PH.D PROGRAMMEE
IN COMPUTER SCIENCE

CYCLE 31TH

PH.D THESIS

Analysis of Threats and Design Flaws in Hardware and Software Cryptographic Systems

SSD: INF/01

COORDINATOR OF THE PH.D PROGRAMME
Prof. Riccardo FOCARDI

SUPERVISOR
Prof. Riccardo FOCARDI

CANDIDATE
Francesco PALMARINI
Matricola 823027

Abstract

In the last two decades the use of cryptography in computer systems has constantly increased. Ranging from personal devices to cloud services and critical infrastructures, cryptography is pervasive and variegated. It is thus of crucial importance to conduct security evaluations of existing cryptographic design and implementations, both at the software level and based on secure hardware.

Since cryptography is closely related to the secure management of cryptographic keys, in this thesis we first investigate on the design and implementation of Java keystores, the standard password-protected facility to handle and securely store keys in Java applications. We define a general threat model and distil a set of significant security properties. We report on undocumented details about the implementation and disclose novel attacks and weaknesses in keystores that do not adhere to state-of-the-art standards or use obsolete and ad-hoc cryptographic mechanisms.

Typically, security sensitive applications adopt protected and tamper-resistant cryptographic hardware. To this end, we study the low-level `APDU` protocol used to communicate with PKCS#11 devices such as USB tokens, smartcards and Hardware Security Modules. We present a new threat model for the PKCS#11 middleware and discuss new attacks that exploit proprietary implementation weaknesses allowing attackers to leak, as cleartext, sensitive cryptographic keys in devices that were previously considered secure.

Complex cryptographic implementations is also found in the firmware of embedded systems and interconnected Internet-Of-Things (IoT) devices. The research for security flaws in the firmware by means of reverse-engineering can be blocked by security mechanisms which prevent unauthorized flash memory readout to protect the intellectual properties contained. In this work we present novel attacks for extracting the firmware from six protected microcontrollers of three different manufacturers and, in particular, we present a new voltage glitching technique for improving the attack performance. Finally, we conduct a thorough evaluation of the results against the state-of-the-art in voltage glitching.

Acknowledgements

First of all I would like to express my deep and sincere gratitude to my supervisor *prof.* Riccardo Focardi for his valuable and friendly guidance over the years. I owe a debt of gratitude to *prof.* Matteo Maffei and *prof.* Stefano Zanero who accepted to devote their time to reading and reviewing this work.

I am grateful to a number of people I had the pleasure to collaborate with during the past few years, in particular *dr.* Stefano Calzavara, *dr.* Graham Steel, *dr.* Marco Squarcina, *prof.* Flaminia Luccio, Claudio Bozzato and Mauro Tempesta. I thank Claudio, Marco and Mauro again as part of the c00kies@venice hacking team, along with Andrea Baesso, Francesco Benvenuto, Francesco Cagnin, Marco Gasparini, Andrea Possemato, and Lorenzo Veronese. All these people are not just colleagues and team mates, but first of all they are friends and comrades in our great adventures.

I would like to thank my soulmate Jessica who has always been ready to give me the strength to accomplish all of this. Finally, thanks to everyone who believe in me day-by-day and put their trust in me, in particular my parents and my family, my old friends Laura and Maria Luisa and every single wonderful individual for inspiring me and lighting up my life.

Without you this journey would not have been possible, so I dedicate this to you.

Contents

Preface

The work presented in this thesis is based on the research that I conducted during my Ph.D. studies in Computer Science at Università Ca' Foscari, Venezia from September 2015 to September 2018. In this time frame, my research has primarily focused on the topics of applied cryptography, security of embedded systems, network and systems security. My research has also developed after several collaborations and consultancies that I conducted for private companies.

Introduction

We are facing an incredibly fast phenomenon of digitalization and interconnection of everyday devices, ranging from computers and smartphones to critical devices such as control units of self-driving vehicles, health devices (*e.g.*, defibrillators, heart rate monitors, insulin pumps), payment and ID cards. We live in the so called Internet-Of-Things (IoT) era where devices, connected to each other, to the cloud and controlled by digital units permeate our lives and, at the same time, physically interact with humans. Digitalization has increased the need for technologies that enable for securely storing, managing and transmitting sensitive data as well as strong and reliable authentication mechanisms. As a result, cryptography has become a fundamental and pervasive component of the software and hardware development process. Cryptography is however still complex and variegate, often requiring engineers and developers to combine different algorithms, schemes and mechanisms in non trivial ways. Furthermore, the increasing market of small, relatively cheap, electronic devices imposes trade-off and limitations on the available resources, for instance associated to the energy consumption for battery-powered devices [6], making it inconvenient to adopt computation-intensive security mechanisms such as standard cryptographic protocols.

The attack surface of cryptographic implementations is quite large, ranging from an incidental bad parameter choice, to major design flaws, software bugs and low-level hardware attacks. As an example, in the last two decades several timing [58] and cache [85] side-channels targeting cryptographic software and libraries have been made public. In particular, the Spectre [59], Meltdown [64] and SGX [27] vulnerabilities affecting modern microprocessors proved that even sound cryptographic implementations can be successfully attacked. It is thus of crucial importance to conduct studies on existing cryptographic designs and implementations, accounting for the entire attack surface. In this thesis we address this tangled problem from three different perspectives: software, hardware and physical.

Software Keystores

First, we address the problem of the secure management of cryptographic keys in software which need to be protected and securely stored by applications. Leaking cryptographic keys, in fact, diminishes any advantage of cryptography, allowing attackers to break message confidentiality and integrity, to authenticate as legitimate users or impersonate legitimate services. Quoting [102], "key management is the hardest part of cryptography and often the Achilles' heel of an otherwise secure system".

In the recent years we have faced a multitude of flaws related to cryptography (*e.g.*, [18, 11, 69, 68]). Some of these are due to the intrinsic complexity of cryptography, that makes it hard to design applications that adopt secure combinations of mechanisms and algorithms. For example, in padding oracle attacks, the usage of some (standard) padding for the plaintext combined with a specific algorithm or mechanism makes it possible for an attacker to break a ciphertext in a matter of minutes or hours [122, 21, 11]. Most of the time this is not a developer fault as, unfortunately, there are well-known flawed mechanisms that are still enabled in cryptographic libraries. In other cases, the attacks are due to flaws in protocols or applications. The infamous Heartbleed bug allowed an attacker to get access to server private keys through a simple over-read vulnerability. Once the private key was leaked, the attacker could decrypt encrypted traffic or directly impersonate the attacked server [69]. Thus, breaking cryptography is not merely a matter of breaking a cryptographic algorithm: the attack surface is quite large and the complexity of low-level details requires abstractions. Crypto APIs offer a form of abstraction to developers that allows to make use of cryptography in a modular and implementation-independent way. The Java platform, for example, provides a very elegant abstraction of cryptographic operations that makes it possible, in many cases, to replace a cryptographic mechanism or its implementation with a different one without modifying the application code.

Crypto APIs, however, do not usually provide security independently of the low-level implementation: default mechanisms are often the weakest ones, thus developers have to face the delicate task of choosing the best mechanism available for their needs. For example, in the Java Cryptography Architecture (JCA), ECB is the default mode of operation for block ciphers [52] and PKCS#1 v1.5 is the default padding scheme for RSA [53], which is well know to be subject to padding oracle attacks [21]. Additionally, crypto APIs that promise to provide security for cryptographic keys have often failed to do so: in PKCS#11, the standard API to cryptographic tokens, it is possible to wrap a sensitive key under another key and then just ask the device to decrypt it, obtaining the value of the sensitive key in the clear [32], and violating the requirement that "sensitive keys cannot be revealed in plaintext off the token" [97].

In this thesis we analyse in detail the security of key management in the Java ecosystem and, in particular, of Java keystores. Password-protected keystores are, in fact, the standard way to securely manage and store cryptographic keys in Java: once the user (or the application) provides the correct password, the keys in the keystore become available and can be used to perform cryptographic operations, such as encryption and digital signature. The `KeyStore` Java class abstracts away from the actual keystore implementation, which can be either in the form of an encrypted file or based on secure hardware. As discussed above, this abstraction is very important for writing code that is independent of the implementation but developers are still required to select among the various keystore *types* offered by Java. Unfortunately, the information in the keystore documentation is not enough to make a reasoned and informed choice among the many alternatives. More specifically, given that the Java Keystore API does not provide control over the cryptographic mechanisms and parameters employed by each keystore, it is crucial to assess the security provided by the different implementations, which motivated us to perform the detailed analysis reported in this thesis. In fact, our work is the first one studying the security of keystores for general purpose Java applications.

We have estimated the adoption rate and analysed the implementation details of seven different Java keystores offered by the Oracle JDK and by Bouncy Castle, a widespread cryptographic library. Keystores are used by hundreds of commercial applications and open-source projects, as assessed by scraping the GitHub code hosting service including leading web applications servers and frameworks, *e.g.*, Tomcat [4], Spring [112], Oracle Weblogic [126]. Additionally, keystores have been found to be widespread among security-critical custom Java software for large finance, government and healthcare companies audited by the authors.

The security of keystores is achieved by performing a cryptographic operation C under a key which is derived from a password through a function F called Key Derivation Function (KDF). The aim of the cryptographic operation C is to guarantee confidentiality and/or integrity of the stored keys. For example, a Password-Based Encryption (PBE) scheme is used to protect key confidentiality: in this case C is typically a symmetric cipher, so that keys are encrypted using the provided password before being stored in the keystore. In order to retrieve and use that key, the keystore implementation will perform the following steps: (a) obtain the password from the user; (b) derive the encryption key from the password using F; (c) decrypt the particular keystore entry through C, and retrieve the actual key material. Notice that different passwords can be used to protect different keys and/or to achieve integrity. To prevent attacks, it is highly recommended that C and F are implemented using standard, state-of-the-art cryptographic techniques [73, 99].

Interestingly, we have found that the analysed keystores use very diverse implementations for C and F and in several cases they do not adhere to standards or use obsolete and ad-hoc mechanisms. We show that, most of the time, keystores using weak or custom implementations for the key derivation function F open the way to password brute-forcing. We have empirically measured the speed-up that the attacker achieves when these flawed keystores are used and we show that, in some cases, brute-forcing is three orders of magnitude faster with respect to the keystores based on standard mechanisms. We even found keystores using the deprecated cipher RC2 that enables an attacker to brute-force the 40-bit long cryptographic key in a matter of hours using a standard desktop computer.

Our analysis has also pointed out problems related to availability and malicious code execution, which are caused by *type-flaws* in the keystore, *i.e.*, bugs in which an object of a certain type is interpreted as one of a different type. In particular, by directly tampering with the keystore file, an attacker could trigger denial of service (DoS) attacks or even arbitrary code execution. Interestingly, we also found that the use of standard key derivation functions can sometimes enable DoS attacks. These functions are parametrized by the number of internal iterations, used to slow down brute-forcing, which is stored in the keystore file. If the number of iterations is set to a very big integer, the key derivation function will hang, blocking the whole application.

Cryptographic Hardware API

Cryptographic hardware such as USB tokens, smartcards and Hardware Security Modules has become a standard component of any system that uses cryptography for critical activities requiring a higher degree of protection compared to software-based solutions. Cryptographic hardware allows operations to be performed inside a protected, tamper-resistant environment, without the need for the application to access the (sensitive) cryptographic keys. In this way, if an application is compromised the cryptographic keys are not leaked, since their value is stored securely in the device.

Cryptographic hardware is accessed via a dedicated API. PKCS#11 defines the RSA standard interface for cryptographic tokens and is now administered by the Oasis PKCS11 Technical Committee [89, 90]. In PKCS#11, fresh keys are directly generated inside devices and can be shared with other devices through special key *wrapping* and *unwrapping* operations, that allow for exporting and importing keys encrypted under other keys. For example, a fresh symmetric key k can be encrypted (wrapped) by device d_1 under the public key of device d_2 and then exported out of d_1 and imported (unwrapped) inside d_2 that will perform, internally, the decryption. In this way, key k will never appear as cleartext out of the devices. One of the fundamental properties of PKCS#11 is, in fact, that keys marked as *sensitive* should never appear out of a device unencrypted.

In the last 15 years, several API-level attacks on cryptographic keys have appeared in literature [1, 22, 23, 26, 32, 39, 65]. As pioneered by Clulow [32], the attributes of a PKCS#11 key might be set so to give the key conflicting roles, contradicting the standard *key separation* principle in cryptography. For example, to determine the value of a sensitive key k_1 given a second key k_2, an attacker simply wraps k_1 using k_2 and decrypts the resulting ciphertext using k_2 once again. The fact that a key should never be used to perform both the wrapping of other keys and the decryption of arbitrary data (including wrapped keys) is not explicitly stated in the specification of PKCS#11 and many commercial devices have been recently found vulnerable to Clulow's attack [26].

In this thesis, we describe new attacks that work at a different API-level. The PKCS#11 API is typically implemented in the form of a middleware which translates the high-level PKCS#11 commands into low-level ISO 7816 Application Protocol Data Units (`APDUs`) and exposes results of commands in the expected PKCS#11 format. In our experiments, we noticed that this translation is far from being a 1-to-1 mapping. Devices usually implement simple building blocks for key storage and cryptographic operations, but most of the logic and, in some cases, some of the sensitive operations are delegated to the middleware.

We have investigated how five commercially available devices implement various security-critical PKCS#11 operations, by analysing in detail the APDU traces. Our findings show that APDU-level attacks are possible and that four out of the five analysed devices leak symmetric sensitive keys in the clear, out of the device. We also show that, under a reasonable attacker model, the authentication phase can be broken, allowing for full access to cryptographic operations. Interestingly, we found that most of the logic of PKCS#11 is implemented at the library level. Key attributes that regulate the usage of keys do not have any importance at the APDU-level and can be easily bypassed. For example, we succeeded performing signatures under keys that do not have this functionality enabled at the PKCS#11 level. For one device, we also found that RSA session keys are managed directly by the library in the clear violating, once more, the PKCS#11 basic requirement that sensitive keys should never leave the token unencrypted.

In this thesis we focus primarily on USB tokens and smartcards, so our threat model refers to a typical single-user desktop/laptop configuration. In particular, we consider various application configurations in which the PKCS#11 layer and the authentication phase are run at different privileges with respect to the user application. Protecting the PKCS#11 middleware turns out to be the only effective way to prevent the APDU-level attacks that we discovered, assuming that the attacker does not have physical access to the token. In fact, physical access would enable USB sniffing, revealing any key communicated in the clear from/to the token. Separating authentication (for example using a dialog running at a different privilege) offers additional protection and makes it hard to use the device arbitrarily through the PKCS#11 API. However, an attacker might still attach to the process and mount a Man-In-The-Middle attack at the PKCS#11 layer, injecting or altering PKCS#11 calls.

Physical Attacks in Embedded Systems

Side-channel attacks are considered among the most powerful physical attacks against embedded devices and secure (*e.g.*, smartcards) or specialized hardware (*e.g.*, FPGAs or ASICs). There exist two classes of side-channel attacks: passive and active [111]. Passive attacks exploit information that is spontaneously leaked by the device such as power consumption [28], timing information [58], electromagnetic emissions [41] or even acoustic emanations [8]. Active attacks (also known as fault injection attacks), instead, influence the system with internal or external stimuli. For instance, optical fault injection is a powerful technique that exposes the silicon to high intensity light sources, *e.g.*, laser and UV, to induce errors or tamper with the data. Since this technique involves decapsulating [110] the chip from its package, technical expertise and specialized equipment are required. Electromagnetic fault attacks

(EM-FI) avoid the need of chip decapsulation since faults are injected through the package using an EM injector [101]. However, some degree of specialized equipment, *e.g.*, a high precision positioning system [84] or an RF amplifier, can still be necessary to conduct complex attacks.

Given the level of sophistication required by some fault injection attacks, capabilities and performance are not the only relevant factors for classifying and evaluating them: the cost also plays a crucial role. In [12] Barenghi *et al.* consider as *low cost* the injection methods requiring less than $3000 of equipment, which are within the means of a single motivated attacker. The authors point out that "these fault injection techniques should be considered as a serious threat to the implementations of secure chips that may be subjected to them". Moreover, the Common Criteria provides a rating for evaluating the attack potential [34], that assigns higher severity scores to side-channel and fault injection attacks that require little special equipment and can be mounted under lower cost and expertise. This metric is used in practice by testing laboratories, in order to quantify the resistance of secure devices against these classes of attacks.

In this thesis, we focus on power supply fault injection, also called voltage fault injection (V-FI), a technique that involves creating disturbances, namely voltage glitches or spikes, on an otherwise stable power supply line. V-FI is a widely used technique because of its effectiveness and, in particular, its low cost. Both in the literature and in industry, efforts have been made to ease mounting complex V-FI attacks [78, 76]. For instance, commercial tools and open source frameworks such as the ChipWhisperer[1] provide an abstraction layer for controlling the attack parameters at the software level, reducing the electronic skills required. This allows for scientists with different background (*e.g.*, algorithms, statistics, machine learning) to focus on the attack logic and apply their own expertise to the hardware side-channel field. A recent study [131] shows that the disturbances induced in the chip via V-FI are effectively caused by the rising and falling edges of the injected glitch. In the literature however, the injected pulses are typically generated as a squared or v-shaped voltage signal, described by a limited set of parameters such as supply voltage, glitch lowest voltage and pulse duration.

In this work, we move one step forward and propose a new V-FI technique which is based on fully arbitrary voltage glitch waveforms. We analyse the attack performance, repeatability and feasibility, in terms of generating this type of glitches using off-the-shelf and low cost equipment. In order to experimentally assess the effectiveness of the arbitrary glitch waveform approach, we present six unpublished attacks against general purpose microcontrollers from three manufacturers. The injected faults are used to alter the execution-flow of the integrated serial bootloader, skipping security checks or generating side-channels that can, in turn, be exploited to gradually leak sensitive data, including the firmware code. We divide the firmware

[1] `https://newae.com/tools/chipwhisperer`

extraction case studies in two classes based on the design and runtime complexity. In the former, the number of successful faults required is from low to moderate ($\leq 100\,\mathrm{k}$) with a straightforward attack logic. On the contrary, the second class represents particularly challenging attacks that require several days to complete, exploit multiple vulnerabilities and inject over one million glitches. Finally, we perform a thorough evaluation of arbitrary glitch waveforms by comparing the attack performance against two popular V-FI techniques in the literature.

We selected firmware extraction for our case studies since firmware protection plays a crucial role in several industrial applications, *e.g.*, for IP and sensitive data protection. Moreover, firmware extraction is a fundamental part of the reverse engineering process performed by researchers to assess the security of embedded systems, ranging from domestic appliances to critical devices such as automotive control units and health devices (*e.g.*, defibrillators, heart rate monitors, implants). In this respect, the six unpublished attacks presented in this thesis contribute, by themselves, to the state of the art, demonstrating the unsuitability of the attacked devices for security sensitive applications. Along a responsible disclosure policy, all the vulnerabilities that we have found and the firmware extraction attacks have been timely reported to the manufacturers: STMicroelectronics, Texas Instruments and Renesas Electronics.

Structure of the Thesis

The following is an overview of the thesis organization and the topic of each chapter.

- Chapter 1 explains the terminology and provides some background on the topics presented in the thesis;
- Chapter 2 analyses the security provided by Java keystores, the standard storage facilities that allow for storing and managing cryptographic keys securely in Java applications;
- Chapter 3 focuses on the security of tamper-resistant cryptographic hardware. In particular, we examine the low-level `APDU` protocol which is used to communicate with five commercially available PKCS#11 smartcards and USB secure tokens;
- Chapter 4 studies fault injection, a class of powerful physical attacks targeting secure hardware and embedded systems.

Summary of Contributions

For what concerns the analysis of Java keystores, in Chapter 2 we define a general threat model for password-protected keystores and we distil a set of significant security properties, and consequent rules, that any secure keystore should adhere to. We conduct a thoughtful analysis of seven keystores, report undocumented details about their cryptographic implementations, and provide a detailed description of the mechanisms used by each keystore to enforce confidentiality and integrity of the stored keys and certificates. We also evaluate the strength of the password-based cryptography used to protect the keystores and we report on several attacks on implementations that do not adhere to state-of-the-art cryptographic standards. Finally, we empirically estimate the brute-force speed-up due to bad cryptographic implementations with respect to the most resistant keystore and also to the NIST recommendations.

In the context of secure cryptographic hardware, we contribute in Chapter 3 by examining the low-level `APDU` protocol which is used to communicate with PKCS#11 smartcards and USB secure tokens. We show that the PKCS#11 API is typically implemented in the form of a software, called middleware, which translates the high-level PKCS#11 commands into the low-level `APDU` protocol. We devise the first threat model for PKCS#11 middleware and we show that `APDU`-level attacks are possible. We present, in particular, new attacks for leaking symmetric sensitive keys in the clear and bypass the authentication from five commercially available tokens and smartcards, some of which were previously considered secure. Moreover, we provide a detailed security analysis of the vulnerabilities with respect to the attacker model and suggest a series of fixes and mitigations.

In Chapter 4 we contribute on physical attacks against secure hardware and embedded systems. Specifically, we investigate on voltage fault injection, a class of powerful active side-channel attacks. We study the effect of different glitch waveforms and, in particular, we propose a new method for the generation of arbitrary glitch waveforms using a low-cost and software-managed setup. We describe an unsupervised method based on genetic algorithms for identifying the set of parameters that maximize the attack performance. We report on unpublished vulnerabilities and weaknesses in six microcontrollers from three major manufacturers. Then, by combining these vulnerabilities, we describe novel attacks for extracting the firmware from the internal read-protected flash memory. Finally, we assess the effectiveness of our proposed approach by conducting an extensive experimental campaign in which we compare the speed, efficiency and reliability of our solution against two other major V-FI techniques.

Chapter 1

Background

In this section we explain the terminology and provide some background on the topics presented in the following chapters. Readers familiar with the PKCS#11 and ISO 7816 standards, microcontrollers and the principles of voltage fault injection attacks can safely skip this section.

1.1 Architecture of the Cryptographic Hardware API

PKCS#11 defines the RSA standard interface for cryptographic tokens and is now developed by the Oasis PKCS11 Technical Committee [89, 90]. The PKCS#11 API is typically implemented in the form of a middleware which translates the high-level PKCS#11 commands into low-level ISO 7816 Application Protocol Data Units (`APDUs`) and exposes results of commands in the expected PKCS#11 format. Thus, from an architectural perspective, the PKCS#11 middleware can be seen as the combination of two layers: the PKCS#11 API and the device API. All of the devices we have analysed are based on the PC/SC specification for what concerns the low-level device API.[1] This layer is the one that makes devices operable from applications and allows for communication with the device reader, exposing both standard and proprietary commands, formatted as ISO 7816 `APDUs`. In the following, we will refer to this layer as the `APDU` layer.

The PKCS#11 and the `APDU` layer are usually implemented as separate libraries. As an example, in Windows systems PC/SC is implemented in the `winscard.dll` library. Then, a separate, device-specific PKCS#11 library links to `winscard.dll` in order to communicate with the device.

It is important to notice that, even if PC/SC provides a standard layer for low-level communication, different devices implement the PKCS#11 API in various, substantially different ways. As a consequence, each device requires its specific PKCS#11 library on top of the PC/SC one. Figure 1.1 gives an overview of a typical PKCS#11 middleware architecture with two cards requiring two different PKCS#11 libraries which communicates with the cards using the same PC/SC library. In Section 1.1.1 and 1.1.2 we illustrate the PKCS#11 and the `APDU` layers more in detail.

[1] `http://www.pcscworkgroup.com/`

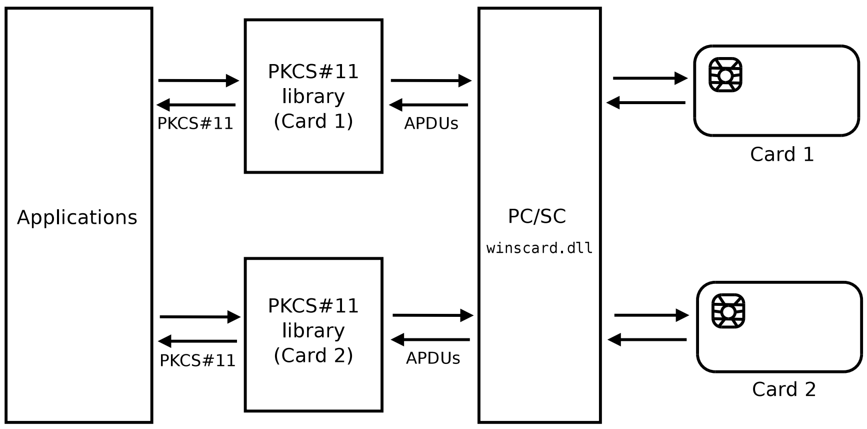


FIGURE 1.1: PKCS#11 middleware for two PC/SC (`winscard.dll`) cards with different PKCS#11 libraries.

1.1.1 The PKCS#11 Layer

As well as providing access to cryptographic functions – such as encryption, decryption, sign and authentication – PKCS#11 is designed to to provide a high degree of protection of cryptographic keys. Importing, exporting, creating and deleting keys stored in the token should always be performed in a secure way. In particular, the standard requires that even if the token is connected to an untrusted machine, in which the operating system, device drivers or software might be compromised, keys marked as sensitive should never be exported as cleartext out of the device.

In order to access the token, an application must authenticate by supplying a PIN and initiate a session. Notice, however, that if the token is connected to an untrusted machine the PIN can be easily intercepted, e.g., through a keylogger. Thus, the PIN should only be considered as a second layer of protection and it should not be possible to export sensitive keys in the clear even for legitimate users, that know the PIN (cf. [90], section 7).

PKCS#11 defines a number of *attributes* for keys that regulate the way keys should be used. We briefly summarize the most relevant ones from a security perspective (see [89, 90] for more detail):

`CKA_SENSITIVE` the key cannot be revealed as plaintext out of the token. It should be impossible to unset this attribute once it has been set, to avoid trivial attacks;

`CKA_EXTRACTABLE` the key can be wrapped, *i.e.* encrypted, under other keys and extracted from the token as ciphertext; unextractable keys cannot be revealed out of the token even when encrypted. Similarly to `CKA_SENSITIVE`, it should not be possible to mark a key as extractable once it has been marked unextractable;

`CKA_ENCRYPT`, `CKA_DECRYPT` the key can be used to encrypt and decrypt arbitrary data;

```
/* Session initialization and loading of DESkey has been omitted ... */

CK_BYTE_PTR plaintext = "AAAAAAAA";                   /* plaintext */
CK_BYTE iv[8] = {1, 2, 3, 4, 5, 6, 7, 8};            /* initialization vector */
CK_BYTE ciphertext[8];                                /* ciphertext output buffer */
CK_ULONG ciphertext_len;                              /* ciphertext length */
CK_MECHANISM mechanism = {CKM_DES_CBC, iv, sizeof(iv)};   /* DES CBC mode with given iv */

/* Initialize the encryption operation with mechanism and DESkey */
C_EncryptInit(session, &mechanism, DESkey);

/* Encryption of the plaintext string into ciphertext buffer */
C_Encrypt(session, plaintext, strlen(plaintext), ciphertext, &ciphertext_len);
```

LISTING 1.1: PKCS#11 `DES/CBC` encryption under key `DESkey`.

