Inter-CESTI: Methodological and Technical Feedbacks on *Hardware Devices* Evaluations

ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales, Trusted Labs

Abstract. The aim of the current article is to provide both methodological and technical feedback on the "Inter-CESTI" challenge organized by ANSSI in 2019 with all 10 ITSEFs licensed for the French CSPN scheme. The purpose of this challenge is to evaluate their approaches to attack a common target representative of the "Hardware devices with boxes" domain, which groups products containing embedded software and combines hardware and software security elements. The common target chosen was the open-source and open-hardware project WooKey, presented at SSTIC 2018 [30, 31]. It is a relevant test vehicle both in terms of software and hardware due to its architecture and threat model. The article aims to capitalize on the feedback from the challenge, with a focus on the hardware and software tests that the labs were able to conduct in a white box setting, as well as the identified attack paths.

1 Context

Traditionally, an Information Technology Security Evaluation Facility (ITSEF) is licensed by the ANSSI's *Centre national de certification* (CCN, french for National Certification Body) for a given domain, either software or hardware. ITSEFs licensed for software generally deal more with software evaluations (VPN, anti-virus, disk encryption software, etc.). Whereas ITSEFs licensed for hardware focus on evaluating targets closer to hardware products (smart cards, accelerated encryption hardware, etc.).

The "Hardware devices with security boxes" domain includes HSM (Hardware Security Modules), smart meters and various embedded systems. An analysis of this type of product shows a high degree of interdependence between embedded software and the underlying hardware, a feature not found as strongly in the classical purely software or purely hardware areas. This means that in order to evaluate this type of product, a laboratory must necessarily have a dual competence: software and hardware. The strict distinction between these two areas is thus tending to blur. We have indeed noted during the licensing audits of the ITSEFs that those identified in one of the domains may also have skills in the other domain.

The "Hardware devices with security boxes" domain is unfortunately only found under the Common Criteria (CC) [4] scheme for which only

a part of the ITSEFs are licensed. However, France also has a national scheme, called *Certification de Sécurité de Premier Niveau* (CSPN) [1], but where this domain does not exist. As a result, many ITSEFs that are not licensed for CC cannot demonstrate their competence in evaluating products intertwining software and hardware. A possible new CSPN domain "Hardware Device" is therefore being considered to fill this gap.

It so happens that CCN organizes annual challenges to test the skills of ITSEFs, usually with a separation between hardware and software challenges. In the hereabove described context, it has been decided to organize a common "Hardware Device" challenge in 2019, which reflects this domain, and which would allow skill evaluation of all ten ITSEFs for this potential new CSPN domain. To accentuate the difficulty and stimulate the relevance of the outcome, the ITSEFs are encouraged to step out of their comfort zones via dedicated test plans where the ANSSI has selected security functions: the so-called "software" ITSEFs have been allocated a majority of hardware tests, while the so-called "hardware" ITSEFs have tested more software functions.

For this challege's test vehicle, CCN wanted to find a representative product of the "Hardware Device" domain, with, if possible, an open-source design to ease the characterization of the paths of attack and white box testing. The choice fell on the WooKey project [25, 30, 31] developed at ANSSI for which laboratory experts can more easily appreciate the work of the ITSEFs, and eventually provide them technical assistance.

In this article, we first give a quick description of the WooKey product and the elements that made up the test vehicle. We then briefly present the envisaged attack surface and the different attack paths distributed over the ten ITSEFs. Finally, we give various concrete attack paths explored and/or exploited by the ITSEFs with, for each of them, a context, reproducible results and a small quotation.

2 WooKey: the challenge test vehicle

The WooKey project [30, 31] has been selected as a test vehicle for its very representativeness of the "Hardware Device" domain: its hardware design and software architecture (rich in external interfaces) lead to numerous attack paths to undermine the product security. Beyond the attacks themselves, the methodology and test plans of the ITSEFs are a relevant element taken into account for their evaluation (including assessment of their understanding of the target).

WooKey is composed of two main elements (shown on Figure 1):

— A device containing an STM32 MCU (MicroController Unit), a touch screen for user interaction, a micro-SD slot, two USB ports (full-speed and high-speed) and a slot for inserting a standard credit card size smart card.
— An authentication token in the form of an extractable contact smart card (communicating via an ISO7816 bus).

When running in nominal mode, Wookey is a disk encrypting platform allowing to store user's data on a micro-SD card while ensuring their confidentiality. Access to the platform's master secrets is done after a strong user authentication phase involving two factors: an AUTH smart card and two PIN codes (PetPIN and UserPIN). To unlock his WooKey, the user inserts this authentication token in the device, enters his PetPIN on the touch screen, validates a PetName to ensure that the token is correct, then enters his UserPIN to completely unlock the platform. From there, the device connected via USB to a host PC is presented as a mass storage device, and the user can drop and retrieve data with transparent (de)encryption. PIN unlocking actually allows the decryption master key to be retrieved from the token and injected into the cryptographic accelerator of the platform's microcontroller.

WooKey provides another running mode: Device Firmware Upgrade or DFU. In this mode, the device waits for an encrypted and signed update file that upgrades the embedded firmware. In order to unlock this mode, the user presses the physical button to start the platform in DFU mode, inserts a dedicated token (called DFU token), enters PINs (PetPIN, validates PetName, then UserPIN) relevant to this token, and thus accesses the device from the host PC via USB as a DFU class device.

The firmware is encrypted and signed on a trusted PC, using a third dedicated token named SIG (inserted in a generic smart card reader on the PC) and containing notably the private firmware signing key.

WooKey's security relies on various elements of defense in depth:

— A software protection of the firmware through a microkernel written in ADA (type-safe language), isolated tasks with dedicated userland device drivers (for USB, micro-SD card, smart card, display). Thus, a vulnerability from an exposed interface should be confined to the task managing this interface. Additional software security elements are used to reinforce protection: stack canaries, W⊕X thanks to the MPU (Memory Protection Unit), etc.
— Cryptographic protection of sensitive assets, thanks in particular to tokens based on secure components evaluated at level EAL4+ [11]

of the Common Criteria and ensuring robustness against several classes of attacks.

For the sake of brevity, we do not give more information about the WooKey project here. The reader is invited to check the bibliographic references [30,31] and the project documentation [25] for more details. More information can also be found in the CSPN security target evaluation [26] that has been provided to the ITSEFs for the challenge.

Fig. 1. "Closed" WooKey target

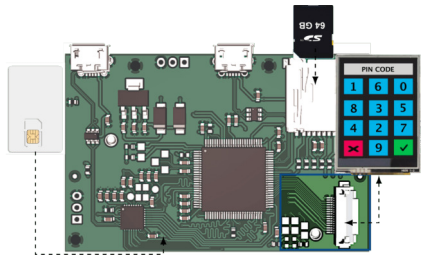

Fig. 2. "Open" WooKey target

3 Evaluation scope of the challenge

The main attacks we are interested in for this challenge are software and hardware pre-authentication attacks, software post-authentication attacks, stealthy hardware pos-authentication attacks, and this on both the WooKey platform and its AUTH and DFU tokens. Attacks bypassing the firmware verification at startup and at update time are also considered. The parts of the tokens covered by their CC EAL4+ certification (notably the Integrated Circuit IC and the Javacard platform) are not considered relevant, only the code of the Javacard applets that runs on these tokens on top of the VM is in the scope of the evaluation.

Hardware trapping and relay attacks, although interesting, are left aside as less relevant to the challenge: the product is inherently susceptible to such attacks. On the other hand, purely software trapping in pre-authentication phases of a platform, or cloning of a platform must be considered. These two scenarios are equivalent in the context of WooKey because it is open-source and open-hardware without hardware "counterfeiting" countermeasures: modifying the firmware of a genuine WooKey or making a new one with the firmware of another brings the same attacks aiming at stealing sensitive assets by pilferage attacks (possibly twice).

The reason motivating our choice concerning the relevant attack paths is the fact that an attacker who has to carry out hardware attacks or combined attacks (hardware and software) will necessarily have an invasive aspect, and will not be able to carry them out during the user authentication phase without him being aware of it. From this observation arise the pre-authentication aspects of hardware attacks. Notable exceptions in post-authentication are stealthy hardware attacks or combined attacks that can be conducted without modifying the board and the token, during the user authentication, and without the user noticing anything. Examples of such attacks are for instance passive secrets recovery at wide-range (with an antenna and so on), exploiting the USB interface consumption as a wiretap to steal sensitive assets, or use voltage glitching through USB to advantageously deflect nominal code execution.

Software attacks are for their part completely relevant in post-authentication: infiltrating the firmware of a WooKey after entering the user's PINs from a malicious USB stack on the host PC or from a trapped SD card to exploit a vulnerability in the platform's SDIO stack are plausible scenarios.

Within the context of the challenge, a CSPN-type security target grouping sensitive assets, usage contexts and security functions was written and provided to the ITSEFs. This security target is made public as a companion document to this article [26]. Finally, each ITSEFs was provided with:

— A completely open WooKey platform (see Figure 2) with JTAG/SWD accessible, and three AUTH/DFU/GIS open and reprogrammable tokens.
— A closed WooKey platform (see Figure 1) with locked JTAG/SWD (in RDP2), and two AUTH/DFU closed tokens (in the GlobalPlatform sense).

4 Identified attack paths and distribution methodology

In order to organize the distribution of the large amount of work covered by the security target [26] to the various ITSEFs, we have split the relevant attack paths between the software and hardware domains. Beyond the search for vulnerabilities, the conformity of the product to the announced specifications must also be verified and validated by the ITSEF.

Each attack path can lead to a partial attack, a combination of them may eventually lead to a complete attack path of the product in various

scenarios such as theft, trapping, trapping with remediation and possibly second flight, etc. As mentioned earlier, in order to be able to evaluate the capacities of software ITSEFs to deal with hardware attacks, and conversely of hardware ITSEFs to deal with software attacks, it was decided to assign each ITSEF attacks at the margin of his area of competence. The results of this distribution are very interesting, and show that the hybrid aspect of the "Hardware Device" domain allows ITSEFs to find relevant attack paths.

On the software side, the goal of the attacker is to exploit a vulnerability in the code to hijack the operation of the product and recover assets. The vulnerabilities considered are those that are purely software, notably Run-Time Errors (stack and heap buffer overflows, integer overflows, etc.) as well as logic errors in state machines. We have identified four axes:

- Static analysis and fuzzing of the exposed code:
  - Exposed software stacks: the most important software elements exposed to the outside are the software stacks managing the communication interfaces. The ISO7816 bus handles communication with the authentication token, the USB bus handles communication with the host PC, the SDIO bus handles communication with the SD card, and the SPI bus handles communication with the touch screen. A vulnerability exploited in one of the software stacks handling them allows the control of the corresponding task (in particular, the tasks managing the AUTH and DFU tokens contain sensitive elements in their memory space).
  - Syscalls: system calls are a potential entry point to either perform a privilege escalation, or to contaminate another task (e.g. via poor IPC management/implementation), or to establish covert channels between tasks.
- The analysis of the Bootloader:
  The WooKey Bootloader is a critcal software element. On one hand it is executed at boot time and decides which partition will be executed on the platform (upon integrity checking and verification of its version). On the other hand it is not updatable. It is therefore crucial to check the proper functioning of its state machines. The goal of the Bootloader study is to find by code proofreading, fuzzing and other analyses, remaining vulnerabilities as well as weaknesses/fragility to software attacks.
- MPU management analysis and privilege separation:
  The MPU (Memory Protection Unit) is the cornerstone of WooKey's software defense in depth since it allows task parti-

tioning and the W⊕X enforcement. It is therefore important to validate by static analysis, code review or dynamic analysis (from each task) the correct configuration of the MPU by the kernel.

— Analysis of the Javacard applets:
On the token side, the applets are implemented using the Javacard language. The underlying NXP JCOP J3D081 platform is certified by the Common Criteria [11] ensuring resistance to some advanced attacks. Nevertheless, it is important to validate that the implementation of state machines and the product applet life cycle is free of vulnerabilities. For instance, insure that it is impossible to extract sensitive assets without prior authentication with PINs (in case of loss or theft with token delivery for example). Moreover, as some algorithms are not exposed or made available by the platform API, they have been fully implemented in Javacard (this is the case for example of the CTR mode of the AES, the HMAC, etc.), as they are not covered by the Common Criteria certification. It is hence important to validate that these custom implementations are not subject to cryptographic weaknesses, and are SCA (Side-Channel Attacks) as well as FIA (Fault Injection Attacks) resistant.

On the hardware side, the attacks foreseen in the CSPN framework are those using hardware at a price considered as "reasonable", e.g. oscilloscope, logic analyser, common and/or accessible electronic hardware (MCU-based development boards, FPGA, ChipWhisperer [3], etc.). Attacks using chip stripping (e.g. heavy chemistry), laser faults, FIB (Focused Ion Beam), etc., are considered to be too highly rated (and therefore out of scope).

In particular, we consider appropriate Fault Injection Attacks (FIA) using voltage glitch [32], clock glitch or EM (electromagnetic emanations) glitch [2] to be in scope. These attacks, as opposed to laser injections, do not require chip stripping neither complex nor expensive hardware. Disrupting the voltage requires, for example, an FPGA and an oscilloscope, costing a few hundred euros (plus a rather "simple" preparation of the target to be attacked by generally minor modifications of the PCB).

— Fault injection and glitch attacks (FIA): These attacks focus on the glitch which allows, via the injection of one or more faults, to hijack security primitives. The rating of such attacks is interesting because it must take into account the target preparation time (for example removing parasitic capacities on the PCB), the mapping time to find exploitable faults (spatial and/or temporal mapping), the probability that an exploitable fault will occur, etc. Stealthy FIA that do not require any preparation or PCB modification (e.g.

discrete injection through USB) are of course very relevant as they either do not require stealing the target for a very long time, or allow a direct attack without any stealing.

  - Attacks against RDP: RDP (Readout Protection) is the technology implemented in STM32 MCUs to protect flash data and remove access to JTAG/SWD. RDP seems to be quite prone to faults as recent publications have shown [21,32,49]. A successful pre-authentication attack on RDP would allow the trapping or cloning of a WooKey.
  - Fragility of the Bootloader: the Bootloader implements itself protections against RDP downgrade, in order to potentially detect a successful fault injection before its execution. The Bootloader is also supposed to check the integrity and manage an anti-rollback mechanism on the partitions present in the internal flash. All these elements are potentially "faulty", and the analysis of their robustness is interesting.
  - Sensitive code fragility: generally speaking, any task or kernel code exposed and sensitive to glitches in pre-authentication is an interesting attack surface to evaluate, and this in relation to and according to the fault model characterized on the target.

- Side-Channel Attacks (SCA):
  Some cryptographic primitives are used in WooKey for pre-authentication, and for those that manipulate secret data it may be interesting to evaluate their robustness to SCA. These attacks would make it possible to extract the secrets via the acquisition of consumption curves or electromagnetic emanations. They can be devastating, specifically if such attacks are possible in a stealthy way (e.g. with an antenna at a wide-range from the target, through monitoring the USB interface, etc.): in this case, even post-authentication attack scenarios must be considered. In this last case, SCA somewhat meet TEMPEST evaluations (see below).
- Analysis of communication buses:
  The target contains several buses on which potentially sensitive data transit. Notably, the ISO7816 bus between the platform and the token is supposed to establish a secure communication channel: it is important to validate its conformity. The SDIO bus allows to interact with the dedicated task, and potentially to take control of it in case of a programming error. Logic analyzers allow bus sniffing, as well as potentially induce malformed packets injections allowing, for example, fuzzing at the lowest protocol level.

— TEMPEST attacks:
PIN entry is done on a touch screen, which implies potential remote EM leaks regarding the keys pressed by the legitimate user during authentication. Knowing that these elements could be captured several meters away using a well-sized antenna (and therefore discreetly), a TEMPEST evaluation of the product is relevant.

Finally, and as previously mentioned, although interesting, attacks by hardware trapping of the platform (especially relay attacks) are also studied but with a lower priority.

5 Attacks overview: a reader's digest

In the sequel of the article, various (partial or complete) attack paths analysis performed by the ITSEFs during the challenge are reported in detail. Although this does not cover all the analysis and all the explored paths (for brevity reasons and to avoid an illegible article), the most relevant results are compiled.

These results provide insightful details about the adopted technical methodology, the necessary equipment and setup (software and hardware), the found vulnerabilities and weaknesses and their possible (partial or total) exploitation in the light of WooKey's threat model and security target [26]. In order to provide the reader a bird's view of these results, the current section classifies them in categories and places them in an exploitation context with regard to WooKey. We also provide links between described partial attacks that would lead to a more complete attack path, as well as some results that can be used as technical inputs to other results, that would improve the overall analysis.

The reader should also be aware that most of the described evaluations have been conducted by various ITSEFs in an independent and parallel way. This can induce some intersections between the descriptions, and some explainable redundancies in the performed tests and explanations. In any case, such a redundancy is interesting since it also allows to capture different approaches for the same target problem. As a matter of fact, WooKey's Bootloader robustness against FIA has been widely stressed, and sections 14, 16, 9 and 15 tackle various methodologies (EM and voltage glitches, formal methods) to find defects and exploit them in this rather critical piece of code. Finally, all the performed tests are not presented in the current article mainly for concision considerations: we have selected what we believe to be relevant and complementary attack paths. For instance, fuzzing campaigns on the USB stack are not presented, although

being apposite, as they provided less interesting feedback and results than other approaches.

Our classification of the evaluations and attacks, albeit somewhat artificial, tries to capture the big thematic axis presented in 4. Table 1 provides a good overview of the 15 attack paths detailed in the current article, with their corresponding section references and the various technical aspects they cover. We can roughly distinguish three kinds of attack paths (but this distinction is not exclusive, an attack path can be in more than one category):

— First, the ones that are oriented towards software exploits (buffer overflows, automatons weaknesses, etc.) and only use software techniques (i.e. no PCB preparation or advanced equipment): 02 involves a pure software fuzzing of the ISO7816 driver and the token abstraction library (exploiting their portability); 04 finds a privilege escalation in the EwoK microkernel through syscalls software fuzzing; 05 explores MPU configuration using dedicated task instrumentation. 01 tries with the help of static code review to check that the automatons in the Javacard tokens do not present software weaknesses. 12 uses formal methods to explore potential RTE (RunTime Errors) in the Bootloader code.
— Then, we have paths that are dedicated to evaluate physical attacks resistance, mainly against SCA and FIA. These attacks usually involve PCB preparation and dedicated equipment to be exploited, and encompass scenarios where the WooKey platform and/or the token are stolen for a certain amount of time. The main notable exceptions are TEMPEST attacks since they require a dedicated equipment, but do not require to steal the target and operate stealthily. 08 reviews ECDSA and ECDH code against SCA during pre-authentications attacks to recover the platform ECDSA key. 09 exploits a leakage in the HMAC computation of the platform keys authenticity tag to extract encrypted platform keys using SCA. 10 uses FIA with voltage glitches to try to bypass the STM32 Readout Protection that prevents adversaries to read the MCU internal flash. 11 explores EM fault injection effects on WooKey's Bootloader, more particularly on internal firmware integrity check. 12 uses voltage glitches to perform FIA and defeat the firmware anti-rollback mechanism. 15 shows the relevance of TEMPEST attacks on the SPI bus, and the results have to be coupled with 14 that analyses SPI communication details and paves the way to

decoding them. Finally, 01 also analyses Javacard applets for heir robustness against physical attacks and provides best practices.

— The inter-CESTI challenge has also exhibited interesting attack paths that are neither purely software nor hardware, but are rather so called *hybrid attacks* that advantageously mix the two aspects. A good example of this is 03 where a voltage glitch is used to trigger and exploit a buffer overflow in the ISO7816 library. 13 deeply analyses the SDIO software layer with bus fuzzing in mind (in order to find software vulnerabilities), which requires a dedicated hardware setup. 12, although already classified both in software (RTE) and physical (FIA) attacks, also finds its place in hybrid ones: static code analysis and formal methods are used to find FIA exploitable spots and deceive anti-rollback. 06 mixes a cryptographic weakness during pre-authentication with SPI bus instrumentation to perform a bruteforce attack on the PetPIN. Finally, 07 checks the conformity of the Secure Channel protocol using specification review and bus ISO7816 sniffing.

	Static code analysis/review	Software exploitation	Software fuzzing	Hardware fuzzing	MPU analysis	Bus sniffing	Bus injection	Crypto attack	SCA	FIA	TEMPEST
❶ Javacard applets analysis (section 6)	x								x	x	
❷ `libiso7816` and `libtoken` fuzzing (section 8)			x								
❸ `libiso7816` glitch attack (section 9)	x	x								x	
❹ EwoK privilege escalation (section 10)		x	x								
❺ MPU configuration review (section 11)			x		x						
❻ PetPIN bruteforce attack (section 7.1)							x	x			
❼ Secure Channel review (section 7.2)	x					x		x			
❽ ECDSA physical attacks (section 12)	x								x		
❾ HMAC physical attacks (section 13)									x		
❿ Bootloader: RDP2 downgrade (section 14)	x									x	
⓫ Bootloader: integrity check bypass (section 15)	x									x	
⓬ Bootloader: anti-rollback bypass (section 16)	x									x	
⓭ SDIO bus analysis (section 17)				x		x					
⓮ SPI bus analysis (section 18)						x					
⓯ SPI TEMPEST (section 19)											x

Table 1. Presented attacks classification

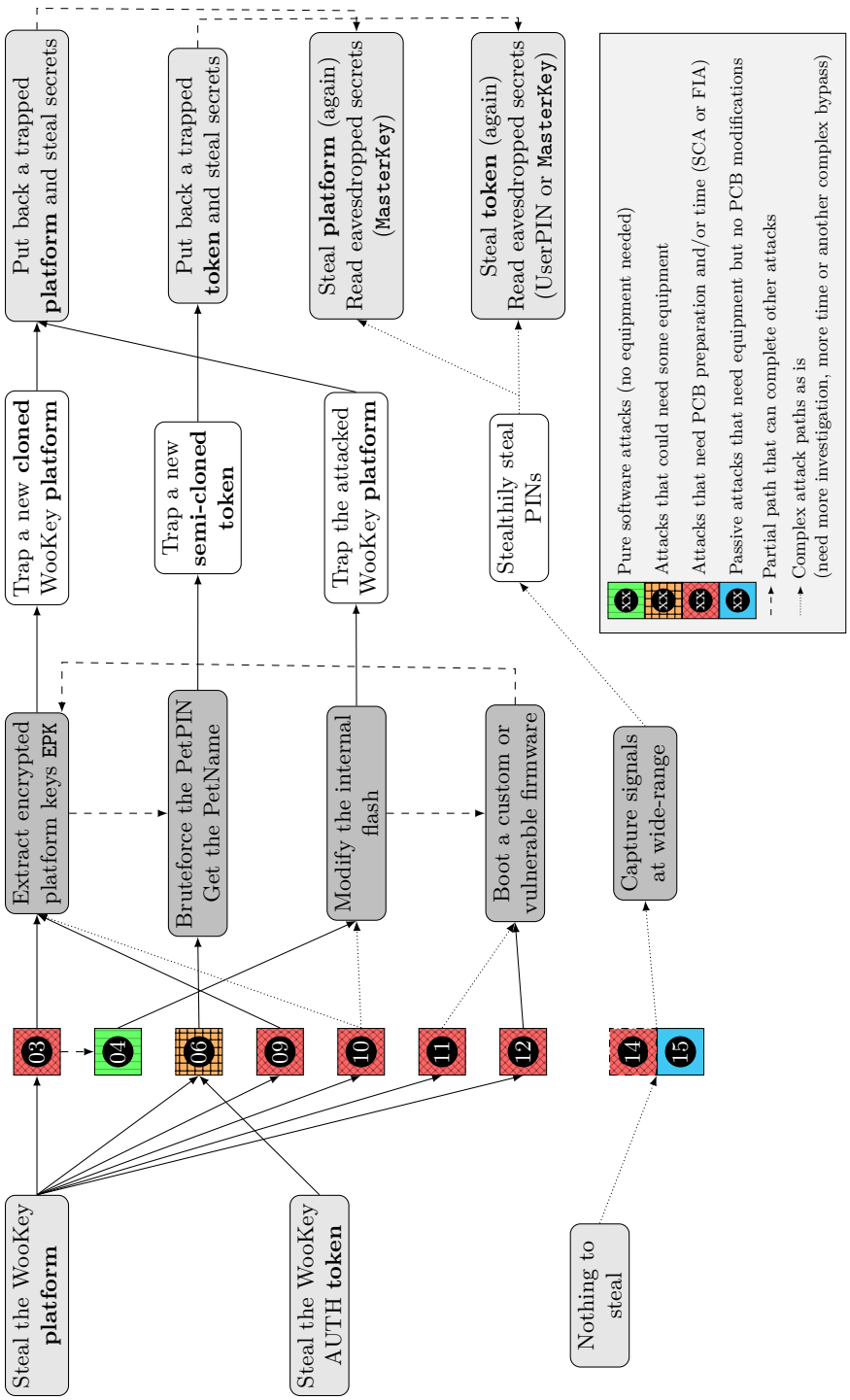

Fig. 3. Attack paths overview

Since the attack paths and their place in the threat mode of the target can be tedious to evaluate, we have represented on Figure 3 an overview of the most representative scenarios that have emerged from the inter-CESTI challenge. From bottom to top of Figure 3, we consider what the adversary must do (steal the platform or the token or only stealthily observe them). Then, what attacks in Table 1 might be used to obtain or modify one of the critical assets (the encrypted platform keys `EPK`, the PetPIN and PetName, the internal flash, etc.). Each asset recovery primitive can lead to trapping the platform and/or the token, and takes place in a more elaborate scenario where the adversary finally recovers the most sensitive assets, i.e. the user data encryption master secrets (`MasterKey`, etc.).

