Shedding too much Light on a Microcontroller's Firmware Protection

Johannes Obermaier
Fraunhofer Institute AISEC
johannes.obermaier@aisec.fraunhofer.de

Stefan Tatschner
Fraunhofer Institute AISEC
stefan.tatschner@aisec.fraunhofer.de

Abstract

Almost every microcontroller with integrated flash features firmware readout protection. This is a form of content protection which aims at securing intellectual property (IP) as well as cryptographic keys and algorithms from an adversary. One series of microcontrollers are the STM32 which have recently gained popularity and thus are increasingly under attack. However, no practical experience and information on the resilience of STM32 microcontrollers is publicly available. The paper presents the first investigation of the STM32 security concept, especially targeting the STM32F0 sub-series. Starting with a conceptual analysis, we discover three weaknesses and develop them to vulnerabilities by demonstrating corresponding Proofs-of-Concept. At first, we discover that a common security configuration provides low protection which can be exploited using our Cold-boot Stepping approach to extract critical data or even readout-protected firmware. Secondly, we reveal a design weakness in the security configuration storage which allows an attacker to downgrade the level of firmware protection, thereby enabling additional attacks. Thirdly, we discover and analyze a hardware flaw in the debug interface, attributed to a race condition, that allows us to directly extract read-protected firmware using an iterative approach. Each attack requires only low-priced equipment, thereby increasing the impact of each weakness and resulting in a severe threat altogether.

1 Introduction

Commercial grade microcontrollers are deployed in countless applications, ranging from industrial systems over automotive control units up to end-user devices. As their capabilities steadily increases, the complexity of their tasks rises and thus their firmware gets more sophisticated.

While previous devices were deployed in stand-alone applications, current systems may be part of large sensor networks or may interact with the Internet-of-Things. Thus, these systems contain valuable Intellectual Property (IP), such as sophisticated measurement or control algorithms. The devices may be license-locked and contain cryptographic material. Altogether, these devices are accompanied by large investments into software development.

At the same time, gaining access to these assets becomes more worthwhile for adversaries. Product piracy has emerged to a large threat, where competitors clone products and cause financial damage to the affected company [7]. As those attackers operate covertly without publishing their exploits, vulnerabilities are often surviving long. Nevertheless, professional researchers as well as hobbyists have also broken several systems in the past, often due to the underlying insufficient hardware security [16, 17]. Especially, Skorobogatov et al. have shown that the chosen security concepts of hardware manufacturers often do not cover all corner cases [13], have weaknesses [11], hidden functions, or even backdoors [12].

Many older microcontrollers were extensively tested for security and often exploited in the last few years. Therefore, the industry shows growing interest in more recent microcontrollers, including the ARM Cortex-M based STM32 series. The wide deployment of these devices finally raised interest into the provided security, mostly in terms of firmware protection.

There are no penetration testing results for STM32 publicly available. Thus, giving a statement regarding the protection of IP is impossible, despite it is often requested. Therefore we undertake a thorough security analysis of the STM32 series in which we answer the crucial question: Does the STM32 series provide a sufficiently strong security concept for firmware protection and, if not, how complex is the exploitation of weaknesses? We start with a high-level conceptual analysis of the security configuration and gradually dig deeper into the hardware implementation. Thereby we cover a large spectrum of attacks by combining software-based as well as hardware-based approaches.

Since the STM32 series is large and contains several sub-series, we will limit our experiments primarily to the STM32F0 family. This series features entry-level microcontrollers at moderate cost for a wide range of commercial products. Where applicable, an estimation of the impact onto other STM32 series is given.

In this paper, we present the following novel contributions:

- conceptual analysis of three STM32F0 security core components
- detection of three weaknesses and presentation of corresponding ideas for their exploitation
  → **Cold-Boot Stepping:** Enforcing single-stepping under limited debugging capabilities in Readout Protection (RDP) Level 1
  → **Security Downgrade:** Leveraging the lock-level design to downgrade the firmware protection setting
  → **Debug Interface Exploit:** Discovery of a race condition in the debug interface in RDP Level 1
- a Proof-of-Concept (PoC) for each weakness, which develops them into three vulnerabilities
- discussion of the impact of each vulnerability on system security and possible countermeasures

In order to encourage discussion and to ensure reproducibility of our results, we publicly provide the source files and additional materials for the PoCs in Section 6.