CKA_WRAP, CKA_UNWRAP the key can be used to encrypt (wrap) and decrypt (unwrap) other `CKA_EXTRACTABLE`, possibly `CKA_SENSITIVE` keys. These two operations are used to export and import keys from and into the device;

CKA_SIGN, CKA_VERIFY the key can be used to sign and verify arbitrary data;

CKA_PRIVATE the key can be accessed even if the user is not authenticated to the token when it is set to false;

CKA_TOKEN the key is not stored permanently on the device (discarded at the end of the session) when it is set to false.

Example 1 *(PKCS#11 symmetric key encryption).* Listing 1.1 reports a fragment of C code performing symmetric `DES/CBC` encryption of plaintext `"AAAAAAAA"` with initialization vector `0x0102030405060708`. PKCS#11 session has already been initiated and `session` contains a handle to the active session. We also assume that `DESkey` is a valid handle to a DES encryption key.

We can see that `C_EncryptInit` initializes the encryption operation by instantiating the `DES/CBC` cryptographic `mechanism` and the cryptographic key `DESkey`. Then, `C_Encrypt` performs the encryption of the string `plaintext` and stores the result and its length respectively in `ciphertext` and `ciphertext_len`. In order to keep the example simple, we skipped checks for errors that should be performed after every PKCS#11 API call (*cf.* [90], section 11). In the following we show how this example is mapped in `APDUs` on one token.

1.1.2 The `APDU` Layer

The ISO/IEC 7816 is a standard for identification, integrated circuit cards. Organization, security and commands for interchange are defined in part 4 of the standard [50]. The communication format between a smartcard and an off-card application is defined in terms of Application Protocol Data Units (`APDUs`). In particular, the half-duplex communication model is composed of `APDU` pairs: the reader sends a Command `APDU` (`C-APDU`) to the card which replies with a Response `APDU` (`R-APDU`). The standard contains a list of *inter-industry commands* whose behaviour is specified

```
0  # The challenge-response authentication is omitted. For details see Section 3.2.1
1
2  # ISO-7816 SELECT FILE command to select the folder (DF) where the key is stored
3  APDU: 00 a4 04 0c 00 00 06 50 55 42 4c 49 43
4  SW: 90 00
5  # ISO-7816 SELECT FILE command to select the file (EF) containing the encryption key
6  APDU: 00 a4 02 0c 00 00 02 83 01
7  SW: 90 00
8  # Encryption of the plaintext (red/italic) using the selected key and the given IV (green/overlined)
9  # The ciphertext is returned by the token (blue/underlined).
10 APDU: 80 16 00 01 00 00 10 01 02 03 04 05 06 07 08 41 41 41 41 41 41 41 41 00 00
11 SW: d2 ef a5 06 92 64 44 13 90 00
```

LISTING 1.2: `APDU` session trace of the PKCS#11 symmetric key encryption.

and standardized. Manufacturers can integrate these standard commands with their own *proprietary commands*.

A `C-APDU` is composed of a mandatory 4-byte header `(CLA,INS,P1,P2)`, and an optional payload `(Lc,data,Le)`, described below:

`CLA` one byte referring to the instruction class which specifies the degree of compliance to ISO/IEC 7816 and whether the command and the response are inter-industry or proprietary. Typical values are `0x00` and `0x80`, respectively for inter-industry and proprietary commands;

`INS` one byte representing the actual command to be executed, *e.g.* `READ RECORD`;

`P1,P2` two bytes containing the instruction parameters for the command, *e.g.* the record number/identifier;

`Lc` one or three bytes, depending on card capabilities, containing the length of the optional subsequent `data` field;

`data` the actual `Lc` bytes of data sent to the card;

`Le` one or three bytes, depending on card capabilities, containing the length (possibly zero) of the expected response.

The `R-APDU` is composed of an optional `Le` bytes data payload (absent when `Le` is 0), and a mandatory 2-bytes status word `(SW1,SW2)`. The latter is the return status code after command execution (*e.g.* `FILE NOT FOUND`).

Example 2 *(Symmetric key encryption in `APDUs`).* We show how the PKCS#11 code of Example 1 is mapped into `APDUs` on the Athena ASEKey USB token. Notice that this token performs a challenge-response authentication before any privileged command is executed. For simplicity, we omit the authentication part in this example but will discuss it in detail in Section 3.2.1.

The encryption operation begins by selecting the encryption key from the right location in the token memory: at line 3, the token selects the directory (called *Dedicated File* in ISO-7816) and, at line 6, the file (*Elementary File*) containing the key. At line 10, the encryption is performed: the Initialization Vector and the plaintext are sent to the token which replies with the corresponding ciphertext.

We describe in detail the the APDU format specification of `SELECT FILE` command at line 3:

`CLA` value `0x00` indicates that the command is ISO-7816 inter-industry;

`INS` value `0xA4` corresponds to inter-industry `SELECT FILE` (*cf.* [50], section 6);

`P1` value `0x04` codes a direct selection of a Dedicated File by name;

`P2` value `0x0C` selects the first record, returning no additional information about the file;

`Lc` the tokens is operating in extended APDU mode, thus this field is 3 bytes long. Value `0x000006` indicates the length 6 of the subsequent field;

`data` contains the actual ASCII-encoded name ("PUBLIC") of the DF to be selected;

`SW1,SW2` the status word `0x90 0x00` returned by the token indicates that the command was successfully executed.

It is important to notice that the `C_EncryptInit` function call sends no `APDU` to the token: we can infer that the low level protocol of the encryption operation is stateless and the state is managed inside the PKCS#11 library. This example shows that the mapping between the PKCS#11 layer and the `APDU` layer is not 1-to-1 and the PKCS#11 library is in some cases delegated to implement critical operations, such as maintaining the state of encryption. We will see how this leads to attacks in Section 3.2.

1.2 Voltage Fault Injection

Voltage fault injection is a non-invasive[2] class of attacks [7, 108] that focuses on creating disturbances on an otherwise stable power supply line in order to cause a misbehaviour in the target. This is the result of setup time violations[3] that can cause incorrect data to be captured [131, 105, 130], allowing an attacker to tamper with the regular control flow, *e.g.*, by skipping instructions, influencing a branch decision, corrupting memory locations, or altering the result of an instruction or its side effects. The disturbances that are induced in the power supply line are called *voltage glitches* or simply *glitches*. A glitch is a transient voltage drop with a duration typically in the ns to µs range, that occurs at a specific instant of time. Glitch timing (also glitch trigger or trigger) is usually calculated as a delay from a specific triggering event such as I/O activity or power-up.

There exist multiple techniques for generating and injecting a voltage glitch into the power supply line of the target device. One of the most commonly used V-FI setup [76, 129], also supported by commercial tools such as the well-known *ChipWhisperer* [78], is represented in Figure 4.2a. A transistor, typically MOS-FET, is

[2]Voltage fault injection requires no physical opening and no chemical preparation of the package.

[3]In digital designs the setup time indicates the minimum time required for an input data to be stable before the active edge clock.

placed in parallel to the power supply line and it is used to briefly short-circuit `Vcc` to ground.[4] Then, the glitch is triggered by a microcontroller (MCU) or a Field Programmable Gate Array (FPGA) managing the attack timing. The main limitations of this technique are the reduced control over the attack parameters: for instance, additional equipment is required for controlling the voltage levels, and the generated glitch can be unpredictable, (*cf.* Figure 4.2b) due to variations in both MOS-FET and target electronic properties.

MCUs integrate processor, flash memory and other peripherals in a single package. However, some microcontrollers also integrate a voltage regulator for providing a fixed and stable power supply to the internal processor and memory, independently from the actual input voltage. Depending on the regulator technology, an external filtering capacitor can be required: in this setting, the voltage glitch source can be connected directly to the capacitor pin in order to bypass the internal regulator and avoid any interference of this component during the attack.

1.2.1 Microcontroller Programming Interfaces

The software running on the MCU, namely the firmware, can typically be loaded (also *programmed*) to the internal flash memory using a debug interface exposed by the MCU: the most common is the standard JTAG / SWD interface as it can also be used for debugging the code, inspecting RAM content and accessing the integrated peripherals. Often a serial bootloader, pre-programmed by the manufacturer, exposes a set of API that can be used for, *e.g.*, write, erase or verify the firmware from a computer.

[4]Voltage glitches below 0 V are common for particular targets, *e.g.*, smartcards [88, 30].

Chapter 2

Software Keystores

In this chapter we address the problem of the secure management of cryptographic keys in software, which need to be protected and securely stored by applications. If an attacker manages to leak cryptographic keys, in fact, she has the ability to break message confidentiality and integrity, to authenticate as legitimate users or to impersonate legitimate services, thus vanishing any advantage of cryptography. In this context we analyse in detail the security of key management in the Java ecosystem and, in particular, of Java keystores. In Java, password-protected keystores are the standard facility to securely manage and store cryptographic keys: once the user (or the application) provides the correct password, the keys in the keystore become available and can be used to perform cryptographic operations, such as encryption and digital signature.

Unless stated otherwise, our findings refer to Oracle JDK 8u144 and Bouncy Castle 1.57, the two latest releases at the time of the first submission of this work in August 2017.

Contributions. The contributions found in this chapter can be summarized as follows:

(i) we define a general threat model for password-protected keystores and we distil a set of significant security properties and consequent rules that any secure keystore should adhere to;

(ii) we perform a thoughtful analysis of seven keystores, we report undocumented details about their cryptographic implementations and we classify keystores based on our proposed properties and rules;

(iii) we report on novel attacks and weaknesses in the analysed keystores. For each attack we point out the corresponding violations of our proposed properties and rules and we provide a precise attacker model;

(iv) we empirically estimate the speed-up due to bad cryptographic implementations and we show that, in some cases, this allows to decrease the guessing time of three orders of magnitude with respect to the most resistant keystore, and four orders of magnitude with respect to NIST recommendations; interestingly, the attack on Oracle JKS keystore that we present in this chapter, and we previously mentioned in a blog post [37], has been recently integrated into the Hashcat password recovery tool;

(v) we discuss the advancements on the security of Oracle and Bouncy Castle keystore implementations following our responsible disclosure. The Oracle Security Team acknowledged the reported issues by assigning three CVE IDs [70, 71, 72] and released the fixes between the October 2017 and the April 2018 Critical Patch Updates [81, 82]. Bouncy Castle developers patched some of the reported vulnerabilities in version 1.58 and the remaining issues were addressed in the following releases.

Structure of the chapter. In Section 2.1 we define the security properties of interest, the rules for the design of secure keystores and the threat model; in Section 2.2 we report on our analysis of seven Java keystores; in Section 2.3 we describe new attacks on the analysed keystores; in Section 2.4 we make an empirical comparison of the password cracking speed among the keystores; in Section 2.5 we discuss the improvements implemented by Oracle and Bouncy Castle following our responsible disclosure; finally, in Section 2.6 we draw some concluding remarks and present the related work.

2.1 Security Properties and Threat Model

In this section, we identify a set of fundamental security properties that should be guaranteed by any keystore (Section 2.1.1). We then distil rules that should be followed when designing a keystore in order to achieve the desired security properties (Section 2.1.2). Finally, we introduce the threat model covering a set of diverse attacker capabilities that enable realistic attack scenarios (Section 2.1.3).

2.1.1 Security Properties

We consider standard security properties such as confidentiality and integrity of keys and keystore entries. Breaking confidentiality of sensitive keys allows an attacker to intercept all the encrypted traffic or to impersonate the user. Breaking integrity has similar severe consequences as it might allow an attacker to import fake CA certificates and old expired keys. Additionally, since the access to a keystore is mediated by a software library or an application, we also consider the effect that a keystore has on the execution environment. Thus, we require the following properties:

P1 Confidentiality of encrypted entries

P2 Integrity of keystore entries

P3 System integrity

Property **P1** states that the value of an encrypted entry should be revealed only to authorized users, who know the correct decryption password. According to **P2**, keystore entries should be modified, created or removed only by authorized users, who know the correct integrity password, usually called *store password*. Property **P3** demands that the usage of a keystore should always be tolerated by the environment, *i.e.*, interacting with a keystore, even when provided by an untrusted party, should not pose a threat to the system, cause misbehaviours or hang the application due to an unsustainable performance hit.

A keystore file should be secured similarly to a password file: the sensitive content should not be disclosed even when the file is leaked to an attacker. In fact, it is often the case that keystores are shared in order to provide the necessary key material to various corporate services and applications. Thus, in our threat model we will always assume that the attacker has read access to the keystore file (*cf.* Section 2.1.3). For this reason we require that the above properties hold even in the presence of offline attacks. The attacker might, in fact, brute-force the passwords that are used to enforce confidentiality and integrity and, consequently, break the respective properties.

2.1.2 Design Rules

We now identify a set of core rules that should be embraced by the keystore design in order to provide the security guarantees of Section 2.1.1:

R1 Use standard, state-of-the-art cryptography

R2 Choose strong, future-proof cryptographic parameters, while maintaining acceptable performance

R3 Enforce a typed keystore format

Rule **R1** dictates the use of modern and verified algorithms to achieve the desired keystore properties. It is well-known that the design of custom cryptography is a complex task even for experts, whereas standard algorithms have been carefully analysed and withstood years of cracking attempts by the cryptographic community [16]. In this context, the National Institute of Standards and Technology (NIST) plays a prominent role in the standardization of cryptographic algorithms and their intended usage [17], engaging the cryptographic community to update standards according to cryptographic advances. For instance, NIST declared SHA1 unacceptable to use for digital signatures beginning in 2014, and more recently, urged all users of Triple-DES to migrate to AES for encryption as soon as possible [119] after the findings published in [19]. The KDF function recommended by NIST [118] is PBKDF2, as defined in the PKCS#5 standard, which supersedes the legacy PBKDF1. Another standard KDF function is defined in PKCS#12, although it has been deprecated for confidentiality purposes in favour of PBKDF2.

Key derivation functions combine the password with a randomly generated salt and iteratively apply a pseudorandom function (*e.g.*, a hash function) to produce a cryptographic key. The salt allows the generation of a large set of keys corresponding to each password [127], while the high number of iterations is introduced to hinder brute-force attacks by significantly increasing computational times. Rule **R2** reflects the need of choosing parameters to keep pace with the state-of-the-art in cryptographic research and the advances in computational capabilities. The latest NIST draft on Digital Identity Guidelines [45] sets the minimum KDF iteration count to 10,000 and the salt size to 32 bits. However, such lower bounds on the KDF should be significantly raised for critical keys according to [118] which suggests to set the number of iterations as high as can be tolerated by the environment, while maintaining acceptable performance. For instance, Apple iOS derives the decryption key for the device from the user password using a KDF with an iteration count calculated by taking into account the computational capabilities of the hardware and the impact on the user experience [5].

Finally, rule **R3** states that the keystore format must provide strong typing for keystore content, such that cryptographic objects are stored and read unambiguously. Despite some criticism over the years [46], the PKCS#12 standard embraces this principle providing precise types for storing many cryptography objects. Additionally, given that keystore files are supposed to be accessed and modified by different parties, applications parsing the keystore format must be designed to be robust against malicious crafted content.

Interestingly, not following even one of the aforementioned rules may lead to a violation of confidentiality and integrity of the keystore entries. For instance, initializing a secure KDF with a constant or empty salt, which violates only **R2**, would allow an attacker to precompute the set of possible derived keys and take advantage of *rainbow tables* [80] to speed up the brute-force of the password. On the other hand, a KDF with strong parameters is useless once paired with a weak cipher, since it is easier to retrieve the encryption key rather than brute-forcing the password. In this case only **R1** is violated.

Additionally, disrespecting Rule **R3** may have serious consequences on system integrity (breaking property **P3**), which range from applications crashing due to parsing errors while loading a malicious keystore to more severe scenarios where the host is compromised. An attacker exploiting type-flaw bugs could indirectly gain access to the protected entries of a keystore violating the confidentiality and integrity guarantees. System integrity can additionally be infringed by violating Rule **R2** with an inadequate parameter choice, *e.g.*, an unreasonably high iteration count value might hang the application, slow down the system or prevent the access to cryptographic objects stored in a keystore file due to an excessive computational load. In Section 2.3 we show how noncompliance to these rules translate into concrete attacks.

2.1.3 Threat Model

In our standard attacker model we always assume that the attacker has read access to the keystore file, either authorized or by means of a data leakage. We also assume that the attacker is able to perform offline brute-force attacks using a powerful system of her choice.

We now present a list of interesting attacker settings, that are relevant with respect to the security properties defined in Section 2.1.1:

S1 Write access to the keystore

S2 Integrity password is known

S3 Confidentiality password of an entry is known

S4 Access to previous legitimate versions of the keystore file

Setting **S1** may occur when the file is shared over a network filesystem, *e.g.*, in banks and large organizations. Since keystores include mechanisms for password-based integrity checks, it might be the case that they are shared with both read and write permissions, to enable application that possess the appropriate credentials (*i.e.*, the integrity password) to modify them. We also consider the case **S2** in which the attacker possesses the integrity password. The password might have been leaked or discovered through a successful brute-force attack. The attacker might also know the password as an insider, *i.e.*, when she belongs to the organization who owns the keystore. Setting **S3** refers to a scenario in which the attacker knows the password used to encrypt a sensitive object. Similarly to the previous case, the password might have been accessed either in a malicious or in honest way. For example, the password of the key used to sign the `apk` of an Android application [3] could be shared among the developers of the team.

In our experience, there exists a strong correlation between **S2** and **S3**. Indeed, several products and frameworks use the same password both for confidentiality and for integrity, *e.g.*, Apache Tomcat for TLS keys and IBM WebSphere for LTPA authentication. Additionally, the standard utility for Java keystores management (`keytool`) supports this practice when creating a key: the tool invites the user to just press the `RETURN` key to reuse the store password for encrypting the entry.

To summarize, our standard attacker model combined with **S1**-**S3** covers both reading and writing capabilities of the attacker on the keystore files together with the possibility of passwords leakage. On top of these settings, we consider the peculiar case **S4** that may occur when the attacker has access to backup copies of the keystore or when the file is shared over platforms supporting version control such as *Dropbox*, *ownCloud* or *Seafile*.

2.2 Analysis of Java Keystores

The Java platform exposes a comprehensive API for cryptography through a *provider*-based framework called Java Cryptography Architecture (JCA). A provider consists of a set of classes that implement cryptographic services and algorithms, including keystores. In this section, we analyse the most common Java software keystores implemented in the Oracle JDK and in a widespread cryptographic library called Bouncy Castle that ships with a provider compatible with the JCA. In particular, since the documentation was not sufficient to assess the design and cryptographic strength of the keystores, we performed a comprehensive review of the source code exposing, for the first time, implementation details such as on-disk file structure and encoding, standard and proprietary cryptographic mechanisms, default and hard-coded parameters.

For reader convenience, we provide a brief summary of the cryptographic mechanisms and acronyms used in this section: Password-Based Encryption (PBE) is

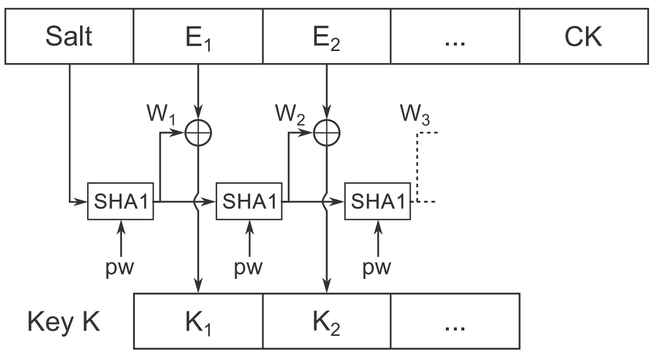


FIGURE 2.1: Decryption in the custom stream cipher used by JKS.

an encryption scheme in which the cryptographic key is derived from a password through a Key Derivation Function (KDF); a Message Authentication Code (MAC) authenticates data through a secret key and HMAC is a standard construction for MAC which is based on cryptographic hash functions; Cipher Block Chaining (CBC) and Counter with CBC-MAC (CCM) are two standard modes of operation for block ciphers, the latter is designed to provide both authenticity and confidentiality.

2.2.1 Oracle Keystores

The Oracle JDK offers three keystore implementations, precisely JKS, JCEKS and PKCS12, which are respectively made available through the providers SUN, SunJCE and SunJSSE [83]. While JKS and JCEKS rely on proprietary algorithms to enforce both the confidentiality and the integrity of the saved entries, PKCS12 relies on open standard format and algorithms as defined in [98].

JKS

Java KeyStore (JKS) is the first official implementation of a keystore that appeared in Java since the release of JDK 1.2. To the time, it is still the *default* keystore in Java 8 when no explicit choice is made. It supports encrypted private key entries and public key certificates stored in the clear. The file format consists of a header containing the file magic number, the keystore version and the number of entries, which is followed by the list of entries. The last part of the file is a digest used to check the integrity of the keystore. Each entry contains the type of the object (key or certificate) and the label, followed by the cryptographic data.

Private keys are encrypted using a custom stream cipher designed by Sun, as reported in the OpenJDK source code. In order to encrypt data, a keystream W is generated in 20-bytes blocks with W_0 being a random salt and $W_i = SHA1(password||W_{i-1})$. The encrypted key E is computed as the XOR of the private key K with the keystream W, hence K and E share the same length. The ciphertext is then prepended with the salt and appended with the checksum $CK = SHA1(password||K)$. The block diagram for decryption is shown in Figure 2.1.

The integrity of the keystore is achieved through a custom hash-based mechanism: JKS computes the SHA1 hash of the integrity password, concatenated with the constant string "`Mighty Aphrodite`" and the keystore content. The result is then checked against the 20 bytes digest at the end of the keystore file.

JCEKS

Java Cryptography Extension KeyStore (JCEKS) has been introduced after the release of JDK 1.2 in the external Java Cryptography Extension (JCE) package and merged later into the standard JDK distribution from version 1.4. According to the Java documentation, it is an alternate proprietary keystore format to JKS "that uses much stronger encryption in the form of Password-Based Encryption with Triple-DES" [52]. Besides the improved PBE mechanism, it allows for storing also symmetric keys.

The file format is almost the same of JKS with a different magic number in the file header and support for the symmetric key type. The integrity mechanism is also borrowed from JKS.

JCEKS stores certificates as plaintext, while the PBE used to encrypt private keys, inspired by PBES1 [73], is based on 20 MD5 iterations and a 64 bits salt. Given that Triple-DES is used to perform the encryption step, the key derivation process must be adapted to produce cipher parameters of the adequate size. In particular, JCEKS splits the salt in two halves and applies the key derivation process for each of them. The first 192 bits of the combined 256 bits result are used as the Triple-DES key, while the remaining 64 bits are the initialization vector.

PKCS12

The PKCS12 keystore supports both private keys and certificates, with support for secret keys added in Java 8. Starting from Java 9, Oracle replaced JKS with PKCS12 as the default keystore type [54].

The keystore file is encoded as an ASN.1 structure according to the specification given in [98]. It contains the version number of the keystore, the list of keys and the certificates. The last part of the keystore contains an HMAC (together with the parameters for its computation) used to check the integrity of the entire keystore by means of a password.

The key derivation process, used for both confidentiality and integrity, is implemented as described in the PKCS#12 standard [98] using SHA1 as hashing function, 1024 iterations and a 160 bit salt. Private keys and secret keys (when supported) are encrypted using Triple-DES in CBC mode. Certificates are encrypted as well in a single encrypted blob, using the RC2 cipher in CBC mode with a 40-bit key. While each key can be encrypted with a different password, all the certificates are encrypted reusing the store password.

2.2.2 Bouncy Castle Keystores

Bouncy Castle is a widely used open-source crypto API. As of 2014, it provides the base implementation for the crypto library used in the Android operating system [35]. It supports four different keystore types via the BC provider: BKS, UBER, BCPKCS12 and the new FIPS-compliant BCFKS. Similarly to the Oracle keystores, all the BC keystores rely on passwords to enforce confidentiality over the entries and to verify the integrity of the keystore file.

BKS

The Bouncy Castle Keystore (BKS) allows to store public/private keys, symmetric keys and certificates. The BKS keystore relies on a custom file structure to store the entries. The file contains the version number of the BKS keystore, the list of stored cryptographic entries and an HMAC, along with its parameters, computed over the entries as integrity check.

Only symmetric and private keys can be encrypted in BKS, with Triple-DES in CBC mode. The key derivation schema is taken from PKCS#12 v1.0, using SHA1 as hashing function, a random number of iterations between 1024 and 2047 which is stored for each entry and a 160 bit salt.

The integrity of the keystore is provided by an HMAC using the same key derivation scheme used for encryption and applied to the integrity password. For backward compatibility, the current version of BKS still allows to load objects encrypted under a buggy PBE mechanism used in previous versions of the keystore[1]. If the key is recovered using an old mechanisms, it is immediately re-encrypted with the newer PBE scheme.

UBER

UBER shares most of its codebase with BKS, thus it supports the same types of entries and PBE. Additionally, it provides an extra layer of encryption for the entire keystore file, which means that all metadata around the keys and certificates are encrypted as well. The PBE mechanism used for encrypting the file is Twofish in CBC mode with a key size of 256 bits. The KDF is PKCS#12 v1.0 with SHA1 using a 160 bits salt and a random number of iterations in the range 1024 and 2047.

The integrity of the keystore is checked after successful decryption using the store password. The plaintext consists of the keystore entries followed by their SHA1 checksum. UBER recomputes the hash of the keystore and compares it with the stored digest.

[1] `https://github.com/bcgit/bc-java/blob/master/prov/src/main/java/org/bouncycastle/jce/provider/BrokenPBE.java`

BCFKS

BCFKS is a new FIPS-compliant [121] keystore introduced in the version 1.56 of Bouncy Castle[2] offering similar features to UBER. This keystore provides support for secret keys in addition to asymmetric keys and certificates.