Regarding the attacks, we distinguish pure software attacks where no complex equipment is necessary from the ones where PCB modifications, instrumentations and/or time (e.g. for multiple acquisitions or trials) are necessary, typically to perform SCA and FIA. represents attacks that might require PCB modifications for readiness, but could optionally be performed otherwise with less efficiency. Finally, illustrates fully passive and stealthy attacks (mainly TEMPEST based ones). Paths represented with ⇢ show how partial attacks can lead to other steps and unlock a full attack path to sensitive user assets recovery, while ⟶ are straightforwardly applicable. Paths represented with ⇢ are considered complex "as is" because they either require more investigation to be practical, or are blocked by other defense in depth mechanisms.

Some interesting elements can be observed on Figure 3: practically recovering the user data encryption assets requires scenarios that involve stealing either the platform or the token, trapping them, putting them back to deceive the legitimate user, and get back the assets by other means (e.g. stealing the platform or token again, or send them using radio communication in an advanced hardware trapping scenario). In all these cases, the stolen elements must be attacked using PCB preparation and hardware attacks that could be destructive: the adversary can provide a hardware clone in this case. The pure software privilege escalation 04 can help to modify the attacked platform when it is fixable to give it back and deceive the user (i.e. this prevents investing too much money in fabricating a new cloned platform).

All in all, many of the discovered attack paths have been extensively **patched** by the WooKey project in recent commits. 04, 06 and 09 have direct and forthright patches that discard the attacks. 01, 07 and 08 advice some code improvements and mitigation (that have been indeed developed) even though no practical attack have been found and/or other

defense in depth mechanisms prevent exploitation. 02, 05, and 13 did not leverage concrete exploitable attack paths, so nothing to do here. 14 and 15, related to TEMPEST attacks, are a work in progress as they require more characterization and could result in hardware modifications. Finally, 03, 10, 11 and 12 are the most tedious to address as they are a direct consequence of the *STM32 susceptibility to FIA*. Best efforts have been put (using double checks and so on) in recent commits to achieve some minimum robustness level, although no absolute "formal proof" can be provided here.[1] It should be however noticed that such attacks take place in quite elaborate and complex scenarios involving stealing (possibly twice) the platform and deceiving the user with a trapped device. Almost equivalent (yet more complex to mount) threats are relay attacks, that are very hard to mitigate in an open-source and open-hardware context.

6 Analysis of the tokens: Javacard applets

The WooKey tokens (AUTH, DFU and SIG) are critical pieces of the project as they protect the main sensitive assets. Hence, an analysis of the applets implementation regarding their automatons and their compliance with Javacard coding best practices is necessary.

6.1 Threat model

The rationale behind the three token types automatons is that almost no command[2] should be allowed outside the Secure Channel that should be established with a legitimate platform through mutual authentication. The idea with limiting the command set here is attack surface reduction at its strict minimum in the pre-authentication phase. Moreover, all the "privileged" commands (retrieving sensitive assets) must only be accessible after a full user authentication using his PetPIN and then his UserPIN.

The underlying possible attacks cover very critical software and logical attacks on the automatons where the assets could be retrieved only by interacting with the token through APDUs without advanced equipment (e.g. using a buffer leak in the APDU memory), by bypassing the user authentication (e.g. bad verification of the PINs), or by exploiting bad implementations of the Secure Channel yielding in command injection without authentication.

1. This is even more true when including compilation and optimization related issues. Software code FIA resistance is an ongoing academic and industrial research topic.

2. The two exceptions are the command used for the platform keys `PK` decryption, and the command establishing the Secure Channel.

Beyond "pure software" attacks, side-channel and faults injection robustness of the code is also put under the microscope. The main issue with such an analysis is that this robustness may heavily rely on the underlying chip and Javacard VM countermeasures. Such elements should however be covered by the EAL4+ certification. Consequently, the assumptions taken in the next analysis are the following:

— All the native algorithms (AES, ECDSA, ECDH, SHA, etc.) and modes (ECB, CBC, etc.) are supposed to be resistant against SCA and FIA. On the other hand, algorithms and modes that do not directly call the Javacard API (i.e. partially or completely implemented in Javacard) should be scrutinized for robustness against such attacks.
— PIN handling primitives (`OwnerPIN` offered by the Javacard API) should be resistant against common attacks.
— GlobalPlatform token locking is working as specified and no new applet can be loaded on the smart cards without knowing the dedicated secret key.

Since in the WooKey project each token is physically dedicated to its role, and since no new applet can be loaded on the token, attacks where a malicious applet tries to attack the project applets (using shared static variables or a bad implementation) can be discarded from the security analysis since only trusted applets are loaded on the locked tokens.

By extension, attacks abusing possible VM vulnerabilities with regard to the Javacard security model such as type confusion and firewall issues [41] can also be discarded (even if such attacks should be covered by the EAL4+ certification).

As a side note, around 20 man days have been allocated to the applets code review.

6.2 Analysis of the Javacard tokens

Overview of the Javacard tokens code The first step of the evaluation consisted in the code architecture analysis of the three token types AUTH (user authentication for nominal mode), DFU (open a firmware update session and derive keys in DFU mode), and SIG (open a signing session and sign the firmware with ECDSA on a trusted PC host).

An overview of the code architecture is presented in Table 2. Three different applets are compiled, one for each token. Most of the code is shared among all the applets and is present in the `common/` folder. For each token, a dedicated applet class (`WooKeyAuth` for the AUTH token, `WooKeyDFU` for the DFU token, `WooKeyDFU` for the DFU token, `WooKeySIG`

for the SIG token) extends a shared `WooKey` class. This shared class implements all the APDU commands that handle the Secure Channel, the user authentication and token life cycle regarding the PINs and PetName. `WooKeyAuth` adds a command to handle the user SD card master encryption key retrieval after a successful authentication. `WooKeyDFU` and `WooKeySIG` implement commands that handle firmware verification and signature using session keys derivation sessions. Finally, the `(private)` folder contains the personalization data for each token as static buffers that are automatically generated by the WooKey SDK.

The tokens have two phases during their life cycle:

1. Personalization phase: at the first `select` of the applets, objects and buffers are allocated and some personalization data is instantiated in dedicated internal objects.
2. Nominal phase: the instantiated objects are used without new allocations.

Folder	Token	Files	Usage
`auth/`	Specific files dedicated to AUTH token	`WooKeyAuth.java` (129 lines)	AUTH token specific code (GetKey command)
`common/`	Files that are common to all tokens	`Aes.java` (353 lines)	AES object and dedicated methods (CBC, CTR, etc.)
		`ECKeyPair.java` (12 lines)	Wrapper object for ECC private and public key pair
		`SecureChannel.java` (406 lines)	Methods for the Secure Channel handling
		`ECCurves.java` (588 lines)	ECCurves object and methods for ECDSA signature and ECDH
		`Hmac.java` (352 lines)	HMAC object and methods
		`WooKey.java` (609 lines)	WooKey object and methods, handling common token commands (opening the Secure Channel, authentication, etc.)
`dfu/`	Specific files dedicated to DFU token	`WooKeyDFU.java` (299 lines)	Code specific to the DFU token (dedicated commands to open an update session and derive keys)
`sig/`	Specific files dedicated to SIG token	`WooKeySIG.java` (427 lines)	Code specific to the SIG token (dedicated commands to open a signature session and derive keys)
`(private)`	Specific files dedicated to private data	`Keys.java` (32 lines)	Static class containing personalization keys, PetName and PINs

Table 2. Overview of the Javacard tokens source code tree

Assets protection analysis In the `(private)` folder, a dedicated Javacard class `Keys.java` is generated by the WooKey SDK (see Listing 1). The elements of this class are passed as arguments to the `WooKey` common class constructor at the personalization phase at the first applet selection as shown on Listing 2. Consequently, the static buffers present in the `Keys` class are used to instantiate internal objects such as ECC keys: for instance, 32 bytes `OurPrivKeyBuf` is used to fill an internal Javacard `ECPrivateKey` object. Since `ECPrivateKey` is provided by the native API, it should be safe to use and covered by the EAL4+ certification. Once the initialization of the `ECPrivateKey` has been performed, the static buffer `Keys.OurPrivKeyBuf` is zeroized.

The same "instantiate a secure object and zeroize" logic is performed for most of the assets, except for four of them. Table 3 presents an overview of static assets buffers from the class `Keys.java` life cycle after the personalization phase. The buffers stored in secure objects are in green, while those that remain in non secure buffers are in red. More specifically, the `MasterSecretKey` that holds the SD card encryption key, `EncLocalPetSecretKey` (`ELK` in the security target), and the PetName remain in non secure buffers.

A potential issue with zeroization of the initialized buffers is the fact that zeroization yields in known values to the attacker. If the attacker tricks an already initialized token (e.g. with a fault injection) making it believe that it is not, zeroes are copied in sensitive assets (e.g. the PINs): it is then possible to establish a Secure Channel and get the `MasterSecretKey`.[4] It is recommended to fill the used buffers with random values.

Even though there is no direct attack path to recover such sensitive assets, it is strongly advised to protect such buffers with dedicated secure objects provided by the Javacard API. This is not an easy task on general purpose buffers. Possible solutions consist either in abusing the existing native Javacard key objects as discussed in section 5.5 of [58], or in locally encrypting such buffers with a key protected in a secure buffer.

```
package wookey_auth;
```

3. See WooKey security target [26] for assets details.

4. Such an attack is a bit trickier to mount though, since the ECC keys will contain zeroes and probably provoke exceptions.

Key / perso data	Type	Size	Asset [3]	Asset cleaning method	Destination	Type destination
OurPrivKeyBuf	static byte[]	32	[A13]	Constructor and self_destroy_card() in WooKey class	W.schannel.OurKey PairWrapper.PrivKey	ECPrivate Key
OurPubKeyBuf	static byte[]	65	[A13]	Constructor and self_destroy_card() in WooKey class	W.schannel.OurKey PairWrapper.PubKey	ECPublic Key
WooKeyPubKey Buf	static byte[]	65		Constructor and self_destroy_card() in WooKey class	W.schannel.WooKey KeyPairWrapper.PubKey	ECPublic Key
LibECCparams	static byte[]	2		self_destroy_card() in WooKey class	W.schannel.ec_context. ECCparams	byte[]
PetPin	static byte[]	4	[A3]	Constructor and self_destroy_card() in WooKey class	W.pet_pin	OwnerPIN
PetName Length	static short	1	[A4]	Never	W.PetNameLength	short
PetName	static byte[]	64	[A4]	self_destroy_card() in WooKey class	W.PetName	byte[]
UserPin	static byte[]	4	[A2]	Constructor and self_destroy_card() in WooKey class	W.user_pin	OwnerPIN
MasterSecretKey	static byte[]	32	[A9]	self_destroy_card() in each applet	None	N/A
EncLocalPet SecretKey	static byte[]	64	[A6]	self_destroy_card() in WooKey class	None	N/A
max_pin_tries	static final byte	1		Never	W.pet_pin and W.user_pin	OwnerPIN
max_secure _channel_tries	static final short	1		Never	W.sc_max_ failed_attempts	short

Table 3. Javacard tokens assets review

```
class Keys {
    static byte[] OurPrivKeyBuf = { (byte)0x2d, (byte)0x87, ... };
    static byte[] OurPubKeyBuf = { (byte)0x04, (byte)0xc3, ... };
    static byte[] WooKeyPubKeyBuf = { (byte)0x04, (byte)0x91, ... };
    static byte[] LibECCparams = { (byte)0x01, (byte)0x01, ... };
    static byte[] PetPin = { (byte)0x31, (byte)0x32, (byte)0x33, (
        byte)0x34, ... };
    static short PetNameLength = 5;
    static byte[] PetName = { (byte)0x57, (byte)0x6f, ... };
    static byte[] UserPin = { (byte)0x31, (byte)0x33, (byte)0x33, (
        byte)0x37, ... };
    static byte[] MasterSecretKey = { (byte)0x27, (byte)0x59, ... };
    static byte[] EncLocalPetSecretKey = { (byte)0xa6, (byte)0x89,
        ... };
    static final byte max_pin_tries = (byte)3;
    static final short max_secure_channel_tries = 10;
}
```

Listing 1. `Keys` class with sensitive assets

```
if((W == null) || (init_done == false)){
    init_done = false;
    W = new WooKey(Keys.UserPin, Keys.PetPin, Keys.WooKeyPubKeyBuf,
        Keys.LibECCparams, Keys.PetName, Keys.
        max_secure_channel_tries);
    init_done = true;
}
```

Listing 2. `WooKey` class construction

Whenever a security issue is detected on the tokens, e.g. too much failed attempts for user authentications or Secure Channel establishment, sensitive assets are destroyed using the `self_destroy_card()` method of the `WooKey` object. This consists of zeroizing the assets in the `Keys` class, but the copies of the assets in the native API secure buffers (`ECPrivateKey` and so on) are not erased while they should be.

Code review, SCA and FIA robustness The code makes extensive use of secure Javacard primitives for buffers manipulations (`arrayCopyNonAtomic`, `arrayFillNonAtomic`, `arrayCompare` and so on), which is a good practice. The evaluation has then focused on all the classical `for()` and `while()` loops to check if they manipulate sensitive assets (hence showing potential issues with regard to SCA leakage or FIA bypass).

A first loop is used in the AES-CTR IV incrementation for Secure Channel encryption (see Listing 3). This loop is not fully balanced but exploiting it in SCA seems complex and of little interest.

```
private void increment_iv(){
    short i;
    byte end = 0, dummy = 0;
    for(i = (short)IV.length; i > 0; i--){
        if(end == 0){
            if((++IV[(short)(i - 1)] != 0))
                end = 1;
        }
        else dummy++;
    }
}
```

Listing 3. AES-CTR IV incrementation

Some ephemeral Secure Channel keys are also handled in a way that could potentially leak information, but their ephemeral aspect renders this exploitation useless.

On the core algorithms side, although SCA attacks are residual and post-authentication, it is recommended to add some masking and shuffling protections:

— The Javacard implementation of HMAC, although masked, manipulates the key bytes in order.
— The AES-CTR `xor` operation is performed unmasked and in order, yielding in a possible leakage.

Logical checks about the automaton sequence and the privileged commands access (post-authentication using PetPIN and UserPIN, and Secure Channel establishment) have been checked to be correctly implemented without apparent loopholes. The Secure Channel and DFU/SIG sessions states (open or closed) handling is performed using a unique `short` variable value `{0xAA, 0x55}` for the privileged state, which seems robust. However, doubling the `if` checks should be enforced in order to add robustness against fault injections and not fully depend on the underlying platform countermeasures. The same advice holds for other parts of the code (PINs checks conditions, Secure Channel failed attempts).

7 Cryptographic mechanisms review

7.1 Pre-authentication phase issue: PetPIN bruteforce

The cryptographic mechanisms are well detailed in the evaluation companion documents, which helps in analyzing and reviewing them. This helped to find an issue in the first phase of the pre-authentication protocol. Since this protocol is quite complex, it won't be detailed here: the reader can refer to the WooKey project documentation [25, 30] and the security

target evaluation [26]. Here are the first steps of this pre-authentication phase:

— The PetPIN is entered on the WooKey touch screen.
— A key called DK (for Derived Key) of 512 bits is computed using a PBKDF2-SHA512 derivation function with 4096 rounds from the PetPIN and a 128-bit salt stored in the WooKey internal flash at personalization phase.
— The DK key is sent to the token in order for it to decrypt a secret named ELK (for Encrypted Local Key). The decrypted form is named KPK (for Keys Platform Key), and is returned to the WooKey platform where it will be used to check a HMAC and decrypt a bag of keys serving to mount the Secure Channel for the next steps of the protocol.

The rationale here is that when the PetPIN is not correct, DK will not be correct and ELK decryption will produce an invalid KPK failing in platform keys HMAC verification and decryption.

The main issue here is that the operation $\texttt{KPK} = Dec_{\texttt{DK}}(\texttt{ELK})$ can be reversed and stays true, whether DK is correct or not. As a consequence, it is possible to enter an incorrect PetPIN on the WooKey, sniff the communication between the platform and the token, extract the incorrect values $\texttt{DK}^*$ and $\texttt{KPK}^*$, and compute $\texttt{ELK} = Enc_{\texttt{DK}^*}(\texttt{KPK}^*)$ to get the secret value in the smart card. For practicality, this attack can be mounted using only a smart card reader on a PC and the target token.

What can be done then? It is possible to create a fake smart card answering to the first step of the protocol but without the timing countermeasures introduced in the real token (i.e. the failing attempts counters and the increasing time with such failing counters). This fake token lays the first brick to mount a bruteforce attack against the PetPIN: the idea is to try every possible PetPIN and check whether a Secure Channel is established by the platform (meaning that the PetPIN is correct). Once the PetPIN is found, the PetName can be recovered on the WooKey screen. What are the following steps for an attacker from here? The PetPIN has been designed to limit theft attacks. Now that PetPIN and PetName are known, the attacker is able to replace the genuine WooKey and token with fake trapped ones: since the user will see the correct PetName after entering his PetPIN, nothing stops him from entering his real UserPIN that will be transmitted to the attacker using e.g. a relay attack. Now that the attacker has the real token and the UserPIN at hand, the SD card encryption secrets (`MasterSecretKey`) can be recovered.

Now back on bruteforcing the PetPIN: a challenging issue is to automate the attack using the WooKey platform by instrumenting its user interface. The keys layout on the virtual keyboard is randomized at each boot, this can be solved by sniffing the SPI traffic between the main board and the screen and interpret the pixels related commands. Now, one must simulate "key presses", which can be performed by injecting SPI traffic between the main board and the touch screen: this part is a bit tedious because of some issues in the code handling the SPI bus. More details about the SPI communication can be found in the dedicated section 18. All in all and after some tuning, a stable and automated solution has been developed. The last element of the attack is the detection, by sniffing the ISO7816 bus between the platform and the token, that an attempt to mount the Secure Channel is triggered or not. All these elements are implemented in a *Teensy* board [19] as shown on Figure 4. The 4 digits PetPIN of the closed WooKey is found in 15 hours, with 4 attempts per minute (41 hours are necessary to cover all the possible 4 digits PINs search space). The counterpart of this attack is that bigger PINs highly increase brute force computation, as the attempts frequency can't be significantly improved.

Fig. 4. WooKey instrumentation with a *Teensy* for PetPIN bruteforce

On the other hand, if an attacker is able, using another attack vector, to extract the WooKey firmware from the internal flash (and all the encrypted key bags with it), then the PetPIN bruteforce can be performed completely offline without the previous necessary instrumentation. This allows to cover all the possible 4 digits PINs in 30 seconds on a common laptop, and all the 8 digit PINs in 4 days, this time increasing exponentially with the number of digits.

As a conclusion, the PetPIN of the WooKey project must have a better protection, and an easy fix would be to use another relation between `DK`, `KPK` and `ELK` to break the exploited reversible operation (e.g. a one-way function). The number of allowed failed attempts to compute `KPK` should also be reduced to limit bruteforce attacks. Increasing minimum PIN size is also an efficient defense in depth countermeasure.

7.2 Secure Channel review and improvement

After the pre-authentication phase where the platform keys `PK` have been decrypted, the Secure Channel is established between the platform and the tokens. The main operations of this protocol are presented on Figure 6: they are basically an ECDSA signed ephemeral ECDH. The platform and the token both draw random scalars $d1$ and $d2$, send each other signed ECDH points $(d1 \times G)$ and $(d2 \times G)$, from which they are both able to compute $(d1 \times d2 \times G)$ and derive the Secure Channel AES-CTR encryption key for confidentiality, HMAC key for integrity, and IV for anti-replay.

The evaluator has verified both on the platform and the token side that the Secure Channel is properly implemented, in accordance with its specification in the project documentation. Sniffing of the ISO7816 bus has been realized using a *Saleae Logic Pro 16* [17] logical analyzer and a software protocol decoder [8], and developing a dedicated *WireShark* [22] `pcap` decoder for WooKey high level APDU and response command parsing as exposed on Figure 5. The analysis has shown that the APDUs and the responses are indeed encrypted and integrity protected, and that the sequence of commands to mount the Secure Channel is respected. Section 12 focuses with more details on the ECDSA and ECDH primitives as implemented on the platform side.

The platform and the token have asymmetrical roles here: because of the ISO7816 bus constraints, the token is a slave waiting for the platform initiator to trigger a Secure Channel. From the protocol design, nothing prevents an attacker without the platform keys from replaying the same sniffed initial value $(d1 \times G)$ and its ECDSA signature that will always be

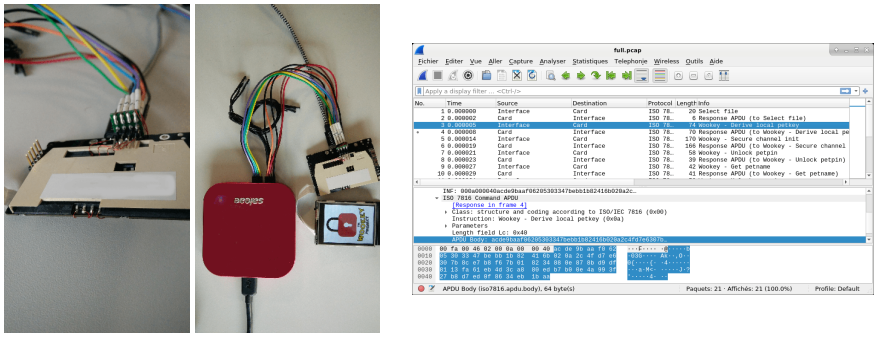

Fig. 5. Sniffing the ISO7816 bus for Secure Channel conformity check

verified by the token. The main consequence of this is that the token will always perform its ECDSA signature computation on a random $(d2 \times G)$. Although this does not leverage a cryptographic vulnerability (the attacker will not be able to compute the shared secret anyways), an undesirable consequence is that the attacker is able to collect randomized signatures from the token, and hence possibly perform SCA to exploit potential leakages.

Notwithstanding the fact that the smart cards used for the tokens are EAL4+ certified (their ECDSA implementation should be immune to such attacks or require very advanced equipment), a defense in depth mitigation is advised by adding a random challenge sent by the token during pre-authentication, and verified at Secure Channel establishment.

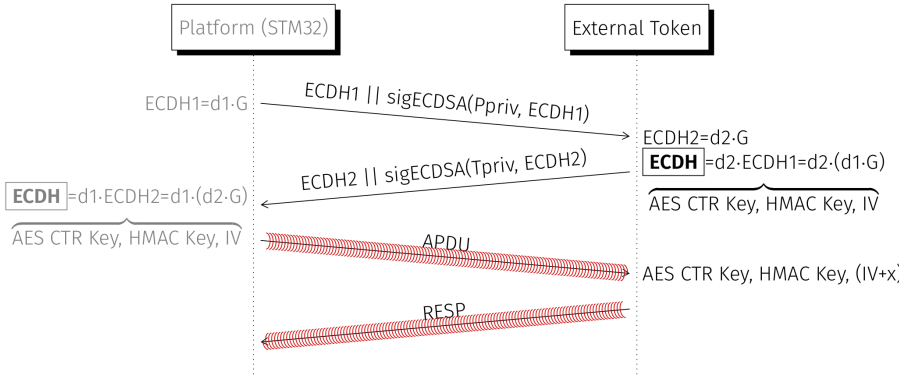

Fig. 6. Secure Channel establishment between the WooKey platform and the tokens

8 Fuzzing the `libiso7816` and `libtoken` libraries

This section is about a fuzzing campaign for the WooKey's implementation of the ISO7816-3 stack and the commands built above (i.e. the token abstract communication protocol). The ISO7816-3 stack is used to communicate with the smart card. This is a really interesting target because it is exposed before the user authentication as the platform needs to detects the smart card before asking the user for the PetPIN. If a potential vulnerability resides in the code and can be exploited before user authentication, then a code execution inside the *SMART* task could be gained. Furthermore, this task contains all the secrets needed to create a fake and backdored clone of the targeted WooKey. Those secrets are the encrypted version of the Platform Key (`PK`) (asset [A8]) and the PKBDF2's seed used to generate the Derived Key (`DK`) (asset [A5]) from the entered PetPIN (see the security target evaluation [26] for the details on the assets). We could imagine a scenario in which the attacker steals the device, runs an exploit against the ISO7816-3 stack, retrieves the described secrets and then creates a trapped clone with those secrets.