We informed ST Microelectronics about our findings, using an adapted responsible disclosure approach.

2 STM32 Security Concept

The flash Readout Protection is the key component of the security concept [15] and is incorporated in every device. It protects the system's firmware, stored in flash memory, against unauthorized readout. Depending on the chip family, there are additional security mechanisms, e.g., a Memory Protection Unit (MPU) and privileged/unprivileged execution modes [14]. Altogether, these technologies aim at enhancing system security.

The flash readout protection is the root of system security, since it protects the firmware as well as the system's security configuration. Therefore, breaking the readout protection mechanism will fully compromise security.

2.1 Flash Readout Protection Levels

The flash readout protection [15] concept consists of three selectable RDP Levels 0, 1 and 2. The protection increases with the level.

RDP Level 0 is the original configuration and imposes no restrictions. The debug interface is active and allows full access to the device. Usually, this level is only used for development.

RDP Level 1 keeps the debug interface active but restricts access to flash memory. As soon as a debugger is connected, flash memory is locked. It can neither be read out directly nor indirectly via DMA nor is the CPU able to execute code from it. Protection can be upgraded to RDP Level 2 but can also be downgraded to RDP Level 0 at the cost of losing all flash memory contents.

RDP Level 2 is restricted the most and provides highest security. Debug access is completely disabled by shutting down the debug interface permanently. The level is irreversible and cannot be downgraded.

Despite RDP Level 2 offers the best protection, RDP Level 1 is still in use. Experience shows, that companies dislike the idea of locking down their devices completely, since it impedes fixing buggy and failed devices. Furthermore, ST warns in their datasheet [15], that defective part analysis cannot be done on devices set to RDP Level 2. Additionally, the STM32F1 series, for example, lacks support for RDP Level 2. Altogether, this results in devices set to RDP Level 1.

RDP Level 1 raises special interest, as it remains unclear, whether this configuration can be considered secure or insecure—a question that is investigated in Section 3.1.

2.2 Readout Protection Design

The RDP level is part of the microcontroller's system configuration, stored in the dedicated *option byte* section. Therein, the available three RDP levels are encoded using 16 bits of non-volatile memory.

This redundancy in security setting storage is crucial for maintaining security under attack. Correctly implemented redundancy raises the bar significantly because several bits must be flipped during an attack. Other already broken systems use only a single bit to distinguish between locked and unlocked configurations [16].

For the STM32, the 16 bits are implemented as two subsequent bytes, named *RDP* and *nRDP*. In every intended configuration, nRDP represents the bitwise complement of RDP. Table 1 shows the mapping of each RDP and nRDP setting to the configured RDP level.

nRDP	RDP	Resulting protection
`0x55`	`0xAA`	RDP Level 0
Any other combination		RDP Level 1
`0x33`	`0xCC`	RDP Level 2

Table 1: Flash readout protection options of the STM32F0 series according to the datasheet [15]

RDP Level 0 and RDP Level 2 are each represented by exactly one complementary pair of bytes. Any other configuration, including non-complementary pairs of bytes, defaults to RDP Level 1.

Although the 16 bit approach appears to be robust—at first, we discover a design issue and defeat the concept in Section 3.2.

2.3 Flash Protection Logic

While RDP Level 0 and RDP Level 2 are straightforward, the technical implementation of RDP Level 1 is only sparsely described.