The entire keystore contents is encrypted using AES in CCM mode with a 256 bits key, so to provide protection against introspection. After the encrypted blob, the file contains a block with a HMAC-SHA512 computed over the encrypted contents to ensure the keystore integrity. The store password is used to derive the two keys for encryption and integrity.

All key derivation operations use PBKDF2 with HMAC-SHA512 as pseudorandom function, 512 bits of salt and 1024 iterations. Each key entry is separately encrypted with a different password using the same algorithm for the keystore confidentiality, while this possibility is not offered for certificates.

BCPKCS12

The BCPKCS12 keystore aims to provide a PKCS#12-compatible implementation. It shares the same algorithms and default parameters for key derivation, cryptographic schemes and file structure of the Oracle JDK version detailed in Section 2.2.1. Compared to Oracle, the Bouncy Castle implementation lacks support for symmetric keys and the possibility to protect keys with different passwords, since all the entries and certificates are encrypted under the store password. The BC provider also offers a variant of the PKCS#12 keystore that allows to encrypt certificates using the same PBE of private keys, that is Triple-DES in CBC mode.

2.2.3 Keystores Adoption

We have analysed 300 Java projects supporting keystores that are hosted on Github to estimate the usage of the implementations examined in this work. Applications range from amateur software to well-established libraries developed by Google, Apache and Eclipse.

We searched for occurrences of known patterns used to instantiate keystores in the code of each project. We have found that JKS is the most widespread keystore with over 70 % of the applications supporting it. PKCS12 is used in 32 % of the analysed repositories, while JCEKS adoption is close to 10 %. The Bouncy Castle keystores UBER and BCPKCS12 are used only in 3 % of the projects, while BKS can be found in about 6 % of the examined software. Finally, since BCFKS is a recent addition to the Bouncy Castle library, none of the repositories is supporting it.

[2]https://github.com/bcgit/bc-java/commit/80fd6825

2.2.4 Summary

In Tables 2.1 and 2.2 we summarize the features and the algorithms (rows) offered by the keystore implementations (columns) analysed in this section. Table 2.1 does not contain the row "Store Encryption" since none of the JDK keystores provides protection against introspection.

To exemplify, by reading Table 2.1 we understand that the JCEKS keystore of the SunJCE provider relies on a custom PBE mechanism based on MD5 using only 20 iterations to derive the Triple-DES key for the encryption of keys. The ✓ mark shows that the keystore supports secret keys, while ✗ denotes that certificates cannot be encrypted.

2.3 Attacks

In the previous section, we have shown that the analysed keystores use very diverse key derivation functions and cryptographic mechanisms and, in several cases, they do not adhere to standards or use obsolete and ad-hoc mechanisms. We now discuss how this weakens the overall security of the keystore and enables or facilitates attacks. In particular, we show that keystores using weak or ad-hoc implementations for password-based encryption or integrity checks open the way to password brute-forcing. During the in-depth analysis of keystores, we have also found security flaws that can be exploited in practice to mount denial of service and code execution attacks.

Attacks in this section are organized according to the security properties violated, as defined in Section 2.1.1. For each attack we provide a detailed description discussing the attacker settings and the rules that are not followed by the keystore implementation (*cf.* Section 2.1.2). We conclude with some general security considerations that are not specific to any particular attack.

Table 2.3 provides a high-level overview of the properties which are guaranteed by the analysed keystores with respect to the attacks presented in this section. We consider versions of Oracle JDK and Bouncy Castle before and after disclosing our findings to the developers. Specifically, we refer to JDK 8u144 and 8u152 for Oracle, while version 1.57 of Bouncy Castle is compared against the development repository as of November 28, 2017.[3] We use the symbol → to point out improvements in newer versions. Details of the changes are listed in Section 2.5. The ✓✓ symbol denotes that a property is satisfied by the keystore under any attacker setting and the implementation adhere to the relevant design rules listed in Section 2.1.2. We use ✓ when no clear attack can be mounted but design rules are not completely satisfied, *e.g.* a legacy cipher like Triple-DES is used. The ✗ symbol indicates that the property is broken under the standard attacker model. When a property is broken only under

[3] `https://github.com/bcgit/bc-java/tree/8ed589d`

		JKS	JCEKS	PKCS12
Provider		Sun	SunJCE	SunJSSE
Support for secret keys		✗	✓	✓*
Keys PBE	KDF	Custom (SHA1)	Custom (MD5)	PKCS12 (SHA1)
	Salt	160b	64b	160b
	Iterations	-	20	1024
	Cipher	Stream cipher	3DES (CBC)	3DES (CBC)
	Key size	-	192b	192b
Certificates PBE	KDF	✗	✗	PKCS12 (SHA1)
	Salt			160b
	Iterations			1024
	Cipher			RC2 (CBC)
	Key size			40b
Store Integrity	KDF	SHA1 with password	SHA1 with password	PKCS12 (SHA1)
	Salt			160b
	Iterations			1024
	Mechanism			HMAC (SHA1)

* since JDK 1.8

TABLE 2.1: Summary of Oracle JDK keystores (Oracle JDK 8u144 and below).

		BKS	UBER	BCFKS	BCPKCS12
Provider		Bouncy Castle	Bouncy Castle	Bouncy Castle	Bouncy Castle
Support for secret keys		✓	✓	✓	✗
Keys PBE	KDF	PKCS12 (SHA1)	PKCS12 (SHA1)	PBKDF2 (HMAC-SHA512)	PKCS12 (SHA1)
	Salt	160b	160b	512b	160b
	Iterations	1024–2047	1024–2047	1024	1024
	Cipher	3DES (CBC)	3DES (CBC)	AES (CCM)	3DES (CBC)
	Key size	192b	192b	256b	192b
Certificates PBE	KDF	✗	✗	✗	PKCS12 (SHA1)
	Salt				160b
	Iterations				1024
	Cipher				RC2 / 3DES (CBC)
	Key size				40b / 192b
Store Encryption	KDF	✗	PKCS12 (SHA1)	PBKDF2 (HMAC-SHA512)	✗
	Salt		160b	512b	
	Iterations		1024–2047	1024	
	Cipher		Twofish (CBC)	AES (CCM)	
	Key size		256b	256b	
Store Integrity	KDF	PKCS12 (SHA1)	SHA1 after decrypt	PBKDF2 (HMAC-SHA512)	PKCS12 (SHA1)
	Salt	160b		512b	160b
	Iterations	1024–2047		1024	1024
	Mechanism	HMAC (SHA1)		HMAC (SHA512)	HMAC (SHA1)

TABLE 2.2: Summary of Bouncy Castle keystores (bc 1.57 and below).

Algorithm 1 JKS 1-block Crack

```
1: procedure JKS_1BLOCKCRACK($Salt, E_{1..n}, CK$)
2:     $known\_plaintext \leftarrow \texttt{0x30} \,\|\, length(E)$
3:     $test\_bytes \leftarrow known\_plaintext \oplus E_1$
4:     for $password$ in passwords do
5:         $W_1 \leftarrow \mathrm{SHA1}(password \,\|\, Salt)$
6:         if $W_1 = test\_bytes$ then
7:             $K \leftarrow \textsc{Decrypt}(Salt, E, password)$
8:             $checksum \leftarrow \mathrm{SHA1}(password \,\|\, K)$
9:             if $CK = checksum$ then
10:                return $password$
```

a specific setting **Sx**, we report it in the table as ✗$_{\mathbf{Sx}}$. If a more powerful attack is enabled by additional settings, we clarify in the footnotes.

As an example, consider the system integrity property (**P3**) in the JCEKS keystore: up to JDK 8u144 included, write capabilities (**S1**) allow to DoS the application loading the keystore; when integrity and key confidentiality passwords are known (**S2** and **S3**), the attacker can also achieve arbitrary code execution on the system (*cf.* note 3 in the table). The rightmost side of the arrow indicates that JDK 8u152 does not include mitigations against the code execution attack.

2.3.1 Attacks on Entries Confidentiality (P1)

JKS Password Cracking

The custom PBE mechanism described in Section 2.2.1 for the encryption of private keys is extremely weak. The scheme requires only one SHA1 hash and a single XOR operation to decrypt each block of the encrypted entry resulting in a clear violation of rule **R1**. Since there is no mechanism to increase the amount of computation needed to derive the key from the password, also rule **R2** is neglected.

Despite the poor cryptographic scheme, each attempt of a brute-force password recovery attack would require to apply SHA1 several times to derive the whole keystream used to decrypt the private key. As outlined in Figure 2.1, a successful decryption is verified by matching the last block (*CK*) of the protected entry with the hash of the password concatenated with the decrypted key. For instance, a single password attempt to decrypt a 2048 bit RSA private key entry requires over 60 SHA1 operations.

We found that such password recovery attack can be greatly improved by exploiting the partial knowledge over the plaintext of the key. Indeed, the ASN.1 structure of a key entry enables to efficiently test each password with a single SHA1 operation. In JKS, private keys are serialized as DER-encoded ASN.1 objects, along the PKCS#1 standard [74]. For instance, an encoded RSA key is stored as a sequence of bytes starting with byte `0x30` which represent the ASN.1 type `SEQUENCE` and a number of bytes representing the length of the encoded key. Since the size of the

	JKS	JCEKS	PKCS12	BKS	UBER	BCFKS	BCPKCS12
(**P1** Entries confidentiality	✗	✗ → ✓	✓[1]	✓	✓	✓ → ✓✓	✓[1]
(**P2** Keystore integrity	✗[2]	✗[2]	✓ → ✓✓	✓	✓	✓ → ✓✓	✓ → ✓✓
(**P3** System integrity	✓✓	$✗^{3}_{\mathbf{S1}}$ → $✗_{\mathbf{S1\text{-}3}}$	$✗_{\mathbf{S1}}$ → ✓✓	$✗_{\mathbf{S1}}$	✓✓	✓✓	$✗_{\mathbf{S1}}$ → ✓✓

[1] only confidentiality of certificates can be violated

[2] under additional settings **S1** or **S4** it might be possible to use rainbow tables

[3] under additional settings **S2** and **S3** it is possible to achieve arbitrary code execution on JDK $\leq$ 8u152

Legend:

✓✓	property is always satisfied
✓	no clear attacks but rules not completely sat-
✗	ified
$✗_{\mathbf{Sx}}$	property is broken in the standard attacker model
	property is broken under a attacker setting **Sx**

TABLE 2.3: Properties guaranteed by keystores with respect to attacks, before and after updates listed in Section 2.5.

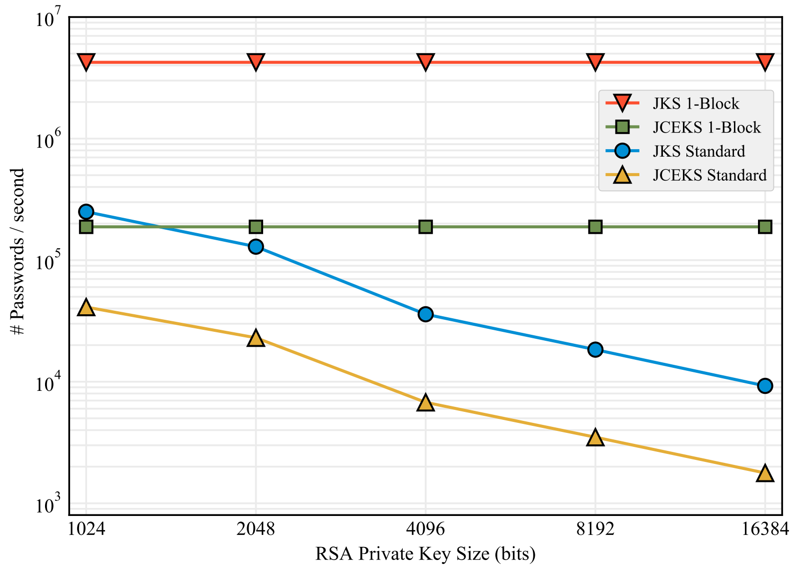


FIGURE 2.2: Performance comparison of password cracking for private RSA keys on JKS and JCEKS using both the standard and the improved 1-block method on a Intel Core i7 6700 CPU.

encrypted key is the same as the size of the plaintext, these bytes are known to the attacker. On average, given n bytes of the plaintext it is necessary to continue decryption beyond the first block only for one password every 256^n attempts.

The pseudocode of the attack is provided in Algorithm 1, using the same notation introduced in Section 2.2.1. We assume that the algorithm is initialized with the salt, all the blocks of the encrypted key and the checksum. The XOR operation between the known plaintext and the first encrypted block (line 3) is performed only once for all the possible passwords. As a halt condition, the result is then compared against the digest of the salt concatenated to the tested password (lines 5-6). To further verify the correctness of the password, a standard decrypt is performed.

A comparison between the standard cracking attack and our improved version is depicted in Figure 2.2. From the chart it is possible to see that the cost of the single block attack (referred to as 1-block) is independent from the size of the encrypted entry, while the number of operations required to carry out the standard attack is bound to the size of the DER-encoded key. As an example, for a 4096 bit private RSA key, the 1-block approach is two orders of magnitude faster than the standard one.

Based on our findings, that we previously mentioned in a blog post [37], this attack has been recently integrated into Hashcat 3.6.0[4] achieving a speed of 8 billion password tries/sec with a single NVIDIA GTX 1080 GPU.

[4]https://hashcat.net/forum/thread-6630.html

JCEKS Password Cracking

The PBE mechanism discussed in Section 2.2.1 uses a custom KDF that performs 20 MD5 iterations to derive the encryption key used in the Triple-DES cipher. This value is three orders of magnitude lower than the iteration count suggested in [45], thus violating both rules **R1** and **R2**. Given that keys are DER-encoded as well, it is possible to speed up a brute-force attack using a technique similar to the one discussed for JKS. Figure 2.2 relates the standard cracking speed to the single block version. Notice that the cost of a password-recovery attack is one order of magnitude higher than JKS in both variants due to the MD5 iterations required by the custom KDF of JCEKS.

PKCS#12 Certificate Key Cracking

Oracle PKCS12 and BCPKCS12 keystores allow for the encryption of certificates. The PBE is based on the KDF defined in the PKCS#12 standard paired with the legacy RC2 cipher in CBC mode with a 40 bit key, resulting in a clear violation of rule **R1**. Due to the reduced key space, the protection offered by the KDF against offline attacks can be voided by directly brute-forcing the cryptographic key. Our serialized tests, performed using only one core of an Intel Core i7 6700 CPU, show that the brute-force performance is 8,300 $passwords/s$ for password testing (consisting of a KDF and decryption run), while the key cracking speed is 1,400,000 $keys/s$. The worst-case scenario that requires the whole 40-bits key space to be exhausted, requires about 9 days of computation on our system. This time can be reduced to about 1 day by using all eight cores of our processor. We estimate that a modern high-end GPU should be able to perform this task in less than one hour.

Notice, however, that although finding the key so easily makes the encryption of certificates pointless, an attacker cannot use the key value to reduce the complexity of cracking the integrity password since the random salt used by the KDF makes it infeasible to precompute the mapping from passwords to keys.

2.3.2 Attacks on Keystore Integrity (P2)

JKS/JCEKS Integrity Password Cracking

The store integrity mechanism used by both JKS and JCEKS (*cf.* Section 2.2.1) only relies on the SHA1 hash digest of the integrity password, concatenated with the constant string “`Mighty Aphrodite`” and with the keystore data. In contrast with rule **R1**, this technique based on a single application of SHA1 enables to efficiently perform brute-force attacks against the integrity password. Section 2.4 reports on the computational effort required to attack the integrity mechanism for different sizes of the keystore file.

Additionally, since SHA1 is based on the Merkle-Damgärd construction, this custom approach is potentially vulnerable to extension attacks [40]. For instance, it may be possible for an attacker with write access to the keystore (**S1**) to remove the original digest at the end of the file, extend the keystore content with a forged entry and recompute a valid hash without knowing the keystore password. Fortunately, this specific attack is prevented in JKS and JCEKS since the file format stores the number of entries in the keystore header.

JKS/JCEKS Integrity Digest Precomputation

The aforementioned construction to ensure the integrity of the keystore suffers from an additional problem. Assume the attacker has access to an empty keystore, for example when an old copy of the keystore file is available under a file versioning storage (**S4**). Alternatively, as special case of **S1**, the attacker may be able to read the file, but the interaction with the keystore is mediated by an application that allows to remove entries without disclosing the store password. This file consists only of a fixed header followed by the SHA1 digest computed using the password, the string "`Mighty Aphrodite`" and the header itself. Given that there is no random salting in the digest computation, it would be possible to mount a very efficient attack to recover the integrity password by exploiting precomputed hash chains, as done in rainbow tables [80].

2.3.3 Attacks on System Integrity (P3)

JCEKS Code Execution

A secret key entry is stored in a JCEKS keystore as a Java object having type `SecretKey`. First, the key object is serialized and wrapped into a `SealedObject` in an encrypted form; next, this object is serialized again and saved into the keystore.

When the keystore is loaded, all the serialized Java objects stored as secret key entries are evaluated. An attacker with write capabilities (**S1**) may construct a malicious entry containing a Java object that, when deserialized, allows her to execute arbitrary code in the application context. Interestingly, the attack is not prevented by the integrity check since keystore integrity is verified only after parsing all the entries.

The vulnerable code can be found in the `engineLoad` method of the class `JceKeyStore` implemented by the SunJCE provider.[5] In particular, the deserialization is performed at lines 837-838 as follows:

```java
// read the sealed key
try {
    ois = new ObjectInputStream(dis);
    entry.sealedKey =
        (SealedObject) ois.readObject();
    ...
```

[5] `http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/5534221c23fc/src/share/classes/com/sun/crypto/provider/JceKeyStore.java`

Notice that the cast does not prevent the attack since it is performed after the object evaluation.

To stress the impact of this vulnerability, we provide three different attack scenarios: (*i*) the keystore is accessed by multiple users over a shared storage. An attacker can replace or add a single entry of the keystore embedding the malicious payload, possibly gaining control of multiple hosts; (*ii*) a remote application could allow its users to upload keystores for cryptographic purposes, such as importing certificates or configuring SSL/TLS. A crafted keystore loaded by the attacker may compromise the remote system; (*iii*) an attacker may even forge a malicious keystore and massively spread it like a malware using email attachments or instant messaging platforms. Users with a default application associated to the keystore file extension (*e.g.*, keystore inspection utilities such as KSE [6]) have a high probability of being infected just by double clicking on the received keystore. Interestingly, all the malicious keystores generated during our tests did not raise any alert on antivirus tools completing a successful scan by *virustotal.com*.

We checked the presence of the vulnerability from Java 6 onwards. We were able to achieve arbitrary command execution on the host with JDK $\leq$ 7u21 and JDK $\leq$ 8u20 by forging a payload with the tool `ysoserial`.[7] Newer versions are still affected by the vulnerability, but the JDK classes exploited to achieve code execution have been patched. Since the deserialization occurs within a Java core class, the classpath is restricted to bootstrap and standard library classes. However, by embedding a recursive object graph in a JCEKS entry, an attacker can still hang the deserialization routine consuming CPU indefinitely and thus causing a DoS in the target machine. We were able to mount this attack on any version of the Oracle JDK $\leq$ 8u144.

The implementation choice for storing secret keys in JCEKS is a clear violation of Rule **R3**, since these entities are essentially stored as Java code. The correct approach is to adopt standard formats and encodings, such as the PKCS#8 format used in the PKCS12 keystore.

JCEKS Code Execution After Decryption

When the attacker knows the integrity password and the confidentiality password of a secret key entry (**S2**, **S3**) in addition to **S1**, the previous attack can be further improved to achieve arbitrary command execution even on the latest, at the time of writing, Java 8 release (8u152). This variant of the attack assumes that the application loading the JCEKS keystore makes use of one of the widespread third-party libraries supported by `ysoserial`, such as *Apache Commons Collections* or the *Spring* framework: such libraries have been found [124] to contain vulnerable gadget chains that can be exploited by the malicious payload.

[6] `http://keystore-explorer.org`
[7] `https://github.com/frohoff/ysoserial`

When a `SealedObject` wrapping a secret key is successfully loaded and decrypted, an additional deserialization call is performed over the decrypted content. The `SealedObject` class extends the classpath to allow the deserialization of any class available in the application scope, including third-party libraries. By exploiting this second deserialization step, an attacker may construct more powerful payloads to achieve command execution.

The exploitation scenarios are similar to the ones already discussed in the previous variant of the attack. Additionally, we point out that even an antivirus trained to detect deserialization signatures would not be able to identify the malicious content since the payload is stored in encrypted form in the keystore.

DoS by Integrity Parameters Abuse

Many keystores rely on a keyed MAC function to ensure the integrity of their contents. The parameters of the KDF used to derive the key from the store password are saved inside the file. Thus, an attacker with write capabilities (**S1**) may tamper with the KDF parameters to affect the key derivation phase that is performed before assessing the integrity of the keystore file. In particular, the attacker may set the iteration count to an unreasonably high value in order to perform a DoS attack on applications loading the keystore.

We found that Oracle PKCS12, BKS and BCPKCS12 implementations are affected by this problem. Starting from valid keystore files, we managed to set the iteration count value to $2^{31} - 1$. Loading such keystores required around 15 minutes at full CPU usage on a modern computer. According to [118] the iteration count should not impact too heavily on the user-perceived performance, thus we argue that this is a violation of Rule **R2**.

2.3.4 Bad Design Practices

During our analysis we found that some of the keystores suffered from bad design decisions and implementation issues that, despite not leading to proper attacks, could lead to serious security consequences.

Our review of the Oracle PKCS12 keystore code showed that the KDF parameters are not treated uniformly among MAC, keys and certificates. During a store operation, the Oracle implementation does not preserve the original iteration count and salt size for MAC and certificates that has been found at load time in the input keystore file. Indeed, iteration count and salt size are silently set to the hard-coded values of 1024 and 20 byte, respectively. Since this keystore format is meant to be interoperable, this practice could have security consequences when dealing with keystores generated by third-party tools. For instance, PKCS12-compatible keystores generated by OpenSSL default to 2048 iterations: writing out such keystore with the Oracle JDK results in halving the cost of a password recovery attack.

The Bouncy Castle BCPKCS12 implementation suffers a similar problem: in addition to MAC and certificate parameters, also the iteration count and the salt size used for private keys are reverted to default values when the keystore is saved to disk. Following our report to the Bouncy Castle developers, this behaviour is currently being addressed in the next release by preserving the original parameters whenever possible.[8]

Lastly, the construction of the integrity mechanism for the UBER keystore could cause an information leakage under specific circumstances. After a successful decryption using the store password, UBER recomputes the hash of the keystore and compares it with the stored digest. This MAC-then-encrypt approach is generally considered a bad idea, since it can lead to attacks if, for example, there is a perceptible difference in behaviour (an error message, or execution time) between a decryption that fails because the padding is invalid, or a decryption that fails because the hash is invalid (a so-called padding oracle attack [122]).

2.3.5 Security Considerations

We now provide general considerations on the security of Java keystores. The first one is about using the same password for different purposes. If the integrity password is also used to ensure the confidentiality of encrypted entries, then the complexity of breaking either the integrity or the confidentiality of stored entries turns out to be the one of attacking the weakest mechanism. For instance, we consider a keystore where cracking the integrity password is more efficient than recovering the password used to protect sensitive entries: as shown in Section 2.4, this is the case of PKCS12 and BCPKCS12 keystores. Under this setting, sensitive keys can be leaked more easily by brute-forcing the integrity password.

Although this is considered a bad practice in general [63], all the keystores analysed permit the use of the same password to protect sensitive entries and to verify the integrity of the keystore. This practice is indeed widespread [42] and, as already stated in Section 2.1.3, prompted by `keytool` itself. Furthermore, our analysis found that the BCPKCS12 keystore forcibly encrypts keys and certificates with the store password. For these reasons, we argue that using the same password for integrity and confidentiality is not a direct threat to the security of stored keys when both mechanisms are resistant to offline attacks and a strong password is used. Still the security implications of this practice should be seriously considered.

The second consideration regards how the integrity of a keystore is assessed. Indeed, a poorly designed application may bypass the integrity check on keystores by providing a null or empty password to the Java `load()` function. All the Oracle keystores analysed in the previous section and BouncyCastle BKS are affected by this problem. On the other hand, keystores providing protection to entries inspection, such as UBER and BCFKS, cannot be loaded with an empty password since the

[8]`https://github.com/bcgit/bc-java/commit/ebe1b25a`

decryption step would fail. Lastly, BCPKCS12 throws an exception if an attempt of loading a file with an empty password is made. If the integrity check is omitted, an attacker can trivially violate Property **P2** by altering, adding or removing any entry saved in the clear. Conversely, the integrity of encrypted sensitive keys is still provided by the decryption mechanism that checks for the correct padding sequence at the end of the plaintext. Since the entries are typically encoded (*e.g.*, in ASN.1), a failure in the parse routine could also indicate a tampered ciphertext.

We also emphasize that the 1-block cracking optimization introduced in 2.3.1 is not limited to JKS and JCEKS. Indeed, by leveraging the structure of saved entries, all the analysed keystores enable to reduce the cost of the decrypt operation to check the correctness of a password. However, excluding JKS and JCEKS, this technique only provides a negligible speed-up on the remaining keystores given that the KDF is orders of magnitude slower than the decrypt operation.

Finally, we point out that the current design of password-based keystores cannot provide a proper key-revocation mechanism without a trusted third-party component. For instance, it may be the case that a key has been leaked in the clear and subsequently substituted with a fresh one in newer versions of a keystore file. Under settings **S1** and **S4**, an attacker may replace the current version of a keystore with a previously intercepted valid version, thus restoring the exposed key. The integrity mechanism is indeed not sufficient to distinguish among different versions of a keystore protected with the same store password. For this reason, the store password must be updated to a fresh one every time a rollback of the keystore file is not acceptable by the user, which is typically the case of a keystore containing a revoked key.

2.4 Estimating Brute-Force Speed-Up

We have discussed how weak PBEs and integrity checks in keystores can expose passwords to brute-forcing. In this section we make an empirical comparison of the cracking speed to bruteforce both the confidentiality and integrity mechanisms in the analysed keystores. We also compute the speed-up with respect to BCFKS, as it is the only keystore using a standard and modern KDF, *i.e.*, PBKDF2, which provides the best brute-forcing resistance. Notice, however, that the latest NIST draft on Digital Identity Guidelines [45] sets the minimum KDF iteration count to 10,000 which is one order of magnitude more than what is used in BCFKS (*cf.* Table 2.2). Thus all the speed-up values should be roughly multiplied by 10 if compared with a baseline implementation using PBKDF2 with 10,000 iterations.

