Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis

Jan Van den Herrewegen[1], David Oswald[1], Flavio D. Garcia[1] and Qais Temeiza[2]

[1] School of Computer Science, University of Birmingham, UK,
{jxv572,d.f.oswald,f.garcia}@cs.bham.ac.uk
[2] Independent Researcher, qaiskhaled744@gmail.com

Abstract. The bootloader of an embedded microcontroller is responsible for guarding the device's internal (flash) memory, enforcing read/write protection mechanisms. Fault injection techniques such as voltage or clock glitching have been proven successful in bypassing such protection for specific microcontrollers, but this often requires expensive equipment and/or exhaustive search of the fault parameters. When multiple glitches are required (e.g., when countermeasures are in place) this search becomes of exponential complexity and thus infeasible. Another challenge which makes embedded bootloaders notoriously hard to analyse is their lack of debugging capabilities.
This paper proposes a grey-box approach that leverages binary analysis and advanced software exploitation techniques combined with voltage glitching to develop a powerful attack methodology against embedded bootloaders. We showcase our techniques with three real-world microcontrollers as case studies: 1) we combine static and on-chip dynamic analysis to enable a Return-Oriented Programming exploit on the bootloader of the NXP LPC microcontrollers; 2) we leverage on-chip dynamic analysis on the bootloader of the popular STM8 microcontrollers to constrain the glitch parameter search, achieving the first fully-documented multi-glitch attack on a real-world target; 3) we apply symbolic execution to precisely aim voltage glitches at target instructions based on the execution path in the bootloader of the Renesas 78K0 automotive microcontroller. For each case study, we show that using inexpensive, open-design equipment, we are able to efficiently breach the security of these microcontrollers and get full control of the protected memory, even when multiple glitches are required. Finally, we identify and elaborate on several vulnerable design patterns that should be avoided when implementing embedded bootloaders.

Keywords: Embedded bootloader, fault-injection attacks, Return-Oriented Programming, binary analysis

1 Introduction

Embedded microcontrollers are at the foundation of our ever-increasingly digital world, steering innovation through data they collect and process. However, with their many advantages and uses come new security concerns. A single vulnerability in an embedded Microcontroller (μC) can lead to the compromise of all embedded systems using that particular type of chip.

An *embedded bootloader* is available on nearly all μCs and typically has full access to the chip's flash/RAM memories and its peripherals before loading the user application. Therefore, chip manufacturers integrate a security mechanism, which we call Code Readout Protection (CRP) in this paper, in the bootloader to safeguard the integrity and secrecy of the firmware binary (and all cryptographic secrets and intellectual property within it).

Because a variety of devices, ranging from automotive Electronic Control Units (ECUs) to Internet of Things (IoT), often use the same or similar µCs, they also include the same, generic bootloader—with no control over its development and no insight into its source code. Hence, the users of µCs find themselves at the mercy of the quality of the chip manufacturer's internal security testing, if any such procedures are in place at all.

Thus, however strong security primitives a specific system is built on, a vulnerable bootloader undoes all of this and makes the device susceptible to various attacks ranging from firmware readout to a full device compromise. Hence, bootloader security is of utmost importance for the integrity of the device and the secrecy of the firmware and the data within it. In notoriously secretive sectors such as the automotive industry, being able to extract firmware from ECUs allows for public scrutiny of the underlying security primitives, which are often of proprietary nature and insecure [dHG18, MV15, MV13, WVdHG+20, GOKP16]. The bootloader is the first piece of software that executes after reset and enforces the chip's CRP. Typically, it initialises essential peripherals (e.g., the internal clock) and loads and executes the application firmware. However, most bootloaders provide an external interface through a serial protocol, which typically uses an internal buffer to write and receive messages, making them prone to well-studied software vulnerabilities such as buffer overflows. These are aggravated by the lack of common mitigation techniques on embedded chips and especially in the bootloader, which often resides in a restricted memory area such as on-chip ROM and cannot be updated.

Hardware-based fault attacks induce a fault in on-chip computations, such as skipping an instruction, by changing the physical operating environment of the chip, e.g., the supply voltage. They do not rely on the presence of a software vulnerability. The literature covers a wide spectrum of hardware-based fault injection methods: the most widely-used techniques include voltage, optical, clock and electromagnetic fault injection. Optical fault attacks require extensive preparation such as decapsulating the chip [SA02], while electromagnetic fault attacks involve specialised hardware [CH17] and have a larger parameter space (e.g., probe positions). On the other hand, voltage fault injection ("glitching") does not require expensive lab equipment: open-source projects such as the Chipwhisperer [OC14] and the Generic Implementation ANalysis Toolkit (GIAnT) [Osw16] significantly lower the entry barrier for voltage glitching. A large amount of research has focused on devising algorithm-specific techniques to recover keys when faults are injected into cryptographic computations (cf. for instance [TMA11, BDL97, BS97, BBKN12, JT12]). Such research usually assumes a specific *fault model* (e.g., a bit-flip in a certain part of a cipher's internal state), and ignores the details of how the faults actually influence the binary code implementing the cryptographic algorithm. However, fault injection such as voltage glitching can often accurately target one particular instruction or memory location, and change the behaviour of normally secure code [MOG+20]. This makes bootloaders especially susceptible to said attacks: if for example the comparison instruction that checks if CRP is enabled can be manipulated, a single fault is sufficient to disable it. This in turn also compromises all cryptographic secrets stored on the µC, without the need for key recovery techniques specific to the implemented cipher. Still, finding the correct fault injection parameters to "hit" a particular location (e.g., in the bootloader binary) is challenging in a real-world attack scenario: most published attacks treat the µC and its firmware as a black box, and thus have to resort to an (optimised) brute force search of the parameter space [BFP19, CPB+13, PBJC14]. However, binary analysis of the bootloader binary could significantly reduce the search space. On the one hand, static analysis, which statically reconstructs the program control flow, reveals the possible bootloader execution paths to the CRP check. On the other hand, dynamic analysis, which leverages the bootloader execution, proves crucial in developing and testing CRP bypass exploits.

In this work, we draw on the target bootloader binaries to gain insight into the bootloader operation, and the enforcement of the CRP mechanism in particular. Furthermore,

we propose several novel methods that bridge the gap between binary analysis and fault injection. We show that our approach enables complex attacks that would be infeasible without analysis of the bootloader binary.

1.1 Our Contribution

Combining software and hardware vulnerabilities, we apply binary analysis techniques on bootloaders of three different chips, which ultimately allow us to bypass their security mechanisms with inexpensive, open-designed hardware. Our approach is widely applicable and, unlike intrusive silicon-level attacks, scales well. Our research reveals several vulnerable design and implementation patterns in a bootloader that makes it vulnerable to attacks in this paper and in the general literature. The contributions of this work are as follows:

- We extract and analyse four embedded bootloaders by three different manufacturers in detail. We show several software and hardware-based attacks on all bootloaders to bypass the CRP mechanisms. We perform all attacks with low-cost, open-design hardware, with a total cost of $\sim$ \$ 250.
- We show how hardware-based fault injection through voltage glitching benefits from static and dynamic binary analysis techniques. Several novel attack methods arise from this approach, including the selection of glitch parameters based on the input-dependent execution path. Furthermore, dynamic glitch profiling on the STM8 ultimately leads to the—to our knowledge—first successful multi-glitch attack applied to a real-world target.
- We demonstrate that software exploitation techniques such as Return-Oriented Programming (ROP), originally developed to bypass stack protection mechanisms for complex processors [Sha07] and later extended to embedded applications [FC08], are also relevant for the bootloader security of simple, constrained µCs.
- We systematise the vulnerability classes identified in bootloaders and describe typical anti-patterns that need to be avoided in the development of secure embedded bootloaders. We explain these anti-patterns in Section 6 and reference them throughout the paper as A1, A2, etc.
- All our tools, including the glitching hardware, will be made available as open source under a permissive license to aid the development of countermeasures and enable independent reproduction of our results[1].

1.2 Responsible Disclosure

We disclosed the vulnerability in the LPC series bootloader (Section 3) to NXP, and they acknowledged the issue. Although NXP previously had cautioned users about limitations of using CRP 1, they updated their developer guidance and recommend to set CRP 2 or 3, where this exploit is not possible. We disclosed the vulnerability in the STM8 series bootloader (Section 4) to ST, and they are currently evaluating the report. Finally, we did not contact Renesas regarding the issue in the 78K0 bootloader (Section 5), because the presence of a vulnerability was already reported in [BFP19].

1.3 Related Work

1.3.1 Hardware-based CRP bypass

Yuce et al. give an overview of commonly used types of hardware-controlled fault injection attacks in [YSW18]. Bypassing CRP on µCs has been an active field of research for

[1] Source available at https://github.com/janvdherrewegen/bootl-attacks

the past two decades, starting with the work presented in [Sko]. In 2002, Skorobogatov et al. showed that they could change a single bit in an SRAM array on a PIC16F84 microcontroller using an off-the-shelf laser pointer [SA02] and use this effect to bypass CRP. In [Sko10], Skorobogatov introduces flash memory bumping; a technique which uses optical fault-injection to force the data bus into a known state. Skorobogatov used this to extract read-protected memory from a NEC 78K0/S microcontroller and a Actel ProASIC3 FPGA. Similarly, researchers have targeted the fuses containing the readout protection bits with UV-C light [bun, OT17, SOR+14, Cesa, SA02]. In [OT17] Obermaier et al. bypassed the CRP of the STM32F0 by injecting a fault to flip one bit in the 16-bit value encoding the CRP level, which ranges from 0 to 2 (0 indicating protection disabled). Because CRP level 2 and 1 only differ by one bit, they downgraded the security level to CRP level 1, which enables the Serial Wire Debug (SWD) interface. The researchers then discovered two additional vulnerabilities in SWD—a timing-based race condition and the reconstruction of control flow based on SRAM contents—leading to a full attack on CRP 1.