This ISO7816-3 stack is written in C language inside the standalone library `libiso7816`.[5] The token related commands are aslo implemented in C in the standalone library `libtoken`.[6] Because of the modularity of the whole project, the libraries can be compiled for any architecture and are hardware independent. This means that in order to fuzz the libraries, we only have to replace the function responsible to retrieve the data from the underlying device (the USART handling ISO7816 in this case) with a function returning characters given by the fuzzer. Since the source code is fully available, we have decided to use `libFuzzer` which is a coverage-based fuzzer.[7]

Technically, `libFuzzer` only needs a `LLVMFuzzerTestOneInput` function which takes as input parameters the fuzzed buffer's address and its size. This buffer will be returned byte by byte to the `libiso7816` through the `platform_SC_getc` function. Thanks to the source code coverage, `libFuzzer` will be able to discover new paths automatically. This path discovery can be visualized using the LLVM's source-based code coverage visualizer.[8]

This fuzzing campaign does not give us any result despite 70 % of code have been visited as shown on Figure 7.

5. https://github.com/wookey-project/libiso7816
6. https://github.com/wookey-project/libtoken
7. https://llvm.org/docs/LibFuzzer.html
8. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

Filename	Function Coverage	Line Coverage	Region Coverage
fuzzing_javacard/libecc/src/nn/nn_config.h	0.00% (0/1)	0.00% (0/5)	0.00% (0/3)
fuzzing_javacard/libecc/src/utils/utils.h	0.00% (0/1)	0.00% (0/6)	0.00% (0/1)
fuzzing_javacard/src/aes_glue.c	85.71% (6/7)	46.26% (105/227)	34.01% (50/147)
fuzzing_javacard/src/aes_soft_unmasked.c	66.67% (8/12)	54.41% (142/261)	58.23% (46/79)
fuzzing_javacard/src/fuzzing.c	100.00% (6/6)	100.00% (58/58)	100.00% (12/12)
fuzzing_javacard/src/hmac.c	100.00% (4/4)	74.07% (100/135)	77.42% (48/62)
fuzzing_javacard/src/libtoken.h	0.00% (0/2)	0.00% (0/19)	0.00% (0/2)
fuzzing_javacard/src/platform_glue.c	66.67% (10/15)	60.42% (29/48)	66.67% (10/15)
fuzzing_javacard/src/smartcard.c	50.00% (7/14)	34.35% (181/527)	40.91% (126/308)
fuzzing_javacard/src/smartcard_iso7816.c	82.00% (41/50)	79.64% (1604/2014)	82.01% (939/1145)
fuzzing_javacard/src/token.c	80.95% (17/21)	75.00% (759/1012)	79.46% (468/589)
fuzzing_javacard/src/token_dfu.c	100.00% (2/2)	90.70% (39/43)	88.89% (16/18)
Totals	**74.81% (101/135)**	**69.28% (3017/4355)**	**72.03% (1715/2381)**

Fig. 7. `libfuzzer` results on `libiso7816` and `libtoken`

9 Glitch attack on the ISO7816 library

The current section describes how a power glitch can be used to attack the `libiso7816` library. This attack allows gaining code execution in the *SMART* task which uses this library. This task hosts the platform encrypted secrets, gaining access to these secrets (even in their encrypted form) allows an attacker to build a clone. Coupled with the kernel privilege escalation described in 10, an attacker can gain privileged code execution and modify the firmware on a closed platform. Due to the security design of the WooKey project, even if these vulnerabilities may be used by an attacker to gain the highest privileges on the WooKey device, encrypted user data are not directly accessible to the attacker. Attack scenarios require the attacker to have a physical access to the device during few hours to clone or modify the device, then interact again with the legitimate user to fool him with a fake clone, and finally relay stolen secrets.

One of the tests given to the ITSEF was the analysis of the Readout Protection of the STM32F4 regarding its resistance to power glitch attacks. These tests were performed on a STM32F4-discovery board since the board has to be modified. The MCU has to be directly powered by an external power supply. To render the injected glitch pulse as narrow as possible, decoupling capacitors responsible of stabilizing the MCU power supply have to be removed. To perform the glitching campaigns, cheap hardware has been used:

— Power supply: DPS3005 (25 €)
— FPGA: Digilent Arty A7-100T (≈200 €)
— Multiplexer: MAXIM4619 (≈2 €)

The FPGA drives the STM32 reset PIN and the Multiplexer enable PIN (see Figure 8). The FPGA allows setting the delay between the MCU reset and the glitch injection and the width of the glitch pulse.

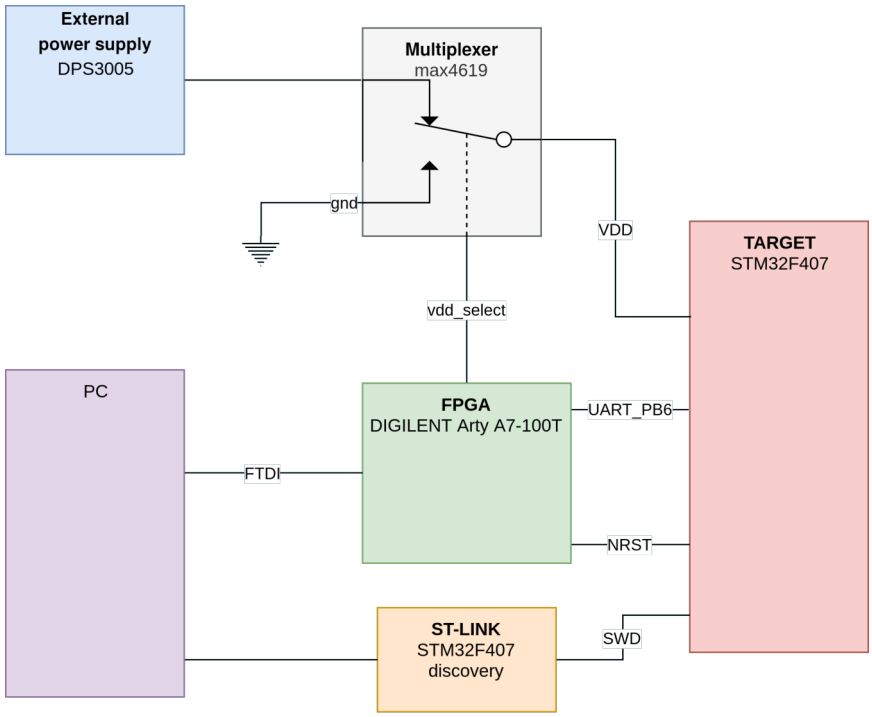

Fig. 8. Glitcher setup

The computer sends delay and width values to the FPGA, checks the results (UART logs + try JTAG), saves these results and tries another couple of delay and width values.

The Romcode and Bootloader execution time can be identified, by looking at the power supply and at the UART I/O as shown on Figure 9.

During the assessment of the Bootloader regarding its resistance to fault attacks, we observe many successful faults using power glitching. Replaying the power glitch parameters of successful glitches gives a good success rate. Unfortunately, the Readout Protection was not bypassed with the Bootloader because of its fault protection mitigation catching the attempts (see section 14 for more details on this). The reproducibility of power glitches in the Bootloader encourages however analyzing the other software components regarding this kind of faults.

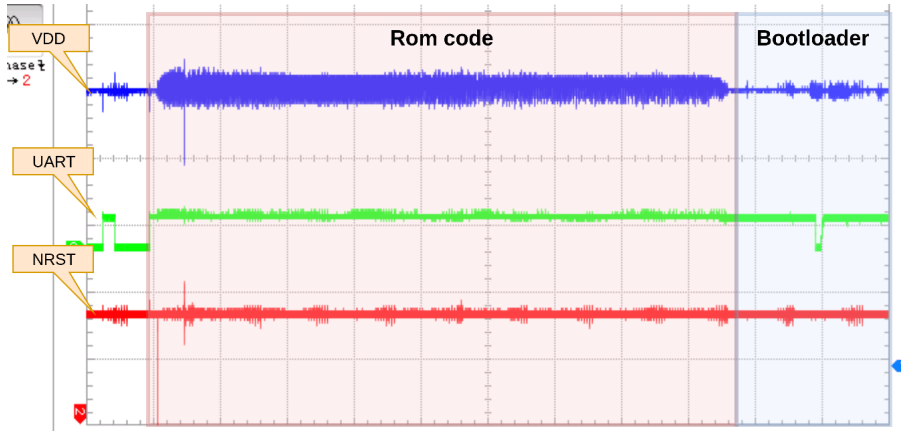

Fig. 9. Boot components timings

One of the only interfaces available on the WooKey platform before authentication is the smart card interface. The ISO7816-3 stack is implemented in the `libiso7816` component. The analysis was focused on this component, especially the part responsible for parsing incoming messages.

Unlike the Bootloader, `libiso7816` does not implement any fault attack mitigation (double checks, state automaton robustness, etc.).

The function `SC_get_ATR` is used to parse the ATR (Answer To Reset) message coming from the smart card, this message is the first message sent by the card (see Listing 4 for the code snippet).

```
int SC_get_ATR(SC_ATR *atr){
  // [...]
  /* Get the historical bytes */
  atr->h_num = atr->t0 & 0x0f;
  for(i = 0; i < atr->h_num; i++){
    if(SC_getc_timeout(&(atr->h[i]), WT_wait_time)){
      goto err;
    }
    checksum ^= atr->h[i];
  }
```

Listing 4. ATR parsing code in `libiso7816`

If a glitch is performed during the masking of the incoming size (`atr->t0`), the size may be fully controlled from the smart card interface. The variable `atr->h[i]` is a stack buffer of 16 bytes, it can be overflowed by 239 bytes.

The WooKey project implements stack cookies as a protection to exploitation of such stack overflows, but at the time of the evaluation a

typo in the build configuration prevents this protection to be applied to generated binaries "`depends on STACK_PROT_FLAGS`" should be "`depends on STACK_PROT_FLAG`" as presented on Listing 5. This issue has been fixed in recent commits after being reported.

```
config STACK_PROT_FLAG
      bool "Activate -fstack-protection-strong"
      default y
...
config STACKPROTFLAGS
      string
      default "-fstack-protector-strong"
      depends on STACK_PROT_FLAGS
```

Listing 5. Typo in the SDK that removes stack cookies

Without stack cookies, exploiting the stack overflow triggered by the glitch is highly simplified. By crafting a dedicated ATR message on the smart card interface, an attacker could gain code execution in the *SMART* task using ROP (Return Oriented Programming) gadgets as the `W⊕X` prevents data execution. To demonstrate this, a similar vulnerable code pattern is integrated to the *BLINKY* task running on STM32F4-discovery board as this development board does not provide a smart card interface and does not have the *SMART* task. The mask applied to the size is targeted in power glitch, and when we have a successful glitch (one successful fault per hour with 5 attempts per second), the Program Counter is controlled by the input (see Figure 10).

10 Kernel privilege escalation

This part describes how a kernel privilege escalation has been found inside the EwoK kernel.[9] EwoK is a secure microkernel targeting embedded systems. It is written in ADA/SPARK language, a strongly typed language often used by domains which need safe and secure software. One of the main security features of EwoK is the strict memory partitioning between tasks. Also, the tasks permissions are fixed at compile time and cannot change at runtime. Like in almost every operating system, EwoK's tasks can discuss with the kernel through *syscalls*. Those *syscalls* are the kernel's main attack surface. If a vulnerability exists inside one of them, an unprivileged task (likely already compromised) could possibly gain kernel privileges.

In order to find basic vulnerabilities inside the kernel, we run a "dumb" fuzzing campaign against EwoK's syscalls. Because EwoK is developed in

9. https://github.com/wookey-project/ewok-kernel

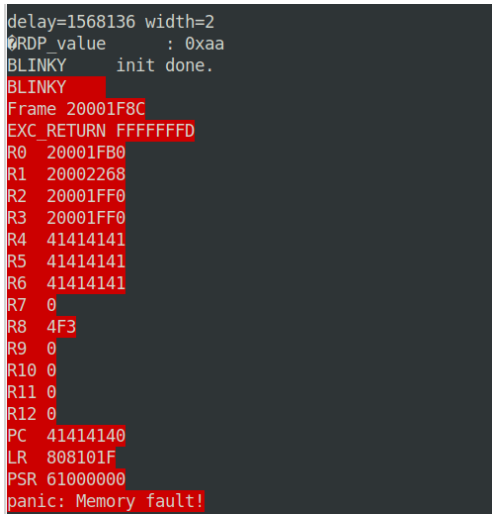

Fig. 10. Controlling the Program Counter with a voltage glitch

the ADA language and is highly dependent on the underlying hardware, running a custom task on a real platform seems to be the simplest way to fuzz the syscalls. Hence, a simple fuzzing task has been developed, with a simple algorithm:

- select a random syscall
- choose 4 arguments between:
  - 0 value
  - a valid pointer inside the task memory, containing random data
  - a random value
- fire the syscall

All the attempts are logged on the UART port. When a kernel panic occurs, it is possible to quickly see which syscall panics EwoK. This fuzzing campaign reveals some crashes and many of them are just arbitrary address dereference and are hardly exploitable.

One bug stands out though. The vulnerability resides inside the `SYS_REGISTER_DMA` syscall, which takes two parameters: `dma_config` and `descriptor`. These parameters are passed by address, it means that the kernel must check that these addresses are part of the caller's memory space. EwoK contains those sanity checks, but performs an affectation to `descriptor` in every failing case as shown on Listing 6.

```
procedure svc_register_dma
 (caller_id   : in ewok.tasks_shared.t_task_id;
  params      : in t_parameters;
```

```
  mode         : in ewok.tasks_shared.t_task_mode)
is
  dma_config   : ewok.exported.dma.t_dma_user_config
     with import, address => to_address (params(1));
  descriptor   : unsigned_32
     with import, address => to_address (params(2));
  index        : ewok.dma_shared.t_registered_dma_index;
  ok : boolean;
begin

  -- Forbidden after end of task initialization
  if is_init_done (caller_id) then
     goto ret_denied;
  end if;

  -- ...
  -- ...
  -- ...
  -- ...

<<ret_inval>>
  descriptor := 0;
  set_return_value (caller_id, mode, SYS_E_INVAL);
  ewok.tasks.set_state (caller_id, mode, TASK_STATE_RUNNABLE);
  return;

<<ret_denied>>
  descriptor := 0;
  set_return_value (caller_id, mode, SYS_E_DENIED);
  ewok.tasks.set_state (caller_id, mode, TASK_STATE_RUNNABLE);
  return;
```

Listing 6. `sys_register_dma` code

This allows a malicious task to write the `NULL` value to an arbitrary address within the kernel space. We choose to exploit this vulnerability by writing `NULL` at the `MPU_CTRL`'s address, hence deactivating the memory partitioning between tasks. Then the task is able to read and write the whole memory. Thus, privileges can be elevated by modifying the kernel's task list to become a privileged task as show on Listing 7.

```
EXPLOIT     MPU_CTRL is @ 0xe000ed94
EXPLOIT     Writing 0...
EXPLOIT     MPU should be turned off !
EXPLOIT     Looking for tasks @ 0x10000000
EXPLOIT     struct task is @ 0x100006e0
EXPLOIT     name = EXPLOIT
EXPLOIT     entry_point = 0x8090001
EXPLOIT     ttype = TASK_TYPE_USER
EXPLOIT     control = 0x3
EXPLOIT     setting to ttype = TASK_TYPE_KERNEL
EXPLOIT     control = 0x2
EXPLOIT     Privileged mode !
```

Listing 7. Privilege escalation on EwoK

11 Analysis of the address spaces of WooKey's tasks

This section presents the analysis of the address space of each task, a test proposed and executed in order to evaluate the conformity of the security function "MPU usage" described in the security target [26].

The main security properties claimed by the EwoK microkernel take root on the restricted access each task has on the resources of the system. The first property is *privilege separation*: tasks are run in unprivileged mode and should only have indirect access to the resources managed by EwoK. The second property is the *confinement* of running applications: tasks should only be able to communicate or interfere with other tasks through authorized kernel interfaces. The purpose of this test is to verify that the MPU is correctly configured and used for privilege separation and confinement.

In regard to these two security properties, the MPU management of EwoK is a critical mechanism in the WooKey platform since ARMv7-M, the architecture of the MCU, is memory-mapped: the resources (e.g RAM, Flash, system registers, peripheral registers) of the system are directly accessed through memory accesses. Consequently, an incorrect MPU configuration could grant a task an unpredicted access to some resources that could be leveraged to corrupt or access data of another task or of the kernel. In case of success, it could mean the direct disclosure of an asset stored on the platform. Data corruption can be a mean to obtain control of the execution flow or of privilege escalation, also leading in the end to the compromise of assets of WooKey.

In any case, the exploitation of an error in the MPU management corresponds to a partial attack path which assumes that the execution flow of one task of WooKey has already been hijacked. This initial compromise is typically obtained through a vulnerability in one of the protocols stacks, executed in a task context, handling one of the external interfaces of the platform.

The evaluator did not identify in the literature specific techniques or tools that can be reused to perform such tests dynamically. However, a static analysis of the binary code, which encompasses MMU aspects through a specific formal specification, targeting higher assurance through formal methods was proposed to verify more general information flow properties of kernels [36]. This work, while being relevant for the analysis of the address space, does not include many hardware aspects - typically specificities of the MCU and its architecture such as particular system

registers. This is a common limitation for static approaches applied on systems with hardware and software interactions.

For this reason, the evaluator decided to favor a dynamic approach for this test. At the beginning of the work, a small preparative code review took place to identify the cases where the MPU is reconfigured by EwoK. From this analysis, the evaluator concluded that the address space of an application is only changed at three occasions: task creation, the end of the *init* phase and the handling of a request to map a device. According to this observation and the fact that other mechanisms inducing changes of the MPU configuration are covered in other items of the test plan, the evaluator focused on the static allocation mapping that is applied during task creation.

The dynamic test implemented follows a simple approach: each application running on WooKey is recompiled to include a procedure that systematically tries read and write accesses while traversing the whole address space. If the access is not refused by the MPU, then the access is considered to be successful. Each time a new accessible memory region is identified by the procedure, a log message is sent to the debug UART of EwoK. The results are captured for all tasks of the nominal mode and analyzed to check for potential communication means through a shared region that is readable by a task and writable by another. Accessible memory regions are also manually reviewed to check for unexpected access to kernel areas.

In the evaluated version of EwoK, upon a memory access outside of the authorized regions configured via the MPU, the MPU fault handler is executed and the kernel kills the task responsible of the fault. This behavior is constraining for the implementation of access testing from user mode. A slight modification, showed in Figure 11, of the MPU Fault handler of EwoK allows to resume execution of the responsible task and to simulate the execution (according to ARM ABI [10]) of a `return 1` statement in the current function. This allows to define a simple access checking primitive following the code skeleton of Figure 12 that returns zero in case of access success and one when the access triggered a MPU fault.

The test procedure traversing the whole address space will check four bytes (aligned) and one byte memory accesses. The idea is to have at least one successful access to some memory mapped register that has specific memory access constraints. In addition, the EwoK bus fault handler is modified similarly to the MPU fault handler because this fault is triggered in the case of an unprivileged access to a system register.

10. On the condition that the current function does not use the stack.

```
   function memory_fault_handler
     (frame_a : t_stack_frame_access)
      return t_stack_frame_access
   is
   new_frame_a : t_stack_frame_access;
   begin
#if CONFIG_KERNEL_CONTINUE_AFTER_FAULT
   frame_a.R0 := 1 ; -- Emulate a "return 1;" executed by the task
   frame_a.PC := frame_a.LR ;
   return frame_a;
#else
      -- On memory fault, the task is not scheduled anymore
      ...
 #endif
   end memory_fault_handler;
```

Fig. 11. Modification of EwoK MPU fault handler

```
int32_t __attribute__ ((noinline)) _check_read_access32(volatile
    uint32_t* addr)
{
    tmp = *addr;
    // access successful
    return 0;
}
```

Fig. 12. Check access primitive

To speed up the complete process for the whole address space, the procedure only checks addresses at the start of a MPU region or subregion. In the ARMv7-M architecture, the bits 4 to 0 of the base address of the MPU region are always 0, and only regions of 256 bytes and larger can be divided equally in 8 subregions, while any MPU region or subregion starts with an address multiple of 32. Therefore, the address space can be processed optimally from address zero using a step of 32.

A first run of the test identified three possible shared regions. The three were false positives located in the Private Peripheral Bus (PPB). The PPB is located in the system register area, which is not subject to access control by the MPU. The ARMv7 reference manual describes cases where the PPB registers are accessible in unprivileged mode. To confirm or infirm the problem, these three regions are exhaustively traversed with read, then write, then read accesses and the two values read are compared. In each of these cases, the write accesses had no effects. Two regions were reserved by ARM, and the last was dedicated for the registers of the Instrumentation Trace Macrocell (ITM), a ARM debugging feature. By default the ITM registers cannot be modified in user mode. However according to ARM documentations, a specific configuration feature can enable user mode access. The final test gives some assurance that it was not the case.

As a result of the whole test, the evaluator concluded that the regions effectively accessible by each task, from creation to the end of the *init* phase, correspond to the mapping defined in the source code. This mapping gives no access to EwoK resources and isolates tasks from each other. This indicates that during the *init* phase of the tasks, the mechanisms restricting the memory accesses correctly forbid tasks access to kernel resources and preserve the two security properties: privilege separation and task confinement. Four man-days were dedicated to this test during the evaluation.

12 Analysis of ECDSA against physical attacks

In the WooKey project, the ECDSA scheme is used to authenticate both the WooKey chip and the smart card when a Secure Channel communication handshake is performed (see section 7.2 for more details on this). If an attacker is able to recover the ECDSA private key of the WooKey platform, he is able to mount a Secure Channel with the token and opens new attack vectors. Such a private key is also a first step to cloning attacks

to create fake devices, fuzz a legitimate token to find new vulnerabilities, and steal other secrets by fooling the legitimate user.

To counter this kind of threat, it is necessary to design protections against physical attacks during the execution of the ECDSA primitive. In the WooKey platform, the ECDSA implementation is provided by the `libecc` project [10], and this section focuses on the study of its protections against physical attacks. It should however be noted that these attacks are only partial as they require that the Platform Keys `PK` have already been decrypted using the dedicated key `KPK` derived from the PetPIN and the token. The attacker needs first to steal the platform, the token, and somehow guess the PetPIN (e.g. using bruteforce attacks such as the ones described in section 7.1).

The ECDSA signature algorithm is provided in Appendix A. The goal of the attacks is to recover the private key d. It is well known that, if the attacker is able to recover the ephemeral scalar k, the static private key d is easily recovered given a valid signature. In fact, as reminded in the recent Minerva attack [24], only the knowledge of a few bits of the ephemeral scalar used for several signatures is necessary to recover the static private key.

First, we describe the platform that was used to get the power consumption during the execution of atomic operations of `libecc`. Then, the core analysis of physical attacks against `libecc` is provided. This analysis leads to the discovery of a partial vulnerability, which is discussed.

12.1 Platform description for analysis

The ChipWhisperer-Lite board [3] was used for the different tests described in the next subsection, together with the STM32F303CT7 MCU. Note that the MCU used in the WooKey platform is STM32F439VIT6. The main differences are:

— A hardware AES implementation is embedded within STM32F439VIT6; this does not affect the analysis of `libecc`;
— STM32F439VIT6 operates at 180 MHz whereas STM32F303CT7 operates at 72 MHz.

The ChipWhisperer-Lite can handle up to 105 million samples per second, which is enough for the STM32F303CT7 MCU whereas it would not suffice if tests were performed with the STM32F439VIT6.