It is out of the scope of this work to investigate brute-forcing strategies. Our tests only aim at comparing, among the different keystores, the actual time to perform the key derivation step and the subsequent cryptographic operations, including the check to assess key correctness. Our study is independent of the actual password guessing strategy adopted by the attacker.

Algorithm 2 Confidentiality password cracking benchmark

```
1:  procedure BENCHCONFIDENTIALITY($test\_duration$)
2:      $encrypted\_entry \leftarrow (B_1, ..., B_{2000})$
3:      $passwords \leftarrow (pw_1, ..., pw_n)$                    ▷ all 10-bytes passwords
4:      $salt \leftarrow$ constant
5:      $counter \leftarrow 0$
6:      while ELAPSEDTIME $< test\_duration$ do
7:          $password \leftarrow$ next($passwords$)
8:          $key \leftarrow \mathrm{KDF}_{key}(password, salt)$
9:          $iv \leftarrow \mathrm{KDF}_{iv}(password, salt)$              ▷ not in JKS, BCFKS
10:         $plaintext \leftarrow$ DECRYPTBLOCK($encrypted\_entry, key, iv$)
11:         VERIFYKEY($plaintext$)
12:         $counter \leftarrow counter + 1$
13:     return $counter$
```

Algorithm 3 Integrity password cracking benchmark

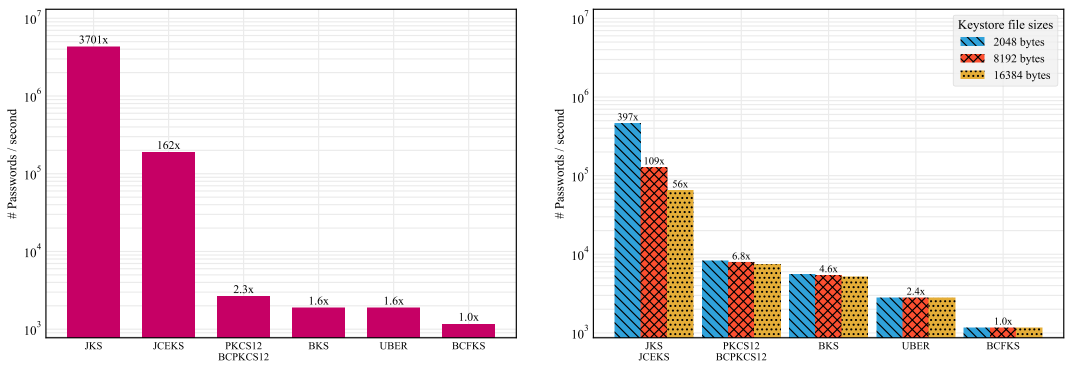
```
1:  procedure BENCHINTEGRITY($test\_duration$)
2:      $keystore\_content_{small} \leftarrow (B_1, ..., B_{2048})$
3:      $keystore\_content_{medium} \leftarrow (B_1, ..., B_{8192})$
4:      $keystore\_content_{large} \leftarrow (B_1, ..., B_{16384})$
5:      $passwords \leftarrow (pw_1, ..., pw_n)$                    ▷ all 10-bytes passwords
6:      $salt \leftarrow$ constant
7:      $counter_{(small,medium,large)} \leftarrow 0$
8:      for all $keystore\_content, counter$ do
9:          while ELAPSEDTIME $< test\_duration$ do
10:             $password \leftarrow$ next($passwords$)
11:             $key \leftarrow \mathrm{KDF}_{mac}(password, salt)$       ▷ not in JKS, JCEKS
12:             $mac \leftarrow$ MAC($keystore\_content, key$)
13:             VERIFYMAC($mac$)
14:             $counter \leftarrow counter + 1$
15:     return $counter_{(small,medium,large)}$
```

2.4.1 Test Methodology

We developed a compatible `C` implementation of the key decryption and the integrity check for each keystore type. Each implementation is limited to the minimum steps required to check the correctness of a test password. This procedure is then executed in a timed loop to evaluate the cracking speed. Algorithms 2 and 3 show the pseudocode of our implementations. Note that, in both algorithms, we set the password length to 10 bytes because it is an intermediate value between trivial and infeasible. Similarly, since the iteration count in BKS and UBER is chosen randomly in the range 1024 and 2047, we set it to the intermediate value 1536.

Confidentiality

The confidentiality password brute-forcing loop (Algorithm 2) is divided into three steps: key derivation, decryption and a password correctness check. The last step is included in the loop only to account for its computational cost in the results. Both PBES1 (PKCS#5) and PKCS#12 password-based encryption schemes, used in all keystores but BCFKS, require to run the KDF twice to derive the decryption key and the



(A) Speed comparison of password recovery attack for key encryption (confidentiality).

(B) Speed comparison of password recovery attack for keystore integrity, considering different keystore sizes.

FIGURE 2.3: Comparison of keystores password cracking speed. Bar labels indicate the speed-up to the strongest BCFKS baseline.

IV. On the other hand, in BCFKS the initialization vector is not derived from the password but simply stored with the ciphertext. During our tests we set *encrypted_-entry* to a fixed size to resemble an on-disk entry containing a 2048 bits RSA key. However, in Section 2.3.1 we have shown how the partial knowledge of the plaintext structure of a JKS key entry can be leveraged to speed-up brute-forcing. This shortcut can be applied to all the analysed keystores in order to decrypt only the first block of *encrypted_entry*. For this reason, the key size becomes irrelevant while testing for a decryption password.

Integrity

Similarly, the integrity password cracking code (Algorithm 3) is divided into three steps: key derivation, a hash/MAC computation and the password correctness check. The key derivation step is run once to derive the MAC key in all keystores, with the exception of JKS and JCEKS where the password is fed directly to the hash function (*cf.* Section 2.2.1). As described later in this section, the speed of KDF plus MAC calculation can be highly influenced by the keystore size, thus we performed our tests using a *keystore_content* of three different sizes: 2048, 8192 and 16384 bytes.

Test configuration

We relied on standard implementations of the cryptographic algorithms to produce comparable results: the OpenSSL library (version 1.0.2g) provides all the needed hash functions, ciphers and KDFs, with the exception of Twofish where we used an implementation from the author of the cipher.[9] All the tests were performed on a desktop computer running Ubuntu 16.04 and equipped with an Intel Core i7 6700 CPU; source code of our implementations has been compiled with GCC 5.4 using `-O3 -march=native` optimizations. We run each benchmark on a single CPU core

[9]`https://www.schneier.com/academic/twofish/download.html`

because the numeric results can be easily scaled to a highly parallel systems. To collect solid and repeatable results each benchmark has been run for 60 seconds.

2.4.2 Results

The charts in Figure 2.3 show our benchmarks on the cracking speed for confidentiality (Figure 2.3a) and integrity (Figure 2.3b). On the x-axis there are the 7 keystore types: we group together different keystores when the specific mechanism is shared among the implementations, *i.e.*, PKCS12/BCPKCS12 for both confidentiality and integrity and JKS/JCEKS for integrity. On the y-axis we report the number of tested passwords per second doing a serial computation on a single CPU core: note that the scale of this axis is logarithmic. We stress that our results are meant to provide a relative, inter-keystore comparison rather than an absolute performance index. To this end, a label on top of each bar indicates the speed-up relative to the strongest BCFKS baseline. Absolute performance can be greatly improved using both optimized parallel code and more powerful hardware which ranges from dozens of CPU cores or GPUs to programmable devices such as FPGA or custom-designed ASICs [56, 31, 66].

Confidentiality

From the attack described in Section 2.3.1, it follows that cracking the password of an encrypted key contained in JKS - the default Java keystore - is at least three orders of magnitude faster than in BCFKS. Even without a specific attack, recovering the same password from JCEKS is over one hundred times faster due to its low (20) iteration count. By contrast, the higher value (1024 or 1024-2047) used in PKCS12, BKS and UBER translates into a far better offline resistance as outlined in the chart.

Integrity

Similar considerations can be done for the integrity password resistance. Finding this password in all keystores but JKS is equivalent, or even faster than breaking the confidentiality password. Moreover, the performance of these keystores is influenced by the size of the file due to the particular construction of the MAC function (*cf.* Section 2.2.1). The speed gain (w.r.t. confidentiality) visible in PKCS12, BKS and UBER is caused by the missing IV derivation step which, basically, halves the number or KDF iterations. Interestingly, in BCFKS there is no difference between the two scores: since the whole keystore file is encrypted, we can reduce the integrity check to a successful decryption, avoiding the computation overhead of the HMAC on the entire file.

2.5 Disclosure and Security Updates

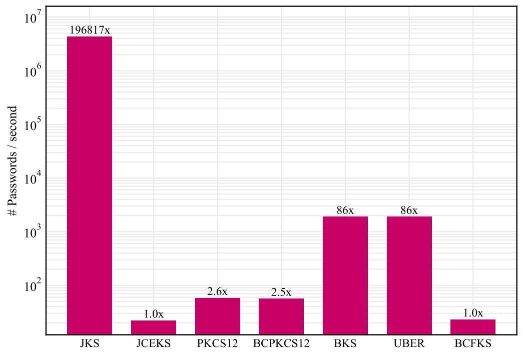
We have timely disclosed our findings to Oracle and Bouncy Castle developers in May 2017. The Oracle Security Team has acknowledged the reported issues with CVE IDs [70, 71, 72] and has released most of the fixes in the October 2017 Critical Patch Update (CPU) [81]. The fix for the second JCEKS deserialization vulnerability was later released in the April 2018 CPU [82]. In the following list, we summarize the changes already published by Oracle:

- `keytool` suggests to switch to PKCS12 when JKS or JCEKS keystores are used;
- improved KDF strength of the PBE in JCEKS by raising the iteration count to 200,000. Added a ceiling value of 5 millions to prevent parameter abuse;
- in PKCS12 the iteration count has been increased to 50,000 for confidentiality and 100,000 for integrity. The same upper bound as in JCEKS is introduced;
- fixed the first JCEKS deserialization vulnerability described in Section 2.3.3 by checking that the object being deserialized is of the correct type, *i.e.*, `SealedObjectForKeyProtector`, and by imposing a recursion limit to prevent infinite loops;
- corrected the second JCEKS deserialization vulnerability by introducing a built-in Java serialization filtering mechanism [51] which provides a way to narrow down the classes that can be deserialized, and a set of metrics for evaluating the deserialization graph size and complexity.

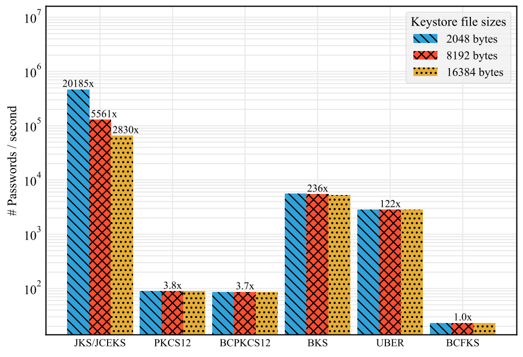
In version 1.58 of the library, Bouncy Castle developers fixed the parameter abuse vulnerability of BCPKCS12 by adding an optional Java system property that imposes an upper bound for the KDF iteration count. Moreover, the following changes apperead in version 1.59:

- in BCFKS, the iteration count is raised to 51,200 for both confidentiality and integrity;
- in BCPKCS12, the iteration count is increased to 51,200 and 102,400 for confidentiality and integrity, respectively.

Table 2.3 outlines the improved security guarantees offered by keystore implementations following the fixes released by Oracle and Bouncy Castle. Additionally, in Figure 2.4 we show the updated results of the brute-force resistance benchmarks to reflect the improved KDF parameters. JCEKS and BCFKS now offer the best resistance to offline brute-force attacks of the confidentiality password. However, JCEKS still provides the weakest integrity mechanism. Thus, if the same password is used both for key encryption and for keystore integrity, then the increased protection level can easily be voided by attacking the latter mechanism. On the other hand, both the confidentiality and the integrity mechanisms have been updated in PKCS12. This keystore, which is now the default in Java 9, offers a much higher security level with respect to the previous release.



(A) Speed comparison of password recovery attack for key encryption (confidentiality).



(B) Speed comparison of password recovery attack for keystore integrity, considering different keystore sizes.

FIGURE 2.4: Revised password cracking benchmarks after library updates.

2.6 Discussion

Keystores are the standard way to store and manage cryptographic keys and certificates in Java applications. In the literature there is no in-depth analysis of keystore implementations and the documentation does not provide enough information to evaluate the security level offered by each keystore. Thus, developers cannot make a reasoned and informed choice among the available alternatives.

In this chapter we have thoroughly analysed seven keystore implementations from the Oracle JDK and the Bouncy Castle library. We have described all the cryptographic mechanisms used to guarantee standard security properties on keystores, including offline attacks. We have pointed out that several implementations adopt non-standard mechanisms and we have shown how this can drastically speed-up the brute-forcing of the keystore passwords. Additionally, we reported new and unpublished attacks and defined a precise threat model under which they may occur. These attacks range from breaking the confidentiality of stored keys to arbitrary code execution on remote systems and denial of service. We also showed how a keystore can be potentially weaponized by an attacker to spread malware.

We have reported the security flaws to Oracle and Bouncy Castle. The issues in the Oracle JDK have been fixed between the October 2017 and the April 2018 Critical Patch Updates [81, 82] following CVE IDs [70, 71, 72]. Similarly, Bouncy Castle developers committed changes to address several problems discussed in this work.

Following our analysis and succeeding fixes, it appears evident that the security offered by JKS, the default keystore in Java 8 and previous releases, is totally inadequate. Its improved version JCEKS still uses a broken integrity mechanism. For these reasons, we favourably welcome the decision of Oracle to switch to PKCS12 as the default keystore type in the recent Java 9 release. After the previously discussed updates this keystore results quite solid, although certificate protection is bogus and key encryption relies on legacy cryptography.

Alternatives provided by Bouncy Castle have been found to be less susceptible to attacks. Among the analysed keystores, the updated BCFKS version clearly sets the standard from a security standpoint. Indeed, this keystore relies on modern algorithms, uses adequate cryptographic parameters and provides protection against introspection of keystore contents. Moreover, the development version of Bouncy Castle includes preliminary support for *scrypt* [86, 87] in BCFKS, a *memory-hard* function that requires significant amount of RAM. Considering the steady nature of keystore files, we argue that in addition to approved standard functions, it would be advisable to consider future-proof cryptographic primitives so to be more resistant against parallelized attacks [20, 24].

Related Work

Cooijmans *et al.* [35] have studied various key storage solutions in Android, either provided as an operating system service or through the Bouncy Castle cryptographic library. The threat model is very much tailored to the Android operating system and radically different from the one we consider in this chapter. Offline brute-forcing, for example, is only discussed marginally in the paper. Interestingly, authors show that under a root attacker (*i.e.*, an attacker with root access to the device), the Bouncy Castle software implementation is, in some respect, more secure than the Android OS service using TrustZone's capabilities, because of the possibility to protect the keystore with a user-supplied password. Differently from our work, the focus of the paper is not on the keystore design and the adopted cryptographic mechanisms.

Sabt *et al.* [100] have recently found a forgery attack in the Android KeyStore service, an Android process that offers a keystore service to applications and is out of the scope of our work. However, similarly to our results, the adopted encryption scheme is shown to be weak and not compliant to the recommended standards, enabling a forgery attack that make apps use insecure cryptographic keys, voiding any benefit of cryptography.

Li *et al.* [63] have analysed the security of web password managers. Even if the setting is different, there are some interesting similarities with keystores. In both settings a password is used to protect sensitive credentials, passwords in one case and keys in the other. So the underlying cryptographic techniques are similar. However the kind of vulnerabilities found in the paper are not related to cryptographic issues. Gasti *et al.* [42] have studied the format of password manager databases. There is some similarity with our work for what concerns the threat model, *e.g.*, by considering an attacker that can tamper with the password database. However, the setting is different and the paper does not account for cryptographic weaknesses and brute-forcing attacks.

Chapter 3

Cryptographic Hardware API

In this chapter we investigate on the security of dedicated cryptographic hardware which, with respect to software-based cryptography, is recognized as a practical solution for ensuring stronger protection against attacks. For this reason tamper-resistant devices such as smartcards, USB tokens and Hardware Security Modules are often adopted for critical activities, *e.g.*, in financial and large organizations. In this respect, we focus on cryptographic devices that are accessible via the PKCS#11 API and, in particular, we study the translation from the PKCS#11 API to the low-level communication protocol, namely `APDU`, which is used to interact with the device. We analysed five commercially available devices and investigated on how these devices implement various security-critical PKCS#11 operations, by studying in detail the `APDU` traces. We found that the PKCS#11 API is typically implemented in the form of a middleware translating the high-level PKCS#11 commands into low-level ISO 7816 Application Protocol Data Units (`APDUs`) and exposing the results of commands in the expected PKCS#11 format. In our experiments, we noticed that this translation is far from being a 1-to-1 mapping. Devices usually implement simple building blocks for key storage and cryptographic operations, but most of the logic and, in some cases, some of the sensitive operations are delegated to the middleware.

Contributions. In this chapter we present:

(*i*) a new threat model for PKCS#11 middleware used by cryptographic hardware;

(*ii*) novel `APDU`-level attacks on commercially available tokens and smartcards, some of which were considered secure;

(*iii*) a security analysis of the vulnerabilities with respect to the threat model.

Structure of the chapter. The chapter is organized as follows: in Section 3.1 we present the threat model; in Section 3.2 we illustrate in detail our findings on five commercially available devices; in Section 3.3 we analyse the attacks with respect to the threat model and in Section 3.4 we draw some concluding remarks and present the related work.

3.1 Threat Model

In this section we analyse various threat scenarios and classify them based on the attacker capabilities.

We consider a typical scenario in which the target token is connected to a desktop or laptop computer running in a single-user configuration. We describe the threat model by focusing on the following sensitive targets:

PIN If the attacker discovers the PIN he might be able to perform cryptographic operations with the device when it is connected to the user's host or in case he has physical access to it;

Cryptographic operations The attacker might try to perform cryptographic operations with the token, independently of his knowledge of the PIN;

Cryptographic keys The attacker might try to learn sensitive keys either by exploiting PKCS#11 API-level attacks such as Clulow's wrap-and-decrypt [32] (cf. Section 1.1.1) or by exploiting the new `APDU`-level vulnerabilities we will discuss in Section 3.2.

3.1.1 Administrator Privileges

If the attacker has administration privileges, he basically has complete control of the host. He can modify the driver, replace the libraries, intercept any input for the users and attach to any running process[1]. As such, he can easily learn the PIN when it is typed or when it is sent to the library, use the PIN to perform any cryptographic operations and exploit any PKCS#11 or `APDU` level attacks to extract cryptographic keys in the clear.

3.1.2 User Privileges

The most common situation is when the attacker has user privileges. In this case we have different scenarios:

Monolithic. The application is run by the same user as the attacker and directly links both the PKCS#11 and the `APDU` library. The attacker can easily sniff and alter data by attaching to the application process and by intercepting library calls. The attacker can trivially learn the PIN when it is sent to the library, use the PIN to perform any cryptographic operations and exploit any PKCS#11 or `APDU` level attacks to extract cryptographic keys in the clear.

Separate authentication mechanism. The application is run by the same user as the attacker and directly links the PKCS#11 library but authentication is managed by a separate software or hardware which is not directly accessible with user privileges. Examples could be a separate dialog for entering the PIN running at different privileges or some biometric sensor integrated in a USB token. The attacker cannot directly log into the token but can still sniff and alter data by attaching to the application process and by intercepting library calls. If the attacker is able to place in the middle and alter data, he could additionally exploit PKCS#11 or `APDU`-level attacks to extract cryptographic keys in the clear. Notice that, knowing the PIN, this can be done by simply opening a new independent session. Without knowledge of the PIN, instead, the attacker needs a reliable Man-In-The-Middle (MITM) attack.

Separate privileges. If the middleware layer is run as separate process at a different privilege level, the attacker cannot attach to it and observe or alter `APDUs`. The

[1]This is typically done by using the operating system debug API to instrument or inspect the target process memory. Examples are the Event Tracing API for Windows and the Linux ptrace() syscall.

Attacker	Application	Attacker can access		Attacker can exploit			
		PKCS#11	APDU	PIN	PKCS#11	APDU passive	APDU active
Admin	Any	✓	✓	✓	✓	✓	✓
User	Monolithic	✓	✓	✓	✓	✓	✓
	Sep. Auth.	✓	✓	✗	✓[1]	✓	✓[1]
	Sep. Privileges	✓	✗	✓	✓	✗	✗
	Sep. Auth.&Priv.	✓	✗	✗	✓[1]	✗	✗
Physical	Any	✗	✓	✓[2]	✓[1]	✓[3]	✓[1,3]

[1] Requires MITM.
[2] Through a keylogger or a USB sniffer.
[3] Only APDU payloads, cannot access middleware memory.

TABLE 3.1: Threats versus attackers and applications.

attacker can still try to access the token directly, so if there are ways to bypass authentication he might be able to perform cryptographic operations and exploit PKCS#11 or APDU-level attacks.

3.1.3 Physical Access

If the attacker has physical access to the user host he might install physical keyloggers and USB sniffers. This is not always feasible for example if the devices are integrated, as in laptops. In the case of a key-logger, the attacker can easily discover the PIN if it is typed through the keyboard. The case of directly sniffing APDUs passing, e.g., through USB, is interesting and more variegated since different sensitive data could be transmitted through the APDU layer, as we will illustrate in Section 3.2.

3.1.4 Summary of the Threat Model

Table 3.1 summarizes what the various attackers can access and exploit in different settings. We distinguish between passive APDU attacks, where the attacker just sniffs the APDU trace, and active APDU attacks, where APDUs are injected or altered by the attacker. In some cases active APDU attacks require mounting a MITM, e.g., when the PIN is now known or when the attacker does not have access to the middleware, as in physical attacks.

We point out that, if the application is monolithic, an attacker with user privileges is as powerful as one with administrative privileges. The maximum degree of protection is when the application offers separate authentication and the middleware runs with different privileges. We notice that the attacker can still perform PKCS#11-level attacks without knowing the PIN by mounting a MITM and altering or hijacking the API calls. Finally, physical attacker can in principle perform all the attacks, except the ones that are based on inspecting process (or middleware) memory and assuming, in some cases, MITM capabilities.

3.2 APDU-level Attacks on Real Devices

We have tested the following five devices from different manufacturers for possible APDU-level vulnerabilities.

- Aladdin eToken PRO (USB)
- Athena ASEKey (USB)
- RSA SecurID 800 (USB)
- Safesite Classic TPC IS V1 (smartcard)
- Siemens CardOS V4.3b (smartcard)

For readability, in the following we will refer to the above tokens and smartcards as eToken PRO, ASEKey, SecurID, Safesite Classic and Siemens CardOS, respectively. These five devices are the ones tested in [26] for which we could find APDU-level attacks. It is worth noticing that we could not inspect the APDU traces of some other devices analysed in [26] because they encrypt the APDU-level communication. We leave the study of the security of encrypted APDUs as a future work.

We have systematically performed various tests on selected sensitive operations and we have observed the corresponding APDU activity. We have found possible vulnerabilities concerning the login phase (Section 3.2.1), symmetric sensitive keys (Section 3.2.2), key attributes (Section 3.2.3), private RSA session keys (Section 3.2.4).

Quite surprisingly we have found that, in some cases, cryptographic keys appear as cleartext in the library which performs cryptographic operations in software. Moreover, we have verified that the logic behind PKCS#11 key attributes is, in most of the cases, implemented in the library. We have also found that all devices are vulnerable to attacks that leak the PIN if the middleware is not property isolated and run with a different privilege (which is usually not the case). Moreover, attackers with physical access could sniff an authentication session through the USB port and brute-force the PIN once the authentication protocol has been reverse-engineered.

Our findings have been timely reported to manufacturers following a *responsible disclosure* process and are described in detail in the following sections.[2]

3.2.1 Authentication

In PKCS#11 the function `C_Login` allows a user to authenticate, in order to activate a session and perform cryptographic operations. For the five devices examined, we found that authentication is implemented in two different forms: plaintext and challenge-response.

[2]Official answers from manufacturers, if any, will be made available at `https://secgroup.dais.unive.it/projects/apduattacks/`.

`C_Login` session trace	Device
0 `# Custom Get challenge:` 1 `APDU: 80 17 00 00 08` 2 `SW:   df 89 61 34 62 05 13 36 90 00` 3 `# Custom External authenticate:` 4 `APDU: 80 11 00 11 0a 10 08 64 d5 97 15 4a 44 eb 23` 5 `SW:   90 00`	Aladdin eToken PRO
0 `# Standard ISO-7816 Get challenge:` 1 `APDU: 00 84 00 00 00 00 08` 2 `SW:   bb 8b ec f8 a3 a8 62 63 90 00` 3 `# Standard ISO-7816 External authenticate:` 4 `APDU: 00 82 02 00 00 00 18 00 00 11 12 8f e3 fa a6 a8 a8 07 10 47` `      e0 af 90 65 20 42 43 2d f0 47 16` 5 `SW:   90 00`	Athena ASEKey USB

0 `# Send 8 random bytes:`
1 `APDU: 80 50 81 01 08 c9 ff 3c d6 63 a2 13 b0`
2 `SW:   61 1c`
3 `# Standard ISO-7816 Get response:`
4 `APDU: 00 c0 00 00 1c`
5 `SW:   35 34 95 09 14 02 1d 3a 03 2a 81 01 03 2a ec a5 97 cc d0 ea`
`      8a cb 05 59 94 78 e1 04 90 00`
6 `# Custom External authenticate:`
7 `APDU: 84 82 03 00 10 fb bb dd 65 5f 0d 70 cc 41 a7 23 47 1d af b0`
`      72`
8 `SW:   90 00`
	RSA SecurID 800

0 `# Standard ISO-7816 Select file:` 1 `APDU: 00 a4 04 00 0c a0 00 00 00 18 0a 00 00 01 63 42 00` 2 `SW:   90 00` 3 `# Standard ISO-7816 Verify:` 4 `APDU: 00 20 00 01 08 31 32 33 34 00 00 00 00` 5 `SW:   90 00`	Safesite Classic TPC IS V1
0 `# Standard ISO-7816 Select file:` 1 `APDU: 00 a4 04 0c 0c a0 00 00 00 63 50 4b 43 53 2d 31 35` 2 `SW:   90 00` 3 `# Standard ISO-7816 Verify:` 4 `APDU: 00 20 00 81 05 31 32 33 34 35` 5 `SW:   90 00`	Siemens CardOS V4.3b

TABLE 3.2: APDU session trace of the PKCS#11 `C_Login` function for the five devices.