According to the datasheet [15], the microcontroller uses two modes of operation in RDP Level 1, called "User mode" and "Debug [...] mode". The microcontroller initially runs in unrestricted user mode and switches over to debug mode when a debugger is attached. Once the system enters debug mode, access to flash memory is denied for the debugger and microcontroller. The datasheet [15] also claims that flash memory is then "totally inaccessible" and "[...] even a simple read access generates a bus error and a Hard Fault interrupt.". The datasheet does not contain any information about the inner workings of the flash protection mechanism and under which conditions debug mode is entered. Altogether, this blurry image of the flash readout protection implementation encourages a deeper analysis and reverse-engineering—whose results are presented in Section 3.3.

3 Attacking the Security Concept

The key components of STM32 security require a detailed investigation. In Section 3.1, we present an analysis of RDP Level 1 security, Section 3.2 deals with the strength of the physical implementation of RDP Level 2, and Section 3.3 discovers limitations of the flash protection logic.

3.1 Cold-Boot Stepping

The datasheet [15] assures, that flash memory is read-protected in RDP Level 1 during debugger access. One may easily overlook, that this statement refers only to flash memory but not to SRAM and peripherals. No information is given on the behavior of these components in RDP Level 1.

In order to investigate the missing details, a microcontroller is programmed with a sample firmware and set to RDP Level 1. As soon as the debugger is attached, the microcontroller is halted, since the core can no longer fetch any instructions from flash memory. The behavior is in accordance with the datasheet. Nevertheless, the SRAM is fully readable with all application data in place. Attaching the debugger does not trigger zeroization or any comparable protection mechanism and the data remains intact. Therefore the question arises to which extent the exposed data in SRAM poses a threat to system security.

3.1.1 Concept of SRAM Snapshot Generation

The readable SRAM can be considered a Cold-Boot scenario [6]. The system has ceased operation, but data still resides in memory. There is the trivial case, in which secret data in SRAM becomes exposed directly. This situation requires no more investigation, as the threat is self-evident.

As a countermeasure against such attacks, common implementations of cryptographic algorithms keep keys in RAM only for a short time during usage, down to a few milliseconds or even less. Therefore, this data cannot be retrieved by manually attaching a debugger, as hitting the right moment in time is practically impossible. Furthermore, the microcontroller has several kilobytes of SRAM, whose memory map is unknown and hard to reconstruct from scratch. Stepping through the firmware is also impossible in RDP Level 1, thus the internal control flow is hidden.

To overcome these issues, we present Cold-Boot Stepping (CBS), a method to precisely take snapshots of the system's SRAM at the moment of our choice. The idea is, that the system is effectively halted every few clock cycles, the SRAM is read out, and a snapshot is created. The general idea is partly derived from the very specific approach [17], used on a PIC microcontroller.

The generalized approach is outlined as follows. At first, CBS sets the system into a well-defined initial state, e.g., by applying a reset. Next, the system is allowed to run for a precisely controlled duration and is then stopped. Next, the SRAM is read out and the snapshot is created. Afterwards, the next iteration starts with an increment in the execution duration T, representing a few clock cycles.

By observing the changes in SRAM from snapshot to snapshot, one can reverse-engineer the basic control flow, find out the usage of specific addresses and may also discover briefly visible secret data. This is similar to stepping through the firmware, since only a few instructions are executed between each SRAM snapshot.

This general CBS approach can be adapted to STM32 microcontrollers by using several tricks to control code execution and SRAM data freezing.

1. **Reset system:** The system is brought into the initial state at the beginning of each iteration.
    (a) **Power OFF:** A full reset (Power-On Reset) can only be performed by a power cycle. It will allow the system to access and execute from its flash memory again.

(b) **Assert Reset:** Reset must be asserted before the system is powered up. This allows to power up the system without starting code execution.

(c) **Power ON:** The system is powered up under reset. The internal circuitry becomes ready to start execution, but is inhibited by the reset signal.

2. **Run System for $(n \cdot T)$:** The firmware is executed for a timespan of exactly $(n \cdot T)$.

   (a) **Deassert Reset:** Releasing the reset line starts the execution of the firmware.

   (b) **Wait for $(n \cdot T)$:** The firmware is executed for a timespan of $(n \cdot T)$. The code will advance to the moment of interest.