12.2 Physical attacks against ECDSA

The attacker can use different ways to recover the ephemeral scalar k (or parts of it) using physical attacks:

— during the generation of k;
— during the Elliptic Curve Scalar Multiplication (ECSM) within ECDSA signature;
— during the other operations of ECDSA signature that manipulate k in order to build the s component of the signature.

The analysis primarily focuses on the ECSM operation, for it is the operation most prone to physical attacks.[11] Many physical attacks against ECC (and particularly targeting the ECSM execution) have been published since the publication by Coron in 1999 [35]. For an overview of state-of-the-art of physical attacks and protections, one can refer to [48].

It is recalled that a new ephemeral scalar is randomly generated for each signature. This fact excludes vertical attacks such as CPA [35] and CPA on addresses [39], in particular. Therefore, only attacks requiring a single consumption trace, such as the SPA [35] and more advanced horizontal attacks, are considered.

For analyzing the protections implemented in `libecc`, the code analysis of the ECSM has been performed. In particular, the file `prj_pt_monty.c` implements the ECSM core. The code of the main loop of the ECSM is provided in Appendix B.

Regarding SPA, as seen in the source code (see Appendix B), the Double-and-Add always countermeasure is implemented: a point addition is systematically performed and the result is discarded if the current scalar bit is 0. Also, the code does not contain any branching condition depending on the current scalar bit, particularly in the function `nn_getbit`. We verified in the assembly code [12] that no optimization was made by the compiler to add branching conditions.

Because of the critical aspect of this function that directly manipulates the scalar bits, we performed measures during the execution of the `nn_getbit` function. The results are provided in Figure 13 and we concluded that the attacker is not able to distinguish between the two possible results of the function.

Then, we investigated protection against advanced horizontal physical attacks. The base point randomization - that is the randomization of the

11. The modular inversion of k and other calculations using k for the signature generation may also be prone to attacks, but their analysis did not expose weaknesses and is not described here for brevity.

12. The code has been compiled with the `-S` option.

```c
#define WORD_BITS (32)
#define WORD(A) (UINT32_C(A))

typedef uint8_t u8;

int main(void)
{

  init();

  word_t a[] = {
    0x5AC635D8, 0xAA3A93E7,
    0xB3EBBD55, 0x769886BC,
    0x651D06B0, 0xCC53B0F6,
    0x3BCE3C3E, 0x27D2604B
  };

  volatile u8 bit_value;

  // give time before
      launching
  // targetted observation
  HAL_Delay(500);

  TRIGGER_HIGH();

  bit_value = nn_getbit(a, 4);
  // bit_value = nn_getbit(a,
      5);

  TRIGGER_LOW();

  while (1);
}
```

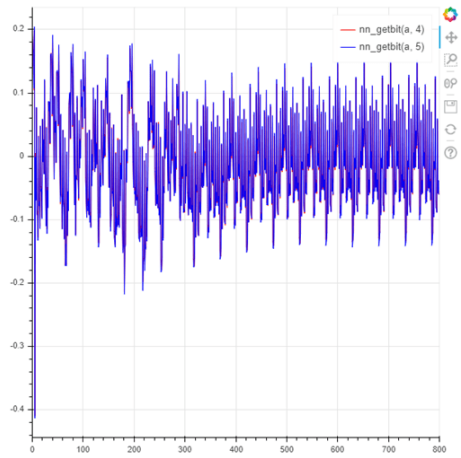

Fig. 13. `nn_getbit` - Program executed on the ChipWhisperer (left) and associated consumption traces (right) with different values of bit position (which yields different results returned by the function)

base point $(X, Y, Z) \rightarrow (lX, lY, lZ)$ with a random non-zero field element l - is implemented in `libecc`. This countermeasure thwarts the horizontal CPA [34]. Also, the random register addresses countermeasure [40] is implemented (this can be seen in Appendix B). This countermeasure prevents the horizontal address-bit DPA [44].

In addition, we analyzed the assembly code of the main loop of the ECSM *mbit*, $(rbit \oplus mbit)$ and *rbit_next* during the points copies. We isolated the few assembly instructions that have an interest, and performed measures on the ChipWhisperer. The results are provided in Figure 14 and we concluded that the attacker is not able to gain any information on the bit scalar *mbit* or on the bits of the mask r.

```
int main(void)
{

  init(); // clock, gpio, leds

  TRIGGER_HIGH();

  __asm__ __volatile__(

   "mov  r5, #400   \r\n"
   "mov  fp, #0     \r\n" // a
   "mov  r6, #0     \r\n" // b
   "eor  r6, fp, r6 \r\n"
   "muls r6, r5, r6 \r\n"
  );

  TRIGGER_LOW();


  while (1);
}
```

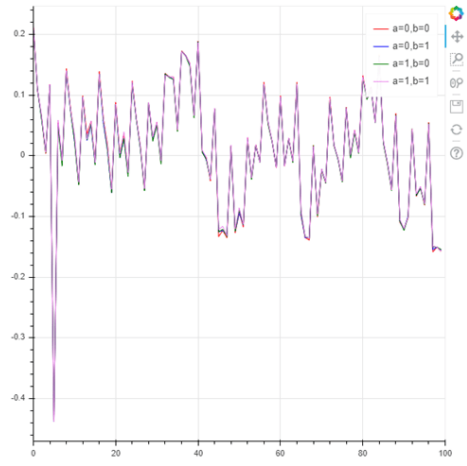

Fig. 14. XOR and MUL - Program executed on the ChipWhisperer (left) and associated observed traces (right) with different values of a and b

Another class of advanced horizontal attacks is considered: the Big Mac-like attacks. The attack, introduced by Walter in [60], consists in detecting possible repetitions of manipulated values within an ECSM.[13] Some related attacks against ECC implementation, with experimental results, were depicted in [28] (targeting a software implementation) and in [48, Sections 8.2.3.2 and 8.14] (targeting a hardware implementation).

Based on the main loop of the ECSM algorithm given in Appendix B, Figure 15 illustrates the operations of two successive iterations performed depending on the scalar bit value, with:

13. In fact, the Big Mac targets modular exponentiation implementation but applies to ECC as well.

— DBL and ADD being the illustration of elliptic curve points doubling and addition respectively; the incoming arrows are inputs and outgoing arrows are the results;
— T is the accumulative point of the ECSM;
— P is the base point of the ECSM.

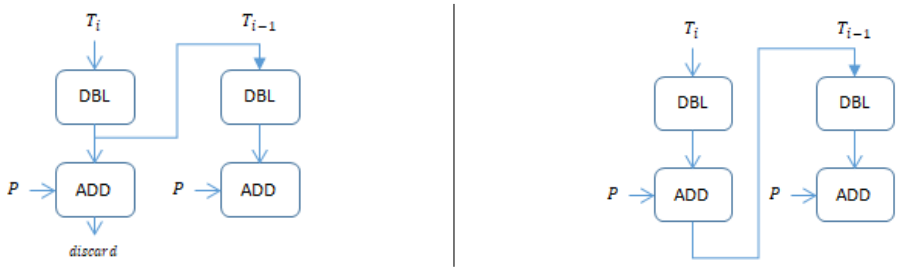

Fig. 15. Operations sequence of two iterations of ECSM if $m_i = 0$ (left) and if $m_i = 1$ (right)

Then, by comparing one input of the addition at iteration i and the input of the doubling at iteration $i - 1$, the attacker is able to deduce m_i. We analyzed the formulas used in `libecc`, in the file `prj_pt_monty.c`. The three input point coordinates are multiplied by other values, in both the doubling and addition formulas. Therefore, three Montgomery multiplications can be used for comparison by the attacker.

From the above analysis, we conclude that `libecc` is vulnerable to a horizontal collision attack.

12.3 Discussion of the exploitation of the vulnerability

Unfortunately, we did not validate the vulnerability with experimental results, due to the time consumed for the `libecc` evaluation within the inter-CESTI challenge time frame. However, we strongly believe that this attack is practical to target individual bits of the scalar k. Indeed, the success rate suggested in [28] is quite high given solely one Montgomery multiplication. Here, we have access to three Montgomery multiplications.

In the Minerva attack [24], only a very few bits per signature are necessary to recover the private key d. However, `libecc` implements a scalar randomization countermeasure, making the Minerva attack unfeasible (more specifically, exploiting the Hidden Number Problem is not possible anymore). Therefore, the attacker would have to perform the attack on all iterations to recover all the bits of k, making it more difficult.

In conclusion, the attack would be practical with a strong expertise in side-channel experimentation and many tries on a legitimate WooKey target.

13 HMAC-SHA256 SCA against the message

13.1 State of the art and attack overview

During the pre-authentication phase, the WooKey platform checks the integrity of its locally stored `EPK` (Encrypted Platform Key). To achieve this step it computes an HMAC-SHA256 over the message `(IV || Salt || EPK)`, which will be called `BigEPK` in the rest of this section. The key which is used to compute the HMAC is called `KPK` and is provided by the token. This `KPK` is intended to be correct only if the PetPIN was correct. If the attacker replaces the original token by its own token or another hardware, he can choose the `KPK` used during the HMAC computation in the platform. Since the static and fix message `BigEPK` is "mixed" with the chosen `KPK`, the adversary is able to attempt Differential Power Attacks (DPA) or Correlation Power Attacks (CPA). If he succeeds, he could know the value of `BigEPK` which is the only secret of the platform (although in encrypted form): the attacker could clone the platform in such a situation. So the attacker needs to steal the platform, execute the attack, make a clone with `BigEPK` but with modified code which allows to memorize the secret assets in internal flash for instance, put back the platform to its owner and finally make later a secondary robbery in order to retrieve all the secrets. The steps are numerous but a successful attack is powerful.

To the best of our knowledge, the HMAC and SHA-2 functions seem to have little scrutiny in the Side Channel Analysis literature. All published attacks against HMAC actually target the embedded hash function, e.g. SHA-2. A first paper was written in 2007 by McEvoy et al. [47] which mount a DPA attack with the Hamming distance model. In 2013 Belaid et al. [29] extend the attack with a leakage in the Hamming weight model. Finally in 2018 Kannwischer et al apply the same attack method as Belaid et al. but against a SHA-2-based PRNG generation for XMSS [45].

The HMAC function is defined as:

```
HMAC(m, k) = H((k xor opad) || H((k xor ipad) || m))
```

where `m` is the message, `k` is the key, and `ipad` and `opad` are fixed constants. `H` is the hash function which is SHA-2 for McEvoy's or Belaid's

studies. In the case of XMSS, the formula for PRNG generation of the i-th block is `SHA-2(0x000..03 || seed || i)`. So in all cases, the known and variable part (`m` resp. `i`) is hashed **after** the unknown fix and secret part (`k` resp. `seed`). This is not the case in WooKey: `m` (equal to `BigEPK`) is an unknown and fix secret whereas `k` (equal to `KPK`) is known and can be manipulated. When previous attacks target the secret key, our attack aims at retrieving the secret message.

Beyond the mere evaluation of WooKey's usage of HMAC, one should notice that our attack would probably be useful against the W-OTS+ hash function used in XMSS. Indeed, as precised in section 3.5 of [45], the construction of the hash function is `f_k(x)=f(0^n||k||x)` where `f` can be SHA-2. In this case, as in our attack, `k` is known and public and `x` is the secret.

13.2 Attack details

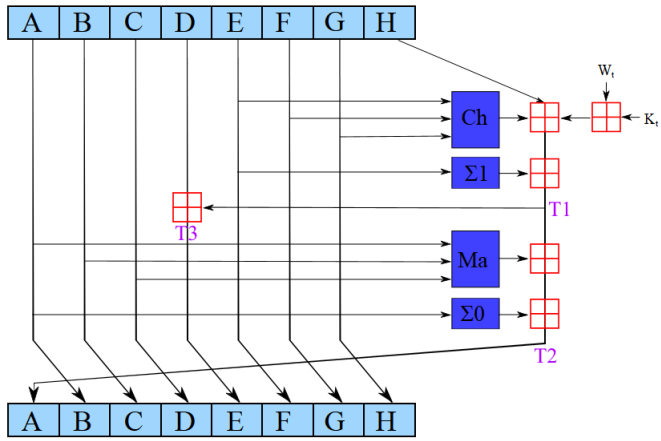

Fig. 16. SHA-2 round function (source: Wikipedia)

Our attack targets the third execution of SHA-2 in the WooKey platform. Indeed the first one computes `h1=SHA-2(KPK xor ipad)` and the second one computes `h2=SHA-2(KPK xor opad)`. The third one computes `h3=SHA-2(h1||m)`. SHA-2 round function is presented on Figure 16. All data `A` to `H`, `Wt` and `Kt` are 32 bits long. Functions in dark blue are composed of `xor`, `and`, `or` and `shift`. The addition is modulo 2^{32} (which is the normal addition on a 32-bits CPU). `Kt` is a known constant which

changes at every round. At first round Wt is the first word of the message (equal to BigEPK in WooKey context) which the attacker wants to retrieve and A to H contain the result of first SHA-2 h1: these values are known but cannot be chosen. For every execution of the HMAC, the attacker can make a guess on Wt value. As other values are known, he can compute T1, T2 and T3 intermediate results. He can then compute the Pearson correlation factor between the Hamming weight (HW) of each of these values and each measurement over time of any physical quantity. Depending on the measurement quality and as we know that no countermeasure has been implemented, the correct Wt word should be found with the guess which has the highest correlation. The attacker can then compute the next values for A to H and reproduce the attack on next round. Finally he can make the attack at every round of SHA-2 and so find the whole message. As Wt is 32 bits wide, the guess space is very large for each time sample of every trace. As a result the need for RAM memory and the computation duration are huge. In order to make it easier and quicker, we have used the same technique as previous studies so called "Partial DPA". It simply consists of considering each byte of Wt independently. At each round, the first guess is done on the least significant byte Wt[0]. So there are only 256 possibilities. It is the same for T1[0], T2[0] and T3[0]. The attacker makes then three Correlation Power Analysis (CPA) with the HW of each of these bytes. When Wt[0] has been found, he makes a guess on Wt[1] and computes T1[1], T2[1] and T3[1]. He realizes again a CPA with the HW of these bytes and finds Wt[1]. Wt[2] and Wt[3] are processed and retrieved the same way. This method could suggest that an error on a lower byte will make the attack on next byte unfeasible: we provide insights and discuss this issue in Appendix C.

13.3 Setup details and characterization on open platform

Before performing our attack on the WooKey platform, we designed a specific board on which only the STM32F439 is routed. The board contains only the minimum of decoupling capacitors. A serial resistor is inserted on the ground in order to measure the current consumed by the chip. We also extracted the HMAC function from the WooKey source code and implemented our own command manager so that we are able to easily change the KPK value. The WooKey kernel is not present and no service can interrupt the manager linear execution, but the internal hardware parameters of the STM32 is the same as on WooKey (like the frequency which is set up at 168 MHz). A GPIO has been used in order to make the synchronization easier.

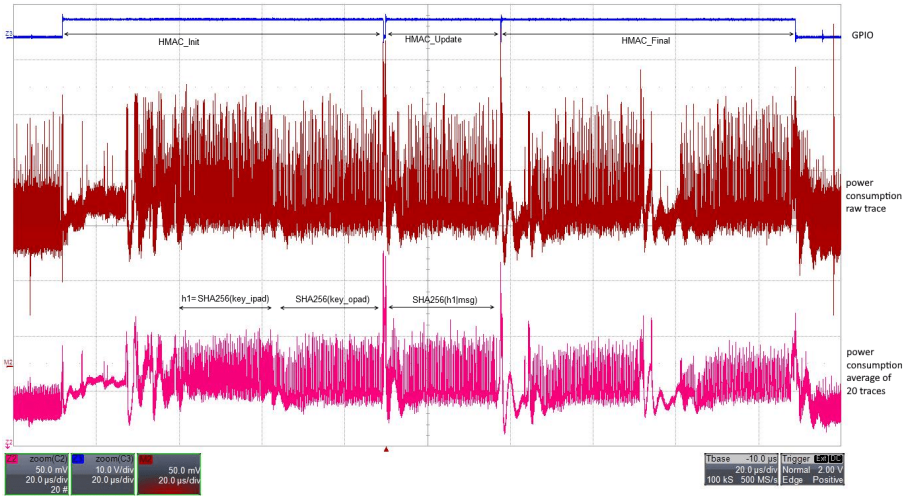

Fig. 17. HMAC execution on the STM32F439

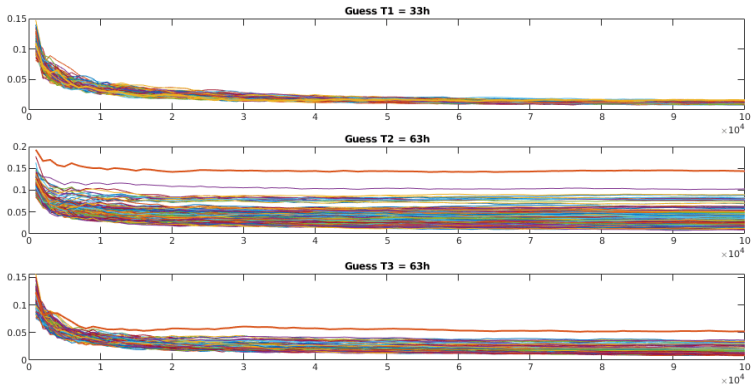

Fig. 18. CPA on HMAC evolution (first round)

With this hardware and software specific setup we probably have better conditions than the direct attack on the WooKey platform, but it will show us if and how we can fully realize the attack. The oscilloscope we used to acquire our power measurements is a high end model with large bandwidth. We did not choose it because of its capabilities which were not fully exploited but simply because it was available in the ITSEF labs. Its sampling rate was set to 500 MS/s so as to be above twice the target frequency. In these conditions, the needed amount of samples are equal to 100,000 to see the whole HMAC execution.

The Figure 17 shows the execution of the HMAC in these conditions. The blue trace is the GPIO that we added in the source code. We used this signal to trigger the beginning of the `HMAC_Update` phase. The red trace is a raw acquisition of the current. The pink trace is the raw average over 20 traces with the same key. No specific post-alignment has been done. We can see that the average has less noise than the raw one whereas the amount of information is still present over time. We could have acquired every trace and then have used them in the CPA but in this case the total transfer would have lasted longer and the disk space would have been also larger. That is why we decided to use the average traces to mount the CPA.

We realized our CPA on the beginning of `HMAC_Update` on intermediate values `T1`, `T2` and `T3`. The results on `Wt[0]` at first round are presented on Figure 18: it shows for every guess the evolution of its maximum of correlation against the amount of average traces. We can see that `T2` and `T3` find the same (and correct) value with very few average traces and that it remains stable when the amount of traces goes up. We note that `T3` is less efficient than `T2` probably due to the different amount of modular additions involved for their computation and this is the only non linear operation in SHA-2 round. Contrary to `T2` and `T3`, `T1` does not work at all even with large amounts of traces, and we cannot really explain why. This might be due to the fact that the quantity of additions is even lower for `T1`, nevertheless we would have expected it to work with higher amount of traces than `T2` and `T3`. Another hypothesis is that `T1` might not be directly computed by the HMAC whereas this variable is present in the source code: the compiler might have made an optimization and `T1` is never directly used at any assembler line. We missed time to investigate this assumption.

The attack works very well on `T2` and `T3` for other bytes and rounds. The Figure 19 shows the results for the three first rounds using `T2`. We can see that the correct values of `Wt` bytes can be found with around 1,000

averaged traces. This is particularly true after the first round. Indeed this one has lower correlation values than next rounds: this is still an open point in our results that could lead to further investigations.

13.4 Acquisitions on the WooKey platform

On the WooKey platform there are two main differences with our specific setup: first, the power consumption cannot be measured as there is no serial resistor on the ground or the `Vdd` and second, there is no synchronization GPIO. Concerning the measurement issue, the attacker could use an electromagnetic sensor but this is an additional tool which needs to be located precisely over the target. Furthermore, EM signal needs much higher sampling frequency: it means also that an expensive scope and larger amount of samples per trace would be needed. There is an easier way without modifying the WooKey platform: we measured the voltage at the `VCAP_1` pin on which the STM32F439 needs an external decoupling capacitor. It is connected to the internal voltage regulator and the goal of this capacitor is to absorb the current spikes when the core needs more power. In order to have a correct synchronization we used the ISO7816 IO signal between the token and the platform. As the attacker has to replace the original token, it means he knows the IO sequence sent by his token and he is able to synchronize on its last answered byte.

The Figure 20 shows an execution of the HMAC on the WooKey platform. Synchronization is achieved through the green IO signal. The blue signal shows the `VCAP_1` voltage and the pink one shows the same signal but after a low pass filter. We observe that every phases of the HMAC are visible on the traces. The `HMAC_Update` phase lasts three times longer as three SHA-2 are needed to hash `BigEPK`. Averaging the traces directly on the scope as we did previously is not a good idea as the IO signal is not completely synchronized. So it seems that post synchronization after the acquisition of each trace is needed. This is an additional step but it does not seem to be difficult. Another interest of traces post-alignment would be to remove traces where an interruption pattern can be observed during SHA-2 processing (due to kernel preemption and so on).

13.5 Attack quotation

The complete attack on the WooKey platform was not realized but we estimate that the nature of the observed signals should lead to the same vulnerability of the HMAC execution against SCA. However it has to be noticed that the attacker would have to replace the WooKey screen by

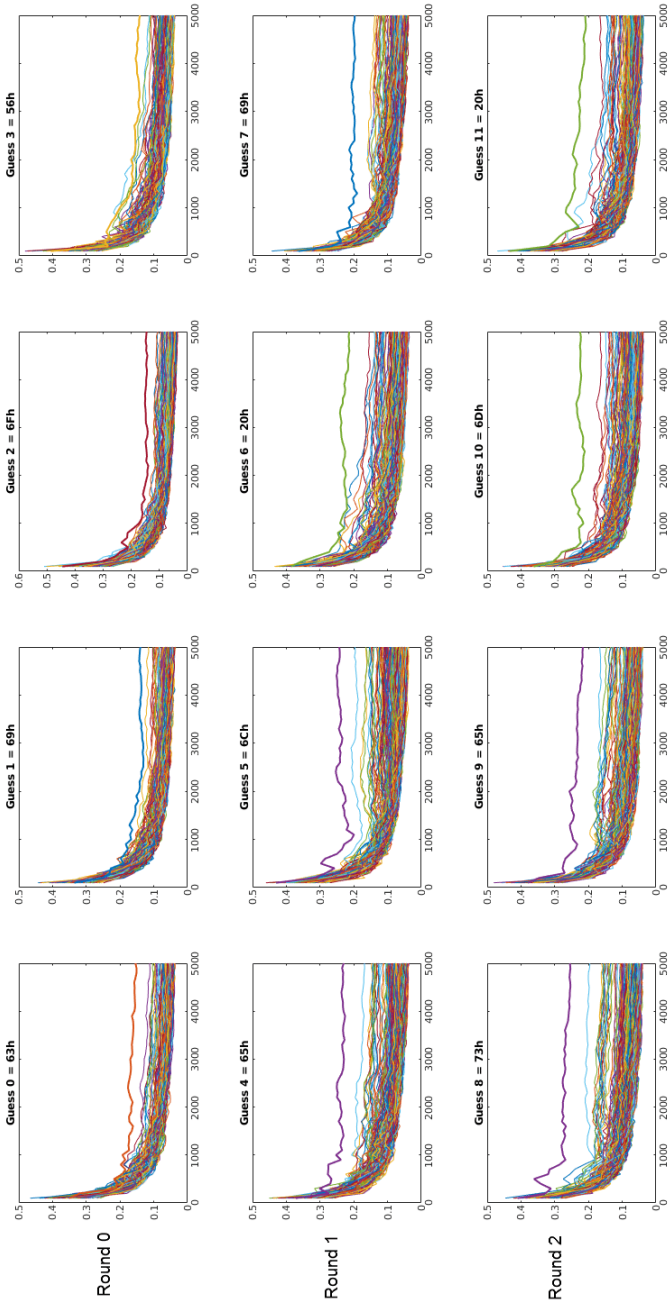

Fig. 19. CPA on HMAC evolution (three first rounds)

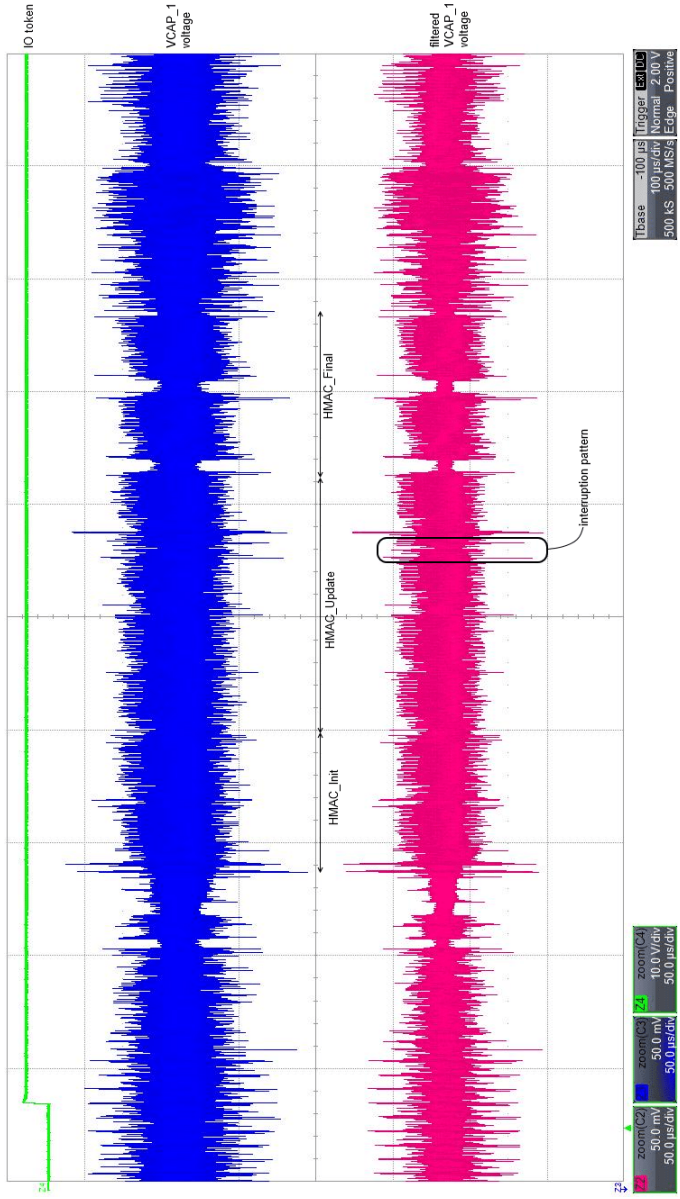

Fig. 20. HMAC execution on the WooKey platform

another SPI driven hardware so that he can repeatedly send the PetPIN with high accuracy instead of using his fingers on the touch screen (see the attack described in section 7 for details on how to perform this). The complete amount of time needed for the evaluation of the HMAC function was 19 days, including 2.5 days for report writing. The used equipment is a high end digital scope which is expensive. As we do not need its full specifications like the high bandwidth for this product, it would be interesting to see if the attack is still doable with cheaper acquisition tools like ChipWhisperer [3] or PicoScopes [12], which is left as future work.