Plain authentication. This authentication method is used by Safesite Classic and Siemens CardOS. When the function `C_Login` is called, the PIN is sent as plaintext to the token to authenticate the session. This operation does not return any session handle at the APDU level, meaning that the low level protocol is stateless: a new login is transparently performed by the library before any privileged command is executed. The fact the PIN is sent as plaintext allows to easily sniff the PIN even without having control of the computer, for example using a hardware USB sniffer.

In Table 3.2 we report an excerpt of a real APDU session trace of the `C_Login` function. We can see that Safesite Classic and Siemens CardOS tokens use (line 4) the standard ISO-7816 `VERIFY` command to authenticate: the PIN, in red colour/italic, is sent as a ASCII encoded string ("1234" and "12345", respectively).

Challenge-Response authentication. In the eToken PRO, ASEKey and SecurID tokens the function `C_Login` executes a challenge-response protocol to authenticate the session: the middleware generates a response based on the challenge provided by the token and the PIN given by the user. At the APDU level, eToken PRO and

ASEKey do not return any session handle thus, as for the previous devices, the low level protocol is stateless and a new login is transparently performed by the library before executing any privileged command. Instead, on the SecurID the challenge-response routine is executed only once for each session as it returns a session handle.

PKCS#11 standard allows PIN values to contain any valid UTF8 character, but the token may impose restrictions. Assuming that the PIN is numeric and short (4-6 digits), which is the most common scenario, an attacker is able to bruteforce the PIN offline, *i.e.* without having access to the device, as it is enough to have one `APDU` session trace containing one challenge and one response. As a proof of concept, we have reverse engineered the authentication protocol of eToken PRO and ASEKey implemented in the PKCS#11 library. This allowed us to try all possible PINs and check whether or not the response computed from the challenge and the PIN matches the one in the trace.

In Table 3.2 we can see that eToken PRO makes use of proprietary commands to request the challenge and provide the response, while ASEKey uses the standard ISO-7816 `GET CHALLENGE` and `EXTERNAL AUTHENTICATE` commands. We have not reverse engineered the challenge-response protocol of the SecurID token but, looking at the `APDU` session trace, we can identify a three-steps authentication protocol. At line 1 eight random bytes are sent to the token; then, a standard ISO-7816 `GET RESPONSE` command is issued to retrieve the challenge (highlighted in red and italic at line 5) and the identifier of the PKCS#11 session (highlighted in green and overlined). Line 7 contains the response generated by the middleware.

On both plain and challenge-response authentication, we have found that tokens implement no protection against MITM: if an attacker can place himself in the middle of the connection he could exploit an authentication exchange to alter user commands or inject his own ones.

3.2.2 Sensitive symmetric keys

We discovered that in Siemens CardOS, eToken PRO and SecurID encryption and decryption under a sensitive symmetric key is performed entirely by the middleware. As a consequence, the value of the sensitive key is sent out of the token as plaintext. This violates the basic PKCS#11 property stating that sensitive keys should never be exported in the clear. We also found that ASEKey surprisingly reuses the authentication challenge (sent in the clear) as the value of freshly generated DES keys.

In the following, we describe the four devices separately.

Siemens CardOS V4.3b. This smartcard does not allow to create symmetric keys with `CKA_TOKEN` set to true, meaning that symmetric keys will always be session keys. According to PKCS#11 documentation, session keys are keys that are not stored permanently in the device: once the session is closed these keys are destroyed. Notice that this does not mean that sensitive session keys should be exported in the

APDU session trace	Token
# DES Key generation: red/italic = plain key value sent to the token APDU: 80 16 01 00 2b 01 01 02 02 02 40 01 03 02 00 18 04 04 11 11 11 11 10 18 17 3f ff ff ff ff 01 08 *3f 44 5f c4 eb 76 f1 86* 06 64 65 73 6b 65 79 00 SW: 90 00	C_GenerateKey sample on Aladdin eToken PRO
# Fetch the key: green/overlined = attributes, red/italic = plain key value, blue/underlined = label APDU: 80 18 00 00 04 0e 02 00 00 18 SW: *17 3f ff ff ff ff 01* 08 *3f 44 5f c4 eb 76 f1 86* 06 *64 65 73 6b 65 79 00* 90 00	C_WrapKey sample on Aladdin eToken PRO
# 3DES Secret key generation APDU: 80 16 00 00 1a 72 35 *be 4e aa de 2d 47 72 b2 8b 47 5f de 63 4d 7e 30 a5 f0 ac 5f c0 56 c6 90* SW: 90 00	C_GenerateKey sample on RSA SecurID 800
# 3DES key is read in the clear even if CKA_SENSITIVE is set to true APDU: 00 c0 00 00 18 SW: *36 90 fa c9 4e 82 55 b1 71 1d 81 e4 3c d1 bd fa 44 9c bb c3 b1 8b 1e 8d* 90 00	C_GetAttribute-Value sample on RSA SecurID 800
# Get challenge (Standard ISO-7816): APDU: 00 84 00 00 00 00 08 SW: *b7 c8 14 4b 4e 5f e6 3e* 90 00 # External authenticate (Standard ISO-7816): APDU: 00 82 02 00 00 00 18 00 00 11 12 95 fa da de 0d 70 42 d9 21 c2 27 a4 8b af 7a 8b 90 47 ae 54 SW: 90 00 # Get an RSA modulus (in red/italic) SW: *79 23 57 33 9a be 2a dd ba ae 2e 09 4c d0 3d 57 8b d0 07 e4 ... (omitted) ... 19 6d 15 ea b6 aa cc 2b e8 30 c3 e8 cf* 90 00 # Send the encrypted key to the token APDU: 80 24 00 80 00 00 a0 *20 5b f1 f9 cd 67 c8 3d e0 cf 9b 1b c7 ... (omitted) ... 33 0b 85 1a 27 7e cd 69 95 71 ca 2e 88 33 a7 f6 4a 97 22 a0* SW: 90 00	C_GenerateKey sample on Athena ASEKey

TABLE 3.3: Leakage of sensitive symmetric keys during PKCS#11 operations.

clear out of the token. What distinguishes a session key from a token key is *persistence*: the former will be destroyed when the session is closed while the latter will persist in the token.

We observed that encryption under a sensitive key sends no APDUs to the token. This gives evidence that encryption takes place entirely in the middleware. Moreover, we verified that even C_GenerateKey function does not send any APDU: in fact, it just calls the library function pkcs11_CryptGenerateRandom to generate a random key value whose value is stored (and used) only inside the library.

Aladdin eToken PRO. In Table 3.3 (first row), we show that symmetric key generation in eToken PRO is performed by the middleware. We can see, in red and italic, a DES key value sent to the token in the clear.

The value of symmetric keys stored in the eToken PRO can be read by using the proprietary APDU command 0x18. No matter which attributes are set for the key, its value can be read. We tested it over a DES key with attributes CKA_TOKEN, CKA_PRIVATE, CKA_SENSITIVE set to true. In order to perform this attack a valid login is required. Since symmetric key operations are performed by the library, this APDU command is used to retrieve the key from the token before performing operations in software.

As an example, in Table 3.3 (second row) we see part of a C_WrapKey operation that retrieves a the DES cryptographic key from the token. We can see the value of the key in the clear.

RSA SecurID 800. In Table 3.3 (third row), we show that symmetric key generation in SecurID is also performed by the middleware. We can see, in red and italic, a 3DES key value sent to the token in the clear.

We were also able to retrieve the value of a sensitive key stored inside the SecurID by just issuing the correct APDU command. In fact, when trying to use the C_GetAttributeValue function, the library correctly returns the CKR_ATTRIBUTE_-SENSITIVE error. However, what really happens is that the key is read from the token but the library just avoids to return it. In Table 3.3 (fourth row) we can see (in red and italic) the value of the sensitive key leaked by the token.

Athena ASEKey. The most surprising behaviour is shown by the ASEKey: the value of token sensitive symmetric keys cannot be read arbitrarily via APDU commands, as they are stored in a separated Dedicated File (DF) which requires authentication. Nonetheless the key value is unnecessarily leaked when the key is generated.

In Table 3.3 (fifth row) we report an excerpt of APDU session for the C_GenerateKey function. We notice that C_GenerateKey sends (line 9) the key encrypted under RSA with a modulus (line 7), using the public exponent 0x010001. In fact, the library encrypts the whole Elementary File (EF) containing the key value, that is going to be written in the token. This means that special care was taken to avoid leaking the value as plaintext when importing it in the token. Unfortunately the key value already appeared in the clear: quite surprisingly, key generation re-uses the 8-bytes random string which is used by the authentication step (line 2) as the sensitive key value.

As a proof of concept, we encrypted a zero-filled 8-bytes buffer using the C_-Encrypt function with the generated key and a null initialization vector. We then performed the same encryption using the 8-bytes challenge as the DES key value obtaining the same value.

3.2.3 Bypassing Attribute Values

In all five tokens examined, PKCS#11 attributes are interpreted by the middleware and do not have any import on the internal behaviour of the token. We performed a simple test by signing a text using an RSA key having the attribute CKA_SIGN set to false:

1. take a private RSA key with CKA_SIGN false;
2. verify that it cannot sign a message via the PKCS#11 API, as expected;

```
0 # Manage security environment
1 APDU: 00 22 41 b6 06 80 01 12 84 01 07
2 SW:   90 00
3 # Custom perform security operation
4 APDU: 80 2a 9e ac 16 90 14 59 b7 b5 0c 2e 69 4e 3f 7e 2f 06 7f 07 1d 8e dd de ba 8c c0
5 SW:   61 80
6 # Custom getData
7 APDU: 80 c0 00 00 80
8 SW:   9d 70 aa 8d c4 af 7a 88 ba e4 6c ab 47 3e 02 19 81 e5 85 53 8a 6a 1b 83 8c 73 39
        29 9e 49 bb 24 a7 27 4f 8e 38 60 b6 d1 71 c6 92 75 58 fe 33 78 d2 fe 99 5c 96 4e
        3e 43 15 9d 67 f9 db 7b 8b 3c 29 d4 97 d5 ec 2e 46 7e 2b c9 c4 92 0f 38 eb 65 11
        2b e1 ba 61 33 7c a1 03 62 f4 2c 2c f2 52 85 2a ee ab 77 ca 6e 37 8e 3b 5a 57 dd
        c1 64 ea d0 76 71 2a 46 0b bc d4 2a ef c0 6c 32 77 c3 5e 79 90 00
```

LISTING 3.1: Forced signature sample

```
>>> signed = 0x9d70aa8dc4af7a88bae46cab473e021981e585538a6a1b838c7339299e49bb24a7274f8e3860b6d171c6927558f
    e3378d2fe995c964e3e43159d67f9db7b8b3c29d497d5ec2e467e2bc9c4920f38eb65112be1ba61337ca10362f42c2cf252852
    aeeab77ca6e378e3b5a57ddc164ead076712a460bbcd42aefc06c3277c35e79
>>> modulus = 0xc1886b5f26ad5349426b8e8bfc9f73385d14f6cf2b2f1d95b080ae2df7a1db11b91d36db33f3b98f1687177471
    1c03b22d7d97939062031df2d15371173b468f9986701d144f315005ec99a71b226fc71b956608c60747ceb4ac0c3725b7d044
    84ac286196975f18911361e28ec50b661273362131b4a4183e01667b090c96f9
>>> pubkey = 0x010001
>>> hex(pow(signed, pubkey, modulus))
'0x1ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ff0059b7b50c2e694e3f7e2f067f071d8edddeba8cc0L'
```

LISTING 3.2: Signature verification in Python

3. perform the sign operation manually, via `APDU`, using the private key and the message. Some tokens use the standard ISO-7816 command `PERFORM SECURITY OPERATION` and some others use a proprietary command but, in both cases after sniffing, it is easy to replicate any valid `APDU` trace for a signature.

This confirms that the low-level behaviour of the token is not compliant to PKCS#11 specification as it allows to perform signature under a key that has `CKA_SIGN` attribute set to false. Since the behaviour of all five tokens is similar, in Listing 3.1 we illustrate the case of Safesite Classic as a representative `APDU` example trace. At line 4 the message is sent to the token and, at line 8, the corresponding signature is returned.

We can verify that signature corresponds using Python shell, as shown in Listing 3.2. In particular, notice that the obtained message corresponds to the one we signed.

3.2.4 RSA Session Keys

When using session RSA keys on the eToken PRO, we discovered that key generation, encryption and decryption operations are performed inside the library. This means that the value of the private key is exposed in the clear out of the token.

Even if one might regard to session keys as less important than long-term keys, as we already discussed in Section 3.2.2 for Siemens CardOS, PKCS#11 still requires that if such keys are sensitive they should not be exported out the token in the clear. For example we can generate a session key which, at some point before the end of the

Token	Auth.	Sensitive symmetric keys		Bypassing attribute values	RSA session keys	
		PKCS#11[1]	APDU		PKCS#11[1]	APDU
eToken PRO	✓[2]	✓	✓	✓	✗	✓[4]
ASEKey	✓[2]	✗	✓[3]	✓	✗	✗
SecurID	✓[2]	✓[5]	✓	✓	✗	✗
Safesite Classic	✓	✗	✗	✓	✗	✗
Siemens CardOS	✓	✗	✓[4]	✓	✗	✗

[1] PKCS#11-level attacks discovered in [26], for comparison.
[2] Requires reverse engineering of the authentication algorithm and bruteforcing.
[3] Leakage occurs only during generation.
[4] Requires access to middleware memory.
[5] Possible for RSA Authentication Client version < 3.5.3.

TABLE 3.4: Summary of the vulnerabilities found .

session, is persisted in the token's memory by calling the `C_CopyObject` function. Clearly this newly created object cannot be considered secure as the value of the private RSA key has already been leaked in the clear out of the token.

3.3 Security Analysis

In Table 3.4 we summarize the APDU-level attacks we found on the five devices. In the columns labelled PKCS#11 we also report the PKCS#11 attacks from [26], for comparison. In particular, the only token that allows for PKCS#11 Clulow-style attack extracting a sensitive key in the clear is eToken PRO. For SecurID we reported that it was possible to directly read the value of sensitive symmetric keys and RSA released a fix starting from RSA Authentication Client version 3.5.3.[3] In the literature we found no known API-level attacks on sensitive keys for the remaining devices.

All devices are affected by attacks on the PIN, some of which requiring reverse engineering and brute forcing, and by attacks bypassing key attributes. For what concerns sensitive keys, only Safesite Classic is immune to attacks. For the remaining four tokens we have reported new attacks that compromise sensitive keys that are instead secure when accessed from the PKCS#11 API.

In order to clarify under which conditions the attacks are possible we cross-compare Table 3.1 with Table 3.4 producing table Table 3.5. In particular, for each device we take the vulnerabilities reported in Table 3.4 and we check from Table 3.1 if the combination attacker / application offers the necessary conditions for the attack. We omit the Admin attacker as it is in fact equivalent to the User attacker when the application is monolithic. In particular, we observe that:

[3]See `https://secgroup.dais.unive.it/projects/tookan/`

Attacker	Application	Auth.	Sensitive symmetric keys		Bypass attribute values	RSA session keys	
			PKCS#11[4]	APDU		PKCS#11[4]	APDU
Aladdin eToken PRO							
User	Monolithic	✓	✓	✓	✓	✗	✓
	Sep. Auth.	✗	✓[1]	✓	✓[1]	✗	✓
	Sep. Privileges	✓	✓	✗	✗	✗	✗
	Sep. Auth.&Priv.	✗	✓[1]	✗	✗	✗	✗
Physical	Any	✓[2,5]	✓[1]	✓	✓[1]	✗	✗
Athena ASEKey							
User	Monolithic	✓	✗	✓	✓	✗	✗
	Sep. Auth.	✗	✗	✓[6]	✓[1]	✗	✗
	Sep. Privileges	✓	✗	✗	✗	✗	✗
	Sep. Auth.&Priv.	✗	✗	✗	✗	✗	✗
Physical	Any	✓[2,5]	✗	✓	✓[1]	✗	✗
RSA SecurID 800							
User	Monolithic	✓	✓[7]	✓	✓	✗	✗
	Sep. Auth.	✗	✓[1,7]	✓	✓[1]	✗	✗
	Sep. Privileges	✓	✓[7]	✗	✗	✗	✗
	Sep. Auth.&Priv.	✗	✓[1,7]	✗	✗	✗	✗
Physical	Any	✓[2,5]	✓[1,7]	✓	✓[1]	✗	✗
Safesite Classic TPC IS V1							
User	Monolithic	✓	✗	✗	✓	✗	✗
	Sep. Auth.	✗	✗	✗	✓[1]	✗	✗
	Sep. Privileges	✓	✗	✗	✗	✗	✗
	Sep. Auth.&Priv.	✗	✗	✗	✗	✗	✗
Physical	Any	✓[2]	✗	✗	✓[1]	✗	✗
Siemens CardOS V4.3b							
User	Monolithic	✓	✗	✓	✓	✗	✗
	Sep. Auth.	✗	✗	✓	✓[1]	✗	✗
	Sep. Privileges	✓	✗	✗	✗	✗	✗
	Sep. Auth.&Priv.	✗	✗	✗	✗	✗	✗
Physical	Any	✓[2]	✗	✗	✓[1]	✗	✗

[1] Requires MITM.
[2] Through a keylogger or a USB sniffer.
[3] Only APDU payloads, cannot access middleware memory.
[4] PKCS#11-level attacks discovered in [26], for comparison.
[5] Requires reverse engineering of the authentication algorithm and bruteforcing.
[6] Leakage occurs only during generation.
[7] Possible for RSA Authentication Client version $< 3.5.3$.

TABLE 3.5: Summary of vulnerabilities with respect to attackers and applications.

User / Monolithic the attacker can attach to the process and eavesdrop the PIN at the PKCS#11 level. Knowing the PIN the attacker can perform any operation and inspect the process memory. So all attacks of Table 3.4 are enabled;

User / Separate authentication mechanism the attacker cannot eavesdrop the PIN directly. Interestingly PKCS#11-level attacks and attribute bypass are still possible through a MITM on the middleware. Moreover, `APDU`-level attacks on keys are still valid as they only require to eavesdrop the `APDUs`;

User / Separate privileges the attacker can still eavesdrop the PIN and work at the PKCS#11 level but all `APDU`-level attacks are prevented. In this setting the only insecure token is eToken PRO since it allows for PKCS#11-level attacks on sensitive keys;

User / Separate authentication and privileges this is the more secure setting: the attacker con only perform PKCS#11-level attacks on eToken PRO through a MITM, since he cannot learn the PIN. All the other tokens are secure;

Physical / Any application through a keylogger or a USB sniffer the attacker can learn the PIN. In case of a USB sniffer, for the tokens adopting challenge-response it is also necessary to reverse-engineer the protocol in the library and perform brute-forcing on the PIN. `APDU`-level attacks are possible only when the keys are transmitted from / to the device. So, for eToken PRO RSA session keys and Siemens CardOS symmetric keys the attacks are prevented, as keys are directly handled by the library and are never transmitted to the device. Other attacks can be performed only through a MITM at the USB level.

3.3.1 Fixes and Mitigations

Compliant PKCS#11 devices should implement all the cryptographic operations inside the hardware. This would prevent all of the attacks we have discussed so far, except for the ones on authentication. However, fixing this at the hardware level requires to redesign the device and is probably just not affordable, in general.

We have seen, however, that having separate authentication and privileges is a highly secure setting that fixes the problem of cryptographic operations implemented at the library level and, at the same time, protects PIN authentication. It is worth noticing that running the middleware with separate privileges can be done transparently to the application while having separate authentication requires to modify the application so that the login step is managed by separate software or hardware.

An alternative solution to mitigate attacks on PIN, with no changes in applications, could exploit the OTP functionality of the devices with a display, such as SecurID. A one-time PIN might be generated by the token and shown on the display asking the user to combine it with the secret token PIN. In this way, offline brute-forcing would be slowed down by the longer, combined PIN and, even if successful, would require physical access to the token in order to re-authenticate since part of the PIN is freshly generated by the token each time the user authenticates.

3.4 Discussion

We have presented a new threat model for the PKCS#11 middleware and we have analysed the `APDU`-level implementation of the PKCS#11 API for five commercially available devices. Our findings show that all devices present `APDU`-level attacks that, for four of them, make it possible to leak sensitive keys in the clear. The only smartcard immune to attacks to keys is Safesite Classic. We have also found that all devices are vulnerable to attacks that leak the PIN if the middleware is not property isolated and run with a different privilege (which is usually not the case). Moreover, attackers with physical access could sniff an authentication session through the USB port and brute-force the PIN once the authentication protocol has been reverse-engineered.

We have reported our finding to manufacturers following a responsible disclosure principle and we have interacted with some of them to provide further information and advices.

Related Work

Many cryptographic API-level attacks have been published in the last 15 years. The first one is due to Longley and Rigby [65] on a device that was later revealed to be a Hardware Security Module manufactured by Eracom and used in the cash machine network. In 2000, Anderson published an attack on key loading procedures on another similar module manufactured by Visa [1] and presented more attacks in two subsequent papers [22, 23]. Clulow published the first attacks on PKCS#11 in [32]. All of these attacks had been found manually or through ad-hoc techniques. A first effort to apply general analysis tools appeared in [128], but the researchers were unable to discover any new attacks and could not conclude anything about the security of the device. The first automated analysis of PKCS#11 with a formal statement of the underlying assumptions was presented in [39]. When no attacks were found, the authors were able to derive precise security properties of the device. In [26], the model was generalized and provided with a reverse-engineering tool that automatically refined the model depending on the actual behaviour of the device. When new attacks were found, they were tested directly on the device to get rid of possible spurious attacks determined by the model abstraction. The automated tool of [26] successfully found attacks that leak the value of sensitive keys on real devices.

Low-level smartcard attacks have been studied before but no previous `APDU`-level attacks and threat models for PKCS#11 devices have been published in literature. In [10], the authors showed how to compromise the APDU buffer in Java Cards through a combined attack that exploits both hardware and software vulnerabilities. In [61], the authors presented a tool that gives control over the smart card communication channel for eavesdropping and man-in-the-middle attacks. In [75], the authors illustrated how a man-in-the-middle attack can enable payments without knowing the card PIN.

In [43] a subset of the authors investigated an automated method to systematically reverse-engineer the mapping between the PKCS#11 and the APDU layers. The idea is to provide abstract models in first-order logic of low level communication, on-card operations and possible implementations of PKCS#11 functions. The abstract models are then refined based on the actual APDU trace, in order to suggest the actual mapping between PKCS#11 commands and APDU traces. The two papers complement each other: the present one illustrates real attacks with a threat model and a security analysis, while [43] focuses on automating the manual, non-trivial reverse engineering task. All of the attacks presented here have been found manually and some of them have been used as test cases for the automated tool of [43].

Finally, for what concerns the threat model, in the literature we find a number of general methodologies (e.g., [106, 115, 125]) that do not directly apply to our setting. In [38] the authors discussed threat modelling for security tokens in the setting of web application while [103] described in details all the actors and threats for smart cards, but none of these papers considered threats at the PKCS#11 middleware layer. To the best of our knowledge, the threat model we propose in this work is the first one in the setting of PKCS#11 tokens and smartcards which takes into account the APDU layer as an attack entry point.

Chapter 4

Physical Attacks in Embedded Systems

Attackers with physical access to a target device can exploit hardware side-channels to leak data, even under the assumption of sound cryptographic designs and implementations. In this chapter we investigate on voltage fault injection, a class of active side-channel attacks that induces faults by creating disturbances, namely voltage glitches, in the power supply of a device. We study how variations in the glitch waveform influence the attack performance and describe a new technique for generating this type of faults using low-cost equipment. As a case study for assessing the effectiveness of our approach, we present unpublished attacks that allow for extracting the firmware from the internal protected memory of six popular microcontrollers. Interestingly, this is the result of several consultancies for the analysis and reverse-engineering of cryptographic implementations that we operated in the automotive field. Cryptography is, in fact, widely used in the read-protected firmware found in on-board electronics of modern vehicles.[1]

Contributions. Our contributions can be summarized as follows:

(i) We investigate the effect of different glitch waveforms in the setting of voltage fault injection attacks and, in particular, we propose a new method for the generation of arbitrary glitch waveforms using a low-cost and software-managed setup;

(ii) we report on unpublished vulnerabilities and weaknesses in six microcontrollers from three major manufacturers: STMicroelectronics, Texas Instruments and Renesas Electronics. We combine these vulnerabilities and describe the attacks for extracting the firmware from the internal read-protected flash memory. All the attacks are non-destructive and can be performed with a black-box approach, *i.e.*, without any knowledge of the firmware code;

(iii) we evaluate the attack performance of our method by comparing the speed, efficiency and reliability of our solution against two popular V-FI techniques.

Structure of the chapter. In Section 4.1 we describe our experimental setup and equipment; in Section 4.2 we introduce our arbitrary glitch waveform technique and we show how to automatically identify and optimize the glitch shape; in Section 4.3 and 4.4 we report on unpublished vulnerabilities of six microcontrollers and describe the attacks for extracting the firmware; in Section 4.5 we empirically evaluate our technique by comparing it with two popular V-FI techniques, we discuss limitations and propose possible improvements; finally, in Section 4.6 we draw some concluding remarks and present the related work.