   (c) **Assert Reset:** As soon as the time is up, reset is applied again. This stops code execution and resets the CPU, but data in SRAM is persistent and becomes frozen.

3. **Dump Memory:** System memory, usually SRAM, is read out and written to a file.

   (a) **Start Debugger:** The debugger attaches to the microcontroller under reset. The debugger takes control and forces the system into permanent halt mode.

   (b) **Deassert Reset:** Reset is deasserted to allow access to the AHB system bus [1] and SRAM. The SRAM state is still preserved, because the system was halted by the debugger and does neither start nor continue execution.

   (c) **Dump SRAM:** The debugger reads the SRAM and writes its data into a file.

4. **n = n + 1:** After these steps have been completed, the system prepares for the next iteration. Therefore the time span of $(n \cdot T)$ is increased by one step to $(n+1) \cdot T$.

5. **Repeat:** Each iteration is started with an increased execution duration. Iterations are performed until the whole time span of interest has been covered.

The schematic of the setup is shown in Figure 1. The laptop on the left orchestrates the attack. It is attached to the debug interface of the device under attack using an ST-LINK debugger. The laptop has a UART connection to the attack control board. A microcontroller on the attack control board handles the reset line and power supply of the device under attack. Switching power on and off can be difficult, since the microcontroller in the device under attack is usually part of a larger device, e.g., an AC powered system. Therefore we employ a relay that

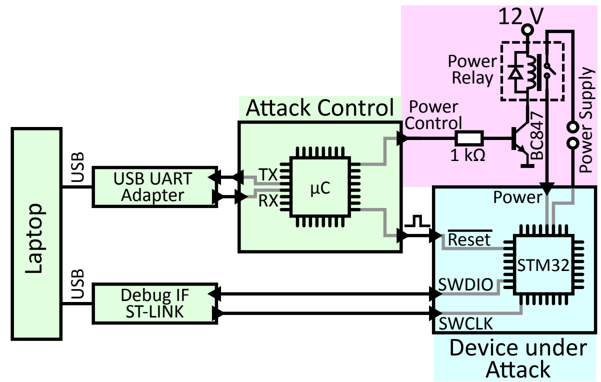

Figure 1: CBS setup schematic

enables the attack control microcontroller to switch high voltages and currents.

CBS requires precise timing with microsecond accuracy, which cannot be provided by a common computer system. Even with an RT-patched Linux kernel, the high timing jitter and reproducibility is unsatisfying. Consequently, the duration of execution cannot be controlled by the debugger, as neither the computer system nor its USB interface guarantees real-time operation.

Therefore, an attack control board, shown in Figure 1, is employed. Its microcontroller provides a low-jitter timing source as the system is primarily dedicated to duration control. Jitter-generating modules, e.g., interrupts, are disabled thereby enabling fully deterministic timing behavior. At the beginning of each iteration, the computer system sends the desired execution duration to the attack control board. Next, it autonomously starts up the device under attack, waits for the configured duration and stops the device again. Thus, timing-critical tasks are exclusively handled by the attack control board. This provides sub-microsecond accuracy. By using the board, precisely stepping to a moment of choice becomes feasible.

3.1.2 PoC: CBS Firmware Extraction

As the attack remains theoretical up to this point, we present a PoC. We created a variant of the attack in which a fully automatized setup indirectly extracts firmware from a locked device. This anonymized example is based on a real case, in which we exploited a commercial product set to RDP Level 1.

Common practice is, that a bootloader is placed on the microcontroller which checks the application firmware image before execution. The integrity check is usually based on a checksum algorithm, such as CRC32. The algorithm iterates over each single byte and may, depending on the implementation, store intermediate CRC results in SRAM.