14 Voltage fault injection attack on Readout Protection

Like most microcontrollers with integrated flash, the MCU embedded in WooKey offers a protection against firmware readout or tampering. This feature allows to protect sensitive assets, like cryptographic keys, in confidentiality and integrity. In addition, the WooKey Bootloader performs a verification to enforce the activation of this protection in a dedicated state denoted `RDP_CHECK`, as shown on Figure 21.

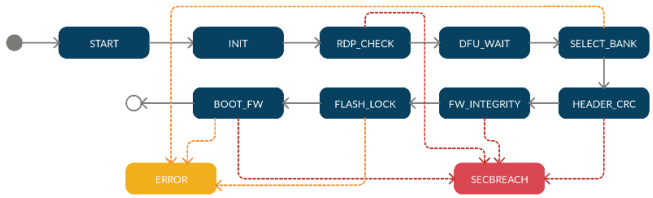

Fig. 21. Bootloader state automaton, initial implementation

Readout protections, implemented in most MCUs, are known to be weak and prone to fault injection attacks. The main goal of the attack detailed hereafter is to dump `EPK`, the encrypted platform keys used to ensure the authenticity of the WooKey board during user unlocking. Indeed, the dumped firmware (or only `EPK` as the firmware is public) can be used to build a malicious firmware which will be injected back into the WooKey board or on a clone of it, and then steal user secrets by deceiving him. In the remainder of this section, a two-step attack is considered: first, disabling the readout protection to gain access to the JTAG interface and then bypassing the software verification implemented in the Bootloader. In case of success, assets can be dumped to build a malicious device. To

perform this attack, the WooKey board needs to be stolen and trapped, returned to its owner and eventually stolen again.

14.1 STM32 RDP attack state of the art

The STM32 series features a security function for JTAG and memory lock called RDP (Readout Protection). There are 3 different protection levels:

- Level 0: no read protection. RDP option byte is set to `0xAA`.
- Level 1: no access to flash memory or backup SRAM can be performed once a debug probe is connected or while booting from SRAM or system memory bootloader (the bootROM). This protection level is not permanent and can be reverted by rewriting the option bytes. Downgrade to level 0 causes the flash memory and backup SRAM to be mass-erased. In order to activate the level 1 protection, any value (except `0xAA` or `0xCC`) has to be set in the RDP option byte.
- Level 2: in this level, all protections provided by level 1 are active. Additionally, booting from SRAM or system memory is no longer possible. JTAG interface is also disabled. Setting the RDP level to level 2 is irreversible because, in this mode of operation, option bytes can no longer be changed. In order to activate the level 2 protection, `0xCC` value has to be written into RDP option byte.

In [32], the RDP level 2 has been attacked with voltage glitch fault injection allowing a downgrade to RDP level 1. Downgrade to RDP level 0 by modifying the value using glitch fault injection is found not possible because of the required precise bit manipulation. In this paper, the attack is performed on a STM32F3 during the power-up phase. The main difference between STM32F3 and STM32F4 (used by the WooKey product) is the duplication of the RDP value in flash memory. However, this work demonstrates that this additional protection does not protect against RDP downgrade.

RDP level verification To fully validate the attack path, the verification of the RDP level performed by the WooKey Bootloader must be weak against fault injection. A source code analysis of this mechanism is therefore carried out. The Listing 8 illustrates the corresponding piece of code. A successful fault attack would cause the execution flow to go through `FLASH_RDP_CHIPPROTECT` case. Considering that the normal case when no fault is injected is the `FLASH_RDP_MEMPROTECT` case, a double

fault injection is needed: one to bypass the `FLASH_RDP_MEMPROTECT` case and another to enter the `FLASH_RDP_CHIPPROTECT` case. Additionally, the decompiled assembly (using Ghidra [6]) is analyzed to ensure that no optimization performed by the compiler could lead to a single fault injection.

```
 static loader_request_t loader_exec_req_rdpcheck(loader_state_t
     nextstate)
{
    /* entering RDPCHECK */
    loader_set_state(nextstate);
    /* default next req */
    loader_request_t nextreq = LOADER_REQ_SECBREACH;
#if CONFIG_LOADER_FLASH_RDP_CHECK
    /* RDP check */
    switch (flash_check_rdpstate()) {
        case FLASH_RDP_DEACTIVATED:
            goto err;
        case FLASH_RDP_MEMPROTECT:
            goto err;
        case FLASH_RDP_CHIPPROTECT:
            dbg_log("Flash is fully protected\n");
            dbg_flush();
            /* valid behavior */
            nextreq = LOADER_REQ_DFUCHECK;
            break;
        default:
            break;
    }
#else
    nextreq = LOADER_REQ_DFUCHECK;
#endif
    return nextreq;
}
```

Listing 8. RDP check extracted from the attacked WooKey Bootloader

```
loader_request_t loader_exec_req_rdpcheck(loader_state_t nextstate)
{
  t_flash_rdp_state tVar1;
  loader_request_t nextreq;
  loader_set_state(nextstate);
  nextreq = LOADER_REQ_SECBREACH;
  tVar1 = flash_check_rdpstate();
  if (tVar1 == FLASH_RDP_DEACTIVATED) {
    NVIC_SystemReset();
    do {
                    /* WARNING: Do nothing block with infinite loop
                       */
    } while( true );
  }
  if (tVar1 == FLASH_RDP_CHIPPROTECT) {
    dbg_log("Flash is fully protected\n");
    dbg_flush();
```

```
    nextreq = LOADER_REQ_DFUCHECK;
  }
  else {
    if (tVar1 == FLASH_RDP_MEMPROTECT) {
      NVIC_SystemReset();
      do {
                    /* WARNING: Do nothing block with infinite loop
                       */
      } while( true );
    }
  }
  return nextreq;
}
```

Listing 9. Compiled code of the RDP check function

Reverse engineering of the binary presented on Listing 9 shows that the `FLASH_RDP_CHIPPROTECT` case is actually handled before the `FLASH_RDP_MEMPROTECT` case leading to the exploitation of the weak verification with only a single fault injection.

14.2 Setup of the attack

To ease the fault injection setup and avoid chip replacement on WooKey open platform (due to a potential destruction of the chip), these tests are performed on a STM32F439 chip placed in a custom board with TQFP100 socket. As the `BYPASS_REG` pin in not accessible on TQFP100 package, the voltage regulator cannot be deactivated. Thus, injecting voltage glitch through `Vdd` power supply is less efficient. However, `Vcap` pin is accessible allowing to inject glitches directly on the CPU power supply, after the voltage regulator (see Figure 22 left). To inject voltage glitches, a PMOS transistor is placed in parallel of the capacitor of one of the two `Vcap` pins (see Figure 22 right). The PMOS transistor is chosen to allow fast switching ($t_{RISE} + t_{FALL} < 20$ ns).

14.3 Test description

To sum up, a double fault injection attack scenario is found possible: one fault on the STM32F4 core boot sequence for the RDP level downgrade (to RDP level 1), and another fault on WooKey Bootloader to bypass the RDP level verification. In case of success, the JTAG probe is connected during the firmware integrity check operation which goes through the whole firmware to compute its SHA-256 hash. Connecting the JTAG probe in RDP level 1 halts immediately the STM32 core allowing to dump the part of code which is being hashed. Repeating the attack, by incrementing

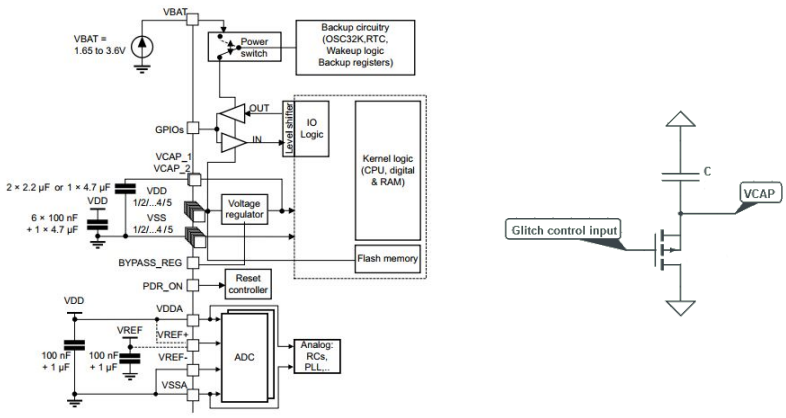

Fig. 22. Power supply on STM32F4 (left) - Fault injection setup (right)

the timing where the JTAG probe is connected, allows to retrieve the whole firmware. Knowing the exact position of the `EPK` key in the firmware can accelerate the attack, requiring only few iterations of the attack. We describe each partial attack individually hereafter before presenting the full attack path exploitation.

RDP level downgrade First, a signal analysis is performed to find the attack timing where the glitch has to be injected. The MCU power consumption is recorded at the start-up of the chip, before the execution of the user's firmware.

As the boot process is targeted in this attack, the chip needs to be power cycled at each iteration. A programmable power supply unit is used to do that. The JTAG probe, controlled by a python script, is used to verify if the attack succeeded. After each fault injection, the JTAG probe tries to read the SRAM. If the probe cannot be connected to the ARM core, the attack did not succeed.

Figure 23 highlights a fault injection timing where exploitable faults were obtained. In this timing window, several timings are found where a downgrade to RDP level 1 is possible. A statistical evaluation over 20,000 runs allows to identify the best one with the highest success rate. Finally, downgrading to RDP level 1 using glitch fault injection is found possible with a success rate of 1.5 %.

RDP level verification bypass A fault injection on the RDP level verification mechanism is then performed. A chip is configured in RDP

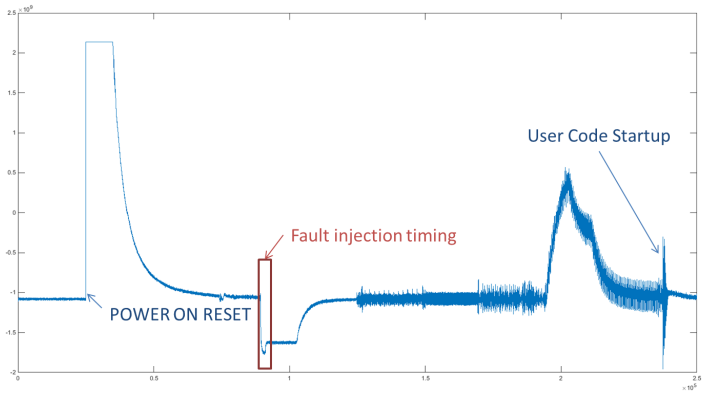

Fig. 23. Signal analysis of STM32 boot

level 1 to emulate a success of the first fault injection. To ease the signal analysis, a GPIO is raised before the execution of the RDP level verification. According to the code analysis, the chip restarts when the actual RDP

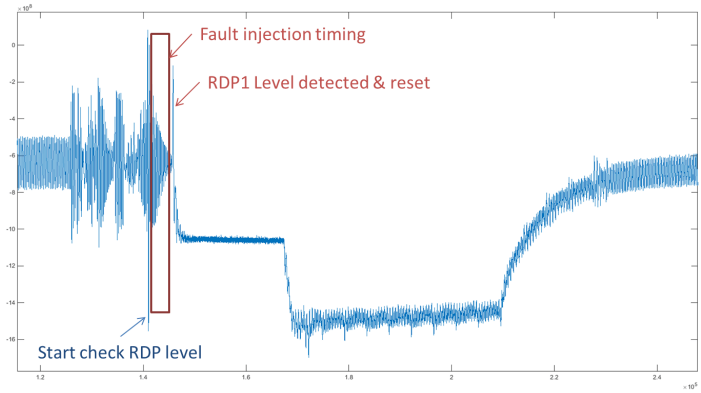

Fig. 24. Signal analysis of RDP level verification mechanism

level does not match the expected one. This is used to identify the end of the RDP level verification (see Figure 24).

Then, the window identified during the signal analysis is scanned using glitches until the right glitch parameters (i.e. the parameters for which the chip does not restart) are found. Finally, a timing is identified where

the reset did not occur and the firmware continues its execution. After an optimization of the fault injection parameters, the success rate for this attack is around 10 %.

Full attack path Finally, the full attack has been tested. Figure 25 shows the fault injection timing for each pulse. Without fault injection, the firmware continues its execution due to RDP level 2 protection.

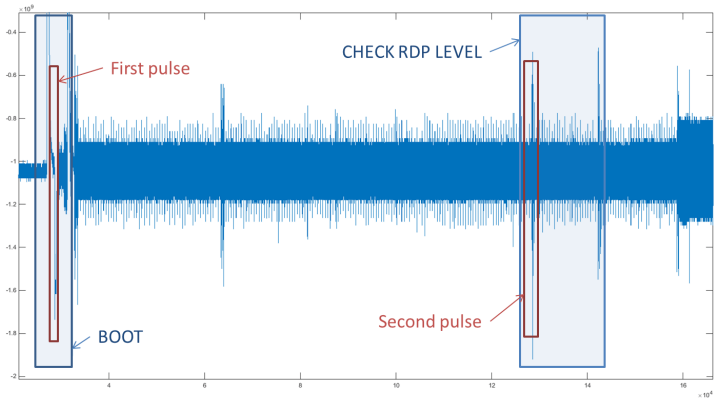

Fig. 25. Signal analysis of both boot and RDP verification

The firmware being public, a simulation of its execution is performed allowing to find the area in SRAM used for firmware manipulation during the firmware integrity check. Thus, only the SRAM area identified has to be dumped to extract the product firmware.

Then, JTAG probe is connected 2.1 seconds after the chip start-up, corresponding to the timing where firmware check is performed. This limitation did not allow to perform as many fault injections as for the partial attacks. Despite of this, combining the two fault injections succeed with a success rate of 0.1 % in means: 16 bytes of the firmware can be dumped and verified using the `HEX` file corresponding to the loaded firmware.

14.4 RDP level verification improvement

A new code version has been deployed to fix the exploitable vulnerability described previously. This time, the RDP level verification is performed

several times during the boot process which significantly complicates the attack path. This is done through a modification of the Bootloader state automaton in which the `RDP_CHECK` state is executed between each other state, as shown on Figure 26.

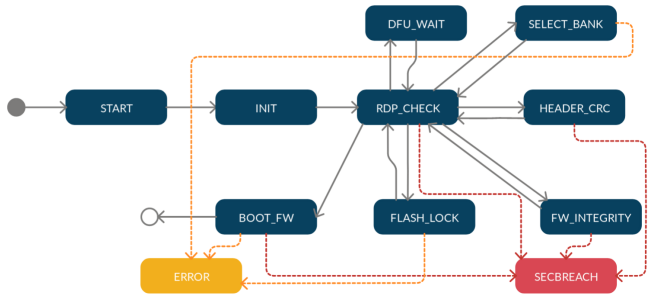

Fig. 26. Bootloader state automaton, new implementation

An analysis of the assembly code highlights that bypassing the normal process flow of the loader by jumping directly into the integrity check step requires a total of 5 pulses with 4 pulses produced in few CPU cycles. Furthermore, altering the process flow and jumping to another function results in a security breach detection and a mass flash erase which invalidates this attack path. This analysis has been confirmed through simulations on the WooKey firmware.

Therefore, another approach is adopted. The goal is to downgrade to RDP level 1, to load a crafted payload in SRAM and to try to perform malicious operations through it. Actually, the STM32 documentation states that the flash memory is unreachable when a JTAG probe is connected or when booting in SRAM if RDP level is above 0. To ensure that the flash is really fully disconnected, a code which jumps to a given flash memory address is loaded in SRAM. The execution works well when RDP level is set to 0 but a hard fault interruption occurs when RDP level is set to 1. Modifying the `VTOR` register to relocate the interrupt vector table in SRAM results in triggering nested hard faults. Considering that the flash memory is really not accessible, this attack path is hence found not exploitable.

14.5 Conclusion

This attack shows that both the hardware readout protection mechanism and the corresponding software check of WooKey were initially

vulnerable allowing to dump the firmware. Glitch fault injection method has the advantage to be a low cost attack, easy to setup for an attacker. The complexity of this attack lies in the precision required for the fault injection timing in order to optimize the double pulse success rate. Indeed, dumping the whole firmware or only some secret keys require to reproduce this attack multiple times.

Finally, this vulnerability is no longer exploitable on the last version of the WooKey firmware. The work done to correct this vulnerability shows that even if the hardware itself is still vulnerable, software solutions exist to overcome (or at least limit) this weakness.

15 EM fault injection attacks on the Bootloader

Electromagnetic (EM) based fault injections have been experimented on the WooKey platform, and more specifically against the Bootloader. This kind of attacks has become affordable and relatively easy to setup, notably thanks to the ChipSHOUTER platform [2]. However, a first necessary step to achieve a working fault injection bench is to have an XYZ table. A cheap yet efficient solution is to use a 3D printer or a CNC driven with *gcode* based scripts. The bench is also completed with an external trigger in order to achieve a better time resolution in the delay programming, as well as for the pulse width. All these elements, as well as the target MCU, are driven using Python scripts and four UARTs.

The first step is to characterize the injection coils that are the main EM pulse source on the MCU surface. All the possible pulse widths are not achievable, and it is necessary to observe the voltage and the current generated by the pulses using an oscilloscope plugged to the dedicated SMAs on the ChipSHOUTER. As a matter of fact, the original coil head of 1 mm is only able to produce significant pulses with widths between 20 and 40 ns, with a global width of 60 to 100 ns.

An external trigger has been developed in Verilog on an ICE40 FPGA and running at 240 MHz. Its basic time unit is consequently 4.2 ns. In order to get a 28-bit counter and achieve programmable delays up to 1 second, a raw adder cannot be used since there is not enough time to propagate carries along 28 bits in 4.2 ns. The trick consists in using a 28-bit LFSR (Linear Feedback Shit Register), at the expense of more computations for the initial state depending on the expected number of cycles. The delay and the pulse width are programmable using an UART synthesized inside the FPGA, and a 200 ns delay is added after the pulse in order to avoid the noise generated by the ChipSHOUTER (this can

create a chain reaction since the pulse could be interpreted as a trigger, generating new pulses and so on).

In order to understand the pulses effects on the target MCU, tests have been conducted using a simple unrolled loop with two interlaced counters, and observing their states after the fault injection. As opposed to voltage glitches, EM fault injections have many setup parameters to explore: the coil choice, the coil direction (and for horseshoe coils their angle), the coil XYZ positions in space, the pulse delay, the injection voltage, the pulse repeat factor, and the pulse shape when it is controllable.

When fixing some of the parameters, it is possible to perform a cartography of the pulse effects depending on the other variable parameters (e.g. the position of the coil in the XY plan). An example of such a cartography is provided on Figures 27 and 28. The blue colored zones represent hangs of the STM32, the purple ones represent reboots, and the yellow/orange/red capture cases where one to hundreds instructions have been skipped.

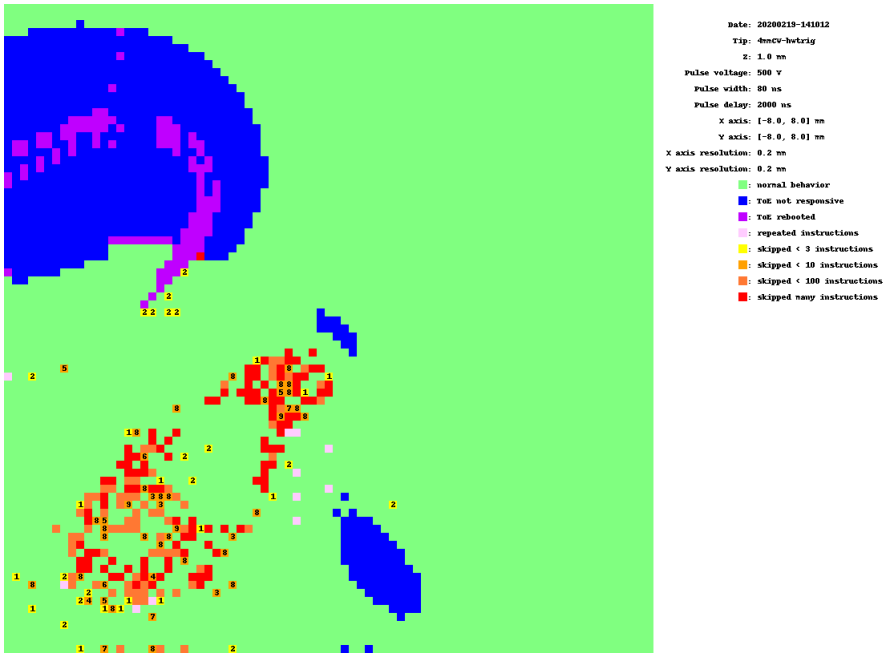

Fig. 27. EM fault injection cartography example

Some variations of the setup have been tested during the evaluation time without finding large zones with enough interesting effects and repeatability: this yields in a probabilistic process and results.

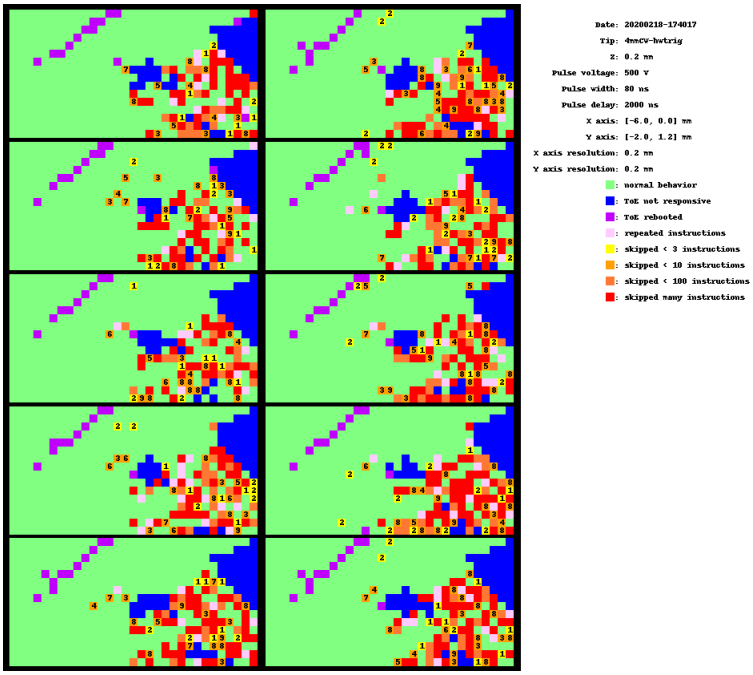

Fig. 28. Repeatability study of a fault on a zone example

Then, the tests have been performed on WooKey's Bootloader code, and more specifically around the `loader_exec_req_integritycheck` function that handles the firmware integrity: a manual source review shows that the integrity test result is not doubled against fault attacks (contrary, for example, to meta-data CRC32 check) as we can see on Listing 10.

```
if (check_fw_hash(ctx.fw, part_addr, part_size) != sectrue)
{
    dbg_log("Error while checking firmware integrity! Leaving \n");
    dbg_flush();
    goto err;
}
```

Listing 10. Firmware integrity test

It should be however noted on the produced assembly code (see Figure 29) that error handling immediately follows the tests, hence removing this test using one fault will not be enough. Faulting the `jump` to the `err` label should however do the trick.