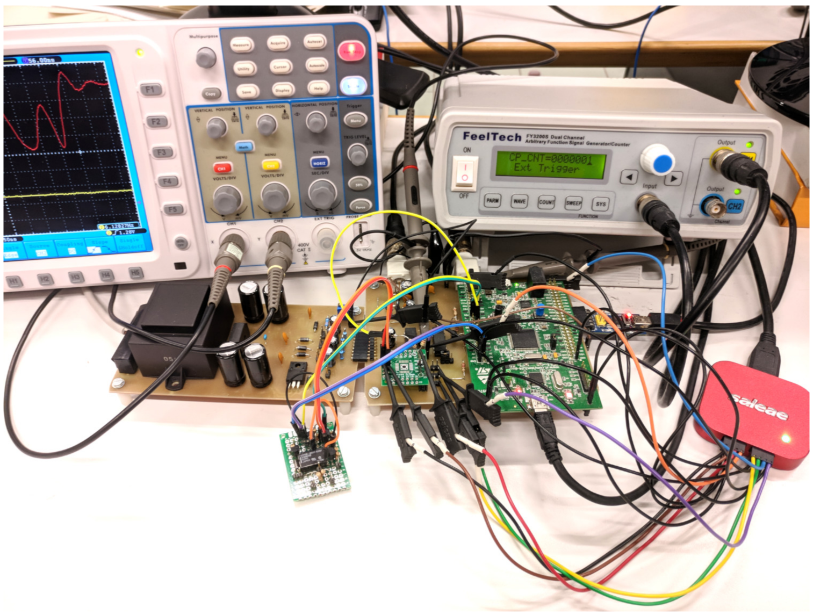
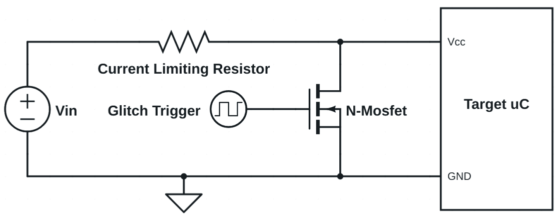
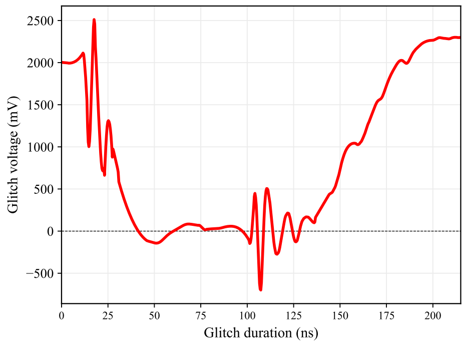


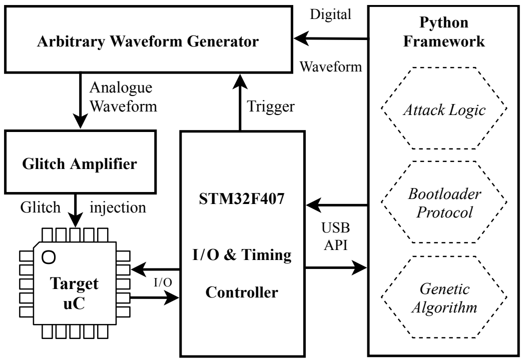
FIGURE 4.1: Picture of the actual experimental setup, including the arbitrary voltage glitch generator.



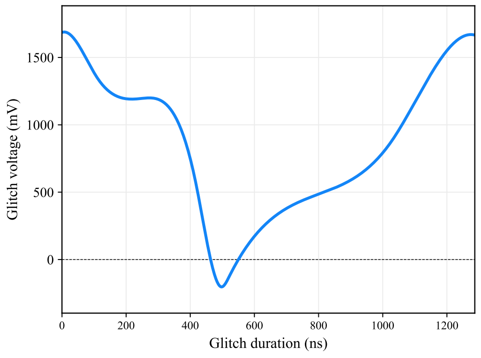
(A)



(B)



(C)



(D)

FIGURE 4.2: A typical transistor-based V-FI setup (4.2a) and a generated glitch (4.2b). The oscillations depend on the target, components in use, and electronic properties of the power supply line. Our V-FI setup for generating arbitrary glitch waveforms (4.2c) and a generated glitch (4.2d).

4.1 Experimental Setup

Based on the work in [57], we developed a low-cost programmable V-FI setup that enables us to overcome the limitations of the transistor-based glitch generation circuit (*cf.* Section 1.2). Our setup, depicted in Figure 4.1, is designed around the Digital Direct Synthesis (DDS, *cf.* [36]) technology: a DDS signal generator outputs an arbitrary waveform from a software-defined set of parameters. Similarly to the generation of analogue audio from a digital source, the digital waveform is fed to a Digital-to-Analog Converter (DAC) for producing the equivalent analogue signal. Since in this work we target general purpose microcontrollers that operate at sub-GHz speed, we chose an off-the-shelf DDS device with a reasonable trade-off between performance and price: the FeelTech FY3200S, a very low cost (about 50 $) Arbitrary Waveform Generator with 6 MHz analog bandwidth and ± 10 V output range. This model has an internal waveform memory of 2048 points and allows for controlling the output waveform with 12-bit vertical resolution, using a publicly available protocol over USB.[2] We use this device as a source of both glitch signal and constant voltage for regular MCU operation: the transition from a constant voltage to the glitch is controlled by an external input trigger. As depicted in Figure 4.2c, we wired the generator to a custom designed board which has has three major functionalities: (i) provide the glitch amplification stage and signal path from the generator to the target MCU; (ii) interface with the target MCU, handling the low-level communication and time-critical operations; (iii) expose a convenient API to control any aspect of the attack from a computer.

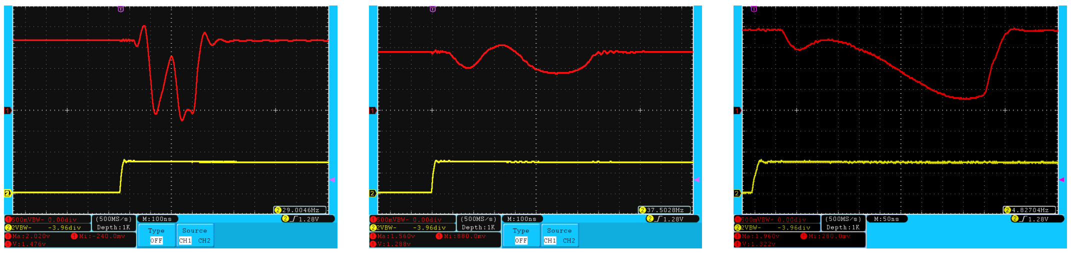
The amplification stage is designed after the arbitrary waveform generator bandwidth and the power requirements of a general purpose, low-speed MCU. We used a THS3062[3] current-feedback, high slew rate operational amplifier with 150 mA output current capability working in a 2-stage, non-inverting configuration. A reed relay is placed in-between the amplifier output and the target, allowing to fully cut off the device power supply when needed, without any intervention on the generator output.

An ARM STM32F407 microcontroller operating at 168 MHz is responsible for running the firmware that controls the board. We designed the firmware using the minimum code required for handling the low-level communication with the target MCU (*e.g.*, UART, I^2C, SPI) and glitch triggering. Upon detection of an external event, the built-in hardware timer guarantees a 10 ns resolution for signalling the generator to inject the glitch after a specific time delay. An API allows for controlling the board and interacting with the target MCU from a computer via a USB link.

[1]Due to non disclosure agreements, we are not allowed to disclose further details about the specific vehicle manufacturers and cryptographic algorithms that we reverse-engineered.

[2]Notice that, since the waveform upload speed is low, we modified the generator to bypass the built-in upload mechanism, improving the upload speed from about 30 s to 200 ms

[3]`http://www.ti.com/lit/ds/symlink/ths3062.pdf`



(A) STM32F373 (B) TI MSP430F5172 (C) Renesas 78K0R

FIGURE 4.3: Oscilloscope trace of the voltage glitch for the STM32F373 (4.3a), the TI MSP430F5172 (4.3b) and the Renesas 78K0R (4.3c). The red (upper) trace represents the waveform of the voltage glitch, while the yellow (lower) is the trigger.

All the complex tasks or algorithms, the attack logic and the specific communication protocol used by the target are implemented in a custom Python framework that assists the design and execution of an attack. As a result, the task of mounting an attack and switching to a different target is substantially simplified. The framework is also responsible for commanding the waveform generation and for controlling the attack parameters, including the search and optimization phase (*cf.* Section 4.2.1).

4.2 Arbitrary Waveform Voltage Glitch

The DAC-based voltage glitch generator described in Section 4.1 enables high flexibility by allowing the attacker to control all typical V-FI parameters (*i.e.*, power supply voltage, glitch voltage, timing and duration) in software and to produce both negative and positive voltage spikes. Up to minor variations due to trace capacitance and impedance, the generated waveform is also repeatable and predictable and it is not influenced by the characteristics of the particular MOS-FET transistor in use, *e.g.*, on-state resistance, capacitance, rise and fall times. However, the most important feature of this setup that we are interested in, is the ability to generate glitch waveforms with arbitrary shape.

In the literature it has been shown (see [131]) that rising and falling edges of a voltage glitch play a crucial role in producing oscillations of the core voltage of a FPGA. Since these oscillations cause computation errors that amount to setup time violations in the circuit, in the present work we move a step forward and experiment on the effectiveness of using non-standard glitch waveforms for fault injection on general purpose microcontrollers. In the literature (*cf.* [117, 57]) Digital-to-Analog converters have already been proven effective for V-FI. However, to the best of our knowledge this work is the first that investigates on using a DAC as a source of arbitrary glitch waveforms, which range from sharp pulses to smooth and variegate waveforms, as exemplified in Figure 4.3. Depending on its characteristic, the generated waveform can induce effects resembling regular voltage glitches or, for instance, a combination of underpowering [14, 105, 130], negative, positive or multiple voltage glitches. Our experiments (*cf.* Section 4.5) suggest that, when performing a fault injection attack, the attack success is strongly influenced by the particular waveform of the glitch. However, we point out that thorough investigations are still necessary to identify the precise, low-level effects of different waveforms on the setup time of combinational logic. We leave this as a future work.

Parameter space. Typically, the set of attack parameters that need to be adjusted are timing, glitch length, glitch voltage and possibly power supply voltage [30, 88, 76]. Our technique adds extra parameters for defining the glitch waveform, described as a function of time where the result is the instantaneous voltage generated. This function is translated into the parameter space as a finite set (from 4 to 10) of (x, y) coordinates that are interpolated by cubic interpolation on a 2048-by-4096 grid, and fed into the DAC. Then, the glitch length is encoded as frequency or period of the arbitrary waveform generated.

4.2.1 Parameter Search and Optimization

We define parameter search as the task of finding one set of parameters that successfully induces one or more faults, implementing a given attack logic. To improve the attack performance, a further optimization phase can be employed for identifying the set of injection parameters that *maximize the probability* of a successful fault. The search phase is mandatory for designing and mounting an attack while the optimization step is subject to the specific requirements and complexity of the attack.

In the following we summarize the the steps performed by the attacker for designing and optimizing a V-FI attack: (i) perform an initial parameter search; (ii) implement the attack scheme and identify $N_{glitches}$ and T_{glitch}, that is the amount of successful faults required and the time spent for injecting one fault, respectively; (iii) define a target time T_{attack} representing the duration under which the attack is considered practical. This can vary from hours to days depending on the attack complexity and attacker's expectations; (iv) define a maximum time $T_{timeout}$ for the optimization step, typically as a fraction of T_{attack}; (v) iterate the optimization step until the success rate $R_{success}$ of the fault attack is such that $R_{success} \geq (T_{glitch} \cdot N_{glitches}) \ / \ T_{attack}$, or the timeout $T_{timeout}$ is reached.

Given the increased number of parameters introduced by our technique, it is important to devise efficient techniques that make optimization feasible and practical. In the following, we describe a semi-automated supervised search (*cf.* Section 4.2.1) and a fully automated unsupervised search based on genetic algorithms (*cf.*, Section 4.2.1).

Supervised Search

Since finding the correct parameter setting is a highly nondeterministic process [88], during our early experiments we used a human-supervised random search approach inspired by the *Adaptive zoom & bound* method proposed by Carpi *et al.* in [30]. First, we randomly generate and interpolate the set of (x, y) points describing the candidate arbitrary glitch waveform. Then, we iteratively select a random sample from each parameter interval and test the obtained combination. This process is repeated and the results are manually evaluated, reducing the parameter space accordingly until one solution is found. Clearly this approach is slow to converge and requires expertise for evaluating the results. Additionally, the parameters are not independent: for instance, altering the glitch waveform or duration can affect the glitch trigger position.

Unsupervised Genetic Algorithm

Based on the work of Picek *et al.* [88], we developed a framework that enables for identifying and optimizing the attack parameters in an unsupervised way. It is designed over a classic genetic algorithm (GA) structure, where an initial population of candidate solutions (the attack parameters) is randomly sampled and an iterative process is responsible for finding a solution that maximizes a *fitness value* F. The fitness value is typically represented by the number of successful glitches produced by a specific set of parameters, but it can be further improved by accounting for additional factors, *e.g.*, the success / failure ratio and the amount of target hangs or reset, as a penalty factor. As an example, in Section 4.4.1 we assign a negative score to the fitness value in the case of a false positive, *i.e.*, an incorrect byte extracted. A solution is composed of one combination of all the members in the parameter space and, at each *generation*, the solutions *evolve* and the attack results are used to evaluate the new fitness value. Since our goal is to find a working solution which is also optimal, *i.e.*, the parameters providing the best attack performance, we repeatedly test one candidate solution and calculate $F(solution) = \frac{S}{T}$, where S is the number of successful attacks and T the total number of tests. We start with 50 tests per candidate and increment this value at each generation: as a result, the first generations enable to test more solutions, while the last are more accurate in evaluating the candidates performance.

At each generation, the population of solutions is improved through repetitive application of the selection, mutation, crossover, and replacement operators. We tuned these operators to the specific characteristics of our V-FI technique:

Selection we tested both the *fitness proportionate selection* and the *tournament selection* standard GA methods of selecting an individual from the population of individuals, and found that both produce acceptable results;

Crossover we use a *uniform crossover* so that, in particular, every single (x, y) point in the glitch waveform can be mixed between two parents with a 0.5 probability;

Mutation every parameter has a different mutation probability. The glitch duration parameter has the highest probability; the glitch waveform has a greater probability in the first generations, together with a higher likelihood of mutating by a small extent;

Replacement a *replace-worst* strategy is adopted, which replaces the worst individual of the current population.

Results. With respect to the parameters optimized by an expert using the supervised approach of Section 4.2.1, our experimental results show that the solutions identified by this algorithm produce the same, or higher attack performance. For each case study presented in this chapter, in fact, the attack parameters (*cf.* Section 4.3.3 and Section 4.4.2) have been automatically optimized using GA. As described above in Section 4.2.1, item (v), the optimization converges when a solution that delivers an acceptable performance level is found, *i.e.*, when the success rate $R_{success}$ is good enough to make the attack complete within time T_{attack}; otherwise it is stopped when the timeout $T_{timeout}$ is reached. During our experiments the average time to converge was in the range of 30 minutes to 10 hours, depending on the target device, the vulnerability and the size of the parameter search space. Finally, we point out that the attack and the parameter optimization can be interleaved so to achieve a continuous performance improvement, avoiding unnecessary voltage glitches dedicated exclusively to the optimization phase.

4.3 Scattered-glitch Attacks

In this section, we assess the effectiveness of arbitrary glitch waveforms described in Section 4.2 against two case studies of low/moderate complexity (Section 4.3.1 and Section 4.3.2). Specifically, the presented attacks exploit a single vulnerability, require a limited amount of glitches ($\leq 100\,\text{k}$) and can be completed in a short time frame: from minutes to a few hours. All the presented attacks are novel and extract the firmware from the internal flash memory of the target microcontroller, by exploiting vulnerabilities either in the bootloader or in the debug interface.

In Section 4.4 we will consider a third case study of increased complexity.

Attacker model. We assume the attacker knows the MCU model under attack and has physical access to the target device. As such, the attacker can directly connect to the exposed pins of the chip, desolder it from the PCB or tamper with the PCB in order to isolate the chip from the other electronic components, minimizing interference. The attacker has no information about the running firmware and the flash memory content but, although not mandatory, she has the ability to inspect the bootloader code in order to identify a suitable instruction to fault.

4.3.1 Case Study 1: STMicroelectronics

We consider two STM32 ARM MCUs belonging to the F1 and F3 series and manufactured by STMicroelectronics. We select this microcontroller family since it is one of the most widespread in consumer electronics, with over 1 billion units sold between 2007 and 2015 [114]. Most STM32 can be programmed either via JTAG / SWD or via the integrated serial bootloader. On USB-enabled MCUs the standard Device Firmware Upgrade (DFU) protocol [120] is often available.

STM32 F1

We select the STM32F103 as a representative of the F1 series. This model is equipped with a 32-bit ARM Cortex-M3 core operating at 72 MHz.

Security mechanisms. The bootloader offers a security mechanism to lock the device and prevent any read or write operations on the flash memory. In particular, we are interested in the *Readout Protect* command that enables the read protection (RDP) feature. If enabled, the bootloader returns a negative response (NACK) when a *Read Memory* command is issued. The *Readout Unprotect* command disables the read protection at the cost of a complete flash memory erasure.

Attack. We easily bypass this protection mechanism by attacking the *Read Memory* command. After the user requests a read operation, the CPU checks the RDP value and returns the positive (ACK) or negative (NACK) response. By injecting a fault during the RDP checking phase, the bootloader can be deceived into returning an ACK despite the active read protection mechanism. Thus, it is enough to issue a *Read Memory* command over a memory block followed by a voltage glitch, and repeat this until an ACK is received and the content of the selected memory block is returned. The attack is then iterated over the subsequent memory blocks.

STM32 F3

We select the STM32F373 as a representative of the F3 series, equipped with an ARM Cortex-M4 core.

Security mechanisms. A hardware memory protection unit (MPU) implements runtime access control to memory and the SRAM parity errors are checked in hardware. The CPU power supply is provided by an internal voltage regulator and a power supply supervision (PVD) circuit is responsible for holding the device in reset state while the input voltage is outside the working range. Compared to the F1 series (*cf.* Section 4.3.1), the flash memory read protection mechanism is enhanced by using a configurable RDP with three levels of protection. At Level-0 the MCU is unprotected. Level-1 grants access to main memory only when in user mode. *i.e.*, when executing regular firmware code. If, instead, the CPU is running the bootloader or is in debug mode (*e.g.*, via JTAG / SWD), then the flash memory is inaccessible. Finally, the Level-2 protection disables the bootloader and any CPU debugging capability.

RDP Level	RDP	$\overline{\text{RDP}}$	Security Features
Level-0	`0xAA`	`0x55`	None (unprotected)
Level-1	*Any other value*		Debug w/o flash memory access
Level-2	`0xCC`	`0x33`	No debug (maximum protection)

TABLE 4.1: Values of RDP and complement ($\overline{\text{RDP}}$) bytes with respect to the RDP protection levels in the STM32 F3. Notice how a single bit flip can downgrade the protection mechanism to Level-1.

Moreover, programming the RDP to Level-2 is an irreversible operation both for the user and for STMicroelectronics. Interestingly, the reference manual [113] points out that the RDP is not a software protection mechanism but it is rather implemented at the hardware level, possibly in the MPU, since any access to protected memory generates a bus error and a hard fault.

Attack. In a recent paper [79], the Level-2 protection of a STM32 F0 microcontroller has been bypassed by decapsulating the chip and using UV-C light to alter the value of RDP byte stored in flash memory.

We bypass the Level-2 protection by glitching the MCU during the power-up phase, in order to interfere with the RDP security mechanism. The first step of the attack is to identify the correct timing. Since the bootloader is disabled, the glitch trigger cannot be synchronized to a bootloader command as in the case of the STM32F1. The reference manual [113] suggests that the RDP loading takes place at the beginning of the boot process thus, ideally, the glitch should be triggered right after the start of the boot process. In fact, the MCU can be successfully downgraded to Level-1 by injecting a glitch at just 11 µs after the boot starts. The attack is repeatable and makes both the bootloader and the JTAG / SWD accessible. Notice that detecting the start of the boot process in not immediate: the presence of a *Power-On Reset* circuit[4] makes it necessary to observe the reset pin in order to recognize when the CPU effectively starts booting.

Security implications. The attack can effectively downgrade the RDP from Level-2 to Level-1 but not to Level-0. This can be explained by observing the RDP values for the various levels, reported in Table 4.1. Since Level-1 is enabled by any value different from `0xCC33` (Level-2) and `0xAA55` (Level-0), it is enough to corrupt a single bit to switch to Level-1 from the other levels. Instead, downgrading to Level-0 would require to precisely alter the value to `0xAA55` which might not be feasible through voltage glitching. At Level-1, the flash memory is not accessible when in debugging mode. However, the debugger is still allowed to read any RAM address or register value. This feature enables an attacker to dump sensitive data (*e.g.*, encryption keys and passwords) by attaching the debugger when a particular firmware routine is being executed. Additionally, an automatic checksum verification of the firmware is

[4]This circuit holds the microcontroller in reset state for 1.5 ms to 4.5 ms after power on, allowing the power supply to stabilize.

often used by vendors to ensure flash data integrity: for instance, the ARM application note 277 [67] suggests to perform a CRC-based ROM self-test as part of the boot process. In such a scenario an attacker could extract the firmware by iteratively attaching the debugger and dumping RAM and registers content while the checksum code is being executed. In [79] the authors have successfully mounted this attack against a microcontroller of the STM32 F0 series.

4.3.2 Case Study 2: Texas Instruments

The MSP430 line from Texas Instruments (TI) integrates a 16-bit CPU and it is optimized for low power applications. These microcontrollers can be found in a large number of consumer and industrial devices [116], ranging from utility meters and burglar alarms, to safety-critical applications such as fire detectors, medical equipments and physical access control systems. Similarly to the STM MCUs (see Section 4.3.1), the MSP430 integrates a software bootloader (BSL) that allows the user to program and verify the firmware.

MSP430 F5xx ultra-low power

The first device under test is the MSP430F5172.

Security mechanisms. The main security mechanism is a user-defined password that guards every data access command. The BSL can also be set to automatically erase the flash memory whenever an incorrect password is provided. The microcontroller has a Supply Voltage Supervisor (SVS) and a Brownout Reset (BOR) circuit that reset the device in the case of low voltage.

Attack. We have found that the user is asked to authenticate only when the first read command is issued. Every subsequent command is executed without asking for the password again. We suppose that an authentication flag is stored in RAM and checked before the execution of every read operation; for this reason, we target the flag check routine of the *TX Data Block* command, which can read up to 250 bytes. However, the attack allows us to only dump a single byte, and a subsequent analysis of the BSL code has confirmed that the authentication flag is checked for *every* byte read.

The attack iterates over the following steps, for all addresses *addr* that need to be dumped: (*i*) request a single byte at address *addr*; (*ii*) the BSL responds with ACK (byte `0x00`) indicating that the command is well formatted; (*iii*) apply a delay T_{trig} starting from the ACK reception, to align with the instruction that checks the authentication flag; (*iv*) inject a voltage glitch in the power supply line; (*v*) if the BSL responds positively, it also returns the value of requested address from flash memory.

MSP430 FRxx FRAM nonvolatile memory

We consider a second target manufactured by Texas Instruments, the MSP430FR5725.

	Extraction time	Total glitches	Successes	Parameter search	Repeatability
STM32F103	1 m (128 kB)	9 k	5 %	20 m	High
STM32F373*	N/A	∼25	∼4 %	2 h	Moderate
MSP430F5172	16 m (32 kB)	34 k	98 %	1 h	High
MSP430FR5725	50 m (8 kB)	100 k	8 %	3 h	Moderate

*STM32F373: results for one Level-2 to Level-1 downgrade; complete firmware dump not feasible using fault injection only.

TABLE 4.2: Results of the attacks on STMicroelectronics and Texax Instruments microcontrollers described in Section 4.3.1 and Section 4.3.2.

Security mechanisms. This microcontroller adopts a Ferromagnetic RAM (FRAM) non-volatile memory technology, instead of the regular flash memory. In particular, the presence of an integrated FRAM error correction coding (ECC) circuit makes this MCU family an interesting case study to assess the effectiveness of our voltage glitching technique. Similarly to MSP430F5172 (*cf.* Section 4.3.2), the BSL of this microcontroller is password-protected.

Attack. We successfully applied the same attack logic used for the MSP430F5172, described in Section 4.3.2.

4.3.3 Experimental Results and Considerations

The results for the attack performance of the two case studies are highlighted in Table 4.2. In the table we indicate, for each microcontroller model: the extraction time, which is the total time required to dump the firmware of the target MCU (the flash memory size is reported in parenthesis); the total number of injected glitches during the attack; the percentage of successful faults over the total injected glitches; the time required for the genetic algorithm to search for optimal parameters (including the glitch waveform) used during the attack (see Section 4.2.1); the repeatability,[5] *i.e.*, the effort for reproducing the attack against a different microcontroller of the same model, loosely indicated as High or Moderate. Higher repeatability scores indicate, in particular, that switching MCU do not require a full parameter search and optimization, since the attack parameters can be largely reused for attacking the new target.

Since the STM32F103 is quite sensitive to voltage glitches, and the maximum length for a read operation is 256 bytes, we managed to dump a 128 kB firmware in under 60 seconds. On average, the attack requires a total of just 9000 glitches, which corresponds to a success ratio of about 5 %. We did not manage to perform a flash dump of the STM32F373 using fault injection only, thus the result represents a single triggering of the Level-2 to Level-1 downgrade vulnerability. As an example,

[5]We point out that this term is also used in the literature (*cf.* [107]) as metric for the ability to inject a specific fault and obtain the same result.

in the case of the CRC32 attack (*cf.* Section 4.3.1) the downgrade must be successfully performed once for every extracted byte.

The performance of the attack against the MSP430F5172 microcontroller is excellent and we managed to dump over 2 kB per minute. Since the ratio of successful glitches over the total is above 98 %, the attack speed is limited only by the low data rate (9600 bps) of the BSL serial interface. On the contrary, the MSP430FR5725 attack success rate is noticeably lower than the previous target, despite the prolonged parameter search phase. As a result, 1 kB of FRAM memory is dumped every 6 minutes, thus one order of magnitude slower.

The MSP430 microcontrollers have a Supply Voltage Supervisor (SVS) and a Brownout Reset (BOR) circuit that resets the device in the case of low voltage. Interestingly, the genetic algorithm (*cf.* Section 4.2.1) managed to identify the correct set of parameters that are sufficient to induce a fault in the computation, without triggering any of the two monitoring circuits. Note that the resulting waveform (see Figure 4.3b), does not resemble the typical squared glitch shape and cannot be generated using the MOS-FET V-FI setup described in Section 4.1. The power supply voltage identified by the algorithm is close to the minimum working value, which makes this MCU exceedingly sensitive to minimal power disturbances. As a result, the waveform voltage range is extremely compressed and, moreover, the smooth transitions from the power supply voltage (1400 mV) to the lower glitch voltage (880 mV) allow for injecting the glitch undetected.

4.4 Complex Attacks

The result of the attacks discussed in Section 4.3 indicates that arbitrary glitch waveforms can help to automatically bypass on-chip protection mechanisms such as brownout detectors or voltage supervisors. In this section we investigate the usage of voltage glitching against particularly challenging attacks, where the total time required is in the range of several days and the number of *successful glitches* is measured in the order of 100 k to over 1 M.

In Section 4.4.1 we present a third case study that we conducted on two MCU families manufactured by Renesas Electronics. Similarly to those presented in Section 4.3, these attacks are novel and unpublished and enable for extracting the firmware from the read-protected internal flash memory. We point out that the firmware can only be dumped if multiple vulnerabilities of the on-chip serial bootloader are combined and exploited; the data leaked by each vulnerability alone is indeed insufficient. The attacker model is similar to the one described in Section 4.3, although with an interesting distinction: the attack is conducted following a full *black-box* approach, *i.e.*, with no information about the running firmware or the flash memory content and, in particular, without reverse engineering the bootloader code.[6]

[6]During regular operation the bootloader code is not memory mapped and thus cannot be dumped.