Other research has shown that an attacker can disable CRP on chips through voltage fault injection [BFP19, Lim19, Ger, Cesa, Cesb, RND, Kra, Gil, WVdHG+20]. By changing the voltage level (*i.e.*, glitching) at the time the chip evaluates or loads the CRP value, they can bypass the protection mechanism and gain read/write access to the flash memory. For instance, the CRP value is initially loaded from flash into the RAM in NXP LPC µCs, and the respective checks can be manipulated through glitches [Ger] (anti-pattern A7). Roth et al. show the practical challenges of voltage glitching attacks: extracting the private key from an STM32-based hardware cryptocurrency wallet required a 3-month profiling phase to determine the correct glitch parameters (offset, width and voltage, cf. Section 2) through exhaustive search [RND]. Finally, Obermaier et al. analyse the security of a real and several counterfeit STM32 chips, uncovering various software and hardware vulnerabilities in the debug interface and chip design [OSM20].

To reduce the time for determining glitch parameters, several parameter optimisation strategies for voltage glitching have been proposed in the literature. Carpi et al. investigate several parameter search strategies on smartcards, of which a generic zoom-and-bound approach proves the most effective, although a genetic algorithm also generates promising results [CPB+13]. Picek et al. follow up on this in [PBJC14] and propose an improved solution based on a genetic algorithm. Finally, in [BFP19], Bozzato et al. continue along the same line and focus on the glitch shape in their genetic search strategy. Note that all these strategies treat the chip under attack as a black box and do not take the executed firmware binary into account, whereas the attacks we propose in this paper are inconceivable without analysis of the bootloader.

1.3.2 Software-based CRP bypass

Bootloaders providing reprogramming functionality at the very least require code to handle communication and to read/write flash memory. This has lead a number of published software vulnerabilities. Temkin et al. found a stack overflow in the USB code of NVIDIA's Tegra bootloader, leading to code execution and CRP bypass [Tem]. Goodspeed et al. make use of the fact that the bootloader is placed at a fixed memory location when blindly exploiting stack-based buffer overflows in embedded application code [GF09]. Timing dependencies in the code that verifies a password protecting the access to the bootloader (anti-pattern A5) have lead to the compromise of the M16C [q3k17] and MSP430 [Goo08] µCs. Additionally, through single-stepping instructions and reading RAM or register contents over a debug interface, researchers have bypassed the copy protection of NRF51822 [Bro], STM32F0 [OT17] and STM32F1 [SO] chips (anti-pattern A2).

Even when the debug interface is properly protected, it has been shown that attackers can recover sensitive data from RAM or data flash once the program flash is erased and thus the readout protection reset [Lau, Goo09] (anti-pattern A4). Similarly, if the chip

allows erasing per flash sector, an attacker can overwrite the boot section with a program that reads out the firmware [Mer10] (cf. Section 3.3 and anti-pattern A3).

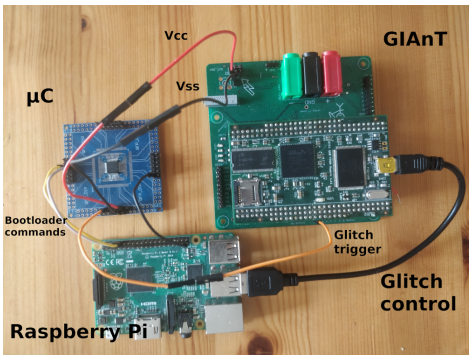
1.4 Outline

This paper is structured as follows: Section 2 describes our fault injection setup and attacker model. In Section 3, we present software vulnerabilities in the LPC1xxx bootloader, before continuing with a complex multi-glitch fault attack in Section 4 enabled through dynamic analysis. We show how symbolic execution benefits fault attacks in Section 5. We present common anti-patterns occurring in vulnerable bootloaders in Section 6, before concluding in Section 7.

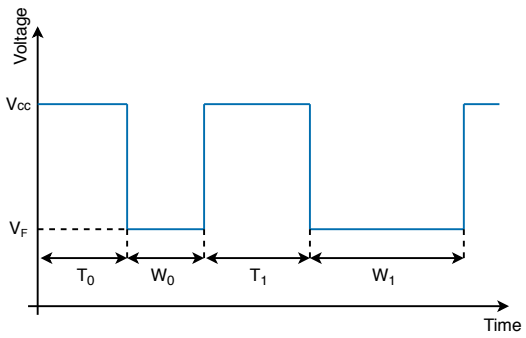
2 Setup and Attacker Model

Hardware setup For the voltage glitch attacks we use a modified version of the GIAnT [Osw16] for generating the glitch waveforms. A Raspberry Pi 3 interfaces with the on-chip bootloader of the target µC over Universal Asynchronous Receiver Transmitter (UART) (cf. Section 4) or Serial Peripheral Interface (SPI) (cf. Section 5). The Raspberry Pi also sends the glitch parameters to the GIAnT over USB, which in turn introduces a glitch on the chip's V_{CC}.

Glitch parameters Figure 1b illustrates the glitch parameters that we consider in this paper together with our notation conventions. We refer to the normal operating voltage as V_{CC}, and the voltage of the glitch as V_F. T_0 denotes the offset in time of the first glitch from the trigger point (e.g., chip reset or a command sent over the communication interface), while W_0 is the width of the first glitch. In case of multi-glitch attacks, T_1 is then the offset of the second glitch from the end of the first glitch, W_1 the width of the second glitch, etc. In case of attacks that use only a single glitch, we omit the indices and simply use T and W for offset and width, respectively. Note that due to limitations of the GIAnT, the time resolution of width and offset is 10 ns.



(a) Hardware setup for the voltage glitching attacks.



(b) Parameter conventions for voltage glitches.

Figure 1: The glitch setup and parameter conventions used throughout this paper.

Attacker model In this paper, we consider the *hardware fault attacker* as introduced in [YSW18], who can change the execution flow by introducing faults but cannot alter the bootloader binary. We assume that the adversary can obtain an identical, freely programmable chip or development board to profile the attacks and retrieve the bootloader

code. Both assumptions are common in practice for embedded devices, where a physical attacker (aiming at firmware recovery) is often part of the threat model and where development kits for most µCs are readily available. We show for each chip in this paper how an attacker can recover the bootloader binary by reading out the memory space in which the bootloader resides. This is trivial for the LPC1xxx (cf. Section 3) and STM8 (cf. section 4) bootloaders, which are always mapped in memory, while the 78k0 bootloader (cf. Section 5) is mapped after writing to a specific flash register. Thus, we assume a capable adversary can always recover the bootloader binary.

3 Finding software vulnerabilities through static and dynamic analysis: NXP LPC1xxx bootloader

In this section, we study the bootloader of the LPC1343 [NXPb] as an example of complex bootloaders containing software vulnerabilities (that do *not* require voltage glitching). We show how analysis of the bootloader binary is crucial for identifying and exploiting said issues, in particular a vulnerability that gives the adversary control over the stack and hence the program counter. We illustrate that the fixed memory layout facilitates exploitation with ROP [Sha07, FC08] techniques, which so far have received relatively little attention for embedded bootloaders.

The LPC1343 is from LPC1xxx family of NXP that encompasses a number of different chips based on an ARM Cortex M3 core. The results in this section likely generalise to other chips from this family as well. The LPC1343 bootloader implements two interfaces to access the chip's flash memory: (*i*) a character-oriented UART protocol and (*ii*) an emulation of a USB storage medium containing a single file representing the flash memory (cf. anti-pattern A10). We focus on UART, but note that the second interface also exposes substantial attack surface that we leave for future research.

The bootloader implements a CRP mechanism with five different access levels (cf. anti-pattern A8): NO_CRP, CRP 1, CRP 2, CRP 3, and an additional level called NO_ISP. These levels increasingly restrict the available bootloader commands: while NO_CRP gives full read/write access to the chip's memories, CRP 1 prevents read access to memory, restricts writes, and also disables the SWD interface. CRP 2 further restricts capabilities to essentially only a full chip erase, while CRP 3 permanently disables all programming functionality. NO_ISP disables the invocation of the bootloader, but leaves the SWD interface active, which can still be used to debug the processor and read memory (cf. anti-patterns A2, A9).

When CRP is enabled, the RAM region used by the bootloader (including the loaded CRP value) is not writable through the bootloader's "Write to RAM" command to protect against straightforward disabling of CRP. However, in this section, we show that the stack area in RAM is not protected on CRP 1, leading to a full bypass of the readout protection. During the responsible disclosure process, NXP confirmed that the vulnerability is present in all LPC1xxx series devices that do not incorporate a Memory Management Unit (MMU). Concretely, based on the datasheets, we believe the following device series to be vulnerable: (*i*) LPC800 (*ii*) LPC1100 (*iii*) LPC1200 (*iv*) LPC1300 (*v*) LPC1500 (*vi*) LPC1700 (*vii*) LPC1800 (note that some LPC1500/1700/1800 feature an MMU).

3.1 Analysis of the LPC1xxx Bootloader

The bootloader resides in the 16 kB ROM from address `0x1FFF0000` to `0x1FFF4000`. We extracted the bootloader by transmitting that memory range over UART on a *profiling* device and loaded it into IDA Pro. We also used the bootloader's "Read from RAM" command for dynamically analysing the behaviour of the code when necessary.