We can observe that it is complex to protect the code against faults that skip one instruction. The compiled assembly code must be checked, and as

```
.text:08000C58 loc_8000C58                             ; CODE XREF: loader_exec_req_integritycheck+24↑j
.text:08000C58                 LDR     R3, =(ctx - 0x8000C5E)
.text:08000C5A                 ADD     R3, PC          ; ctx
.text:08000C5C                 LDR     R3, [R3,#(ctx.fw - 0x20000010)]
.text:08000C5E                 LDR     R2, [R7,#0x10+partition_size] ; partition_size
.text:08000C60                 LDR     R1, [R7,#0x10+partition_addr] ; partition_base_addr
.text:08000C62                 MOV     R0, R3          ; fw
.text:08000C64                 BL      check_fw_hash
.text:08000C68                 MOV     R2, R0
.text:08000C6A                 LDR     R3, =0xAA55AA55
.text:08000C6C                 CMP     R2, R3
.text:08000C6E                 BEQ     loc_8000C80
.text:08000C70                 LDR     R3, =(a41merrorWhileC - 0x8000C76)
.text:08000C72                 ADD     R3, PC          ; "\x1B[41mError while checking firmware i"...
.text:08000C74                 MOV     R0, R3          ; fmt
.text:08000C76                 BL      dbg_log
.text:08000C7A                 BL      dbg_flush
.text:08000C7E                 B       err
.text:08000C80 ; ---------------------------------------------------------------------------
.text:08000C80
.text:08000C80 loc_8000C80                             ; CODE XREF: loader_exec_req_integritycheck+5E↑j
.text:08000C80                 LDR     R3, =0x500000C
.text:08000C82                 B       loc_8000C88
.text:08000C84 ; ---------------------------------------------------------------------------
.text:08000C84
.text:08000C84 loc_8000C84                             ; CODE XREF: loader_exec_req_integritycheck+30↑j
.text:08000C84                                         ; loader_exec_req_integritycheck+3C↑j
.text:08000C84                 NOP
.text:08000C86
.text:08000C86 err                                     ; CODE XREF: loader_exec_req_integritycheck+6E↑j
.text:08000C86                 LDR     R3, =0xFF35CFCF
.text:08000C88
.text:08000C88 loc_8000C88                             ; CODE XREF: loader_exec_req_integritycheck+72↑j
.text:08000C88                 MOV     R0, R3
.text:08000C8A                 ADDS    R7, #0x10
.text:08000C8C                 MOV     SP, R7
.text:08000C8E                 POP     {R7,PC}
.text:08000C8E ; End of function loader_exec_req_integritycheck
```

Fig. 29. Compiled assembly code for `loader_exec_req_integritycheck`

a matter of fact most of the faults we have obtained usually skip more than one instruction, or have other effects. In order to perform the tests without damaging the WooKey platform, we have ported the Bootlader code on a NUCLEO-F439ZI board with minor modifications to fit to this slightly different platform. Flash writing functions have also been removed since they trigger a mass erase whenever a fault is detected. Printed messages on the UART have also been added in order to follow the Bootlader state, a LED is turned on just before the targeted integrity check, and the system clock is kept at 16 MHz (the WooKey platform reconfigures it to 168 MHz). In the integrity check code, the hash function computation is completely removed to gain time during the tests – since its resulted hash value would be incorrect and the purpose of the pulse is to skip it anyways. The compiled code is compared to the one from the WooKey binary: in order to get the same result a `-O0` compilation flag must be used to turn off optimizations. From `loader_exec_req_integritycheck` to the result processing, a hundred of microseconds are necessary (i.e. 1600 cycles at 16 MHz).

A first test campaign is performed with pulses of 400 V and 80 ns width on some of the yellow-orange zones of the cartography presented on Figure 28 with random delays between 0 and 100 μs, with a 80 tests per minute rate. A first observation, rather unexpected, is that we regularly

see large portions of the firmware dumped on the UART. Since the UART is disabled on the production WooKey board, this is not a relevant attack path. Nonetheless, similar results could be obtained when attacking the USB enumeration as it has been demonstrated by Micah Elizabeth Scott in PoC||GTFO [54] on another hardware platform. After 500 attempts, a first firmware integrity check bypass can be successfully observed as shown on Listing 11.

```
Next state: REQ_INTEGRITYCHECK.
Locking flash write
^[[7mBooting FLIP in nominal mode
^[[0mJumping to FW mode: 8020189
Next state: REQ_RDPCHECK
RDP0, Flash is readable
Next state: REQ_BOOT
Geronimo !
```

Listing 11. Firmware integrity check bypass UART log

The UART message seems to advocate for a complete function call bypass rather than an integrity check test bypass. From here, adjusting the parameters allows to get a 7 % success rate. With more time and tuning, a better success rate and other fault injection positions might be obtained.

The conclusions of these experiments is that a firmware integrity check bypass is possible although being hard to exploit: the attacker must find a way to inject a corrupted firmware in the internal flash, then have a successful pulse on a platform running at a ten times frequency clock, and avoid being detected by the Bootlader to prevent a mass erase. As a matter of fact, and in order to achieve a better robustness against faults, the Bootlader code could benefit from more elaborate CFI (Control Flow-Integrity) checks.

A second EM fault injection attack that has been explored is the RDP2 to RDP1 downgrade (similar to what has been obtained with voltage glitches in section 14). In order to be as close as possible to the experimental voltage glitch setup on the STM32F3 of [32], the NUCLEO-F439ZI board has been configured in RDP2 and adapted so that a reset instruction sent to the embedded ST-Link chip triggers a power cut-off, yielding a Power-On-Reset on the target MCU. Observing the `NRST` SWD pin allows to have a synchronization signal as close as possible to this target. Contrary to the case where we attack a chosen code running on the MCU (such as WooKey's Bootlader code), attacking the RDP2 check is performed "blindly": there is no debug feature and feedback that allow

to know whether the fault parameters are more or less successful. Having only a binary result (RDP2 bypassed or not) is hence more challenging.

Since the article [32] exhibits a successful attack with a 11 μs delay, we have covered random delays from 0 to 20 μs with a 4 μs resolution in our test campaigns. The complete MCU surface (100 mm^2) have been covered with random positions and 400 V/80 ns width pulses, using a custom hand made horseshoe coil, and resulting in 250 tests per minute. In order to quickly check the success of the attack, we try to connect to the MCU through JTAG. A first campaign of 150,000 tests has unfortunately provided no interesting result. Consequently, the explored surface has then been expanded to 200 mm^2, and the coil switched to the 4 mm CCW one provided with the ChipSHOUTER. The target MCU has sadly died, becoming unresponsive, during this second campaign of 360,000 tests. This ended our experiments on the RDP2 to RDP1 downgrade using EM based fault injections.

16 WooKey's Bootloader: a formal analysis approach

WooKey's Bootloader is a critical piece of code that cannot be upgraded: it must therefore be free of security issues. This includes the absence of run-time errors (RTE), the respect of functional properties and the resistance to fault injection attacks (FIA).

Even when the source code is available (white box evaluation), it is still a challenging task to find vulnerabilities especially in the context of a time-limited code audit (four days were allocated to this analysis during the challenge). Therefore, an efficient methodology needs to be applied leveraging the best of human understanding and automated static analysis. The purpose of the current section is to provide an insight of applying such a methodology to the Bootloader.[14]

16.1 The methodology

Fully automated analysis may work very well to detect undefined behaviors or some CWE (Common Weakness Enumeration) registered weaknesses, but the final verdict regarding security remains a (subjective) human decision. One way to efficiently achieve this difficult task is to manually browse the code while being assisted by generic tools that can

14. The Bootloader is 10 kloc, but such a methodology can be applied to more complex projects of hundreds of kloc and more entry points.

be configured and customized to help checking properties and obtaining certainty about facts.

A key aspect is time, though, and the evaluator needs to efficiently obtain results even from a subset of the code. Partial analysis is a consequence of this time constraint, and might also be a necessary approach when dealing with precise analysis techniques that do not scale well with the code size. A way to simplify the analysis is to follow the modular structure of the code base (sometimes with cross-modules analysis paths).

Another consequence of the time constraint is to prevent from manually annotating the code to express a formal specification (properties, contracts) to be verified by deductive verification, for example with Frama-C WP [5].

However, a middle approach consists in focusing on the global properties that have to be verified across multiple functions, avoiding the complexity of writing contracts for every function. This approach has only been very lightly applied during the WooKey challenge by specifying very simple properties based on assertions (more elaborated global properties can be specified as presented in [53]).

The technique called "value analysis" implemented for example by Frama-C Eva [15] may prove, without additional annotations, the lack of erroneous state violating global properties by abstract interpretation. However, over-approximation may lead to uncertainty: warnings can correspond to real erroneous states or just be false alarms. Disambiguation can be performed by finding concrete paths that reach the erroneous state. Finding such paths could be done manually for a very simple code. Another approach makes use of precise analysis techniques called Dynamic Symbolic Execution (DSE) [50]: they automatically search path conditions or a given oracle, i.e. the property to violate. If all the paths existing in the code can be covered then the search is both sound (no missing state) and precise (no false alarm): this so called all-path coverage is usually not achievable in complex code due to path explosion.

Properties can also be violated as a result of paths perturbed by fault injection (FIA). These paths can be found manually by modifying the source code to simulate faults, or automatically with a DSE based tool called Lazart that simulates multiple fault injections with several fault models as explained in [51].

In the following sections, the evaluator follows a 2-step methodology that consists in using the Frama-C platform [55] to understand the behavior of the Bootloader, and define some functional properties to be either verified or violated by counter-examples also called attack paths.

16.2 Understanding the behavior of the Bootloader

No precise documentation of the Bootoader is given by the WooKey project [25, 30]. But the implementation review in the code is always a valuable source of information, as well as the compiled binary (through decompilation using Ghidra [6], see section 16.4).

A value analysis with Frama-C Eva always starts by listing the available entry points of the module to be analyzed (the roots of the callgraph). The Bootloader has several entry points: the `main` function and interruption handlers. This analysis focuses on `main` launching the Bootloader automaton in charge of booting the firmware. The perimeter of the partial analysis is composed of a subset of the implementation (`.c` files) and all the required include files (`.h`). When only the prototype of a function is provided (the implementation is missing), the Frama-C kernel automatically generates a minimal specification expressed in ACSL. Such a contract respects the over-approximation (soundness) unless some global variables are modified by this function. Therefore, the perimeter of a partial analysis needs to be large enough to include all the side effects that could have an impact on the analyzed behavior. The analysis usually begins with a small subset of the implementation and more content is added if the understanding of the behavior shows that some important dependencies are missing. Only a subset of the source files located in the directory `loader` of the project have been selected to start the analysis, in particular the file `main.c` that implements the Bootloader automaton. The initial state of the Bootloader includes its context (e.g. DFU mode), the firmware area called SHR, and some registers like the RDP state. The Bootloader context is assigned with precise values (the initial values), whereas the fields of SHR and registers are imprecise, i.e. assigned with the largest interval depending on their type.

After having launched the value analysis with Eva, the evaluator checks the results by directly browsing the source code with Frama-C GUI. A value analysis computes intervals for all the variables (fixed point computation), and separately propagates several states at each statement (trace partitioning) in particular to precisely unroll loops (for a bounded number of iterations). The Bootloader automaton is made of an infinite loop that dispatches requests to functions that are in charge of the state transitions, for example checking the integrity of the firmware, and booting (which is the last transition of the automaton). Syntactic unrolling allows to duplicate the code for each loop iteration and visualize in Frama-C GUI the nominal sequence of the Bootloader as represented in Table 4.

Current state	Request	Next state
START	REQ_INIT	INIT
INIT	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_DFUCHECK	DFUWAIT
DFUWAIT	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_SELECTBANK	SELECTBANK
SELECTBANK	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_CRCCHECK	HDRCRC
HDRCRC	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_INTEGRITYCHECK	FWINTEGRITY
FWINTEGRITY	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_FLASHLOCK	FLASHLOCK
FLASHLOCK	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_BOOT	BOOTFW

Table 4. Nominal sequence of the Bootloader as inferred by Frama-C

The redundancy of RDP check transitions (interleaved 6 times in the nominal sequence) is a countermeasure against an RDP downgrade attack (see section 14 for more details), checking several times if the RDP level has not been faulted before booting.

Syntactic unrolling also shows that some erroneous states are detected, for example if the firmware has been corrupted (integrity check failure). In some cases, the request `REQ_ERROR` ends the automaton by triggering a system reset, and in other cases a `SECBREACH` triggers a mass erase. As the Bootloader automaton is very simple, the evaluator can use Frama-C GUI to make sure (visually) that there is no unexpected sequence of transitions A more complex automaton would have required to verify properties about the expected sequences, for example with MetACSL and E-ACSL combined with DSE [50, 53]. This approach has not been experimented here.

```
// Checking the validity of the transition
if (! loader_is_valid_transition(state, req)) {
// Transition REQ_ERROR is decided.
... dead code detected by Eva ...
}
```

Listing 12. `REQ_ERROR` dispatching

Some generic properties are checked: absence of some C undefined behaviors, also called RTE as defined by [15], and accessibility of code sections (detection of dead code and of potentially reachable code). No RTE has been detected in the analyzed perimeter, even warnings. Several dead code sections appear in Frama-C GUI (with a red background).

The following one is particularly interesting: when an invalid transition is detected, the request `REQ_ERROR` should be dispatched by the automaton ending then in a system reset (see the code on Listing 12).

The value analysis shows that the result returned by the function checking the validity of the transition is never the C boolean `FALSE` (whose integer value is 0) but the set {`0x55aa55aa`, `0xaa55aa55`} which contains secured magic values respectively representing `FALSE` and `TRUE`. The precision level is low enough [15] to over-approximate all the potential states even invalid transitions (that should not happen without FIA). Therefore, the dead code section reveals a bug in the way the condition detecting an invalid transition is tested. This bug is a weakness in the protection against FIA: invalid transitions are not detected and are normally handled by the automaton.

The other dead code sections show protections against FIA, i.e. countermeasures that should not be normally executed. All the countermeasures are not detected (seen as dead code) because of the lack of precision: the over-approximation includes states that are caused by fault injection.

16.3 Checking a functional property of the Bootloader

The security function SF12 defined in [26] should prevent the Bootloader of a dual-bank WooKey from booting the previous firmware version. Exploiting a vulnerability in this anti-rollback mechanism would lead to a full attack path.

Two other security mechanisms are mentioned by [26] when describing threats: verifying the integrity of the firmware, and checking the RDP level (STM32 register) before booting. Related vulnerabilities are less interesting to exploit because in each case a preliminary attack is necessary to obtain a full path: attacking DFU to load a corrupted firmware (see section 15 for such an attack path with EM faults), and forcing a lower RDP level (see section 14 for a successful downgrade with a voltage glitch).

The property stating that the booting firmware is not the result of a rollback can be expressed by checking the value of the booting address (global variable `ctx.next_stage`). One assertion is expressed for each case depending on the DFU mode and Flip/Flop versions (more details are given in [26]). Cases that should not happen trigger a false assertion. The code presented on Listing 13 implements the property verification.

15. The precision level is progressively increased during the analysis. With a low precision, the result is over- approximated. With a higher precision, in particular more trace partitioning and splitting, the result is the value `TRUE` which means that no invalid transition can happen without fault injection.

```
if (flip_shared_vars.fw.fw_sig.version > flop_shared_vars.fw.fw_sig.
    version) {
if (ctx.dfu_mode == sectrue) assert(ctx.next_stage == DFU1_START);
else if (ctx.dfu_mode == secfalse) assert(ctx.next_stage ==
    FW1_START);
else assert(false);
} else if (flip_shared_vars.fw.fw_sig.version <
flop_shared_vars.fw.fw_sig.version) {
if (ctx.dfu_mode == sectrue) assert(ctx.next_stage == DFU2_START);
else if (ctx.dfu_mode == secfalse) assert(ctx.next_stage ==
    FW2_START);
else assert(false);
} else assert(false);
```

Listing 13. Anti-rollback property verification

The goal is to find unexpected paths caused by a corrupted initial state. The symbolic state is composed of the firmware header, the RDP state, and the loader context. A specific test is written to set the initial state (concrete and symbolic variables), invoke the automaton, and check the property (see Listing 14).

```
// 1) Set the initial state: concrete and symbolic variables
...
// 2) Invoke the automaton
loader_set_state(LOADER_START);
loader_exec_automaton(LOADER_REQ_INIT);
// 3) Check the property "no rollback on boot"
...
```

Listing 14. Anti-rollback property check setup

Some modifications of the source code are needed to make the automaton execution terminate for every path (some stubs are also generated in particular for hash and CRC computations): infinite loops are removed, the primitive `system_reset` simply returns, booting does not call the specified firmware address but ends the automaton loop, errors also end the loop.

A DSE analysis with KLEE [9] does not detect paths violating the property despite an all-path coverage. Assuming that no potentially corrupted data in the initial state has been forgotten (missing symbolic variables), the analysis proves that the anti-rollback property is verified in the normal behavior of the Bootloader, i.e. without fault injection.

16.4 Vulnerability of the anti-rollback mechanism to FIA

The anti-rollback mechanism is assumed to be resistant to double fault injection as shown by the implementation of the function handling

the request `REQ_SELECTBANK`: a "sanity check against fault on rollback" (as commented in the code) is performed three times. The evaluator has to ensure that the protection provided by this countermeasure is secure enough, i.e. if a single or even a double fault cannot bypass the countermeasure and force a rollback.

Lazart [16] [51] offers several fault models, and the ability to efficiently inject multiple faults. In the current version of Lazart, the most useful model for our usage is "test inversion" as it is applied automatically (without manual configuration) and systematically to every conditional branching in the source code. So once a target function (entry point) has been identified and tested with DSE (to ensure that the functional behavior is correct), then an analysis can be immediately launched without the need to configure the way faults are injected.

The same test as in section 16.3 has been used for the analysis. But the symbolic initial state has been made fully concrete (no symbolic variable) to decrease the complexity of the analysis. Therefore, two test cases are needed, one called "Flip to Flop" trying to force a Flop instead of the expected Flip boot, and the opposite one "Flop to Flip".

The C instruction `switch` may be compiled in different ways, leading to different vulnerabilities (and different number of faults) when the fault model "test inversion" is applied by Lazart. The Bootloader binary has been decompiled with Ghidra [6] to obtain a C representation of each `switch` that is composed of the equivalent branching instructions.

Two paths with a single fault are detected by Lazart for the case "Flip to Flop". These paths exploit in a similar way a vulnerability in the function handling the request `REQ_SELECTBANK`, that first checks if both banks are bootable, and if not, systematically boots on Flop if it is bootable. Therefore, a single fault is enough to negate the first test and then simply branch to the Flop boot. In the first test, two conditions can be negated (Flip bootable or Flop bootable), hence making two attack paths as shown on Listing 15.

```
// Fault injection to negate one of the following branching:
if (flip_shared_vars.fw.bootable == FW_BOOTABLE &&
flop_shared_vars.fw.bootable == FW_BOOTABLE) {
...
}
// And to continue below, booting Flop:
/* only FLOP can be started */
if (flop_shared_vars.fw.bootable == FW_BOOTABLE) {
```

16. Provided by Verimag through the CLAPS project (funded thanks to the French ANR "Programme d'Investissement d'Avenir IRT Nanoelec" ANR-10-AIRT-05).

```
...
```

Listing 15. "Flip to Flop" attack paths

Regarding the test case "Flop to Flip", a path with a single fault exploits a vulnerability of the function handling the request `REQ_FLASHLOCK`, that only checks once if the bank to boot is Flip. Therefore, a single fault is enough to force Flip (see Listing 16).

```
if (ctx.boot_flip == sectrue) { // Fault injection to force this
    condition
...
```

Listing 16. "Flop to Flip" attack paths

A last element to be noticed regarding these attack paths on anti-rollback is that they can be tested without triggering the flash mass erase emergency state that the Bootloader executes when it detects non nominal behaviors. As we can see on the automaton represented in Figure 21, `FLASH_LOCK` state execution leads to the `ERROR` state (contrary to other states such as `FW_INTEGRITY` whose failures lead to `SECBREACH` and mass erase). This explains why FIA against the anti-rollback mechanism seem more successful than faults against integrity check (section 15) or Readout Protection anti-downgrade (section 14).

An exploitation of the case "Flip to Flop" has been attempted with a power glitch attack as presented in the next section.

16.5 Experimental setup for fault injection

Power glitches through USB The PC or the USB cable may glitch the target, allowing stealthy fault attack compared to LASER or electromagnetic attack. It could be a voltage glitch on the `Vbus` of the USB bus either with positive or negative glitch. Even if there is a voltage regulator between the USB cable and the STM32, that voltage glitch could allow code rerouting on the MCU.

However, tests have shown that the WooKey target seems protected against voltage glitch through USB. Indeed, the electronic architecture with a diode and decoupling capacitors next to the MCU tends to inhibit the effects of the glitches. Moreover, the MCU is directly connected to the `Vbus` through a GPIO in order to probe the `Vbus` voltage. This connection leads to destroy the MCU during large voltage glitches.

Direct power glitches The authors of [32] show that glitching with arbitrary waveform is more effective than traditional glitching (using pulse setup or MOSFET). However, the setup to perform arbitrary waveform is more expensive (around 100 € versus 2 € for MOSFET) and more complex to implement. Furthermore, recent attacks with traditional glitching setup [21] show that it is still very effective against unprotected target. Therefore, to comply with a low level attacker, the glitcher is implemented using a MOSFET driver (MAX17602) and a MOSFET (IRF3205) which will short-circuit the `Vcap` pin to ground during glitching.

The Nucleo-F439ZI will be the target as the MCU is the same as the one on the WooKey board. On the two capacitors connected to `Vcap1` and `Vcap2`, one is removed and the other one is replace by a 130 nF capacitor. The glitcher is directly connected to the `Vcaps` pin. The MOSFET's drain is connected to `Vcap` and the source is connected to `GND`. The delay and the pulse width are controlled by a second Nucleo board.

16.6 Exploitation of the anti-rollback mechanism vulnerability

The vulnerability identified in section 16.4 is targeted using the direct power glitch setup. By sweeping the glitch width against some dummy code, it can be found that the optimal glitch width to corrupt the processing of the circuit is around 150 ns. Such a glitch is illustrated on Figure 30.

Then, the glitch is swept over the boot process in order to find the correct timing to exploit the vulnerability. In order to reduce the jitter, the oscilloscope (rigol DS1054Z) is used to synchronize on the power consumption of the target. The right timing to perform the attack is 80 µs after the re-synchronization. This timing is shown on Figure 31. Applying a power glitch at this particular time slot of the boot process allows to force the Flop boot instead of the regular Flip one. It shows that the anti-rollback mechanism vulnerability is exploitable with an achieved success rate of 0.4 % (17 successful attempts over 4,000 trials).

16.7 Conclusion

In this section, a full exploitation was performed from code analysis to a real attack using power glitches on the STM32 circuit. The exploited vulnerabilities could have been discovered in black box testing, i.e. without code analysis. However, as the source code of WooKey is available, experimenting the ability of the exposed code analysis methodology to detect non-obvious vulnerabilities in an exhaustive way is complementary

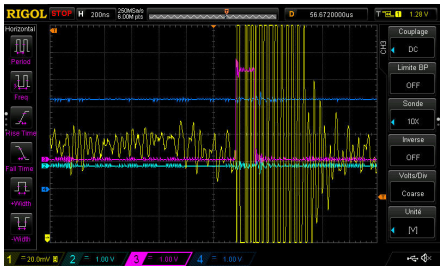

Fig. 30. Applied glitch.
Magenta: glitcher's command
Yellow: power consumption.