In Section 4.4.2 we show the attack results and discuss the issues and the challenges that have emerged and that are specifically related to this class of complex attacks.

4.4.1 Case Study 3: Renesas Electronics

We tested two microcontrollers from the 78K family manufactured by Renesas Electronics, specifically series 78K0/Kx2 (8-bit core) and 78K0R/Kx3-L (16-bit core). The manufacturer suggests that these MCUs are suitable for a wide range of applications, from home appliances to more critical ones such as healthcare and automotive. A 2016 document from Renesas [94] reports that 920 millions MCUs / SoCs have been sold in 2015 and, on average, every new vehicle contains 11 Renesas MCUs installed in the onboard Electronic Control Unit.

The attacks described in this section target the 78K Flash Memory Programming Interface (FMPI) [92, 93], *i.e.*, the bootloader used to load a firmware into the internal flash memory. The microcontroller can be set to boot from the FMPI, exposing the common programming functionalities, *e.g.*, write, erase, verify, to the user.

Security mechanisms. As opposed to what found in Section 4.3.1 and Section 4.3.2, this interface does not provide a command to directly read a memory address. All the commands that could potentially leak the flash memory content (*e.g.*, checksum, verify) are enforced to operate on 256 bytes aligned memory segments. This constraint disallows, for instance, a one byte increment of the checksum segment or to perform an efficient brute-force by verifying a single byte at a time. Additionally, the 78K offers a mechanism to further protect the content of the flash memory in production devices: a security flag field, controlled by the *Security set* command, can be set to disallow *Boot block rewrite*, *Programming*, *Block erase* and *Chip erase* commands. Since the security flag can be reverted only with a full memory erase, disabling the *Chip erase* command is an irreversible operation.

FMPI Vulnerabilities

In this section we provide a description of the vulnerabilities that we found in the FMPI interface. In Section 4.4.1 we combine these vulnerabilities to mount three different attacks for dumping the flash memory content.

FlagBypass Restrictions on *program*, *erase* and *chip erase* commands can be bypassed by injecting a fault while the *Security flag* value is being evaluated. We have found that in order to attack this command two separate glitches are required and thus the rate of success is very low, *i.e.*, about one per minute or lower.

ShortVerify and **ShortChecksum** By glitching the routine that checks *start* and *end* parameters sent to the *verify* and *checksum* commands, we are able to force these commands to operate on 4 bytes rather than the intended 256 bytes.

ChecksumLeak The *checksum* command can be exploited as a side-channel by causing an error during its calculation, which amounts to iteratively subtracting each byte from the starting value `0x10000`. An error introduced by the glitch could cause the checksum routine to miss one byte during its calculation, making it possible to compute the value of this byte through a subtraction from the correct checksum. We point out that this vulnerability can produce false positives (*i.e.*, bytes that are not actually in memory) and that it can be difficult to precisely recover the position of the leaked byte in the flash memory.

Bitflip In flash memories, the write operation changes the state of a bit from 1 to 0 while, on the contrary, the erase operation switches it back to 1 (see [29]). Since the bootloader does not enforce a flash erase before writing, the *program* command can be used to alter existing flash memory content. As a consequence, by using solely the *program* command we are able to turn `101` into, *e.g.*, `100` or `000` but not into `111`.

Mounting the Attacks

By combining the vulnerabilities of Section 4.4.1 we mounted three different attacks for dumping the read-protected internal flash memory of the 78K 8-bit and 16-bit MCUs.

SequentialDump By combining the ShortVerify, ShortChecksum and ChecksumLeak vulnerabilities it is possible to discover four bytes from the flash. The process, depicted in Algorithm 4, works as follows: (*i*) use the ShortChecksum vulnerability (line 2) to obtain the checksum value of the target 4 bytes; (*ii*) use ShortChecksum and ChecksumLeak vulnerabilities to leak 4 different byte values (lines 5 and 6) (*iii*) process and combine (line 8) leaked bytes to obtain a new set of candidates for the 4 bytes that has not been checked already; (*iv*) perform a first check (line 10) for the validity of the candidate by comparing its checksum with the known *bytesChecksum*. This does not require any interaction with the hardware; (*v*) verify the candidate using the ShortVerify vulnerability (line 11).

The attack is feasible thanks to the ShortVerify and ShortChecksum vulnerabilities, that allow to selectively work on 4 bytes. The API would only allow to perform *verify* and *checksum* for blocks of 256 bytes.

Erase & Write We inject a custom software routine in the firmware that directly dumps the firmware through a serial communication channel with a computer. The attack is mounted as follows: (*i*) use Algorithm 5 to dump the first *n* bytes of the flash;[7] (*ii*) use the FlagBypass vulnerability to erase the first *n* bytes; (*iii*) use the FlagBypass vulnerability once more to write the custom routine into the erased memory; (*iv*) set the microcontroller to boot from the custom routine and receive the dump from the serial interface.

This translates into a considerable performance improvement: a full flash dump can be performed in about three to five hours, while the number of required glitches

[7]Since the minimum erase size is 1024 bytes, then $n \geq 1024$.

Algorithm 4 Attack for extracting 4 bytes.

```
function FourBytesDump($addr$)
    $bytesChecksum \leftarrow$ ShortChecksum($addr$)
    $oldCandidates \leftarrow \varnothing$
    $leak \leftarrow \varnothing$
    while $|leak| < 4$ do
        $leak \leftarrow leak \cup$ ChecksumLeak($addr$)
    while True do
        $newCandidates \leftarrow$ Combine($leak$) $\setminus oldCandidates$
        for all $guess \in newCandidates$ do
            if Checksum($guess$) $= bytesChecksum$ then
                if ShortVerify($addr, guess$) then
                    return $guess$
        $oldCandidates \leftarrow oldCandidates \cup newCandidates$
        $leak \leftarrow leak \cup$ ChecksumLeak($addr$)
```

Algorithm 5 Memory dump using the *SequentialDump* attack.

```
function FlashDump($startAddr, endAddr$)
    $data \leftarrow \varnothing$
    while $startAddr \neq endAddr$ do
        $data \leftarrow data \,\|\,$ FourBytesDump($startAddr$)
        $startAddr = startAddr + 4$
    return $data$
```

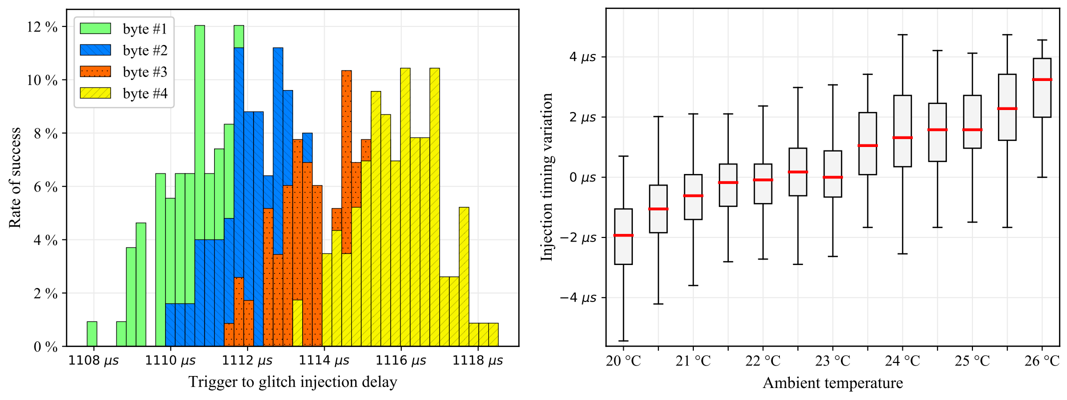
is about one order of magnitude lower with respect to the full attack. This attack was tested on 78K0R only. Attacking the 78K0 series might also be possible, although our preliminary tests have been unsuccessful.

Bitflip & Write We found that the firmware of several commercial devices does not fill the available flash space completely. For instance, the unused blank[8] memory segment could be left for future firmware updates. A checksum or verify command is sufficient to locate any blank segment in the flash memory. We exploit these empty segments to further optimize the firmware extraction strategy.

The attack is mounted as follows: (*i*) use the `Bitflip` vulnerability to store the firmware-dump routine of the *Erase & Write* attack in an unused memory area; (*ii*) use Algorithm 4 to dump the first 4 bytes to identify the location of the boot section;[9] (*iii*) use Algorithm 5 to dump the first 256 bytes of the boot section; (*iv*) analyse the dumped bytes to identify a suitable candidate for bit-flipping: for instance, a `FF FF` sequence is sufficient for encoding a branch instruction; (*v*) use the `Bitflip` vulnerability to replace all the instructions up to the `FF FF` sequence with NOPs (byte `0x00`) followed by a branch instruction to the firmware dump routine; (*iv*) set the microcontroller to boot from the custom routine and receive the dump from the serial interface.

[8] A segment is considered blank if all bytes have value 0xFF, *i.e.*, the block is erased.

[9] In the 78K architecture the first 4 bytes of the flash memory hold the address of the firmware boot section (*i.e.*, the entry point).



(A) (B)

FIGURE 4.4: Frequency distribution of 4 bytes leaked by the Renesas *ChecksumLeak* vulnerability (4.4a): on the x-axis the delay between the trigger and glitch injection, on the y-axis the rate of successful faults. The effect of ambient temperature variations on the optimal injection timing (4.4b). The reference measurement is at 23 °C and 1110 µs.

	Extraction time	Total glitches	Successful glitches	Parameter search	Repeatability
SequentialDump	2 d 12 h	3.3 M	549 k	5 h	Moderate
Erase & Write	∼3 h	513 k	45 k	1 h	High
Bitflip & Write	<1 h	204 k	15 k	30 m	High

TABLE 4.3: Results of the three attacks on Renesas 78K microcontrollers described in Section 4.4.1. The values are obtained by averaging the results of three complete firmware extractions.

4.4.2 Experimental Results and Considerations

In Table 4.3 we summarize the results of the Renesas attacks described in Section 4.4.1. As one would expect, the *SequentialDump* attack is the slowest one as it dumps all the flash memory by using fault injection only, thus requiring a very high glitch count. On the contrary, the software dump routine loaded using the *Erase & Write* and *Bitflip & Write* attacks leads to a major improvement in firmware extraction time. To this end, since these attacks extract few bytes using fault injection, a trade-off can be achieved between the bare extraction speed (*i.e.*, the glitch success ratio) and the time required for the parameter optimization phase. However, we point out that to trigger the FlagBypass vulnerability, required by the *Erase & Write* and *Bitflip & Write* attacks, two repeated glitches are necessary, resulting in an extremely low success ratio: we managed to achieve one success in about 15 to 30 minutes.

The *SequentialDump* attack can run fully automated and unsupervised with a reasonable degree of repeatability. Indeed, we performed several full firmware dumps from different 78K0 and 78K0R microcontrollers. Typically, switching MCU requires a re-optimization phase of the attack parameters, including the glitch waveform, in order to achieve good glitch success ratio. Interestingly, we experienced that each exploited vulnerability best performs with a specific glitch waveform. Although a single glitch waveform can be sufficient to trigger multiple vulnerabilities, the success ratio of such a waveform is low.

Our tests revealed also that, even with the correct parameters, both the attack performance and the repeatability of long-running attacks can be influenced by timing errors and external variables such as the ambient temperature.

Injection Timing

Fault injection aims at causing an error during the computation of a specific task, so the timing is a critical parameter. We refer to *injection timing* as the delay that we introduce between an external trigger event and the injection of the glitch. During our experiments we experienced glitch timing inaccuracy that affected the attack performance. In general, this could be caused by external and physical variables such as temperature, clock stability, trigger precision or interferences. As an example, Figure 4.4a depicts the effect of timing variations on the output of the Renesas *ChecksumLeak* vulnerability presented in Section 4.4.1: it appears evident the correlation between injection delay and the probability of leaking one of the four bytes. The $\pm 2\,\mu s$ error is introduced by the inaccurate trigger event (*i.e.*, the command transmission to the bootloader) combined with the fluctuation in the checksum computation time caused by small variations in the internal oscillator frequency. As a result, this timing error makes it difficult to recover the exact position of the leaked byte because of the overlapping probability distributions. We point out that timing errors could be minimized with the use of synchronization techniques such as frequency locking [109, 77] or side-channel power analysis [78].

Ambient Temperature

Extreme temperatures are known to facilitate fault injection and side channel attacks on several targets [47, 48, 91]. In fact, we have found that also small variations in the ambient temperature can affect the attack performance, requiring multiple adjustments of the injection parameters and thus affecting the attack repeatability. When targeting microcontrollers that are running on the integrated oscillator, attacks using long injection delays ($\geq 100\,\mu s$) can be particularly sensitive to temperature changes. This is particularly interesting when performing attacks that span over more than one day: for instance, heating or cooling systems can be turned off during the night causing a noticeable temperature variation. To verify our observations we measured the impact of a $\pm 3\,°C$ ambient temperature excursion on the Renesas *ChecksumLeak* vulnerability. We repeated the attack for one hour by using, at each iteration, a different injection timing that is randomly sampled in the range $1100 \pm 5\mu s$. The box plot in Figure 4.4b collects all the glitch timings that lead to a successful attack: the plot suggests that the value of the glitch delay is proportional to the ambient temperature.

This behaviour is caused by slight variations in the frequency of the internal oscillator (*cf.* [96, 123]). In particular, an increase in ambient temperature causes a decrease of execution speed in the Renesas 78K microcontroller, which misaligns the target instruction with respect to the injected glitch. We managed to reduce the error caused by temperature variations by applying a $\sim 0.1\,\%/^{\circ}\mathrm{C}$ compensation factor to the injection timing. This factor can be easily calculated from the results of the above test.

4.5 Evaluation

In order to evaluate the attack performance of our approach, referred to as `AGW` in this section, we conducted a series of tests against the two other main voltage glitching techniques: an ubiquitous transistor-based setup, namely `Mosfet`, and its generalization using a DAC-generated pulse that we will refer to as `Pulse`.

4.5.1 Performance Analysis and Comparison

In the following we describe the three setups that we used during the tests:

`Mosfet` This is the classic configuration often adopted in the literature [78, 76], similar to the one described in Section 1.2 and depicted in Figure 4.2a. Specifically, we use a VN2222 N-channel MOS-FET paired with an ADP3623 driver to ensure fast and sharp switching times. This setup allows for configuring glitch duration and timing. The MCU power supply voltage can be varied manually and the glitch voltage (*i.e.*, the peak low voltage of the glitch waveform) is fixed at 0 V since the source pin of the MOS-FET is tied to ground.

`Pulse` With respect to the previous setup, this allows for improved configurability and control over the generated glitch. The glitch is, in fact, more predictable and it is not influenced by the characteristics of the specific MOS-FET in use. We implemented this setup using our arbitrary function generator (*cf.* Section 4.1), enabling us additional control over the power supply voltage and the glitch voltage, duration, timing and, in particular, the rise and fall times of the glitch edges. To the best of our knowledge, this setup resembles the behaviour of the industry standard Riscure VC Glitcher [95] which is, however, missing the ability to alter the rise and fall times of the generated glitch.

`AGW` Our proposed setup (*cf.* Section 4.1) enhances the output capabilities of the `Pulse` method, allowing the attacker to produce fully arbitrary glitch waveforms. Moreover, this setup is capable of producing voltage glitches with 20 V peak-to-peak amplitude and ± 10 V output range, for supporting a broad variety of targets.

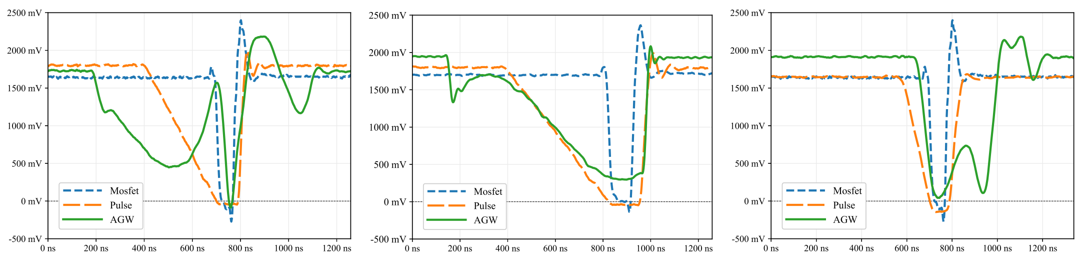
Vulnerability	Technique	Success	False Positive	Reset	Reset/Success	Glitch Count
ShortVerify	`Mosfet`	668 (2.6 %)	1	1780 (6.9 %)	2.66	25701
	`Pulse`	969 (3.7 %)	0	1685 (6.4 %)	1.74	26180
	`AGW`	1291 (6.8 %)	1	2786 (14.6 %)	2.16	19044
ShortChecksum	`Mosfet`	474 (2.1 %)	1	1862 (8.3 %)	3.93	22322
	`Pulse`	689 (2.8 %)	1	1632 (6.6 %)	2.37	24931
	`AGW`	728 (4.4 %)	2	2912 (17.7 %)	4.01	16475
ChecksumLeak	`Mosfet`	412 (4.9 %)	254 (3.0 %)	2481 (29.8 %)	6.02	8329
	`Pulse`	455 (5.6 %)	158 (1.9 %)	2510 (30.9 %)	5.52	8136
	`AGW`	687 (8.6 %)	42 (0.5 %)	2515 (31.5 %)	3.66	7977

TABLE 4.4: Performance comparison of three vulnerabilities described in Section 4.4.1 using different voltage glitching techniques. Results are obtained by averaging 4 independent runs of 10 minutes each. Values inside the parenthesis indicate the percentage relative to the total glitch count.

The high runtime complexity of the Renesas attacks (*cf.* Section 4.4.1) makes the *ShortVerify*, *ShortChecksum* and *ChecksumLeak* vulnerabilities an interesting benchmark for evaluating the performance of these three V-FI techniques. The experiments were conducted by attacking a microcontroller of the 78K0/Kx2 family, preprogrammed with known memory content so to verify the correctness of the extracted data. All the attack parameters were computed by running the algorithm described in Section 4.2.1 for 8 hours.[10] In Table 4.4 we present the performance results obtained by averaging 4 independent runs of 10 minutes each that we conducted for every combination of vulnerability and technique. Between each run, the glitch timing was adjusted to compensate for temperature variations (*cf.* Section 4.4.2).

In the table we indicate: *(i)* the number of successes, *e.g.*, the amount of glitches that lead to a successful verify operation or to extract the correct byte; *(ii)* the number of false positives, such as an incorrect byte extracted or a bad short-checksum; *(iii)* how often we reset the microcontroller for becoming unresponsive after a glitch; *(iv)* the total glitch count, injected during the 10 minutes run. The results show that `AGW` outperforms the other techniques. In particular we observe that the absolute number of successful glitches is noticeably higher. Similarly, `AGW` presents a higher ratio of success over the total injected glitches, thus our technique is both faster and more efficient. The false positive count for the *ChecksumLeak* vulnerability is 6 times and almost 4 times lower with respect to `Mosfet` and `Pulse` techniques; interestingly, in Section 4.5.1 we show how this enables a major reduction in the firmware extraction time. When a glitch makes the microcontroller non-responsive, a reset operation is performed at the cost of additional overhead, due to the bootloader re-initialization. The results show that, in general, our technique induces more resets in the target,

[10]Although in some cases the optimal set of parameters could have been found in a shorter period, we forced the algorithm to run for 8 hours in order to guarantee fairness and comparability of the results.



(A) ShortVerify (B) ShortChecksum (C) ChecksumLeak

FIGURE 4.5: Comparison of the voltage glitch waveforms for the three V-FI techniques and vulnerabilities evaluated in Table 4.4.

Technique	Tested combinations	# ShortVerify	# ChecksumLeak	# ShortChecksum	Total glitch count	Total dump time
`Mosfet`	351 k	13.9 M	3.1 M	699 k	18.1 M	6 d 19 h
`Pulse`	142 k	3.8 M	2.6 M	582 k	7.1 M	3 d 16 h
`AGW`	105 k	1.5 M	1.5 M	351 k	3.3 M	2 d 12 h

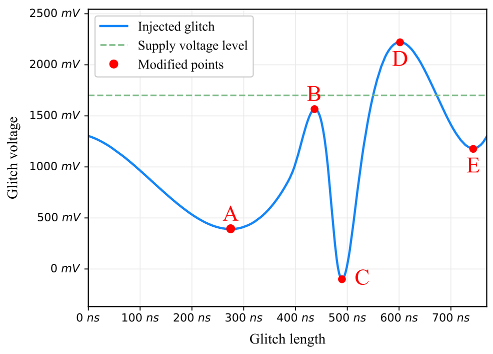
TABLE 4.5: Full firmware extraction from a 60 kB flash memory using the Renesas *SequentialDump* attack. Overview of the performance variation among different voltage glitching technologies.

thus limiting the number of glitches injected in the 10 minutes run. This limitation is, however, compensated by a higher fault efficiency, which contributes in rising the overall attack performance.

Finally, in Figure 4.5, for each vulnerability we plot the glitch waveforms of the three V-FI techniques. The `Mosfet` setup shows the shorter glitch duration and the maximum voltage amplitude as a consequence of both undershooting below 0 V and overshooting above `VCC`. While the edge rise time is sharp for all the three vulnerabilities, the fall time in the `Pulse` setup appears to be much longer, about 400 ns in the case of Figure 4.5b. Interestingly, the duration of the low-end (close to 0 V) part of the `AGW` glitch is similar to those of the other two techniques.

Firmware Extraction Time

After evaluating the performance of the single vulnerabilities, we tested the *SequentialDump* attack presented in Section 4.4.1. The results, depicted in Table 4.5, represent a full 60 kB flash memory dump. Notice that, for the sake of brevity, each technique was tested on 5 consecutive 256-bytes memory blocks only; the 60 kB result was obtained after calculations. Our technique managed to dump the firmware 32 % and 63 % faster than `Pulse` and `Mosfet`, respectively. Interestingly, our approach is also very efficient, reducing the total number of glitches required to complete the attack: the `Mosfet` produced about five time the number of glitches, followed by the `Pulse` method which doubled the value of `AGW`. In fact, since these techniques show a higher false positives ratio in the *ChecksumLeak* vulnerability, the number of extracted combinations that require to be verified (*cf.* Algorithm 4) is also higher.



(A) Points of perturbation

Point	Variation	Success / minute
A	+200 mV	−22 %
A	+550 mV	−58 %
B	−260 mV	−16 %
C	−170 mV	Not working
C	+140 mV	+7 %
D	−510 mV	−33 %
E	+480 mV	−1 %

(B) Performance variation

FIGURE 4.6: Voltage glitch waveform for the Renesas *ShortVerify* vulnerability (4.6a). Red points are shifted in the y-axis (voltage), reporting the performance effect of each variation in the table (4.6b).

Glitch Waveform Characterization

We conducted a final experiment to characterize how minor perturbations in the *ShortVerify* waveform would impact the success rate. This waveform, used for the `AGW` tests and depicted in Figure 4.6a, is capable of producing about 130 successful verify operations per minute. The test is performed as follows: a point of local minimum or maximum, labelled with letters from A to E, is selected and moved along the y-axis so to lower or raise its voltage; after interpolating the 5 points, the resulting waveform is tested for 10 minutes and the number of successful verify operations is collected. The results of Figure 4.6b highlight a correlation between a specific perturbation and the attack success rate. As an example, if point A is raised by 200 mV, then the performance decreases by about 22 %. Interestingly, lowering point C by 170 mV does not produce any success at all, while raising this point by 140 mV increases the number of successful *verify* operations by 7 %. This particular result indicates that refinements in the parameter optimization algorithm (see Section 4.2.1) could leave room for further performance improvements of the `AGW` technique.

4.5.2 Limitations and Further Improvements

The experimental campaign conducted proved that the success rate of an attack can be improved by selecting specific glitch waveforms and, as described in Section 4.3.3, some countermeasures such as integrated voltage supervisors can be automatically bypassed. However, this improvement comes at the cost of increased complexity in the glitch parameter search (*cf.* Section 4.2.1). In particular, searching and optimizing the glitch waveform might be time consuming, possibly requiring numerous glitches. The choice of the voltage fault injection technique should, thus, account for both the security mechanisms employed by the target (if any) and the overall attack complexity. As a compromise between attack performance and parameter search duration, it might in fact be advantageous to reduce the degrees of freedom of the waveform generation. For instance, starting the glitch waveform optimization from a small, predefined set of shapes, could substantially ease the task of the optimization algorithm.

We plan to conduct more experiments on new microcontrollers and other classes of fault attacks, *e.g.*, against cryptographic implementations, and to target secure microcontrollers or hardware (*e.g.*, smartcards, USB tokens) and high speed Systems-on-a-Chip (SoCs). To this end, our low cost generator (see Section 4.1), which is limited to produce arbitrary waveforms with a maximum frequency of about 6 MHz, will be upgraded to increase both bandwidth and output sampling rate. Finally, we also leave as a future work the study of an improved version of the genetic algorithm presented in Section 4.2.1, and the investigation of other classes of optimization algorithms.

4.6 Discussion

In this chapter we have studied, for the first time, how voltage glitches with arbitrary waveforms affect the success and efficiency of an attack. We have also investigated the feasibility of identifying a valid set of attack parameters, including the glitch waveform, in an automated and unsupervised way, and showed the feasibility of generating these type of glitches using low cost equipment. Furthermore, we have presented novel attacks on six widely used microcontrollers from three manufacturers. These attacks target the bootloader interface and allow for extracting the firmware from the internal protected flash memory. Following a responsible disclosure policy, we have timely reported the security flaws to STMicroelectronics, Texas Instruments and Renesas Electronics. Finally, we have evaluated the performance improvement provided by the arbitrary glitch waveforms against two other major voltage glitching techniques. The results showed an increment in the firmware extraction speed and, in particular, a significantly lower amount of injected glitches required to complete the attack.

Regardless of the arbitrary glitch waveform technique, we believe that the presented attacks are valuable. They provide evidence that an attacker, even with limited resources, can use voltage fault injection to bypass the protection mechanisms offered by the microcontrollers under test. Thus, we certainly discourage the adoption of this kind of microcontrollers for security or safety-involved systems. Even when microcontrollers come with some basic protection mechanisms and voltage supervisors, we have shown how these can be easily and systematically bypassed, allowing for efficient firmware extraction. Moreover, firmware extraction is problematic by itself for intellectual property. So, independently of the criticality of the application, companies should be aware that the protections implemented in budget microcontrollers are insufficient to protect the know-how in the firmware and, consequently, devices could be cloned or tampered by criminals with a low effort and investment.