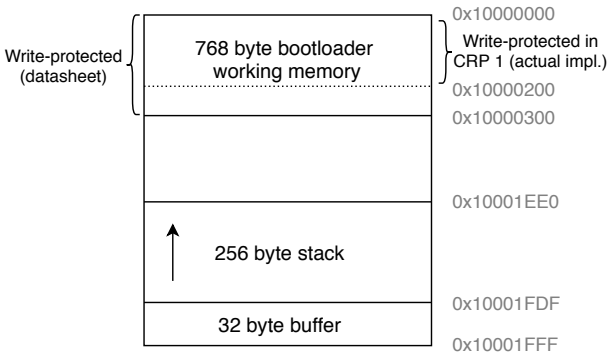


Figure 2: RAM memory layout of the LPC1343 bootloader, indicating write-protected memory areas according to actual implementation and datasheet.

The RAM memory layout (see Figure 2) (esp. of the bootloader) deserves special attention for finding and exploiting potential software vulnerabilities. Therefore, it is described in detail in the following. We first discovered that the bootloader uses the lower part of the chip's RAM up to `0x10000300` as its working memory, as also mentioned in NXP's documentation [NXPb], while it reserves the top 32 byte of the RAM for the flash programming commands. Addresses below the top 32 byte are allocated for the stack area. The stack grows towards lower addresses with a maximum size of 256 byte.

We confirmed that the CRP level is configured with a specific 32-bit value stored at address `0x2FC` in the chip's flash memory: `0x12345678` refers to CRP 1, `0x87654321` to CRP 2, and `0x43218765` to CRP 3, while other values leave the chip unprotected (*i.e.*, NO_CRP). Incidentally, this design anti-pattern A6 facilitates voltage glitching (cf. [Ger]), as we further discuss in Section 6. After reset, the bootloader loads the programmed CRP value from flash into RAM and uses the value in RAM for all subsequent CRP checks.

We then further statically analysed the bootloader commands, specifically the implementation of "Write to RAM" as shown in Listing 1. When the chip is configured in CRP 1, the "Write to RAM" command only proceeds if the target address is $\geq$ `0x10000200`, because memory below this address is used by the bootloader (and includes the buffered CRP level). Note that the start address of the RAM is loaded from a "hidden" configuration part of the flash as further described in [Dom16].

Interestingly, while the manual [NXPb, p. 329] claims that writes are permitted only above `0x10000300` and that the bootloader uses RAM up to to `0x1000025B`, the actual implementation permits writes above `0x10000200`, cf. Figure 2. We practically verified that we are able to write above this bound. We also noticed that the bootloader uses some variables located above `0x10000200` (specifically a pointer stored in `0x10000248` and referenced throughout the code). We have not further investigated whether this behaviour can be exploited, but it appears likely that this mismatch between specification and implementation could be misused.

```
1  ldr         r2, =0x438 // stores 0x10000000
2  ldr         r3, [sp, #0x28 + var_18]
3  ldr         r2, [r2]
4  adds        r2, #0xff
5  adds        r2, #0xff
6  adds        r2, #0x2  // add 0x200
7  cmp         r3, r2
8  // continue if address >= 0x10000200
9  bhs         continue_writing
10 // else set error code
11 movs        r4, #0x13
```

Listing 1: Check in the LPC bootloader for RAM write range starting at offset `0x1fff0d94`

Crucially, we found that the implementation of "Write to RAM" does not protect the stack: there are no checks at all if the target address is on the stack (*i.e.*, $\geq$ `0x10001EE0`). An attacker can exploit this to bypass the protection in CRP 1 and invoke the otherwise disabled "Read Memory" command, as further described in the following Section 3.2.

3.2 CRP 1 Bypass with Stack Overwrite

Because the stack is not write-protected, we can overwrite return addresses on it and hence control the program counter. We use ROP techniques to chain different gadgets in the bootloader to (*i*) jump into the "Read Memory" command and the (*ii*) jump back to the main command handler. Returning to the main command handler code prevents the bootloader from crashing and enables it to keep on receiving subsequent commands. First, we determined that the topmost return address on the stack is at `0x10001f54` while executing the "Write to RAM" command. We then write the following ROP chain, further illustrated in Figure 3, to the stack, starting at that address:

- `FB 0C FF 1F`: return address with a location behind the CRP check inside the "Read Memory" command handler (concretely, `0x1fff0cfa`[2]). This code then dumps 900 byte from a given starting address via the serial connection.
- When executed normally, the above code initially pushes seven registers (`R1`–`R7`) and ends in a corresponding `pop {r1, r2, r3, r4, r5, r6, r7, pc}` instruction. We therefore place seven 4-byte words on the stack. Crucially, the value later poped into `R3` controls the read target address (in the example ROP chain in Figure 3, this is set to `FC 02 00 00` to read the data starting at `0x2fc` in flash).
- `7F 11 FF 1F`: the control flow then returns to a gadget at `0x1fff117e`, which executes a `pop {r4, pc}` instruction. This is necessary to pad the overall stack layout (further explained below).
- This is followed by a 4-byte value to be poped into `R4`, and then the return address to proceed at:
- `81 0E FF 1F`: this is a `pop {r3, r4, r5, r6, r7, pc}` gadget. Due to the above padding, the remainder of the stack after this value is still in its original configuration and not overwritten, because it already contains valid values for `R3`–`R7` and a valid return address (in the main handler).

Applying this exploit repeatedly with different target addresses, we successfully read the complete flash and RAM within a few seconds. As the attack does not require voltage glitching, it can be carried out using a standard UART-USB cable, which is widely available for less than $ 5.

3.3 CRP 1 Bypass with Partial Flash Overwrite

In addition to the above vulnerability, LPC1xxx devices also allow the partial erasure and overwrite of single flash sectors in CRP 1 (cf. anti-pattern A3). In an application note, NXP states that "though it is unlikely, it is conceivable that an attacker with knowledge about a system could partially overwrite firmware in such a way as to gain read access to internal flash memory" [NXPa]. We confirmed that such an attack, akin to the methods demonstrated for PIC18F chips by [Mer10, GdKGVM12], is indeed possible for the LPC1343 and other similar chips from the LPC1xxx family. At high level, the attack proceeds as follows:

[2] Because the bootloader uses Thumb mode, all return addresses are incremented by 1 on the stack, so the written value is `0x1fff0cfb`

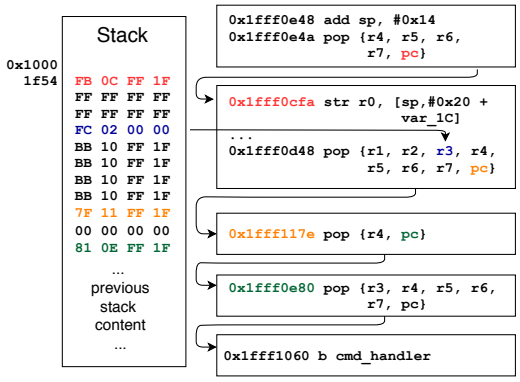


Figure 3: ROP chain for bypassing readout protection of LPC bootloader and reading 900 byte from any start address (here: `0x2fc`). The exploit is applied by invoking the "Write to RAM" command as: `W 268443476 172`

1. The attacker overwrites one flash sector with dumper code padded by `nop` instructions, which outputs the contents of all other sectors e.g., through a UART port.

2. The attacker then resets the device, and the original initially runs as normal. When a jump or call references the overwritten sector, the dumper code is invoked.

3. The attacker then uses a second, identical device to overwrite a different sector with a similar dumper to recover the contents of the sector overwritten in step 2.

We practically verified that this method can be applied to the LCP1343 configured in CRP 1. However, this technique depends on the characteristics of the target program: it only works if the attacker overwrites a sector that contains code that is executed during the operation of the target program. A natural choice would be to use the boot sector, however, this is not possible in case of the LPC bootloader: CRP 1 prevents erasure and overwriting of sector 0. Therefore, the attacker has to potentially overwrite multiple sectors one by one until a sector that contains executed code is found. Furthermore, if only sector 0 contains active code, this method cannot be used to recover the flash contents, while the attack from Section 3.2 is independent of the target program.

4 Glitching guided by dynamic analysis: The STM8 bootloader

ST's STM8 series chips feature a bootloader which provides the user with read, write and erase functionality [ST a]. It incorporates a readout protection mechanism to prevent an attacker from connecting to the bootloader. In this Section, we analyse the embedded bootloaders of two STM8 chips, namely the STM8L152C6 [ST c] and the automotive STM8AF6266 [STM], the latter being used in car immobilisers, a security sensitive anti-theft component [WVdHG+20]. As shown in [RND], where three researchers each spent three months voltage glitching the bootloader of an STM32F2 chip before they eventually succeeded, finding the correct glitch parameters to bypass readout protection is far from trivial.

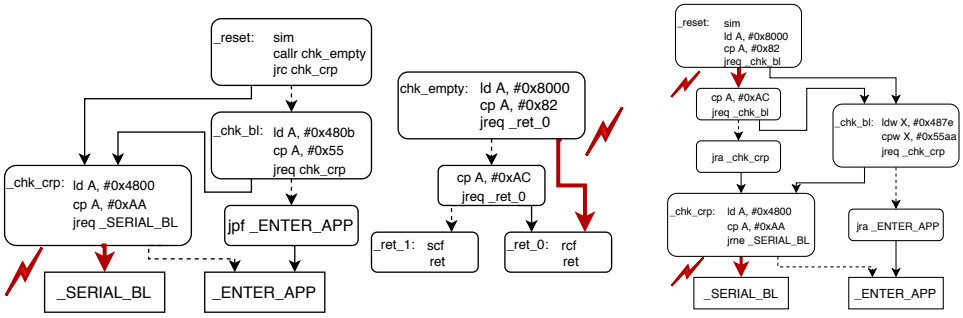
The lack of any feedback when voltage glitching the STM8 bootloader makes it an even more challenging feat. Unlike other bootloaders (cf. e.g., Sections 3 and 5), which allow certain commands and only restrict security-sensitive functionality such as flash reads and writes, the CRP on the STM8 prevents any communication with the bootloader

if enabled. Hence, until the bootloader activates the serial interface and thus pulls the UART receive pin high, we are completely in the dark as to what the injected glitch has achieved. An attacker could employ power analysis or other side channels to determine differences in instruction flow, however this typically requires many traces. Therefore, we use a technique to facilitate bootloader glitching, which we call *bootloader grey-box glitching*. By flashing security critical parts of the bootloader as a user application, we gain an invaluable advantage to profile the glitches individually. We then use that information to set up a complex double-glitch attack.