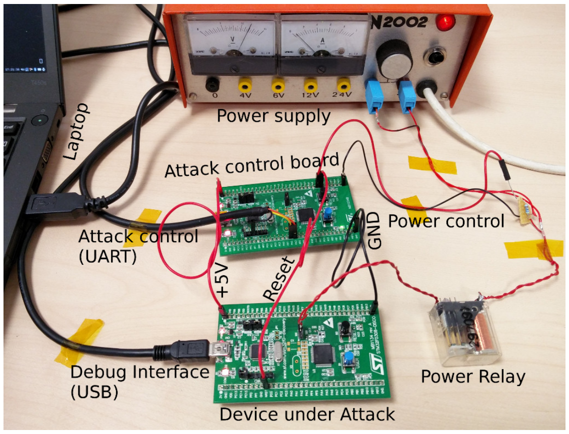

Figure 2: PoC setup for CBS

If an attacker gets hold of subsequent intermediate results of a CRC computation, the source data can be reconstructed. The burst-error detection capability [18] of the CRC32 guarantees, that the solution is unique and that a skipped byte will not yield a solution. The CBS method steps through the computation of the CRC gaining access to the intermediate CRC results. They are used to reconstruct the source data of the CRC which reveals the flash memory contents.

The setup for this attack is shown in Figure 2, which is the practical realization of Figure 1. The device under attack is an STM32F0 discovery board locked to RDP Level 1 which contains a sample-firmware that computes the CRC over itself. The surrounding devices execute the attack as described previously.

The attack runs autonomously. The laptop dynamically adjusts the step width depending on the success of the last extraction iteration. As the attack requires several data transfers between devices, the resulting readout speed is limited to approximately seven bytes per minute. Microcontrollers come with comparably low flash storage, such as 64 KiB for the STM32F051R8T6. Therefore, the attack is practically executable in a few days. In our example, a small firmware image was completely and correctly extracted in a few hours.

The PoC shows, that despite RDP Level 1 prevents direct reading of flash memory, read-access to SRAM poses an invitation to many attacks. We suspect that the attack might work on any STM32 system set to RDP Level 1. Thus the practical protection offered by RDP Level 1 was shown to be futile under these conditions.

3.1.3 Countermeasures against CBS

To prevent the attack, the developer has to set the device into the fully-locked RDP Level 2. This disables the debug interface completely. Without access to the debug interface and SRAM, the attack becomes infeasible. RDP Level 2 is supported by most STM32 microcontrollers, except the STM32F1 series.

Please note, that the attack does not depend on the reversibility of CRC. It might even work more efficiently on hash functions, which often store the result of each round in SRAM. Data hashed with only one round of a hash function is usually reversible. Operating only on CPU registers aggravates the attack, but cannot fully prevent it as the attack can be adjusted to this situation. An advanced debugger can stop the system using precisely timed debug commands. Since the reset signal is not asserted, the CPU registers remain intact and can be read out. However, there is no way around RDP Level 2, if the microcontroller must be well-protected. Additionally, a random delay early in boot might further increase protection.

Experience made us aware of the additional semi-technical issue, that developers have large trust in their toolchain for device protection. The popular debugging software OpenOCD offers only a single command named “lock” to “Lock the entire flash device”. Despite some developers assumed to have their devices fully-secured, the “lock” command does not support RDP Level 2 and only activates RDP Level 1. Therefore we submitted a patch [10], which adds RDP Level 2 support to OpenOCD and clearly displays the configured RDP level.

3.2 Security Downgrade

The previous section demonstrates that the debug interface should not be accessible, thus, it has to be disabled by setting RDP Level 2. As the attack surface is thereby reduced, an attacker might now target the RDP Level with the goal of downgrading security and enabling the debug interface again. This cannot be achieved using official means, since the datasheet [15] claims that “level 2 cannot be removed at all” and is “irreversible”.

3.2.1 Concept of RDP Level Downgrade

The RDP Level protection mechanism is based on the RDP and nRDP bytes in the option byte memory region. At the power-on event, the option bytes are loaded from flash memory and the corresponding RDP Level is set.

At a first glance, the physical RDP Level storage seems to be robustly implemented as it employs 16 bits to store three protection levels. In theory, this redundancy would strengthen system security. However, a closer look reveals that a non-optimal mapping between the RDP Level and the option bytes was chosen.