Related Work

One of the first attacks exploiting the idea of hardware faults was described (but not tested) by Boneh, DeMillo and Lipton in [25]: it recovered the secret factors p and q of an RSA modulus from a correct signature and an hypothetical faulty one. In the same year, Anderson and Kuhn investigated low cost attacks to tamper resistant devices, focussing on a fault injection attack on pay-TV smartcards [2]. Then, in [60] Kömmerling and Kuhn described an extensive range of invasive and non-invasive tampering techniques and mitigations. As such, the paper is considered a milestone in the setting of hardware fault attacks, highlighting power supply glitching attacks as the most practical ones. In [7], authors combined voltage fault injection and power analysis to compromise the confidentiality of cryptographic computations and suggested possible countermeasures. In the following decade, numerous articles (*e.g.,* [9, 49, 44, 57, 62]) have further investigated the feasibility of applying voltage glitching to attack both microcontrollers and secure hardware, such as smartcards. In particular, an extensive survey of the current state-of-the-art is provided in [55] and [12]. The power supply fault injection mechanism has been extensively studied and explained as the result of setup time violations in the combinatorial logic [104, 105, 130, 131].

Fault attacks against cryptographic implementations is also a very active research topic and, in particular, several papers [104, 13, 14, 15] studied the effect of constantly underfeeding a circuit to cause faults. In recent years, fault injection has also been proven effective for achieving privilege escalation on a locked-down ARM processor that was running a Linux-based OS [117] and, in the same year, a paper [33] by Cojocar *et al.* proved that two widely used software countermeasures to fault attacks do not provide strong protection in practice.

Firmware extraction from read-protected microcontrollers is a relatively less explored field: in [44] Goodspeed defeated the password protection mechanism found in older Texas Instruments MSP430 microcontrollers. The author used a timing-based side-channel attack to exploit an unbalanced code in the password check routine and, by using voltage glitching, he bypassed the security feature that allows for disabling the serial bootloader (BSL) completely. In [79] authors showed that it is possible to downgrade hardware-enforced security restrictions and dump the internal firmware of an STM32F0 MCU, but the attack is invasive as it requires to decapsulate the chip and expose the silicon to UV-C light. For what concerns the characterization of V-FI attack parameters, in [30] and [88] authors have successfully applied genetic algorithms and other optimization techniques. Finally, the glitch generation using FPGA combined with Digital-to-Analog converters has been studied in the literature (*cf.* [117, 57]) and adopted by V-FI commercial tools, such as the well-known Riscure VC Glitcher [95].

Our work continues this line of research by focusing on the voltage glitch generation step, in order to optimize the glitching effects that can be exploited by the adversary. In particular we investigate, for the first time, the impact of arbitrary glitch waveforms on the success, performance and repeatability of V-FI attacks. We show that our approach improves on the state of the art by evaluating the attack performance with specific glitch waveforms against the two most popular V-FI generation techniques [49, 57].

Conclusion

In this thesis we have studied existing cryptographic designs and implementations from the software, hardware and physical perspectives. We have assessed the security provided by Java keystores, focusing on seven of these cryptographic storage facilities found in the Oracle JDK and the Bouncy Castle library. We have analysed all the cryptographic mechanisms, including undocumented ones, adopted by keystores to enforce standard security properties. We have also disclosed new attacks that range from breaking the confidentiality of stored keys, to denial of service of the target system and arbitrary code execution. Moreover, we have shown how a keystore can be potentially weaponized by an attacker to spread malware. Finally, we have discussed the advancements on the security of Oracle and Bouncy Castle keystore implementations after our findings.

Then, we have thoroughly analysed tamper-resistant cryptographic hardware, accounting for the low-level APDU communication protocol as a new attack surface. We have presented a new threat model for the PKCS#11 middleware and we have analysed the APDU-level implementation of the PKCS#11 API for five commercially available smartcards and USB tokens. In particular, we have shown that these devices present APDU-level attacks that allow for bypassing the authentication and, for four of them, make it possible to leak sensitive keys in the clear. We have also discussed a series of fixes and mitigations to prevent all the attacks we have presented.

Finally, we have investigated on voltage fault injection, a physical class of attacks that targets cryptographic hardware and embedded systems. We have studied, for the first time, the effect of voltage glitches with arbitrary waveforms on the attack success and efficiency. We have also investigated on the feasibility of identifying a valid set of attack parameters, including the glitch waveform, in an automated and unsupervised way using genetic algorithms, and showed the possibility of generating these type of glitches using low cost equipment. We have presented novel attacks on six widely used microcontrollers from three manufacturers, allowing the attacker to extract the firmware from the internal protected flash memory. To evaluate the performance improvement provided by our approach, we have conducted an experimental campaign showing a measurable increment in the firmware extraction speed and a substantial difference in the total number of injected glitches required to complete the attack.

During the development of this work, we realized that there is a general lack of attention to security in many commercial products. Although in recent years large companies have raised the investments for, *e.g.*, bug bounties, vulnerability assessments and security analyses, software is still often developed by engineers with little to no background in security and, in particular, in cryptography. Interestingly, some of the vulnerabilities related to poor design practices that we found in cryptographic hardware, suggest that this practice might hold for security oriented products too. Computer security is frequently perceived as an additional cost rather than a strategic asset. This policy pose a threat not only to the end user, but also to the company itself, *e.g.*, by damaging the reputation as the result of data breaches or by the loss of intellectual property. Furthermore, securing existing systems that were developed without sound design principles can require extensive re-engineering: legacy libraries, standards and formats, such as the JKS Keystore, are still widespread and difficult to patch while maintaining backward compatibility. In conclusion, although we acknowledge that advancing the state-of-the-art is the primary objective of academic research, we believe that our mission is also to raise the awareness of both the public and the industry with regards to the major role of IT security.

Bibliography

[1] R. Anderson. "The Correctness of Crypto Transaction Sets (Discussion)". In: *Revised Papers from the 8th International Workshop on Security Protocols*. London, UK: Springer-Verlag, 2001, pp. 128–141.

[2] R. J. Anderson and M. G. Kuhn. "Low Cost Attacks on Tamper Resistant Devices". In: *Security Protocols, 5th International Workshop, Paris, France, April 7-9, 1997, Proceedings*. 1997, pp. 125–136.

[3] *Android Studio User Guide: Sign Your App*. URL: `https://developer.android.com/studio/publish/app-signing.html`.

[4] *Apache Tomcat 7 Documentation: SSL/TLS Configuration*. 2017. URL: `https://tomcat.apache.org/tomcat-7.0-doc/ssl-howto.html`.

[5] Apple inc. *iOS Security Guide*. Tech. rep. Mar. 2017. URL: `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`.

[6] M. Armand and J.-M. Tarascon. "Building better batteries". In: *Nature* 451.7179 (2008), p. 652.

[7] C. Aumüller et al. "Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures". In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. 2002, pp. 260–275.

[8] M. Backes et al. "Acoustic Side-Channel Attacks on Printers". In: *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. 2010, pp. 307–322.

[9] H. Bar-El et al. "The Sorcerer's Apprentice Guide to Fault Attacks". In: *IACR Cryptology ePrint Archive* 2004 (2004), p. 100.

[10] G. Barbu, C. Giraud, and V. Guerin. "Embedded Eavesdropping on Java Card". In: *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*. 2012, pp. 37–48.

[11] R. Bardou et al. "Efficient Padding Oracle Attacks on Cryptographic Hardware". In: *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology, CRYPTO 2012*. 2012, pp. 608–625.

[12] A. Barenghi et al. "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures". In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076.

[13] A. Barenghi et al. "Low Voltage Fault Attacks on the RSA Cryptosystem". In: *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, 6 September 2009*. 2009, pp. 23–31.

[14] A. Barenghi et al. "Low Voltage Fault Attacks to AES". In: *HOST 2010, Proceedings of the 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 13-14 June 2010, Anaheim Convention Center, California, USA*. 2010, pp. 7–12.

[15] A. Barenghi et al. "Low Voltage Fault Attacks to AES and RSA on General Purpose Processors". In: *IACR Cryptology ePrint Archive* 2010 (2010), p. 130.

[16] E. Barker. *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175B.pdf`. Aug. 2016.

[17] E. Barker and A. Roginsky. *Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths (Rev. 1)*. `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf`. Nov. 2015.

[18] B. Beurdouche et al. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *Proceedings of the 36th IEEE Symposium on Security and Privacy, S&P 2015*. 2015, pp. 535–552.

[19] K. Bhargavan and G. Leurent. "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016*. 2016, pp. 456–467. URL: `http://doi.acm.org/10.1145/2976749.2978423`.

[20] A. Biryukov, D. Dinu, and D. Khovratovich. "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications". In: *Proceedings of the 1st IEEE European Symposium on Security and Privacy, EuroS&P 2016*. 2016.

[21] D. Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". In: *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO '98*. 1998, pp. 1–12.

[22] M. Bond. "Attacks on Cryptoprocessor Transaction Sets." In: *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*. Vol. 2162. LNCS. Paris, France: Springer, 2001, pp. 220–234.

[23] M. Bond and R. Anderson. "API Level Attacks on Embedded Systems". In: *IEEE Computer Magazine* 34.10 (Oct. 2001), pp. 67–75.

[24] D. Boneh, H. Corrigan-Gibbs, and S. Schechter. "Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks". In: *Proceedings of the 22nd Annual International Conference on the Theory and Applications of Cryptology and Information Security, ASIACRYPT 2016*. 2016.

[25] D. Boneh, R. A. DeMillo, and R. J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)". In: *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*. 1997, pp. 37–51.

[26] M. Bortolozzo et al. "Attacking and fixing PKCS#11 security tokens". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, 2010, pp. 260–269.

[27] F. Brasser et al. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017.* 2017.

[28] E. Brier, C. Clavier, and F. Olivier. "Correlation Power Analysis with a Leakage Model". In: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. 2004, pp. 16–29.

[29] Y. Cai et al. "Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques". In: *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. 2017, pp. 49–60.

[30] R. B. Carpi et al. "Glitch It If You Can: Parameter Search Strategies for Successful Fault Injection". In: *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*. 2013, pp. 236–252.

[31] R. Clayton and M. Bond. "Experience Using a Low-Cost FPGA Design to Crack DES Keys". In: *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2002*. 2002, pp. 579–592.

[32] J. Clulow. "On the Security of PKCS#11". In: *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2003*. 2003, pp. 411–425.

[33] L. Cojocar, K. Papagiannopoulos, and N. Timmers. "Instruction Duplication: Leaky and Not Too Fault-Tolerant!" In: *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*. 2017, pp. 160–179.

[34] Common Criteria Working Group. *CC v3.1. Release 5 — Common Methodology for Information Technology Security Evaluation*. `https://www.commoncriteriaportal.org/files/ccfiles/CEMV3.1R5.pdf`. Apr. 2017.

[35] T. Cooijmans, J. de Ruiter, and E. Poll. "Analysis of Secure Key Storage Solutions on Android". In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM 2014*. 2014, pp. 11–20.

[36] L. Cordesses. "Direct digital synthesis: A tool for periodic wave generation (part 1)". In: *IEEE Signal processing magazine* 21.4 (2004), pp. 50–54.

[37] Cryptosense S.A. *Mighty Aphrodite – Dark Secrets of the Java Keystore*. 2016. URL: `https://cryptosense.com/mighty-aphrodite-dark-secrets-of-the-java-keystore/`.

[38] D. De Cock et al. "Threat modelling for security tokens in web applications". In: *Communications and Multimedia Security*. Springer. 2005, pp. 183–193.

[39] S. Delaune, S. Kremer, and G. Steel. "Formal Analysis of PKCS#11 and Proprietary Extensions". In: *Journal of Computer Security* 18.6 (Nov. 2010), pp. 1211–1245. DOI: `10.3233/JCS-2009-0394`. URL: `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/DKS-jcs09.pdf`.

[40] Y. Dodis, T. Ristenpart, and T. Shrimpton. "Salvaging Merkle-Damgård for Practical Applications". In: *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2009*. 2009, pp. 371–388.

[41] K. Gandolfi, C. Mourtel, and F. Olivier. "Electromagnetic Analysis: Concrete Results". In: *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Generators. 2001, pp. 251–261.

[42] P. Gasti and K. B. Rasmussen. "On the Security of Password Manager Database Formats". In: *Proceedings of the 17th European Symposium on Research in Computer Security, ESORICS 2012*. 2012, pp. 770–787.

[43] A. Gkaniatsou et al. "Getting to know your Card: Reverse-Engineering the Smart-Card Application Protocol Data Unit". In: *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*. 2015, pp. 441–450.

[44] T. Goodspeed. "A side-channel timing attack of the MSP430 BSL". In: *Black Hat USA* (2008).

[45] P. A. Grassi et al. *Digital Identity Guidelines: Authentication and Lifecycle Management*. `https://pages.nist.gov/800-63-3/sp800-63b.html#sec5`. 2017.

[46] P. Gutmann. "Lessons Learned in Implementing and Deploying Crypto Software". In: *Proceedings of the 11th USENIX Security Symposium*. 2002, pp. 315–325. ISBN: 1-931971-00-5. URL: `http://dl.acm.org/citation.cfm?id=647253.720291`.

[47] J. A. Halderman et al. "Lest we remember: cold-boot attacks on encryption keys". In: *Commun. ACM* 52.5 (2009), pp. 91–98.

[48] M. Hutter and J. Schmidt. "The Temperature Side Channel and Heating Fault Attacks". In: *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*. 2013, pp. 219–235.

[49] M. Hutter, J.-M. Schmidt, and T. Plos. "Contact-based fault injections and power analysis on RFID tags". In: *Circuit Theory and Design, 2009. ECCTD 2009. European Conference on*. IEEE. 2009, pp. 409–412.

[50] *Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange*. ISO/IEC 7816-4. 2013.

[51] *Java Core Libraries: Serialization Filtering*. 2018. URL: `https://docs.oracle.com/javase/10/core/serialization-filtering1.htm`.

[52] *Java Cryptography Architecture (JCA) Reference Guide*. 2016. URL: `https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[53] *JDK 7 Security Enhancements*. 2016. URL: `https://docs.oracle.com/javase/8/docs/technotes/guides/security/enhancements-7.html`.

[54] *JDK 9 Early Access Release Notes*. 2017. URL: `http://jdk.java.net/9/release-notes`.

[55] M. Joye and M. Tunstall, eds. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012.

[56] J. P. Kaps and C. Paar. "Fast DES Implementations for FPGAs and Its Application to a Universal Key-Search Machine". In: *Proceedings of the 5th Annual International Workshop in Selected Areas in Cryptography, SAC'98*. 1999, pp. 234–247.

[57] T. Kasper, D. Oswald, and C. Paar. "A Versatile Framework for Implementation Attacks on Cryptographic RFIDs and Embedded Devices". In: *Trans. Computational Science* 10 (2010), pp. 100–130.

[58] P. C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. 1996, pp. 104–113.

[59] P. Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *CoRR* abs/1801.01203 (2018).

[60] O. Kömmerling and M. G. Kuhn. "Design Principles for Tamper-Resistant Smartcard Processors". In: *Proceedings of the 1st Workshop on Smartcard Technology, Smartcard 1999, Chicago, Illinois, USA, May 10-11, 1999*. 1999.

[61] G. de Koning Gans and J. de Ruiter. "The SmartLogic Tool: Analysing and Testing Smart Card Protocols". In: *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*. 2012, pp. 864–871.

[62] T. Korak and M. Hoefler. "On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms". In: *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*. 2014, pp. 8–17.

[63] Z. Li et al. "The Emperor's New Password Manager: Security Analysis of Web-based Password Managers". In: *Proceedings of the 23rd USENIX Security Symposium*. 2014, pp. 465–479.

[64] M. Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018, pp. 973–990.

[65] D. Longley and S. Rigby. "An Automatic Search for Security Flaws in Key Management Schemes". In: *Computers and Security* 11.1 (Mar. 1992), pp. 75–89.

[66] I. Magaki et al. "ASIC Clouds: Specializing the Datacenter". In: *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA 2016*. 2016, pp. 178–190.

[67] Matthias Hertel, ARM. *AN277, ROM Self-Test in MDK-ARM*. `http://www.keil.com/appnotes/files/an277.pdf`.

[68] MITRE. *CVE-2012-4929: CRIME attack*. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4929`. Sept. 2012.

[69] MITRE. *CVE-2014-0160: Heartbleed bug*. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`. Dec. 2013.

[70] MITRE. *CVE-2017-10345*. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10345`. Oct. 2017.

[71] MITRE. *CVE-2017-10356*. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10356`. Oct. 2017.

[72] MITRE. *CVE-2018-2794*. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-2794`. Apr. 2018.

[73] K. Moriarty, B. Kaliski, and A. Rusch. *PKCS#5: Password-Based Cryptography Specification (Version 2.1)*. `https://www.ietf.org/rfc/rfc8018.txt`. Jan. 2017.

[74] K. Moriarty et al. *PKCS#1: RSA Cryptography Specifications (Version 2.2)*. `https://www.ietf.org/rfc/rfc8017.txt`. Nov. 2016.

[75] S. J. Murdoch et al. "Chip and PIN is Broken". In: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. 2010, pp. 433–446.

[76] C. O'Flynn. "Fault Injection using Crowbars on Embedded Systems". In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 810. URL: `http://eprint.iacr.org/2016/810`.

[77] C. O'Flynn and Z. Chen. "Synchronous sampling and clock recovery of internal oscillators for side channel analysis and fault injection". In: *J. Cryptographic Engineering* 5.1 (2015), pp. 53–69.

[78] C. O'Flynn and Z. (Chen. "ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research". In: *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*. 2014, pp. 243–260.

[79] J. Obermaier and S. Tatschner. "Shedding too much Light on a Microcontroller's Firmware Protection". In: *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017.* 2017.

[80] P. Oechslin. "Making a Faster Cryptanalytic Time-Memory Trade-Off". In: *Proceedings of the 23rd Annual International Cryptology Conference on Advances in Cryptology, CRYPTO 2003*. 2003, pp. 617–630.

[81] Oracle Corporation. *Critical Patch Updates, Security Alerts and Third Party Bulletin*. Oct. 2017. URL: `http://www.oracle.com/technetwork/security-advisory/cpuoct2017-3236626.html`.

[82] Oracle Corporation. *Critical Patch Updates, Security Alerts and Third Party Bulletin*. Apr. 2018. URL: `https://www.oracle.com/technetwork/security-advisory/cpuapr2018-3678067.html`.

[83] Oracle Corporation. *Java Cryptography Architecture, Standard Algorithm Name Documentation for JDK 8*. `http://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#KeyStore`. 2014.

[84] S. Ordas, L. Guillaume-Sage, and P. Maurine. "Electromagnetic fault injection: the curse of flip-flops". In: *J. Cryptographic Engineering* 7.3 (2017), pp. 183–197.

[85] D. A. Osvik, A. Shamir, and E. Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*. 2006, pp. 1–20.

[86] C. Percival. "Stronger Key Derivation via Sequential Memory-Hard Functions". In: (May 2009).

[87] C. Percival and S. Josefsson. *The scrypt Password-Based Key Derivation Function*. `https://tools.ietf.org/html/rfc7914`. Aug. 2016.

[88] S. Picek et al. “Evolving genetic algorithms for fault injection attacks”. In: *37th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2014, Opatija, Croatia, May 26-30, 2014*. 2014, pp. 1106–1111.

[89] *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40*. `http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html`. OASIS Standard. Apr. 2015.

[90] *PKCS #11 v2.30: Cryptographic Token Interface Standard*. `http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm`. RSA Laboratories. Apr. 2009.

[91] J.-J. Quisquater and D. Samyde. “Eddy current for magnetic analysis with active sensor”. In: *Proceedings of eSMART*. Vol. 2002. 2002.

[92] Renesas Electronics. *78K0/Kx2 Flash Memory Programming*. `https://www.renesas.com/en-eu/doc/DocumentServer/024/U17739EJ3V0AN00.pdf`.

[93] Renesas Electronics. *78K0R/Kx3-L Flash Memory Programming*. `https://www.renesas.com/en-eu/doc/DocumentServer/026/U19486EJ1V0AN00.pdf`.

[94] Renesas Electronics. *Renesas Mid-Term Growth Strategy*. `https://www.renesas.com/en-in/media/about/ir/event/presentation/2016-12-q2-strategy.pdf`.

[95] Riscure. *VC Glitcher*. `https://www.riscure.com/uploads/2017/07/datasheet_vcglitcher.pdf`.

[96] Robin Walsh, Atmel. *RC Oscillator Frequency Drift Compensation*. `http://ww1.microchip.com/downloads/en/DeviceDoc/article_ac9_atmegaxx8pa-15-rc-oscillator.pdf`.

[97] RSA Laboratories. *PKCS#11 v2.30: Cryptographic Token Interface Standard*. Apr. 2009.

[98] RSA Laboratories. *PKCS#12: Personal Information Exchange Syntax Standard (Version 1.0)*. June 1999.

[99] RSA Laboratories. *PKCS#12: Personal Information Exchange Syntax Standard (Version 1.1)*. Oct. 2012.

[100] M. Sabt and J. Traoré. “Breaking into the KeyStore: A Practical Forgery Attack Against Android KeyStore”. In: *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS 2016), Part II*. 2016, pp. 531–548.

[101] J. Schmidt and M. Hutter. “Optical and em fault-attacks on crt-based rsa: Concrete results”. In: *Proceedings of 15th Austrian Workshop on Microelectronics*. 2007, pp. 61–67.

[102] B. Schneier. *Applied Cryptography (2nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1995.

[103] B. Schneier, A. Shostack, et al. “Breaking up is hard to do: modeling security threats for smart cards”. In: *USENIX Workshop on Smart Card Technology, Chicago, Illinois, USA, http://www. counterpane. com/smart-card-threats. html*. 1999.

[104] N. Selmane, S. Guilley, and J. Danger. “Practical Setup Time Violation Attacks on AES”. In: *Seventh European Dependable Computing Conference, EDCC-7 2008, Kaunas, Lithuania, 7-9 May 2008*. 2008, pp. 91–96.

[105] N. Selmane et al. “Security evaluation of application-specific integrated circuits and field programmable gate arrays against setup time violation attacks”. In: *IET Information Security* 5.4 (2011), pp. 181–190.

[106] A. Shostack. “Experiences threat modeling at microsoft”. In: *Modeling Security Workshop. Dept. of Computing, Lancaster University, UK*. 2008.

[107] D. Skarin, R. Barbosa, and J. Karlsson. “GOOFI-2: A tool for experimental dependability assessment”. In: *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*. 2010, pp. 557–562.

[108] S. Skorobogatov. “Flash Memory ’Bumping’ Attacks”. In: *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*. 2010, pp. 158–172.

[109] S. Skorobogatov. “Synchronization method for SCA and fault attacks”. In: *J. Cryptographic Engineering* 1.1 (2011), pp. 71–77.

[110] S. P. Skorobogatov and R. J. Anderson. “Optical Fault Induction Attacks”. In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. 2002, pp. 2–12.

[111] R. Spreitzer et al. “Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices”. In: *IEEE Communications Surveys and Tutorials* 20.1 (2018), pp. 465–488.

[112] *Spring Crypto Utils Documentation: Keystore*. 2017. URL: `http://springcryptoutils.com/keystore.html`.

[113] STMicroelectronics. *RM0313 Reference manual*. `http://www.st.com/resource/en/reference_manual/dm00041563.pdf`.

[114] STMicroelectronics. *STMicroelectronics Reports 2015 Fourth Quarter and Full Year Financial Results*. `http://www.st.com/web/en/resource/corporate/financial/quarterly_report/c2792.pdf`. Jan. 2016.

[115] F. Swiderski and W. Snyder. *Threat Modeling*. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 0735619913.

[116] Texas Instruments. *MSP430 ultra-low-power MCUs - Applications*. http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/applications.html.

[117] N. Timmers and C. Mune. "Escalating Privileges in Linux Using Voltage Fault Injection". In: *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017*. 2017, pp. 1–8.

[118] M. S. Turan et al. *Recommendation for Password-Based Key Derivation. Part 1: Storage Applications*. http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf. Dec. 2010.

[119] *Update to Current Use and Deprecation of TDEA*. 2017. URL: https://beta.csrc.nist.gov/News/2017/Update-to-Current-Use-and-Deprecation-of-TDEA.

[120] USB Implementers Forum, Inc. *Device Class Specification for Device Firmware Upgrade*. http://www.usb.org/developers/docs/devclass_docs/DFU_1.1.pdf (Version 1.1 - Aug 5, 2004).

[121] A. Vassilev. *Annex A: Approved Security Functions for FIPS PUB 140-2, Security Requirements for Cryptographic Modules*. http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexa.pdf. Apr. 2016.

[122] S. Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." In: *Proceedings of the 21st International Conference on the Theory and Applications of Cryptographic Techniques Advances in Cryptology, EUROCRYPT 2002*. 2002, pp. 534–546.

[123] F. Vincis and L. Fanucci. "A trimmable RC-oscillator for automotive applications, with low process, supply, and temperature sensitivity". In: *12th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2005, Gammarth, Tunisia, December 11-14, 2005*. 2005, pp. 1–4.

[124] *Vulnerability Note VU#576313*. 2015. URL: https://www.kb.cert.org/vuls/id/576313.

[125] L. Wang, E. Wong, and D. Xu. "A Threat Model Driven Approach for Security Testing". In: *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*. SESS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 10–. ISBN: 0-7695-2952-6. DOI: 10.1109/SESS.2007.2. URL: http://dx.doi.org/10.1109/SESS.2007.2.

[126] *WebLogic Integration 7.0: Configuring the Keystore*. URL: http://docs.oracle.com/cd/E13214_01/wli/docs70/b2bsecur/keystore.htm.

[127] F. F. Yao and Y. L. Yin. "Design and Analysis of Password-Based Key Derivation Functions". In: *IEEE Transactions on Information Theory* 51.9 (2005), pp. 3292–3297.

[128] P. Youn et al. *Robbing the bank with a theorem prover*. Tech. rep. UCAM-CL-TR-644. University of Cambridge, Aug. 2005.

[129] B. Yuce, P. Schaumont, and M. Witteman. “Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation”. In: *Journal of Hardware and Systems Security* (2018), pp. 1–20.

[130] L. Zussa et al. “Investigation of timing constraints violation as a fault injection means”. In: *27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France*. 2012.

[131] L. Zussa et al. “Analysis of the fault injection mechanism related to negative and positive power supply glitches using an on-chip voltmeter”. In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014, Arlington, VA, USA, May 6-7, 2014*. 2014, pp. 130–135.