Because the bootloader is shared among all chips of the same version and likely more (according to [ST a], there are only four versions of the STM8 bootloader, whereas there are dozens of different STM8 chips), knowledge of its operation is invaluable when glitching other similar chips.

Attack assumptions We assume that the chip under attack is programmed (e.g., the first flash byte equals `0xAC` or `0x80`) and has the highest level of protection enabled (e.g., the bootloader is disabled and readout protection is active). We would like to emphasise that even though we only demonstrate this attack on the STM8AF6266 and STM8L152C6, we believe the methodology would generalize to other STM8 chips with a different bootloader version.

4.1 Bootloader Extraction and Analysis



(a) Entry point of the STM8L bootloader

(b) Entry point of the STM8A bootloader

Figure 4: Control flow diagrams of the STM8L and STM8A bootloaders. Jumps are displayed with a full line, whereas a dotted line implies a fallthrough path. Glitch paths for each µC with first flash byte `82` are indicated in red.

We obtained the two bootloaders of the analysed STM8 chips by connecting to the on-chip Serial Wire Interface Module (SWIM) [ST b] debug interface of a profiling device and issuing a read command for the address range `0x6000`–`0x8000`, which is the bootloader ROM according to the datasheets. The CRP on the STM8 works as follows: two option bytes, namely the CRP[3] and Bootloader Enable (BL) bytes stored in EEPROM control whether the bootloader activates or loads and executes the application code. If either the chip is empty (according to the datasheet, an STM8 chip is deemed empty if the flash byte on address `0x8000` does not equal `0x82` or `0xAC`) or the BL byte is set to a certain value[4], the bootloader continues to check the CRP byte. Finally, if the CRP byte indicates readout protection is disabled, the bootloader activates its serial interfaces (it supports

[3] ST refer to this byte as `ROP`, however to avoid any confusion with Return-Oriented Programming, we will call it CRP in this paper

[4] cf. datasheet of particular STM8 chip for exact value of option bytes

both UART and SPI) and waits for further programming commands. From here on, we will refer to this part of the bootloader as the *serial bootloader*. The serial bootloader performs no further CRP checks and thus grants full read and write access to the firmware, making the STM8 a suitable target for voltage glitching attacks.

Figure 4 depicts the control flow diagrams following the entry point of both the STM8L152C6 and STM8AF6266 bootloaders. If the µC is blank, or the BL option byte is set to `55(AA)`, it proceeds to check the CRP byte. The CRP on the STM8L152 is disabled if this byte equals `0xAA`, and only then the serial bootloader activates. On the STM8AF6266, readout protection is only enabled if the CRP byte equals `0xAA` (cf. anti-pattern A6).

4.2 Profiling Critical Bootloader Sections

Glitching a certain instruction on a µC often requires precision in the range of *nanoseconds*. Thus, even for a single fault, exhaustively searching the entire glitch parameter space—consisting of the glitch offset T, width W and voltage V_F as well as the normal operating voltage V_{CC}—quickly leads to a state explosion. The STM8 incorporates Brown-Out Reset (BOR) circuitry holding the chip under reset when V_{CC} drops below a user specified threshold (which they can set through an option byte in EEPROM). This threshold can range from 1.8 V to 3 V, so in order to circumvent the BOR circuit altogether, we keep the normal operating voltage at 3.3 V, leaving us with three unknown parameters. The goal of this phase is to optimise the glitch voltage and width by temporarily minimizing its timing aspect.

4.2.1 Critical Bootloader Sections

First, we define a Critical Bootloader Section (CBS) as a logically coherent unit of basic blocks which either check the BL or CRP option byte, or directly precede the serial bootloader. The STM8 bootloader checks both option bytes at the very beginning of its execution and does not continue unless BL is set and CRP is disabled. Thus, by design, identifying the critical sections on this µC does not require extensive reverse engineering.

Next, to *separately* run and fault each CBS on the real µC hardware with dynamic analysis capabilities, we insert the CBSs one by one into the custom user application stub depicted in Listing 2 and flash it as a normal user application onto the µC. On chips which are difficult to fault (e.g., the glitch only succeeds if both V_F and W lie within a narrow range which faults the instruction but does not reset the µC), this technique significantly reduces the parameter space. The stub pulls a GPIO pin high before executing the specific bootloader section on which we trigger the glitch, and indicates success by pulling a different GPIO pin high. We define success as reaching a basic block which the application would never enter in normal operation. As shown in Listing 2, a successful glitch in the `check_empty` section would result in the `scf` instruction being reached (which has been replaced in the template), indicating that the chip is empty and thus making the bootloader progress into the `check_CRP` section.

```
    PE_ODR |= 0x80 // generate trigger
chk_empty:
    ld A, #0x8000
    cp A, #0x82
    jreq _ret_0
    cp A, #0xAC
    jreq _ret_0
_ret_1:
    PE_ODR |= 0x01 // indicate success
_ret_0:
    ret
```

Listing 2: Flashing the `check_empty` CBS enabling the search for voltage and width parameters

4.2.2 Reduced glitch parameter search

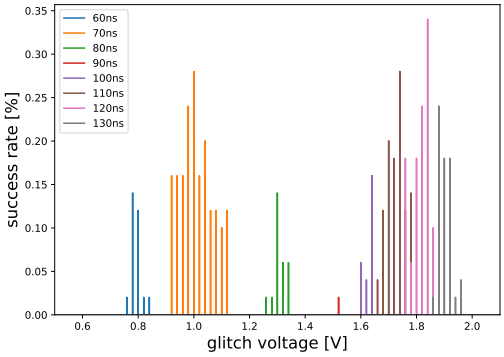


Figure 5: Success rate for glitching the *__enter_app* bootloader section (which immediately precedes the serial bootloader) on the STM8A at a constant offset of 0.34 µs

Intuitively, since the CBS executes immediately after the trigger, the offset search space reduces significantly. This allows us to essentially focus on the glitch voltage and width, which are mutually dependent: a deeper glitch, for example to $V_F = 0.5\,\mathrm{V}$, must be very short ($W \approx 50\,\mathrm{ns}$) in order not to reset the chip. Figure 5 shows the various width/voltage pairs which produce a successful glitch in the `_reset` block in the STM8A bootloader. Additionally, in this phase we get a rough estimate of the *glitch success rate*, however, due to other influences such as the glitch timing, we do not achieve the same rate when glitching the complete bootloader, as shown in Section 4.3.

4.3 Partially Attacking the Bootloader on Reset

In the second phase we move on to glitch the real bootloader, which poses several additional challenges. Firstly, even though the STM8 datasheet states that the µC restarts with an internal 2 MHz clock, we have noticed that the bootloader writes the Clock Master Divider Register (`CK_DIVR`), which controls the CPU frequency, just before loading the user application. Hence, we still have to determine the actual reset frequency of the bootloader. In a similar fashion to [RND], we achieved this by connecting a 30 Ω shunt resistor between the V_{ss} pin and the ground, in essence lifting the chip's ground. This allows us to measure the power consumption of the chip, which reveals the real clock frequency of the µC after reset as shown in Figure 6. A second issue arises due to the built-in Power-On Reset (POR) circuit, which generates a reset signal when the chip powers up. Hence, the bootloader does not execute immediately once we pull the reset pin high. Again, as shown in Figure 6, the power consumption gives us an estimate for when the bootloader starts operating and thus reduces the offset search space significantly. With the glitch voltage and width set to the values acquired in the first phase, we can now scan the offset search space. Before moving on to a fully locked chip, we set the `BL` and `CRP` option bytes individually, such that we only need one glitch to reach to the serial bootloader.

4.3.1 Comparison: STM8L vs. STM8A

Table 1 gives an overview of the various offsets leading to the serial bootloader on the STM8L with all possible combinations of the first flash byte and one option byte set. All glitches are aligned with either the rising or falling edge of a 2 MHz clock from reset, which can be attributed to a stable internal oscillator. In order to achieve a success rate above

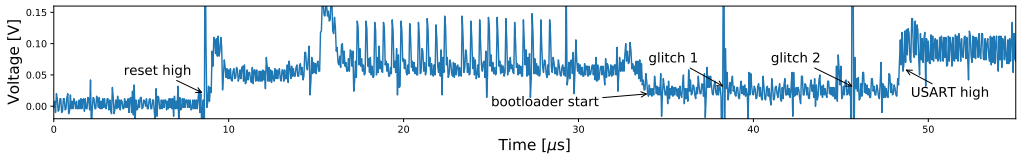


Figure 6: Power consumption of the STM8L152 upon reset measured with a 30 Ω shunt resistor.

0.1%, the glitches need to fall within the vicinity of 20 ns of the given offsets, proving the necessity of the earlier profiling phase.