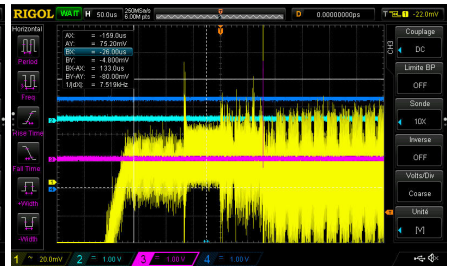

Fig. 31. Corrupted boot process.
Yellow: power consumption. Magenta: glitch command. The re-synchronization is performed on the negative spikes as shown by the trigger's marker.

with a black box approach that may reveal surprising vulnerabilities, but that usually requires too much time to cover every possible attack paths.

17 SDIO interface analysis

The WooKey device requires an SD card to be able to store the encrypted user data. Since it is an external interface, a malicious SD card (or a device able to emulate an SD card) might be used to trigger and exploit vulnerabilities. Exploiting them leverages partial attack paths: other security mechanisms must be bypassed to recover the assets (e.g. memory segregation). Two kinds of vulnerabilities could be triggered:

— Application layer vulnerabilities (when plain text application data are handled by WooKey tasks).
— SD protocol vulnerabilities (in the case of mishandled SD functions, or a weak state-machine).

The purpose of the current section is to assess the exposure to such attacks, and evaluate the complexity to put in place a malicious SD device.

17.1 State of the art of SD attacks

The Secure Digital protocol [20] is not often reviewed from a security standpoint, despite its complexity. Attacks targeting the host are even less studied. Dana Geist and Thom Does (University of Amsterdam) published a report in 2016 on this topic [57]. This project is mainly intended to attack computers using advanced features of SD Cards. This highlights the complexity of implementing a fake SDIO device.

Nonetheless, there are no public tools to fuzz or to interact with an SD card reader (i.e. targeting the host).

17.2 SD card content analysis

Some devices are writing metadata into the memories. In such cases, it is interesting to identify if there are some mishandled parameters. No specific tools are required for exploitation.

This activity has been performed by analyzing an empty SD card after being used by WooKey. Nevertheless, no obvious structure has been identified, and this has been confirmed by looking into the WooKey project source code (no extra data is written into the SD card apart from the encrypted user data). The exposure to malicious SD content is hence minimal.

17.3 SD protocol analysis

The SD card communication protocol is defined by the SD Association, and simplified versions of the specifications are available at [20]. The physical layer of SDIO is composed of the following signals:

— A clock signal, `CLK`.
— A Command signal, `CMD`, which is bidirectional (command and response on the same signal).
— 1, 2, 4 or 8 data signals `D0` to `D7`. Common SD cards use up to 4 lines. The data lines are bidirectional (read and write data on the same signal).

All SD cards should respond to 3.3 V logic, but newer SD cards might allow logic down to 1.8 V to improve the speed and consumption. The communication levels, the clock speed and the bus width are negotiated during the communication.

The card is acting as a slave that only responds to host commands (up to 126 commands can be implemented, including application command `ACMD`). The commands and responses frames are 48 bits long and contain 32 bits of effective data, except for the response `R2` that is 136 bits long (120 bits of data). Additionally, some commands do not expect a response and some commands trigger a read or a write from the data lines. Some SD cards also support commands to be queued. Newer versions of SD specifications also define some advanced features, for media streaming, for connectivity (Wi-Fi, Bluetooth, GPS, etc.), or for security (authentication, encryption).

However, embedded systems such as WooKey do not implement every functionality offered by the SD protocol. Consequently, sniffing the data is a good starting point to identify what is indeed implemented.

SDIO Sniffing The first step consists in determining the maximum clock rate and expected capture duration. This can be done by using an oscilloscope on the clock and command lines. As specified by the SD protocol, the communication starts with a 400 kHz clock (321 kHz measured). The communication speed then increases to 50 MHz.

The first thing that needs to be highlighted is that the communication only starts when the user has been authenticated (UserPIN valid) and the WooKey is plugged in to a computer (`SCSI_CMD_READ_CAPACITY` sent to USB). This puts the attacks targeting the SD card in the post-authentication category or entrapment category. Such attacks can be however stealthy if the malicious SD device looks alike the genuine one.

Sniffing requires to capture with a sampling rate of at least 250 MHz (to get accurately the clock edge) for around 15 seconds, which prohibits the usage of an oscilloscope due to the memory length against the usage of a logic analyzer. There are two kinds of logic analyzers:

— Buffered logic analyzers, which provide fast sampling rate. However, buffers sizes are often limited to several megabytes (advanced logic analyzers provide compression to help spaced events to be captured).
— Streamed logic analyzers, which provide continuously the samples to the computer (almost no memory limitation), but with a moderate sampling rate (communication is the bottleneck).

Since the clock line is always active (even when there is no communication), the usage of a buffered logic analyzer does not fit the requirements as the compression would not be efficient. A *Saleae Logic Pro 16* [17] has been chosen because:

— It allows capturing in streaming mode at 500 MS/s, allowing continuous capture of several gigabytes to terabytes.
— It allows developing custom decoding protocols, with a great community.

At 50 MHz, probing SDIO is not trivial: the input impedance of the *Saleae Logic* is still very low (about 350 Ω); and WooKey does not drive enough current, which introduces some bit errors. Moreover, some coupling effects might occur between signals, increasing the risk of induced errors. The bit-error impacts on data lines are highly amplified when ciphering occurs, and the computer accessing the USB MSC (SCSI mass storage)

protocol reacts randomly while attempting to parse the partitions headers as shown on Figure 32.

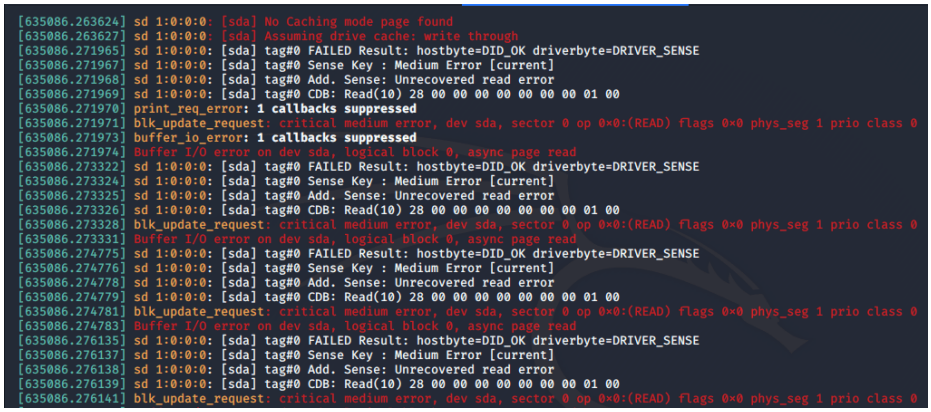

Fig. 32. Linux `dmesg` error while sniffing SDIO

To be slightly less intrusive, some extra pull-up resistors have been placed on signals and the wires were shortened and shielded. This allows capturing both clock and command line without any error, but the data lines still cannot be captured properly. It is still enough for a first analysis of issued commands.

Another solution would be to woks with low capacitance probes, like active probes (expensive). This would be mandatory for dealing with higher logic speed (but in such case, the sample rate of the *Saleae Logic* would probably be the bottleneck).

SDIO decoding An open-source SDIO analyzer software has been developed for the *Saleae Logic* by airbus-seclab: *SDMMC-Analyzer* [18]. This project has mainly been developed for eMMC analysis, and does not handle well the SD card protocol (particularly, eMMC commands and responses are slightly different). Using it reveals some unknown and misinterpreted commands/responses. Consequently, this SDIO Analyzer has been heavily modified to fit the needs of decoding SD protocols (see Figure 33).

The capture highlights a very minimalist SDIO implementation: the host waits for the card to be ready, then increases the clock speed to 50 MHz before having identified the card (`Card_Identification_Data`

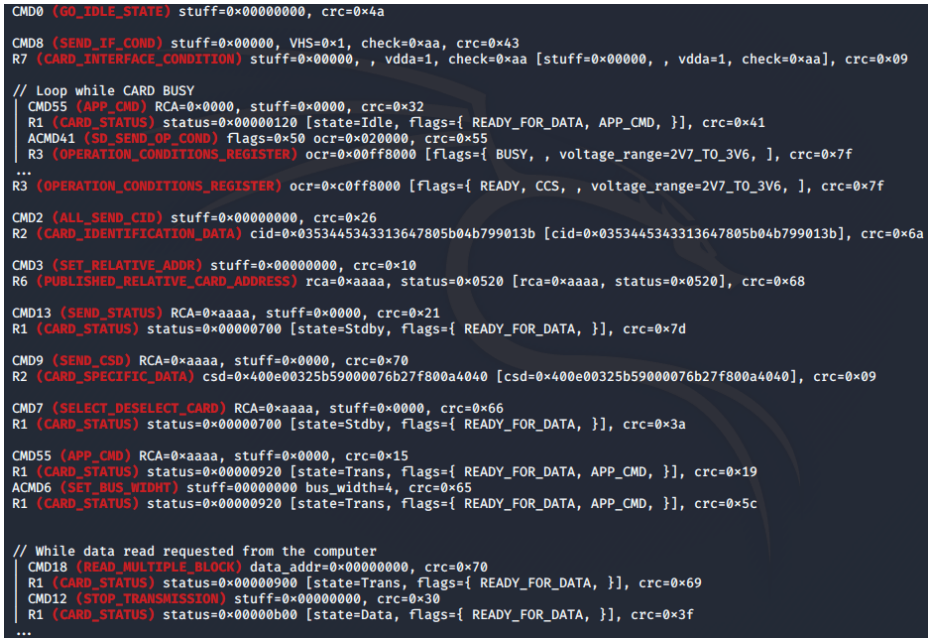

Fig. 33. SDIO capture with the *Saleae* and customized `sdmmc-analyzer` plugin

and `Card_Specific_Data` requests). The card is then selected and stays in the "trans" state, waiting for read/write transfers upon computer request.

Moreover, it is interesting to notice that, despite the card answers that the maximum speed for data line is 25 Mbits/s per line (reaching 50 MBytes/s using the 4 lines), the clock is kept at 50 MHz. This means that some mandatory parameters in the card answers are not taken into account; and this could partially explain the communication issue when probing the data lines (the communication is already out of specifications). By crosschecking with the source code, the responses handled by WooKey are summarized in Table 5.

Except for few error flags checking, the source code analysis reveals that the only field that is effectively handled by WooKey is the card capacity (which is computed from the `Card_Specific_Data` because it is requested by the computer to initialize the SCSI transfers and it is displayed on the WooKey screen).

Finally, there was no request (such as *extended* `Card_Specific_Data` request or advanced features usage) that would require the data lines in

Type	Content	WooKey handling
`R1`	Card Status	Current state only checked against trans to raise error
`R2`	Card Identification Data	Not handled
`R3`	Card Specific Data	Only capacity is handled and used by WooKey
`R4`	Operation Condition Register	Only Card capacity status and Card ready flags checks
`R5`	Relative Card Address Card status bits	Error flag check
`R6`	Card Interface Condition	Error flag check

Table 5. SD responses supported by WooKey

the response. The data lines are only used to provide data between the computer and the SD card (WooKey streams the encrypted data through `DMA` requests, without any particular handling). This means that the only interesting handled fields are the ones related to card capacity, in the `Card_Specific_Data`. Depending on the version, it can be a single 22 bits integer or a pair of integers (12 bits + 3 bits).

SDIO Fuzzing The previous analysis was performed with SDIO fuzzing in mind. Fuzzing a SDIO host is very different from fuzzing a SDIO card, since the fuzzer does not control the communication channel. The only thing that is possible is to respond to messages.

Moreover, the response frame format (including the frame length) depends on the state machine, which is fully controlled by the host. In other words, it is not possible to respond with an unexpected message type. Fuzzing the SDIO consists in fuzzing the content of responses in a way that produces unexpected results.

The initial idea was to place an FPGA in man-in-the-middle position, which only modifies a specific response (triggered from a command). This allows to avoid re-implementing a complex SDIO stack into an FPGA, and only focus on specific fields to be modified. Developing an SDIO fuzzer is interesting for SD interface analysis, targeting for instance:

— The card state machine (given in `R1` responses).
— The case of multiple card responses to the `ALL_SEND_CID` command.
— The bus speed and width with non-standard values.
— Triggering unexpected error flags.

Such a fuzzer might have to deal with communication constraints (interfacing is challenging due to impedances, voltage level change, clock phases, and bidirectional signals).

Finally, and since WooKey does not handle many SDIO data, the effort for developing an SDIO fuzzer is not justified here (and would be highly over-dimensioned for the purpose). Instead, it has been decided to cover SDIO through a tainted code analysis, focusing on the card capacity parameters.

Tainted code review By computing minimum and maximum values for the card capacity fields and analyzing the propagation of the results through the *PIN* application (to display the card size in GBytes) and *USB* application (requested parameter during SCSI initialization), no overflow has been found. The values are stored in an `uint64_t` when necessary, and displaying the pair of 4 digits does not overflow the oversized printed buffers.

17.4 Conclusion

According to the fact that the attack surface on WooKey SDIO interface is minimal (only available after user authentication, with a minimal SDIO stack implementation, and without handling plain text data from the SD card); and that the tainted code analysis does not reveal particular issues while handling the integer parameters, WooKey SDIO interface seems to offer a good level of security.

18 Analysis of the SPI communication with the display

18.1 Attack path

WooKey uses a token for authenticating the user of the USB thumb drive. Because the token is protected with a PIN code, the theft of both the token and the device does not allow an attacker to decrypt the content.

The threat model for WooKey takes into account an attacker trying to steal the PIN code using a fake device, so a legitimate user must first enter a PetPIN and then validate a PetName before entering the UserPIN [25, 30, 31]. The risk of an attacker using a hardware tap on the SPI bus to steal the PIN is considered residual [31] since it requires a physical access on the device to insert the tap, and then stealing the token to extract the master key to decrypt the data.

However, using the electromagnetic emission from the SPI bus might be another possible attack vector to steal the UserPIN, and does not require a direct physical access to the device. In its current design, WooKey uses

a main board for the STM32 and a daughter board for the touchscreen. They are linked together with an unshielded ribbon cable transmitting synchronous serial data using SPI.

In this section, we will focus on a proof-of-concept of a hardware tap to steal the UserPIN as a first step to develop the tools needed to extract the different PIN codes. We only discuss the feasibility of extracting sufficiently accurate data from the electromagnetic emissions to recover both PIN codes since this attack was not performed. The information provided here are to be taken as complementary to the TEMPEST characterization described in 19.

18.2 State of the art

The retrieval of information transmitted on serial lines through electromagnetic emissions has been explored for some time now [56]. Keyboards using both PS/2 and USB interfaces have been studied in depth to show that it is possible to retrieve the keystrokes in a practical environment [59, 61, 62].

18.3 Practical attack

The SPI bus was tapped by directly soldering on the ribbon cable connector (see Figure 34). A CY7C68013A from Cypress was used to act as a logic analyzer, since it can collect 4 channels at 12 MHz. A minimum of 12 Mega samples per second are necessary since the SPI clock is set to 6 MHz for the screen. The following signals were acquired: Clock (`SCLK`), Chip Select (`CS`), Master In Slave Out (`MISO`) and Master Out Slave In (`MOSI`).

The data are recovered by waiting for the `CS` line to be asserted low, and then sampling the current bit on `MISO` and `MOSI` for every rising edge of the `SCLK` line. For the proof-of-concept, *PulseView* [13] was used to record the signals and then the result was exported to raw binary and parsed with a tiny C code for performance reasons. Finally, a Python script, using the *Python Imaging Library* [14], was used to recover the images displayed on the screen. Three commands need to be interpreted:

— `Column_Address_Set` (`0x2A`)
— `Page_Address_Set` (`0x2B`)
— `Memory_Write` (`0x2C`)

The column and page commands take two 16 bits arguments for the starting and ending column or row. The next memory write will then be inside the frame described by the column and page address

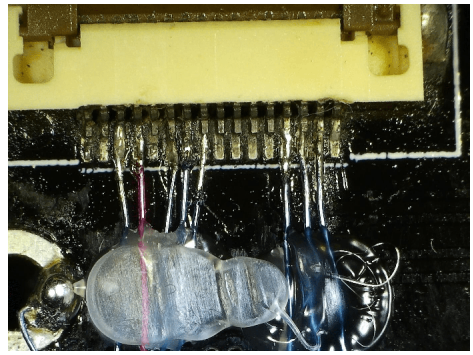

Fig. 34. Soldering for SPI bus sniffing

set commands. The write command takes a variable number of 3 bytes arguments describing the current pixel in RGB format with 6 bits words for each color component. It has to be noted that the number of pixels written can be less than the frame size. By looking for a potential command in the sent buffer, or using timing information, it is possible to detect the end of the write command.

In order to help with parsing, all possible commands sent by WooKey to the screen have been added to the parser with their number of arguments: `Display_OFF`, `Power_Control_1` and `2`, `VCOM_Control_1` and `2`, `Memory_Access_Control`, `Vertical_Scrolling_Address_Start`, `Sleep_Out` and `Display_ON`.

18.4 Results

The recovery of the images displayed on the screen is straight-forward and shown on Figure 35: since we are directly soldered on the SPI bus with a sampling rate satisfying the Nyquist–Shannon sampling theorem, no information is lost.

18.5 Real world feasibility

In a scenario where the signal would be acquired from electromagnetic emissions, some bytes will be corrupted and the `CS` line will not be available to tell us when the bus is actually active. Since the commands to send images are sent in burst with near-constant timing between the bytes, depending on where we are in the sequence, it is possible to use this information to synchronize the decoding of commands and their arguments. The timing reflects the function call for commands and operands in the

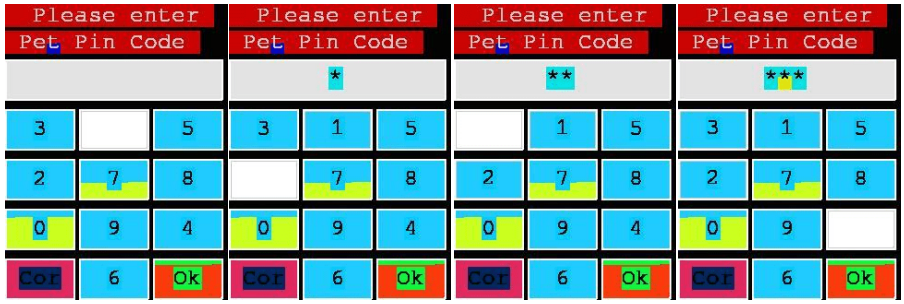

Fig. 35. Capturing '1234' sequence entered on the pinpad

C implementations of the screen driver. The sequence to display a new image (tile) is as follows:

— `Column_Set` followed by `Page_Set`, all bytes separated by 10 to 12 μs
— a `Memory_Write` command directly follows by 10 to 12 μs
— the first pixel red component follows after a 10 to 12 μs gap
— pixel components are separated by 580 to 920 ns

This observation should help to segregate between the command part and the actual image written to the screen. While the timing for the red component is slightly higher than for the green and blue ones, the change in color cannot be detected using only the timing information because the SPI line is much slower than the difference in timing while looking up a new color in the palette for RLE (Run-Length Encoded, see [16]) images. Also, when the drawing of a new tile directly follows the previous one, the same timing of 10 to 12 μs is observed before the new `Column_Set` command. This should help to identify the menu style specific to the drawing of the pinpad and the refreshing of a tile after a touch on the screen.

19 TEMPEST attack on WooKey's screen

19.1 Presentation

The TEMPEST code name captures various security specifications from NSA and NATO about radio, electronic, acoustic or vibrating emanations from an information system. These are considered as data leakage from the system since they are not intentionally produced and are a side effect of the system's operation: one can see them roughly as long-range side-channel leakages. In France, ANSSI edited in 2014 a document explaining how to

mitigate TEMPEST attacks in secured building and installations [27]. The TEMPEST effects may be leveraged as side-channel attacks to partially or totally retrieve secrets from a system.

The efficiency of TEMPEST attacks is not a myth. They supposedly started during the first World War on telephone wires and were actively used during the second World War [42]. Since then, many declassified documents confirm the widespread use of such attacks at state level [43].

In 2015, a laboratory from Tel Aviv University disclosed a vulnerability in GnuPG. They successfully extracted keys from the surrounding field emanating from a regular laptop [38]. In 2017 a cybersecurity team from Fox-IT was able to recover an AES-256 encryption key using common hardware at a distance between 1 m and 30 cm [37]. More recently, so called "screaming channels" [33] make use of classical SCA techniques (template attacks) to achieve key recovery on an AES-128 leaking through the radio front-end of a Nordic Semiconductor nRF52832 at a range of 10 m using 1,500 traces.

Another example application is to extract information from an electromagnetic leakage of a display link. Electromagnetic emanations may occur and, in specific conditions and with appropriate equipment, they can be captured. Easily accessible tools allow anybody to setup a TEMPEST attack: one of the most convincing examples is the *TempestSDR* [46] framework which targets HDMI cables. Thanks to the work of Martin Marinov who released *TempestSDR* in 2014, anyone with less than 100 € of equipment can build his own TEMPEST installation. *TempestSDR* offers the ability to capture and intercept on-the-fly the signal emitted from the cable, is compatible with affordable hardware, and runs on Window and Linux. An article and a presentation at SSTIC 2018 [52] (in french) detail how to work with *TempestSDR* on DVI or HDMI cables.

Critical devices which manipulate confidential data and require a high security level need to be tested and have mitigations against TEMPEST. In the context of the Inter-CESTI challenge, it has been decided to test the robustness of the WooKey platform against these attacks.

19.2 Preliminary work

Before focusing on WooKey, the evaluator validated the setup on a known setting: the TEMPEST attack was reproduced to intercept the image displayed on a screen through an HDMI cable. The setup is the following:

— Radio receiver: USRP N210 (≈ 2,000 €)
— Receiving board: WBX 50-2200 MHz (≈ 500 €)

— Antenna: supplied antenna

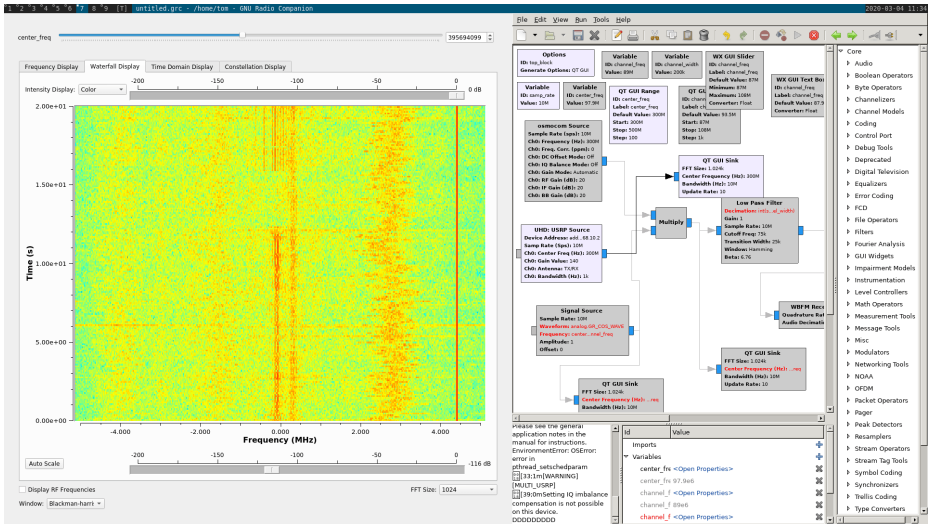

Fig. 36. HDMI Spectrogram

The HDMI 2.1 is bond to a large band with a maximum of 340 MHz. With a setting at 400 MHz, *GNU Radio* [7] has been used to observe the signal and validate the frequency. On Figure 36, we can observe the switch occurring on the screen between a dark image and a bright one, done by hand using Alt+Tab. The waterfall spectogram displays those waves emitted at different frequencies depending on the data passing through the wire. We can now tune *TempestSDR* on the matched frequency of 400 MHz. Figure 37 shows the screen on the right partially restored on the left one.