Figure 3 shows the hexadecimal and binary representation of each RDP Level configuration. RDP Level 0 and 2 map to exactly one setting, where RDP Level 1 covers the

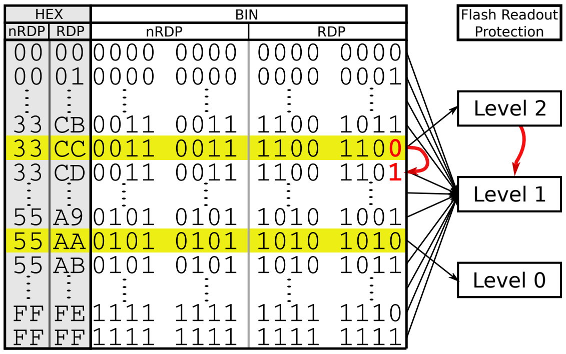

Figure 3: A security downgrade from RDP Level 2 to 1 becomes possible by flipping a single bit

remaining ones. In terms of security, a downgrade from RDP Level 2 to 1 or 0 is of particular interest.

The most powerful attack would cause a transition from RDP Level 2 to 0 as this removes any protection completely. Since the hamming distance of the settings is eight, an attacker has to modify eight bits in total without touching any neighboring bits. More exactly, four specific bits have to be flipped from 0 to 1 and four bits must be changed from 1 to 0 while the remaining eight bits must remain unchanged. Such a modification to the option bytes is difficult, most likely only realizable with highly specialized equipment, and therefore rather expensive. Altogether, this results in a high security margin.

A security downgrade from RDP Level 2 to 1 is an alternative by which the attacker enables the debug interface again. This transition is less difficult, as 65534 of 65536 configurations map to RDP Level 1. Due to this design, the minimum hamming distance between RDP Level 2 and 1 is one, thus there is little security margin as a single bit flip causes a security downgrade. Furthermore, not a specific bit needs to be targeted, flipping *any* bit suffices. Even accidentally flipping multiple bits will still downgrade security. Altogether this imposes a significant risk, because the design undermines RDP Level 2 security.

3.2.2 Proof-of-Concept: UV-C Security Downgrade

We show by experiment, that the risk in fact evolves to a practical threat. As downgrading security involves the manipulation of data, a technique to induce data faults [4] is required. One class of methods is optical fault injection [13]. It uses light to introduce energy at the desired location on the previously decapsulated chip. As the photons interact with the semiconductor, the targeted region on the chip's die starts to malfunction. Depending on the attack parameters, this causes a temporary or permanent data fault.

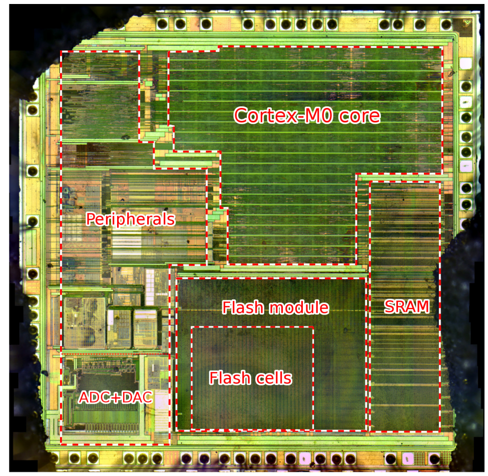

Figure 4: Annotated die of the STM32F051R8T6 (Die size approximately 2700 µm × 2700 µm)

Before an optical fault injection can take place, the device has to be decapsulated. In this process, the package above the chip is removed by chemical etching. This exposes the chip's die for the next step, as shown in Figure 4.

The targeted option bytes are implemented as flash memory. In this technology, data storage is based on electrons, which are trapped on the floating gate [8]. When UV-C light of a wavelength of approximately 254 nm illuminates the floating gate, electrons are ejected, the cell becomes discharged, and the stored bit permanently flips from 0 to 1 [5]. Please note that the method is limited to discharging cells and can only flip bits from 0 (charged) to 1 (uncharged).