Table 1: Glitch success rate and their offsets triggered on reset of the critical bootloader sections of the STM8L. Glitch voltage and width kept constant at $V_F = 1.84\,\text{V}$ and $W = 50\,\text{ns}$

CRP	BL	[8000]	section	T [µs]	success rate [%]
AA	00	82	chk_empty	**29.5**	**0.6**
			_chk_BL	35.75	0.1
		AC	chk_empty	**30.5**	**0.5**
			_chk_BL	36.25	0.1
				36.75	0.1
00	55	82	_chk_CRP	**38**	**0.6**
		AC		**39**	**0.5**

The bootloader code clarifies and helps predicting the glitch timing for different bytes on address `0x8000`. For instance, if the first flash byte equals `0xAC`, the glitch on the CRP byte check falls 1 µs, or two clock ticks, later due to the execution of an additional basic block in the `check_empty` subroutine. As to be expected due to a different internal design, performing the same experiment on the STM8A does not yield exactly the same result. Firstly, the reset time, e.g., the time it takes for the bootloader to start executing, on the STM8 is roughly 78 µs, opposed to 26 µs on the STM8L. In addition, the internal oscillator on the STM8A does not seem to be as stable as its STM8L counterpart, making the glitch offsets fall more within a range of ∼ 6 µs: glitches in the `_reset` section fall in the range of 79.67 µs–86.56 µs, whereas glitching the `_check_crp` section is most successful in the 86.68–93.54 µs range.

4.4 Full Double-Glitch Attack

Finally, once we have the correct parameters for the separate glitches, we have to combine them to attack a fully locked chip. The main challenge is that we receive no feedback until the `USART_RX` pin is pulled high indicating the activation of the serial bootloader. Table 2 gives the final glitch parameters for the chips we have investigated. What stands out is that the success rate for the double-glitch attack on the STM8L is substantially lower than 0.0036%, which is what we would expect when combining both individual glitch success rates. We attribute this peculiarity to the 3-stage pipeline on the STM8: the first glitch makes the bootloader jump directly to the `_chk_crp` section after returning from `chk_empty`, whereas in Section 4.2 the bootloader goes through the `_chk_bl` section first. Thus, the pipeline content differs at the point of the `_chk_crp` glitch in both scenarios. We repeated the partial attack for the µC with an 'empty' chip (e.g., with `00` as first flash byte) and CRP enabled, and indeed, the individual glitch success rate for the `_chk_crp` section in this scenario is 0.02%. For reference, 100k glitches, the average number of attempts needed to bypass the CRP on STM8AF6266, takes ∼ 2.5 min on our setup. Since

Table 2: Glitch parameters for fully locked STM8 chips with first flash byte `82` triggered on reset.

chip	T_0 [µs]	W_0 [ns]	T_1 [µs]	W_1 [ns]	succ. rate [%]
STM8L152C6	29.5	50	7.32	50	0.0001
STM8AF6266	80.75	120	3.91	120	0.001

the CRP check only occurs once at boot time, we reset the chip for each glitch attempt. Given the final success rates for the double glitch attacks, we would like to emphasise the difficulty of a black-box only approach (e.g., without inspecting the bootloader code) to glitch the CRP check. Without knowledge of the execution path and time between both target checks, a full search of the glitch parameters quickly leads to a state explosion. The lack of feedback until both glitches succeed only aggravates this, making a strong argument for our greybox approach.

5 Glitching guided by static analysis: Renesas 78K0 bootloader

From a program analysis perspective, the STM8 bootloaders described in Section 4 are fairly straightforward to analyse, because their CBS executes immediately after reset, and does not continue unless protection is disabled. In contrast, many other bootloaders allow basic functionality and only restrict certain security sensitive commands if CRP is active. Intuitively, glitching such bootloaders poses a number of different challenges. First, the communication protocol and bootloader code are often intertwined, making the identification of CBS (*i.e.*, where we want to inject a glitch) non-trivial. Moreover, different commands typically have their own unique handler code, making the glitch offset dependent on the specific handler and the content of the command, among others. In this section, we leverage symbolic execution to predict glitch offsets based on the execution path taken in the command handler code of the Renesas 78K0 bootloader. The 78K0 is a multi-purpose 8-bit low-power Renesas microcontroller which, similar to the STM8, is often used as the central µC on certain automotive immobiliser systems [WVdHG+20]. We evaluate the effectiveness of our white-box technique and compare it to the black-box attack by Bozzato et al. [BFP19].

5.1 78K0 Bootloader Extraction and Analysis

An external programmer can activate the bootloader on Renesas µCs by pulling the `FLMD0` pin high on reset. Subsequently, it can select either SPI or UART communication modes to interface with the on-chip bootloader [Rena]. The user can increase the level of security by clearing individual bits in the security byte, which respectively turn on write, block erase, chip erase and boot sector write protection on the 78K0 (cf. anti-patterns A8 and A9). Regardless of the security configuration, the bootloader always allows executing certain commands such as `verify` or `checksum`, which confirm and calculate a simple checksum over blocks of 256 byte respectively. Other commands, like `program` or `erase` only succeed if the respective security bit is `1`. All flash-related bootloader commands take a 3 byte start and end address as argument. With the 78K0 being an 8-bit µC, arithmetic on these 3 byte addresses is performed byte-wise.

The bootloader region is not mapped to memory during regular operation and thus cannot be read from a normal application. However, Renesas provides a "flash self programming library", which users can include in their application to interface with on-chip firmware that performs flash operations [Renc]. The library function `FlashInit` sets the µC into flash programming mode by first writing `0xA5` to the `FLMDPCMD` register, which

enables the writing of flash-specific registers. Next, it enables the flash programming mode by writing to the `FLMDMCR` register, which consequently maps the bootloader and on-chip firmware in a memory region marked as "reserved" in the datasheet.

Attack strategy Bozzato et al. propose three attacks to read out the full firmware on a 78K0 by exploiting several bootloader commands [BFP19]. A first attack glitches the `checksum` and `verify` commands to operate on 4 bytes instead of the minimum allowed 256 bytes, which allows an attacker to guess 4 bytes of firmware per successful glitch. They base the guess on a different glitch in the `checksum` command, which can leak bytes individually though is not completely accurate (e.g., the address or value of the leaked byte can be wrong). This is the only known attack on this µC which preserves the original firmware completely. Since all of these commands require a start and end address to operate on, all handlers initially call the same sanity check function. In contrast to Bozzato et al., who employ a genetic algorithm to determine the best glitch offset in a black-box manner [BFP19], we make use of the full knowledge of the bootloader binary.

5.2 Constraint-based Glitching

Symbolic execution is a widely used technique in software testing and program analysis [Kin76]. A symbolic executor tracks constraints over the range of values *symbolised* input variables can take along an execution path. It can use the constraints to generate a viable input value that will cause the execution of that path in a concrete execution. Logically, these constraints can also be used to verify if a given input value will cause the execution of a particular path. We leverage the latter technique to statically create classes of arguments which follow the same execution path through the targeted handler code, and thus will result in the same glitch offset. Our technique operates on the assembly language instructions, as opposed to lifting to an Intermediate Representation (IR). Unfortunately, state-of-the-art symbolic execution engines such as Angr [SWS+16] and KLEE [CDE08] do not provide out-of-the-box support for exotic architectures like the 78K0. However, the main reason for this decision is to retain low-level information such as instruction cycles. This proves crucial in predicting and classifying offsets for hardware-level attacks such as voltage glitching.

5.2.1 Constructing argument equivalence classes

Our framework uses the bootloader control flow graph to statically calculate the constraints along all paths through a certain command handler. The only prerequisites for our technique are: (*i*) Extraction of the bootloader control flow graph from a disassembler such as IDA Pro [Hex] or Ghidra [Nat]: the auto-analysis is usually adequate for simple serial protocols. (*ii*) Identification of the targeted bootloader command handler code: since the bootloader communicates through a serial interface, it typically suffices to follow the corresponding serial interrupt handler to find this. The extensive use of constants (*i.e.*, error codes) can further help locate handler code. We then perform a depth-first search from the command handler entry point to the target instruction. Since we are interested in the constraints along the complete path, we recursively repeat this process for all calls along the path. We mark the input variables to the command handler (cf. Appendix B) as symbolic and record their constraints along each path. We propagate the arguments for each conditional branch to check whether they originate from the initial input variables. Then, we use a simple constraint solver (*i.e.*, `python-constraint`) to obtain all viable paths through the command handler and their respective constraints. We then define an argument's equivalence class within the handler as follows:

Definition 5.1. Given a function f with input arguments $A_n, ..., A_0$, we define an *equivalence class* on this function as the set of all arguments which result in the same execution path through the function.

Finally, we use instruction cycle count information from the datasheet to calculate the number of cycles a path, each corresponding to one argument equivalence class, takes.

5.2.2 Practical application on the 78K0 bootloader

A similar pattern emerges in the handler code for all flash related commands: at the very start, each handler calls a function, shown in Listing 3, which processes both addresses provided in the command buffer. Concretely, it calculates which 1 kB flash block each address resides in and performs certain sanity checks on the arguments, e.g., if the end address falls within the on-chip flash range and whether the start address is lower than the end address.

```
1 int sanity_check(A0, A1) {
2     ...
3     b0 = get_block_no(A0);
4     b1 = get_block_no(A1);
5     if (cmp_addr(A0, Amax) > 0)
6         return -1;
7     if (cmp_addr(A0, A1) > 0)
8         return -1;
9     return 0;
10 }
```

Listing 3: Pseudocode of the `sanity_check` function, with arguments A_0 and A_1 the start and end address provided in the command buffer. A_{max} indicates the highest allowed flash address on the chip.