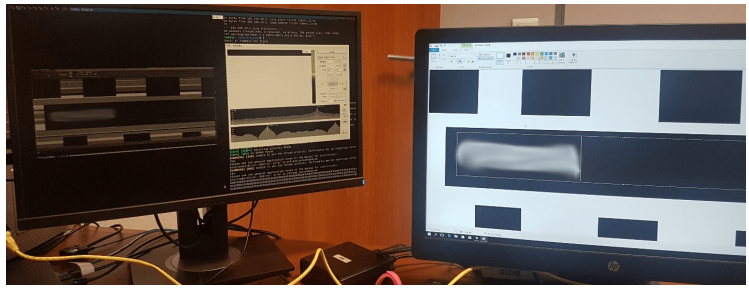

Fig. 37. *TempestSDR* in action

Although the evaluation center is not expert in such attacks, the experiment has been reproduced in a couple of days and with fairly standard equipment. This step had several purposes: first gaining hands-on experience with the hardware and software components used for TEMPEST attacks and validate our setup. With the setup out of the way, it is possible to focus on attacking the WooKey board itself.

19.3 Application to WooKey

In the context of the WooKey project, even though close-range side-channel attacks are considered meaningful only during the pre-authentication phase, TEMPEST attacks are also of interest during the user authentication as they can be long-range and hence be performed stealthily. A partial attack could be conducted by intercepting the radio emissions from the WooKey platform.

The purpose of the following experiments is to retrieve the PetPIN and UserPIN codes of the user at the moment they are entered on the touch screen. The captured trace can be processed later to decode the information. This would allow to recover the PetPIN and UserPIN, then to complete the attack the attacker would need to steal the device from the victim to extract the confidential data.

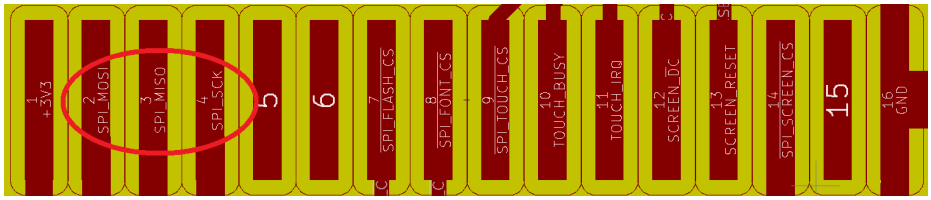

Fig. 38. Schematics of the SoC-Screen connection

WooKey is composed of two boards, one is the main SoC and the other the TFT screen and its controller. The link between the main SoC and the screen board is done by a 16 pins cable. The WooKey's source code was analyzed to understand the configuration of the hardware components:

— `driver-ili9341`: ILI9341 TFT screen userspace driver
— `driver-ad7843`: AD7843 touchscreen userspace driver

These two drivers use functions of the userspace SPI driver `driver-stm32f4xx-spi`.

SPI (Serial Peripheral Interface) is a standard and pretty basic serial communication interface. It uses 4 wires, clock, input/output and slave

selection between devices. The Figure 38 allows to identify the SPI pins on the cable between both boards. WooKey schematics (Figure 39) also show that the touchscreen controller is connected to the SPI wires.

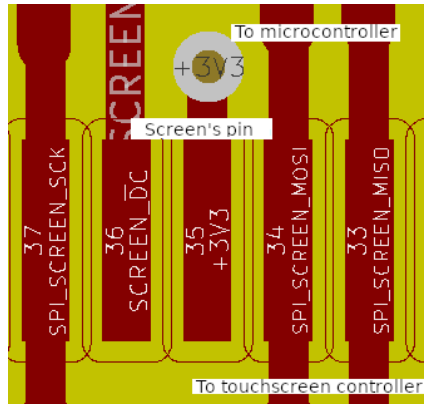

Fig. 39. SPI zoom on the screen board schematics

In order to conduct a TEMPEST attack on WooKey, the attacker will have to listen the data exchanged on this cable at a rate of 6 MHz (from the source code on Listing 17).

```
uint8_t tft_init(void)
{
...
#if CONFIG_WOOKEY_V1
    spi1_init(SPI_BAUDRATE_6MHZ);
#elif defined(CONFIG_WOOKEY_V2) || defined(CONFIG_WOOKEY_V3)
    spi2_init(SPI_BAUDRATE_6MHZ);
...
```

Listing 17. SPI bus frequency in WooKey source code

To work with this rather low frequency (compared to HDMI), the equipment has to be adapted:

— Radio receiver: USRP N210 (≈ 2,000 €)
— Board: LFRX DC-50 MHz (≈ 100 €)
— Antenna: ANT500 (≈ 30 €)

All the following experiments were conducted using *GNU Radio Companion*. The antenna ANT500 has a minimal frequency of 75 MHz, it is thus not appropriate to receive 6 MHz. The wavelength of a wave is $\lambda = \frac{v}{f}$ where v is the speed of light and f the frequency of the wave. Indeed, to

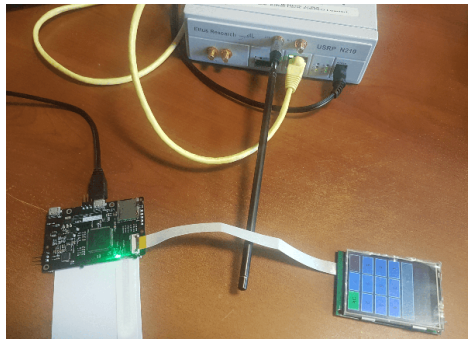

Fig. 40. Setup for TEMPEST attack on WooKey

obtain the length of a proper antenna the speed of light must be divided by the frequency:

$$\frac{300,000,000 \text{ m/s}}{6,000,000 \text{ Hz}} = 50 \text{ m}$$

This size can be cut off by a divider, so it is possible to find a regular size antenna which is still acceptable for our requirements (although it might be less accurate).

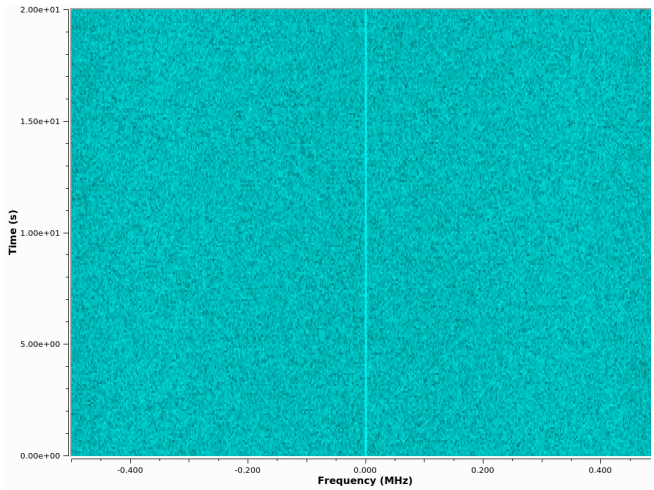

Fig. 41. Inactive WooKey spectrogram

On Figure 41, the first observation can be made when WooKey is powered on and its screen displays the PIN selection interface. The spectrogram is a waterfall plot that shows here a range of frequencies and

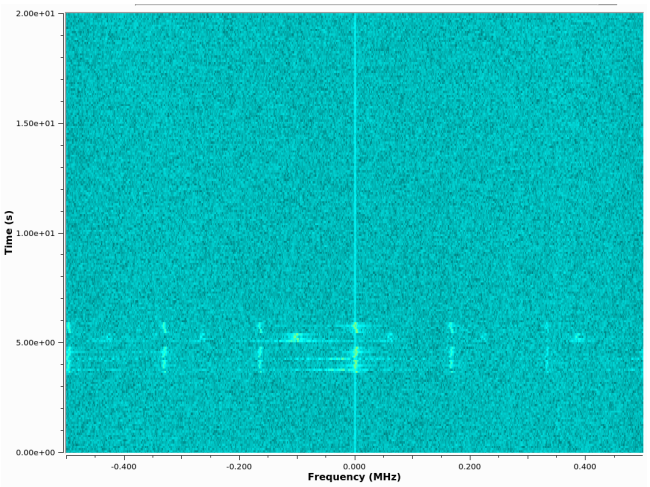

Fig. 42. WooKey initialization spectrogram

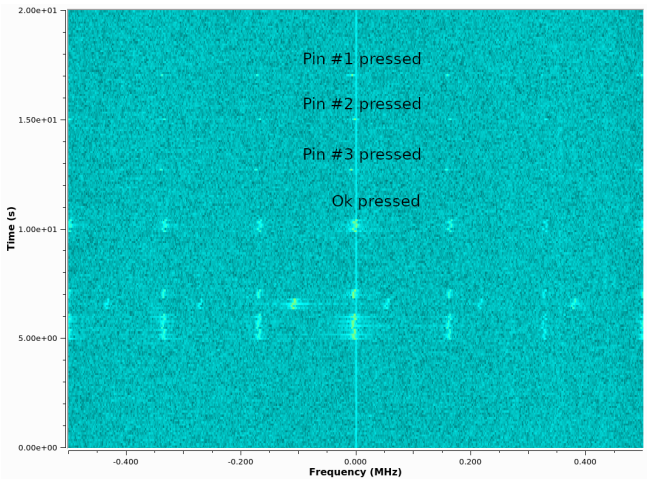

Fig. 43. WooKey usage spectrogram

is able to show multiple signals on a period of time. The figure here displays a history of 2 seconds. In this case, nothing happened, only usual perturbations (white noise).

On Figure 42 the spectrogram shows emanations produced while the WooKey boots up. When the evaluator enters a PIN code on the screen and then presses the OK button, the screen is totally refreshed. Before that, each press on a numbered key slightly changes its color for a short time. These emanations are visible on the spectrogram on Figure 43 (to obtain this the selected frequency is 5 MHz). It should be noted that this signal is repeated approximately every 1 KHz. Figure 44 displays the variation of this frequency over time.

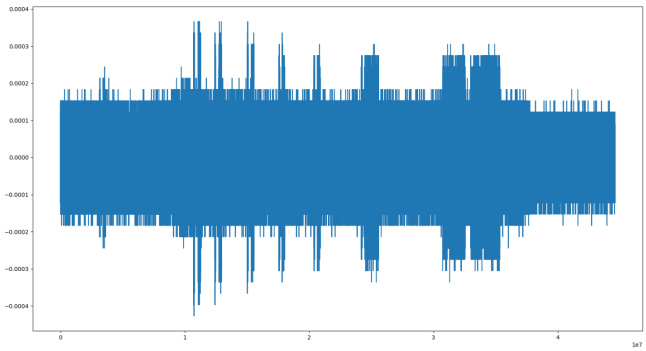

Fig. 44. Frequency variation over time

In the context of the Inter-CESTI challenge, the overall workload allocated to the TEMPEST attack was four days. In this short time frame, it was possible to demonstrate that the hardware components used by the WooKey platform leak information in the form of EM emissions. These emissions could potentially be captured by an attacker in order to recover the victim's PetPIN and UserPIN, which are both sensitive assets of the WooKey product. Several tools, open source or not, are publicly available and the required hardware is affordable. An analyst with modest expertise will be able to setup and capture these signals.

However, to complete the attack, the adversary would need to analyze the captured signal and ideally reconstruct SPI frames. The TEMPEST study described in the current section should be completed with the elements extracted from section 18 dedicated to the SPI communication details. There are no public tools available for such analysis, and developing a framework for long-range SPI decoding with noise requires time and

expertise, yielding in the conclusion that the TEMPEST attack path is *residual* in the context of the WooKey product. Although the vulnerability is present, its exploitation would require an attack potential (i.e. time, means and knowledge) beyond what is considered acceptable for the product.

20 Conclusion

In this article, we have provided a methodological and technical feedback on the inter-CESTI challenge regrouping an overview of various software, hardware and hybrid attacks conducted by the 10 ITSEFs licensed for the french CSPN scheme. The WooKey project (the evaluation target) provided a white box evaluation context thanks to its open-source and open-hardware aspects: this allows advanced instrumentation techniques, leveraging various attack paths optimizations on par with the limited time frame constraints of the inter-CESTI challenge.

The results of the challenge exhibit that the three kinds of attacks (software, hardware, hybrid) can be efficiently performed by the 10 ITSEFs beyond the specialization of each one. Interesting attack paths that involve software exploits, cryptographic weaknesses, side-channels and fault injections have been notably found and exploited. As a matter of fact, physical attacks have proven to be quite easily achievable using cheap and accessible equipment (such as the ChipWhisperer, the ChipSHOUTER, FPGA, etc.), demystifying the fact that such attacks require very advanced adversaries with substantial means outside the CSPN scope.

First of all, this supports the fact that "Hardware devices with boxes" alike targets must be studied and evaluated with all these attack paths in mind (i.e. included in the threat model) to cover all the relevant security aspects. Secondly, the results of the challenge also clearly encourage the creation of a "Hardware Device" in the CSPN scheme: this is under scrutiny within CCN, ANSSI's Certification Body, with the inter-CESTI feedback in mind.

Finally, the outcomes of the challenge have also been a great source of betterment for the WooKey project. For the sake of transparency and security improvement, all the attack paths and enhancement advice provided by (and discussed with) the ITSEFs have been integrated in recent commits.

A ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a signature scheme. It has been standardized in [23].

Input: private key d, an encoded integer $m \in [0, t-1]$ representing a message
Output: Signature (r, s)

1: $k \xleftarrow{\mathcal{R}} \{1, \ldots, t-1\}$
2: $Q \leftarrow [k]G$
3: $r \leftarrow x_Q \bmod t$
4: **if** $r = 0$ **then**
5: **go to** line 1
6: $k_{inv} \leftarrow k^{-1} \bmod t$
7: $s \leftarrow k_{inv}(dr + m) \bmod t$
8: **if** $s = 0$ **then**
9: **go to** line 1
10: **return** (r, s)

Algorithm 1. ECDSA Signature

B Main loop of the ECSM

This code comes from the file prj_pt_monty.c.

```
/* Main loop of Double and Add Always */
while (mlen > 0) {
   int rbit_next;
   --mlen;
   /* rbit is r[i+1], and rbit_next is r[i] */
   rbit_next = nn_getbit(&r, mlen);
   /* mbit is m[i] */
   mbit = nn_getbit(m, mlen);
   /* Double: T[r[i+1]] = ECDBL(T[r[i+1]]) */
   prj_pt_dbl_monty(&T[rbit], &T[rbit]);
   /* Add:  T[1-r[i+1]] = ECADD(T[r[i+1]],T[2]) */
   prj_pt_add_monty(&T[1-rbit], &T[rbit], &T[2]);
   /* T[r[i]] = T[d[i] ^ r[i+1]]
    * NOTE: we use the low level nn_copy function here to avoid
    * any possible leakage on operands with prj_pt_copy
    */
   nn_copy(&(T[rbit_next].X.fp_val), &(T[mbit ^ rbit].X.fp_val));
   nn_copy(&(T[rbit_next].Y.fp_val), &(T[mbit ^ rbit].Y.fp_val));
   nn_copy(&(T[rbit_next].Z.fp_val), &(T[mbit ^ rbit].Z.fp_val));
   /* Update rbit */
   rbit = rbit_next;
}
```

C SCA on WooKey's HMAC-SHA256 details

The method used for performing the CPA could suggest that an error on a lower byte will make the attack on next byte unfeasible. Indeed, for instance, if `Wt[0]` best guess is not the correct value, the carry propagation on `T1[1]` as on `T2[1]` and `T3[1]` will not be correct. Our tests showed that the influence of the carry between two bytes of the same round is relatively low. The attack succeeded on `Wt[1]` whereas `Wt[0]` had been changed with bad values on purpose: the amount of traces to retrieve `Wt[1]` would be a little bit higher than for `Wt[0]` but not so much. However if only one byte is wrong at one round, it is completely impossible to find any byte of `Wt` at the next round. This information could be used to go back to the previous round and find the correct value. With this methodology we consider for each byte only its contribution. The three other ones are considered as noise even if lower bytes are already successfully retrieved. We have tried a second methodology where we consider not only the HW of current byte but also the HW of previous ones. So for `Wt[1]`, the HW was computed on 16 bits; for `Wt[2]`, the HW was done on 24 bits and finally for `Wt[3]`, it was done on 32 bits. For `Wt[1]` and `Wt[2]`, the correlation for correct key was higher than with previous methodology but the correlation for other keys are also higher. For `Wt[3]`, the correct key was not the one with the best correlation. We have two hypothesis which could explain this behavior. The first one is that the HW model doesn't completely fit the leakage of the chip. The second one is that the distribution shape of the HW on 8, 16, 24 or 32 bits is not the same. Considering only one byte, the probability to be on a low or high HW value is not negligible. On 32 bits, when we attack `Wt[3]`, the HW is more often on the center of the distribution which doesn't make it trivial to distinguish the values. This assumption could be explored with simulations to see if it is real or not.

References

1. Certification CSPN. https://www.ssi.gouv.fr/administration/produits-certifies/cspn/.
2. ChipSHOUTER. https://github.com/newaetech/ChipSHOUTER.
3. ChipWhisperer. https://wiki.newae.com/Main_Page.
4. Common Criteria: New CC Portal. https://www.commoncriteriaportal.org/.
5. Frama-C. https://frama-c.com/download.html.
6. Ghidra. https://ghidra-sre.org/.
7. GNU Radio. https://www.gnuradio.org/.
8. ISO7816 Analyzer. https://github.com/nezza/ISO7816Analyzer.
9. Klee. https://klee.github.io/.
10. libecc. https://github.com/ANSSI-FR/libecc.
11. NXP J3D081 Security Target Lite. https://www.commoncriteriaportal.org/files/epfiles/0860b_pdf.pdf.
12. PicoScope. https://www.picotech.com/.
13. PulseView. https://sigrok.org/wiki/PulseView.
14. Python Imaging Library (PIL). https://www.pythonware.com/products/pil/.
15. RTE - Runtime Error Annotation Generation. https://frama-c.com/download/frama-c-rte-manual.pdf.

16. Run-Length Encoding (RLE). https://en.wikipedia.org/wiki/Run-length_encoding.
17. Salae Logic Pro analyzers. https://www.saleae.com/.
18. SD/MMC Analyzer for Logic. https://github.com/dirker/sdmmc-analyzer.
19. Teensy USB Development Board. https://www.pjrc.com/teensy/.
20. The Secure Digital protocol. https://www.sdcard.org.
21. WALLET.FAIL. https://wallet.fail/.
22. WireShark. https://www.wireshark.org/.
23. ANSI X9.62, Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), September 1998. American National Standards Institute, X9-Financial Services.
24. Minerva: Incomplete formulas leak everywhere. https://minerva.crocs.fi.muni.cz/, 2019.
25. The WooKey project documentation. https://wookey-project.github.io/, 2019.
26. WooKey Security Target Evaluation. https://wookey-project.github.io/_downloads/security_target_intercesti.pdf, 2019.
27. ANSSI. La protection contre les signaux compromettants. https://www.ssi.gouv.fr/uploads/IMG/pdf/II300_tempest_anssi.pdf, 2014.
28. Aurélie Bauer, Éliane Jaulmes, Emmanuel Prouff, and Justine Wild. Horizontal Collision Correlation Attack on Elliptic Curves. In *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, pages 553–570. Springer, 2013.
29. Sonia Belaid, Luk Bettale, Emmanuelle Dottax, Laurie Genelle, and Franck Rondepierre. Differential power analysis of HMAC SHA-2 in the Hamming weight model. In *2013 International Conference on Security and Cryptography (SECRYPT)*, pages 1–12. IEEE, 2013.
30. Ryad Benadjila, Arnauld Michelizza, Mathieu Renard, Philippe Thierry, and Philippe Trebuchet. WooKey: designing a trusted and efficient USB device. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 673–686, 2019.
31. Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnauld Michelizza, and Jérémy Lefaure. WooKey: USB Devices Strike Back.
32. Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. https://tches.iacr.org/index.php/TCHES/article/view/7390.
33. Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers. In *Proceedings of the 25th ACM conference on Computer and communications security (CCS)*, CCS '18. ACM, October 2018.
34. Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal Correlation Analysis on Exponentiation. In *Information and Communications Security - 12th International Conference, ICICS 2010, Barcelona, Spain, December 15-17, 2010. Proceedings*, pages 46–61. Springer, 2010.

35. Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, pages 292–302. Springer, 1999.

36. Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple arm-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 223–234, New York, NY, USA, 2013. Association for Computing Machinery.

37. Fox-IT. TEMPEST attacks against AES. `https://resources.fox-it.com/rs/170-CAK-271/images/Tempest_attacks_against_AES.pdf`, 2017.

38. Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing Keys from PCs using a Radio:Cheap Electromagnetic Attacks on Windowed Exponentiation. `http://www.cs.tau.ac.il/%7Etromer/papers/radioexp.pdf`, 2015.

39. Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. Address-Bit Differential Power Analysis of Cryptographic Schemes OK-ECDH and OK-ECDSA. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 129–143. Springer, 2002.

40. Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. A Practical Countermeasure against Address-Bit Differential Power Analysis. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, pages 382–396. Springer, 2003.

41. Jean Dubreuil. Java Card security, Software and Combined attacks.

42. jya@pipeline.com. [...] compilation of the history of TEMPEST [...]. `http://cryptome.org/tempest-time.htm`.

43. jya@pipeline.com. [...] responses to Cryptome's request for information on the history of TEMPEST [...]. `http://cryptome.org/tempest-old.htm`.

44. Ievgen Kabin, Zoya Dyka, Dan Kreiser, and Peter Langendörfer. Horizontal address-bit DPA against montgomery kP implementation. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017, Cancun, Mexico, December 4-6, 2017*, pages 1–8. IEEE, 2017.

45. Matthias J. Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. *Differential Power Analysis of XMSS and SPHINCS*, pages 168–188. 01 2018.

46. Martin Marinov. Remote video eavesdropping using a software-defined radio platform. `https://github.com/martinmarinov/TempestSDR/raw/master/documentation/acs-dissertation.pdf`, 2014.

47. Robert McEvoy, Michael Tunstall, Colin C Murphy, and William P Marnane. Differential power analysis of HMAC based on SHA-2, and countermeasures. In *International Workshop on Information Security Applications*, pages 317–332. Springer, 2007.

48. Cédric Murdica. *Physical security of elliptic curve cryptography. (Sécurité physique de la cryptographie sur courbes elliptiques).* PhD thesis, Télécom ParisTech, France, 2014.

49. Johannes Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller's firmware protection. In *11th {USENIX} Workshop on Offensive Technologies (WOOT 17)*, 2017.
50. Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. StaDy: Deep Integration of Static and Dynamic Analysis in Frama-C. 05 2014.
51. Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *Seventh IEEE International Conference on Software Testing, Verification and Validation*, Cleveland, United States, March 2014.
52. Pierre-Michel Ricordel and Emmanuel Duponchelle. Risques associés aux signaux parasitescompromettants : le cas des câbles DVI et HDMI. `https://www.sstic.org/media/SSTIC2018/SSTIC-actes/risques_spc_dvi_et_hdmi/SSTIC2018-Article-risques_spc_dvi_et_hdmi-duponchelle_ricordel.pdf`, 06 2018.
53. Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. Tame your annotations with metacsl: Specifying, testing and proving high-level properties. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs*, pages 167–185, Cham, 2019. Springer International Publishing.
54. Micah Elizabeth Scott. The FaceWhisperer for USB Glitching; or, Reading RFID with ROP and a Wacom Tablet. In *International Journal of Proof-of-Concept or Get The Fuck Out 13:4*, 2016.
55. Julien Signoles, Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, and Boris Yakobowski. Frama-C: a Software Analysis Perspective. volume 27, 10 2012.
56. Peter Smulders. The threat of information theft by reception of electromagnetic radiation from RS232 cables. *Computers & Security - COMPSEC*, 9:53–58, 02 1990.
57. Thom Does and Dana Geist, Cedric Van Bockhaven. *SDIO: new peripheral attack vector*, pages 1–42. 08 2016.
58. Ebo van der Laan, Erik Poll, Joost Rijneveld, Joeri de Ruiter, Peter Schwabe, and Jan Verschuren. Is java card ready for hash-based signatures? Cryptology ePrint Archive, Report 2018/611, 2018. `https://eprint.iacr.org/2018/611`.
59. Martin Vuagnoux and Sylvain Pasini. Compromising Electromagnetic Emanations of Wired and Wireless Keyboards. *USENIX Security Symposium*, 01 2009.
60. Colin D. Walter. Sliding Windows Succumbs to Big Mac Attack. In *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, pages 286–299. Springer, 2001.
61. Litao Wang and Bin Yu. Research on the Compromising Electromagnetic Emanations of PS/2 Keyboard. In *Proceedings of the 2012 International Conference on Communication, Electronics and Automation Engineering*, pages 23–29, 2013.
62. Litao Wang, ChangLin Zhou, and Bin Yu. Laboratory Test and Mechanism Analysis on Electromagnetic Compromising Emanations of PS/2 Keyboard. pages 657–660, 11 2012.