The first experiment targets the question, whether RDP Level 2 is in fact irreversible. In difference to flash memory, technologies like eFUSE [9] can implement real one time programmable memory by physically blowing an electrical connection. While manipulating a flash cell requires only UV-C light, resetting an eFUSE needs more sophisticated equipment. So the experiment will show, whether an financially limited attacker is able to perform the security downgrade.

In this experiment, a decapsulated chip in RDP Level 2 is exposed to UV-C light. At first, the debugger cannot attach to the microcontroller, since the debug interface is disabled. Then illumination is performed in an EPROM-eraser using an UV-C mercury lamp with the relevant emission line at 254 nm. After a few hours of UV-C illumination, the debugger successfully attaches to the system as it is no longer fully locked. Reading the "read protection level status" in the FLASH_OBR register reveals, that RDP Level 2 is no longer active and was downgraded to RDP Level 1. As expected, a single bit in the option byte was flipped from 0 to 1. In this experiment, the configuration changed from 0x33CC to 0x33CD. The experiment proves, that no physically permanent locking mechanism exists and RDP Level 2 can in fact be downgraded by flipping a single bit.

Nevertheless, the CBS approach is not feasible afterwards. The attack does not only flip bits of the RDP setting. Despite security becomes downgraded, such a coarse full-chip illumination causes too much damage to the firmware.

In order to target the option bytes exclusively, their location must be determined by reverse engineering the flash memory layout and orientation. Based on experience, the flash module is usually a regularly structured region on the chip surrounded by control and read/write circuitry. This narrows down the search to the rectangular section in the lower center region, as marked in Figure 4. In this region, we employ a bisection approach. This technique uses the light-sensitive flash memory as an image sensor. At first, the chip is left unsecured and loaded with an all-zero firmware image. This makes each cell sensitive to incoming UV-C light. At next, a simple plastic mask is created that covers a specific region, e.g., the upper half of the flash module region. Then illumination starts.

The flash is read every few minutes and the resulting bit flips are analyzed. Only regions that are not covered by the mask will exhibit data faults. In order to map the fault to a physical location, the flash layout must be guessed manually. Flash memory is constructed from numerous vertical bitlines and horizontal wordlines, which create a matrix containing a flash cell on every intersection. As this is a digital system, the number of bitlines and wordlines is usually a multiple of two and their product must yield the storage size. This precondition alone does not yield a unique solution, thus additional trial-and-error is necessary. A correct guess will yield a bitflip-pattern resembling the structure of the mask. This means, for example, that only bits in the lower half change their values, if the upper half is covered by a mask.

A successful guess is depicted in Figure 5, which shows the flash layout of an STM32F051R8T6. The orientation of the figure matches the flash cell region orientation in Figure 4. The layout was reconstructed from several bisection steps that yield the following results. The *firmware* flash region has 1024 bitlines and 512 wordlines. As the flash memory is specified as being 32 bit wide there are 32 bit-*columns*, marked in red in Figure 5. Each single bit-column consists of 32 bitlines, since 32 bit-columns times 32 bitlines result in a total of 1024 bitlines.

Additional UV-C illumination tests show, that the option byte flash region is similar in structure and is placed beneath wordline 0 (note the small region beneath wordline 0 in Figure 5). The region consists of the same number of bitlines and only a few wordlines, since the option

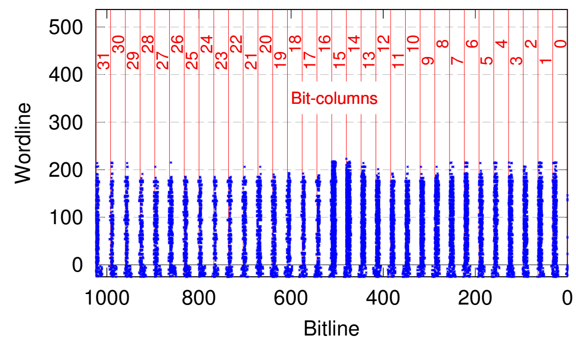

Figure 5: Flash layout containing flipped bits (blue) after UV-C irradiation while the upper half was covered

byte memory is rather small. Since the RDP and nRDP bytes are placed in the 16 LSBs of the option memory, these bytes reside in the right half of the flash cell region. This information helps crafting a mask, that covers the firmware flash region but not the option bytes.