Figure 7 depicts a simplified control flow graph of the function that calculates the block number for a given flash address, called twice from within `sanity_check`. If the address is non-trivial (*i.e.*, not `0` or `ffff`), the code calculates the block number by merging the two least significant bytes into two 8-bit registers, and dividing these by `0x8` and `0x80` consecutively. Unsurprisingly, a 16-bit division on an 8-bit µC takes considerably longer than any other operation: on the 78K architecture, `udivw` completes after 25 clock cycles, whereas a simple `cmp` takes 4 cycles. The `cmp_addr` function follows a similar pattern but results in a smaller execution time difference, because it only compares its arguments byte by byte, starting from the most significant byte, and thus does not include any overly time-consuming operations such as division. For reference, we include the full assembly code in Appendix A.

5.3 Exploitation and Evaluation

In both the `checksum` and `verify` handlers, the targeted length check takes place directly after the code returns from the `sanity_check` function. Thus, applying our symbolic execution technique to each command handler yields nine sets of arguments (with $A_1 = A_0 + 3$, the smallest possible range on the 78K0) that result in a distinct execution path. Figure 8 shows the actual glitch offsets of each of those sets and reveals the main advantage of our technique compared to black-box predictions of a genetic algorithm: by basing the glitch offsets on execution paths, we can accurately predict glitch offsets of other sets of addresses by taking the execution cycles of each path into account.

Table 3 compares the cycle count for each equivalence class path through the `checksum` command handler obtained by our technique with the actual glitch offsets. Our technique calculates all viable execution paths from the start of the command handler to the basic block indicating a parameter error. Firstly, we note that even though in absolute terms the offset difference between classes is less than what we would expect from the cycle difference (e.g., a difference of 12 cycles between `0004` and `0000` translates into a real offset

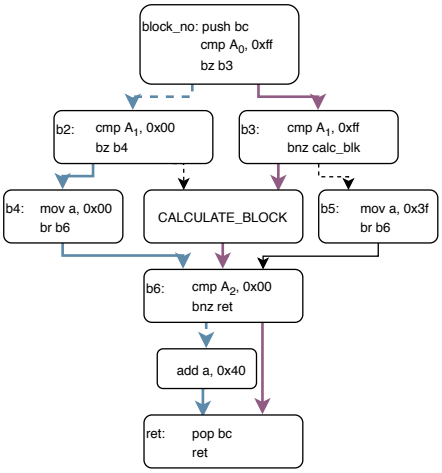


Figure 7: Simplified control flow graph of the `get_block_no` function, which takes as input a 3 byte address $A = A_2A_1A_0$. To illustrate, we depict example execution paths for the equivalence classes of `0x1004c` and `0x5ff`.

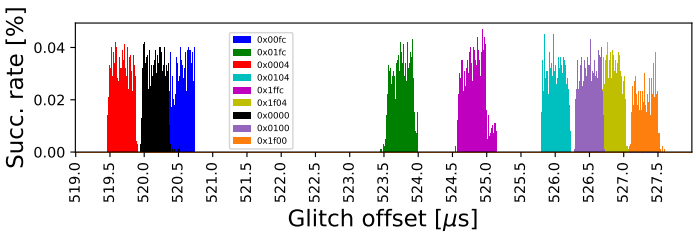


Figure 8: Glitch success rate and offset of the `checksum` command for the first address of each equivalence set. Glitch triggered on falling edge of the SPI clock pin on a 78K0/KC2 µC powered at 3 V clocked by 8 MHz internal oscillator. Glitch voltage and width constant at 0 V and 120 ns, respectively.

difference of $0.5\mu s$ instead of an expected $1.5\mu s$ @ 8 MHz). This could be attributed to the actual internal clock frequency, which depends on the input voltage, ambient temperature and the manufacturing tolerances of the particular chip. However, taking this discrepancy into account, the deltas of the real offsets and the calculated cycles are consistent. Only paths with the least significant byte of the end address equal to `ff` diverge from this. We attribute this to the glitch occurring on a different instruction to the other paths, due to there being a specific check for this byte and thus facilitating the path to the target basic block.

We compare our technique to the genetic algorithm published by Bozzato et al. in [BFP19]. They inject arbitrary glitch waveforms and tailor a genetic algorithm to optimise both the glitch offset and shape. As no source code is available for their approach, we have to rely on the published performance characteristics, which include results for two bootloader commands, namely `checksum` and `verify`. Table 4 summarises the results of the comparison of our technique with the arbitrary waveform injection and a more generic pulse injection, which resembles the GIAnT as used in our work. While we completely focus on the timing aspect of the glitches, and leave the actual shape, voltage and glitch width constant at $V_F = 0\,V$ and $W = 120$ ns, our technique performs better than the black-box approach with a normal pulse style glitch.

Table 3: Comparison of predicted number of cycles and actual glitch offset (first offset with a glitch success rate above 3%) for each equivalence class for the `checksum` command. Differences ($\frac{\Delta T}{\Delta c} = \frac{T_n - T_0}{c_n - c_0}$) are calculated from the offset T_0 of the first encountered equivalence class, `0004`

equiv. class	cycles (c)	$T[\mu s]$	$\frac{\Delta T}{\Delta c}$	equiv. class	cycles (c)	$T[\mu s]$	$\frac{\Delta T}{\Delta c}$
`0004`	454	519.49	n/a	`0104`	610	525.82	0.041
`0000`	466	519.97	0.040	`0100`	624	526.32	0.040
`00fc`	536	520.39	0.011	`1f04`	632	526.65	0.040
`01fc`	614	523.57	0.025	`1f00`	644	527.44	0.042
`1ffc`	642	524.63	0.027				

Table 4: Success rates of different glitch search strategies on `checksum` and `verify` commands. `wvf` and `pulse` use the same genetic algorithm, with the latter using a pulse-shaped glitch and the former also optimising the glitch waveform.

method	`checksum`	`verify`
`wvf` [BFP19]	4.2%	6.8%
`pulse` [BFP19]	2.8%	3.7%
`equiv` [this paper]	3.2%	4.3%

As for the arbitrary waveform injection, as shown in Figure 8, our success rate only surpasses 4.2% on 2% of the offsets, which appears to be the upper limit for our constant voltage and width. Additionally, because the pattern shown in Figure 8 emerges in all other flash-related handlers, glitching another command does not require searching for the parameters again. Assuming a 5 % success rate for guessing a byte (we have not included this glitch since it is very similar to the described `checksum` attack), extracting the firmware of an 8 kB chip would take $\sim$ 10 hours.

Discussion and limitations We would like to emphasise that, even though we only demonstrate it on the 78K0 bootloader, this technique applies to many other µCs as well. In a scenario where either the glitch offset is close to the trigger (e.g., in the range µs), or where we can match a code section in the bootloader to an externally observable anomaly (e.g., a faulty serial message checksum, which is always calculated on the contents of the message), we can compute the code paths to the targeted area. In cases where an initial search of the parameter space does not yield any positive results, or where the bootloader is hardened against glitching attacks (e.g., by performing redundant checks), a more fine-grained search is required. By statically calculating the possible execution flows from a set point to a targeted section, we can reduce the offset search space and shift our focus to other glitch parameters. However, for glitch offsets which occur too long after a trigger, this technique can lead to a state explosion and will require heuristics to reduce the number of possible paths. Finally, because we focus on the glitch timing, a combination of our technique to accurately predict the offset and a different strategy to optimise the remaining parameters might yield even better results.

6 Lessons Learned for Secure Bootloader Design

Bootloader vulnerabilities as presented in this paper are not easily mitigated. Both chip manufacturers and their clients benefit from having the possibility to alter the flash memory content of the chip in case of malfunction or a firmware upgrade. The bootloader is the ideal candidate to incorporate this functionality. However, this requires a significantly larger codebase, potentially leading to software vulnerabilities such as described in Sections 1.3 and 3. Even though hardware fault injection attacks are hard to prevent, manufacturers can

mitigate this risk by including extra components such as sensitive brownout detection [Gil] or a randomised internal clock [KK99] in the chip design. However, these mitigations undermine the overall performance of the chip and increase its costs, making a software-based approach—which would only affect the bootloader performance—attractive. Thus, in order to provide adequate reprogramming functionality without unnecessarily increasing the attack surface, we give several *anti-patterns* aimed at supporting the development of (more) secure bootloaders. These are design patterns we have observed both in our work as in previous research which *weaken* the protection mechanisms and thus must be *avoided* in an embedded bootloader.

A1 - Partial RAM write access in protected state: As shown in Section 3, µCs which have multiple protection levels often permit limited debug access to the chip's memory. Chips without an MMU must ensure that all bootloader memory and the area accessible to the user are separate. If no exploit mitigations are in place, a single compromise of the stack can jeopardise the whole system.

A2 - Partial leakage of memory or registers: Certain µCs still provide read access to RAM and/or registers when CRP is enabled. To exploit this issue, Obermaier et al. introduced cold boot stepping, which reconstructs the control flow of a program based on SRAM snapshots [OT17]. Furthermore, by single stepping a load instruction and manipulating CPU registers, Brosch recovers the firmware of a Bluetooth µC [Bro].

A3 - Partial flash overwrite: Having write access to one sector essentially gives an attacker read access to the chip. In addition to the attack described in Section 3.3, many systems have been broken by overwriting a flash sector with a program that reads out the entire memory of the chip [Mer10, GdKGVM12, WVdHG+20].

A4 - Incomplete or non-atomic chip erase: On many µCs, the CRP can be disabled through a full chip erase. However, as shown in [Lau, Goo09], in some cases that chip erase does not clear the full internal state (e.g., leaves RAM or data flash contents intact) and hence allows to recover information such as cryptographic keys stored in unerased memory. Furthermore, the erase process should be uninterruptible and atomic, that is to say that the bootloader should only disable the CRP at the very end of the erase process.

