machine Specifications*. URL: `https://docs.oracle.com/javase/specs/` (visited on May 13, 2016).

[25] K. Y. C. Jihoon Cho and D. Moon. *Hybrid WBC: Secure and efficient encryption schemes using the White-Box Cryptography*. Cryptology ePrint Archive, Report 2015/800. `http://eprint.iacr.org/`. 2015.

[26] M. Karroumi. "Information Security and Cryptology - ICISC 2010: 13th International Conference, Seoul, Korea, December 1-3, 2010, Revised Selected Papers." In: ed. by K.-H. Rhee and D. Nyang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Chap. Protecting White-Box AES with Dual Ciphers, pp. 278–291. ISBN: 9783642242090. DOI: `10.1007/978-3-642-24209-0_19`. URL: `http://dx.doi.org/10.1007/978-3-642-24209-0_19`.

[27] D. Klinec. *Whitebox-crypto-AES*. URL: `https://github.com/ph4r05/Whitebox-crypto-AES/` (visited on May 13, 2016).

[28] D. Klinec. *Whitebox-crypto-AES-java*. URL: https://github.com/ph4r05/Whitebox-crypto-AES-java (visited on May 13, 2016).

[29] T.-H. Le, C. Canovas, and J. Clédière. "An Overview of Side Channel Analysis Attacks." In: *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*. ASIACCS '08. Tokyo, Japan: ACM, 2008, pp. 33–43. ISBN: 9781595939791. DOI: 10.1145/1368310.1368319. URL: http://doi.acm.org/10.1145/1368310.1368319.

[30] T. Lepoint, M. Rivain, Y. De Mulder, P. Roelse, and B. Preneel. "Selected Areas in Cryptography – SAC 2013: 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers." In: ed. by T. Lange, K. Lauter, and P. Lisoněk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. Chap. Two Attacks on a White-Box AES Implementation, pp. 265–285. ISBN: 9783662434147. DOI: 10.1007/978-3-662-43414-7_14. URL: http://dx.doi.org/10.1007/978-3-662-43414-7_14.

[31] J. A. Muir. *A Tutorial on White-box AES*. Cryptology ePrint Archive, Report 2013/104. http://eprint.iacr.org/. 2013.

[32] Y. D. Mulder, P. Roelse, and B. Preneel. *Revisiting the BGE Attack on a White-Box AES Implementation*. Cryptology ePrint Archive, Report 2013/450. http://eprint.iacr.org/. 2013.

[33] National Institute of Standards and Technology. *ANNOUNCING REQUEST FOR CANDIDATE ALGORITHM NOMINATIONS FOR THE ADVANCED ENCRYPTION STANDARD (AES)*. URL: http://csrc.nist.gov/archive/aes/pre-round1/aes_9709.htm (visited on May 9, 2016).

[34] N. Nethercote. "Dynamic binary analysis and instrumentation." PhD thesis. PhD thesis, University of Cambridge, 2004.

[35] *PlaidCTF2013 Frida DFA*. URL: https://github.com/SideChannelMarvels/Deadpool/tree/master/wbs_aes_plaidctf2013/DFA2 (visited on June 7, 2016).

[36] E. Sanfelix, C. Mune, and J. de Haas. *Unboxing the White-Box: Practical attacks against Obfuscated Ciphers*. Blackhat Europe. https://www.blackhat.com/docs/eu-15/materials/eu-15-Sanfelix-Unboxing-The-White-Box-Practical-Attacks-Against-Obfuscated-Ciphers-wp.pdf. 2015. (Visited on May 10, 2016).

[37] P. Sasdrich, A. Moradi, and T. Güneysu. *White-Box Cryptography in the Gray Box - A Hardware Implementation and its Side Channels*. Cryptology ePrint Archive, Report 2016/203. http://eprint.iacr.org/. 2016.

[38] A. Saxena, B. Wyseur, and B. Preneel. *White-Box Cryptography: Formal Notions and (Im)possibility Results*. Cryptology ePrint Archive, Report 2008/273. http://eprint.iacr.org/. 2008.

[39] K. Schmeh. *Kryptografie: Verfahren - Protokolle - Infrastrukturen (iX-Edition)*. 5., aktualisierte Auflage. dpunkt.verlag GmbH, Feb. 2013. ISBN: 9783864900150. URL: http://amazon.de/o/ASIN/3864900158/.

[40] *Security | Android Open Source Project*. URL: https://source.android.com/security/index.html (visited on May 12, 2016).

[41] *Side-Channel Marvels*. URL: https://github.com/SideChannelMarvels (visited on May 13, 2016).

[42] F.-X. Standaert. "Introduction to side-channel attacks." In: *Secure Integrated Circuits and Systems*. Springer, 2010, pp. 27–42.

[43] P. Teuwen. *Hiding your White-Box Designs is Not Enough, Troopers talk*. URL: https://speakerdeck.com/doegox/hiding-your-white-box-designs-is-not-enough-1?slide=63 (visited on June 7, 2016).

[44] P. Teuwen. *MoVfuscator Writeup*. URL: http://wiki.yobi.be/wiki/MoVfuscator_Writeup (visited on May 12, 2016).

[45] P. Teuwen and M. Eder. *TracerGrind on Android*. URL: https://github.com/SideChannelMarvels/Tracer/wiki/TracerGrind-on-Android (visited on May 12, 2016).

[46] *The Heartbleed Bug*. URL: http://heartbleed.com/ (visited on May 10, 2016).

[47] *Valgrind Lackey*. URL: http://valgrind.org/docs/manual/lk-manual.html (visited on June 10, 2016).

[48] *Valgrind README.android*. URL: http://valgrind.org/docs/manual/dist.readme-android.html (visited on May 12, 2016).

[49] Y. Xiao and X. Lai. "A Secure Implementation of White-Box AES." In: *2009 2nd International Conference on Computer Science and its Applications*. Dec. 2009, pp. 1–6. DOI: 10.1109/CSA.2009.5404239.