A5 - Non-constant time code: Timing leakage on password-protected bootloaders such as the Renesas M16C or TI MSP430 allows an attacker to recover the password byte-by-byte and gain access to the full flash memory [Goo08, q3k17].

A6 - Default to unprotected: A comparison is easier to glitch if only specific value(s) *enable* the readout protection. For example, the LPC1343 bootloader starts with disabled protection unless a few specific values are read. Therefore, a much bigger range of glitches can cause this desired effect, e.g., if a load is forced to all zero or all one.

A7 - Non-redundant check for readout protection: On µCs without hardware countermeasures against fault injection, it is typically possible to bypass a single check with high success rate, e.g., through voltage glitching. However, as evident from Section 4, the success rate decreases exponentially with each redundant check.

A8 - Large number of protection levels: This can confuse developers as to exactly what kind of protection each level relates to. It is not uncommon for developers to use a manufacturer-provided IDE, which in turn could hide the CRP details to the user and thus obscure which level is actually selected.

A9 - Separate On-Chip Debug (OCD) and CRP mechanisms: On many µCs such as the Renesas V850, 78K0R and 78K0, or the TI MSP430, the readout protection mechanism is unrelated to the OCD access, which the user needs to either secure in software [Renb] or by blowing a fuse [Tex]. Due to this ambiguous setup, programmers can lose track of some ways to access the on-chip memory, ultimately undoing all other protection measures.

A10 - Complex bootloader logic: Every feature of the bootloader's communication protocol broadens the attack surface and thus entails more software exploitation risks. For instance, the USB storage emulation of certain LPC µCs, which contains a FAT filesystem implementation [NXPb], could contain more issues, whereas the STM8 does not allow access to any bootloader commands if CRP is enabled. Besides, some Renesas µCs support up to three (UART, single-wire UART and SPI) communication interfaces in the same bootloader, which increases attack surface.

In addition to the above anti-patterns, there are other possible approaches and tradeoffs to be taken into account for secure bootloader design. They include:

Bootloader read protection Some devices incorporate read protection of the bootloader memory space, preventing readout of the bootloader binary. For instance, recent chips may incorporate eXecute-Only-Memory (XOM), which utilises additional hardware to restrict a certain memory area (e.g., the bootloader section) to instruction fetches and disallows any read or write access. This would mitigate the attacks described in this paper, because access to the bootloader binary is a prerequisite for each. However, Schink et al. analyse several XOM implementations in [SO19] and bypass the restrictions in each case to recover the protected code.

In-field & field-return analysis In certain scenarios, e.g., to perform dynamic in-field testing or to determine the cause of device failure, the manufacturer requires privileged access to the chip. This directly contradicts the proposed mitigations and anti-patterns, which are intended to lock down the chip as much as possible. Specifically, the manufacturer must balance anti-patterns A1 and A4 with leaving sufficient debug capabilities on the chip for these scenarios. An appropriate solution, though going against A9, would be to have a separate debug mechanism which only the manufacturer can access and is protected when the device goes into production.

7 Conclusion

Voltage fault injection is a powerful technique that has been widely studied in the context of cryptographic primitives, but comparatively little research has been done in the context of traditional software security. This paper has brought advancements in binary analysis such as symbolic execution and dynamic analysis into the low-level hardware security domain. Furthermore, we also show that exploitation techniques like ROP apply in the context of embedded bootloaders. We demonstrated that symbolic execution can be leveraged to define argument equivalence classes based on their respective execution paths. This gives us valuable insights into their execution times, which allows us to produce precisely-targeted glitch offsets to aid the glitch parameter search as we demonstrated on the 78K0 bootloader. By flashing parts of the bootloader as application code and thus enabling dynamic glitch profiling on the target hardware itself, we have presented here the first fully documented multi-glitch attack against the widely used STM8 family of microcontrollers, which gives full access to the device's memory. The techniques presented in this paper are applicable to other families of microcontrollers and can be fully implemented using inexpensive open-designed hardware.

Acknowledgments

This research is partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/S030867/1, EP/R008000/1 and by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM).

References

[BBKN12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[BDL97] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the Importance of Checking Computations. In *Proceedings of Eurocrypt'97*, pages 37 – 51, 1997.

[BFP19] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):199–224, 2019.

[Bro] Kris Brosch. Firmware dumping technique for an ARM Cortex-M0 SoC . https://blog.includesecurity.com/2015/11/NordicSemi-ARM-SoC-Firmware-dumping-technique.html.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology – CRYPTO'97*, pages 513 – 525, 1997.

[bun] bunniestudios. Hacking the PIC 18F1320. https://www.bunniestudios.com/blog/?page_id=40.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[Cesa] Silvio Cesare. Adventures in glitching PIC microcontrollers to defeat firmware copy protection. https://2015.kiwicon.org/the-con/talks/#e203.

[Cesb] Silvio Cesare. Defeating Firmware Copy Protection Using Glitching. https://www.youtube.com/watch?v=UGlm_I-RI-U.

[CH17] Ang Cui and Rick Housley. BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*, 2017.

[CPB+13] Rafael Boix Carpi, Stjepan Picek, Lejla Batina, Federico Menarini, Domagoj Jakobovic, and Marin Golub. Glitch It If You Can: Parameter Search Strategies for Successful Fault Injection. In *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, pages 236–252, 2013.

[dHG18] Jan Van den Herrewegen and Flavio D. Garcia. Beneath the Bonnet: A Breakdown of Diagnostic Security. In *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, pages 305–324, 2018.

[Dom16] Domen Puncer Kugler. LPC13xx Bootloader Reverse Engineering. https://github.com/domenpk/lpc13xx_boot_analysis, Jan 2016.

[FC08] Aurélien Francillon and Claude Castelluccia. Code Injection Attacks on Harvard-Architecture Devices. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 15–26, 2008.

[GdKGVM12] Flavio D. Garcia, Gerhard de Koning Gans, Roel Verdult, and Milosch Meriac. Dismantling iClass and iClass Elite. In *17th European Symposium on Research in Computer Security (ESORICS 2012)*, volume 7459 of *Lecture Notes in Computer Science*, pages 697–715. Springer-Verlag, 2012.

[Ger] Chris Gerlinsky. Breaking Code Read Protection on the NXP LPC-family Microcontrollers. https://recon.cx/2017/brussels/resources/slides/RECON-BRX-2017-Breaking_CRP_on_NXP_LPC_Microcontrollers_slides.pdf.

[GF09] Travis Goodspeed and Aurélien Francillon. Half-Blind Attacks: Mask ROM Bootloaders Are Dangerous. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT 2009, page 6, USA, 2009. USENIX Association.

[Gil] Brett Giller. Implementing Practical Electrical Glitching Attacks. https://www.blackhat.com/docs/eu-15/materials/eu-15-Giller-Implementing-Electrical-Glitching-Attacks.pdf.

[GOKP16] Flavio D. Garcia, David Oswald, Timo Kasper, and Pierre Pavlidès. Lock It and Still Lose It - On the (In)Security of Automotive Remote Keyless Entry Systems. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, August 2016. USENIX Association.

[Goo08] Travis Goodspeed. Cracking the MSP430 BSL. https://fahrplan.events.ccc.de/congress/2008/Fahrplan/attachments/1191_goodspeed_25c3_bslc.pdf, 2008.

[Goo09] Travis Goodspeed. A 16 Bit Rootkit, and Second Generation Zigbee Chips. https://www.blackhat.com/presentations/bh-usa-09/GOODSPEED/BHUSA09-Goodspeed-ZigbeeChips-SLIDES.pdf, 2009.

[Hex] Hex-Rays. IDA Pro. accessed: 2020/07/27.

[JT12] Marc Joye and Michael Tunstall, editors. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.

[KK99] Oliver Kömmerling and Markus G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In Scott B. Guthery and Peter Honeyman, editors, *Proceedings of the 1st Workshop on Smartcard Technology, Smartcard 1999, Chicago, Illinois, USA, May 10-11, 1999*. USENIX Association, 1999.

[Kra] Kraken. Kraken Identifies Critical Flaw in Trezor Hardware Wallets. https://blog.kraken.com/post/3662/kraken-identifies-critical-flaw-in-trezor-hardware-wallets/.

[Lau] Adam Laurie. Atmel SAM7XC Crypto Co-Processor key recovery. https://adamsblog.rfidiot.org/2013/02/atmel-sam7xc-crypto-co-processor-key.html.

[Lim19] LimitedResults. Pwn the ESP32 Forever: Flash Encryption and Sec. Boot Keys Extraction. https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/, November 2019.

[Mer10] Milosch Meriac. Heart of Darkness - exploring the uncharted backwaters of HID iCLASS security. Technical report, Bitmanufaktur GmbH, 2010.

[MOG+20] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[MV13] Charlie Miller and Chris Valasek. Adventures in Automotive Networks and Control Units. *Def Con*, 21:260–264, 2013.

[MV15] Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. *Black Hat USA*, 2015:91, 2015.

[Nat] National Security Agency (NSA). Ghidra. accessed: 2020/07/27.

[NXPa] NXP. AN10968: Using Code Read Protection in LPC1100 and LPC1300. https://www.nxp.com/docs/en/application-note/AN10968.pdf.

[NXPb] NXP. UM10375: LPC1311/13/42/43 User manual. https://www.nxp.com/docs/en/user-guide/UM10375.pdf.

[OC14] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research. In *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, pages 243–260, 2014.

[OSM20] Johannes Obermaier, Marc Schink, and Kosma Moczek. One Exploit to Rule them All? On the Security of Drop-in Replacement and Counterfeit Microcontrollers. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020.

[Osw16] David Oswald. Generic Implementation ANalysis Toolkit, 2016. available online at https://sourceforge.net/projects/giant.

[OT17] Johannes Obermaier and Stefan Tatschner. Shedding too much Light on a Microcontroller's Firmware Protection. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*, 2017.

[PBJC14] Stjepan Picek, Lejla Batina, Domagoj Jakobovic, and Rafael Boix Carpi. Evolving genetic algorithms for fault injection attacks. In *37th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2014, Opatija, Croatia, May 26-30, 2014*, pages 1106–1111, 2014.

[q3k17] q3k. Renesas M16C programmer. https://github.com/q3k/m16c-interface, July 2017.

[Rena] Renesas Electronics. *78K0/Kx2 Flash Memory Programming.*

[Renb] Renesas Electronics. *78K0/Kx2 User's Manual: Hardware.*

[Renc] Renesas Electronics. *Code Flash Libraries (Flash Self Programming Libraries).*

[RND] Thomas Roth, Dmitry Nedospasov, and Josh Datko. wallet.fail. https://fahrplan.events.ccc.de/congress/2018/Fahrplan/events/9563.html.

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 2–12, 2002.

[Sha07] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[Sko] Sergei P. Skorobogatov. Copy Protection in Modern Microcontrollers. https://www.cl.cam.ac.uk/~sps32/mcu_lock.html.

[Sko10] Sergei Skorobogatov. Flash Memory 'Bumping' Attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, pages 158–172, 2010.

[SO] Marc Schink and Johannes Obermaier. Exception(al) Failure - Breaking the STM32F1 Read-Out Protection. https://blog.zapb.de/stm32f1-exceptional-failure/.

[SO19] Marc Schink and Johannes Obermaier. Taking a Look into Execute-Only Memory. In Alex Gantman and Clémentine Maurice, editors, *13th USENIX Workshop on Offensive Technologies, WOOT 2019, Santa Clara, CA, USA, August 12-13, 2019*. USENIX Association, 2019.

[SOR+14] Daehyun Strobel, David Oswald, Bastian Richter, Falk Schellenberg, and Christof Paar. Microcontrollers as (In)Security Devices for Pervasive Computing Applications. *Proceedings of the IEEE*, 102(8):1157–1173, 2014.

[ST a] ST Microelectronics. STM8 bootloader. https://www.st.com/content/ccc/resource/technical/document/user_manual/e4/83/c1/d6/ee/d8/49/b8/CD00201192.pdf/files/CD00201192.pdf/jcr:content/translations/en.CD00201192.pdf.

[ST b] ST Microelectronics. *STM8 SWIM communication protocol and debug module.*

[ST c] ST Microelectronics. *STM8L151x4, STM8L151x6, STM8L152x4, STM8L152x6 datasheet.*

[STM] STMicroelectronics. *STM8AF Series.*

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157, 2016.

[Tem] Katherine Temkin. Vulnerability Disclosure: Fusée Gelée. https://github.com/Qyriad/fusee-launcher/blob/master/report/fusee_gelee.md.

[Tex] Texas Instruments. *MSP430 Programming With the JTAG Interface.*

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In Claudio A. Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 224–233. Springer, 2011.

[WVdHG+20] Lennert Wouters, Jan Van den Herrewegen, Flavio D. Garcia, David Oswald, Benedikt Gierlichs, and Bart Preneel. Dismantling DST80-based Immobiliser Systems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):99–127, Mar. 2020.

[YSW18] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *J. Hardware and Systems Security*, 2(2):111–130, 2018.

A Full assembly of the `get_block_no` and `cmp_addr` functions from the 78K0 bootloader

```
1 get_block_no:   push    bc
2                 cmp     A0, #0FFh
3                 bz      _b3
4                 cmp     A1, #00h
5                 bz      _b4
6                 mov     a, A0
7                 xch     a, X
```

```
                    mov     a, A1
                    br      calc_blk
_b4:
                    mov     a, #00h
                    br      _b6
_b3:
                    cmp     A1, #0FFh
                    bnz     enter_calc_blk
                    mov     a, #3Fh
                    br      _b6
enter_calc_blk:
                    mov     a, A0
                    add     a, #01h
                    xch     a, X
                    mov     a, A1
                    addc    a, #00h
calc_blk:
                    mov     C, #08h
                    divuw   C
                    mov     C, #80h
                    divuw   C
                    xch     a, X
                    cmp     A0, #0FFh
                    bnz     _b6
                    dec     a
_b6:
                    cmp     A2, #01h
                    bnz     return
                    add     a, #40h

return:
                    pop     bc
                    ret
```

Listing 4: Assembly of the `get_block_no` function in the 78K0 bootloader. Registers `a`, `b` and `c` are depicted in lower case, while the input address $A = A_2A_1A_0$ (with each address byte referencing a location in RAM) is given in uppercase

```
cmp_addr:           mov     a, A2
                    cmp     a, B2
                    bc      ret_0
                    bnz     loc_874
                    mov     a, A1
                    cmp     a, B1
                    bc      ret_0
                    bnz     loc_874
                    mov     a, A0
                    cmp     a, B0
                    bc      ret_0
                    bnz     loc_874
ret_0:
                    clr1    CY
                    ret
ret_1:
                    set1    CY
                    ret
```

Listing 5: Assembly of the `cmp_addr` function in the 78K0 bootloader with input addresses $A = A_2A_1A_0$ and $B = B_2B_1B_0$.

B Example path through the `checksum` command handler

```
addr  Instruction                                cycles

1aa8  call !sanity_check_addr                    7
 892  set1 flash_getbyte_reg.03h                 4
 895  call !sub_FE9                               7
 fe9  mov A, #0FFh                                4
 feb  call !sub_103F                              7
103f  mov !d_resp_0, A                           8
1042  mov A, #01h                                4
1044  mov !byte_FE14, A                          8
1047  ret                                        6
 fee  ret                                        6
 898  movw HL, #msg_buffer_b1                    8
```

```
14  89b  mov A, [HL+03h]                    8
15  89d  mov addr_H, A                      4
16  89f  mov end_addr_H, A                  4
17  8a1  mov A, [HL+04h]                    8
18  8a3  mov addr_M, A                      4
19  8a5  mov end_addr_M, A                  4
20  8a7  mov A, [HL+05h]                    8
21  8a9  mov addr_L, A                      4
22  8ab  mov end_addr_L, A                  4
23  8ad  call !get_block_no                 7
24 117d  push BC                            4
25 117e  cmp addr_L, #0FFh                  6
26 1181  bz loc_1193                        6
27 1193  cmp addr_M, #0FFh                  6
28 1196  bnz loc_119C                       6
29 119c  mov A, addr_L                      4
30 119e  add A, #01h                        4
31 11a0  xch A, X                           2
32 11a1  mov A, addr_M                      4
33 11a3  addc A, #00h                       4
34 11a5  mov C, #08h                        4
35 11a7  divuw C                            25
36 11a9  mov C, #80h                        4
37 11ab  divuw C                            25
38 11ad  xch A, X                           2
39 11ae  cmp addr_L, #0FFh                  6
40 11b1  bnz loc_11B4                       6
41 11b3  dec A                              2
42 11b4  cmp addr_H, #01h                   6
43 11b7  bnz loc_11BB                       6
44 11bb  pop BC                             4
45 11bc  ret                                6
46  8b0  mov end_block_no_0, A              4
47  8b2  mov A, [HL+00h]                    8
48  8b4  mov addr_H, A                      4
49  8b6  mov A, [HL+01h]                    8
50  8b8  mov addr_M, A                      4
51  8ba  mov A, [HL+02h]                    8
52  8bc  mov addr_L, A                      4
53  8be  call !get_block_no                 7
54 117d  push BC                            4
55 117e  cmp addr_L, #0FFh                  6
56 1181  bz loc_1193                        6
57 1183  cmp addr_M, #00h                   6
58 1186  bz loc_118F                        6
59 118f  mov A, #00h                        4
60 1191  br loc_11B4                        6
61 11b4  cmp addr_H, #01h                   6
62 11b7  bnz loc_11BB                       6
63 11bb  pop BC                             4
64 11bc  ret                                6
65  8c1  mov start_block_no, A              4
66  8c3  mov A, end_block_no_0              4
67  8c5  sub A, start_block_no              4
68  8c7  inc A                              2
69  8c8  mov diff_block_no, A               4
70  8ca  call !comp_end_addr                7
71  85a  mov A, end_addr_H                  4
72  85c  cmp A, max_flash_addr_H            6
73  85e  bc loc_872                         6
74  860  bnz loc_874                        6
75  862  mov A, end_addr_M                  4
76  864  cmp A, max_flash_addr_M            6
77  866  bc loc_872                         6
78  872  clr1 CY                            2
79  873  ret                                6
80  8cd  bc loc_874                         6
81  8cf  mov A, addr_H                      4
82  8d1  cmp A, end_addr_H                  6
83  8d3  bc loc_8E7                         6
84  8d5  bnz loc_8E5                        6
85  8d7  mov A, addr_M                      4
86  8d9  cmp A, end_addr_M                  6
87  8db  bc loc_8E7                         6
88  8dd  bnz loc_8E5                        6
89  8df  mov A, addr_L                      4
90  8e1  cmp A, end_addr_L                  6
91  8e3  bc loc_8E7                         6
```

```
92  8e7   clr1 CY                                    2
93  8e8   ret                                       6
94 1aab   bc error_5                                6
95 1aad   cmp addr_L, #00h                          6
96 1ab0   bnz error_5                               6
```

Listing 6: Example path through the checksum command handler for equivalence class `fc`. Our technique marks input bytes (`[HL + 00]` , . . . , `[HL + 05]`) as symbolic and builds up the constraints along the path.