Despite not essential for the attack but to strengthen our results, we delayered the chip's flash using a combination of laser ablation and sulfuric acid etching. The predicted number of vertical bitlines and horizontal wordlines became visible. Additionally, we verified the location of the flash cells by locally injecting non-permanent faults using an IR pulse laser. A reason for the vertical pattern of unflipped bits in each bit-column in Figure 5 was also discovered: Several large vertical metal traces on an inner chip layer block the UV-C light before it reaches the actual cell. Altogether, this proves the results of the UV-C bisection method.

3.2.3 Impact of Increased Precision

After the position of the RDP bytes is known, we target this memory more exactly by using a carefully crafted mask to cover all other memory. By placing the mask diagonally, only the lower right edge of the memory region is exposed, where the LSBs of the RDP bytes reside. Depending on the positioning accuracy, we achieved a security downgrade while the number of additionally flipped firmware bits varied between a few hundred down to none. Our best results in fact yielded a security downgrade to RDP Level 1 with no additional firmware bit faults at all.

Altogether, a weakness in the security design has been shown. Having very few or even no firmware bit faults enable the practical use of RDP Level 1 attacks like CBS. The vulnerability applies to all STM32 microcontrollers using such a security design. However, the feasibility of the attack might strongly differ between the controllers, depending on the physical implementation of the flash.

3.2.4 Countermeasure: RDP Downgrade Detection

There exists no countermeasure to prevent a security downgrade, since the issue persists in hardware. It is up to the manufacturer to enhance the concept, e.g., by using RDP Level 2 instead of 1 as default/fallback setting.

Nevertheless, the effects of an attack can be mitigated. Upon startup, the firmware should immediately check, whether the expected level of protection is still active. This includes the "read protection level status" in the FLASH_OBR register as well as the option bytes themselves. In case of a mismatch between the setting and the expectation, the RDP Level 2 setting should be rewritten to the option bytes, as this will restore missing charge in the flash cells and re-enable security.

3.3 Debug Interface Exploit

The STM32F0 series microcontrollers do not have a JTAG interface but use the Serial Wire Debug (SWD) interface for debugging and flash programming. When a debugger becomes attached to this interface in RDP Level 1, the flash protection logic cuts access to flash memory. This sparsely documented mechanism raises interest and asks for a thorough investigation.

3.3.1 Reverse Engineering

For the analysis, using the common off-the-shelf debugger ST-LINK fails. As soon as the debugger becomes connected the flash is immediately locked down. This issue is caused by the debugger, which triggers the protection mechanism by performing several SWD requests automatically upon connecting. As the immediate lockup prevents a precise analysis, this overly verbose debugger is of no use for our experiment.

In order to gain control down to the *physical layer* of SWD communication, we implement an own rudimentary but versatile debugger. This is feasible, since the SWD interface is well-documented [2]: SWD uses two signals, SWDIO and SWCLK, for synchronous communication between the debugger and the microcontroller. SWDIO is a bidirectional data signal, SWCLK is the clock used in the communication.

Figure 6 shows the wiring of our setup. The debugger (DBG) is controlled by the Laptop using a UART. The debugger is connected to the power, reset and SWD signals of the microcontroller under analysis and attack. The associated firmware gives us full access to all SWD communication layers, thus, a detailed analysis becomes possible.

The first experiment tests, which SWD transactions trigger the flash protection logic. Therefore, the system is set to RDP Level 1, the firmware runs from flash and flashes an LED. The flashing LED shows, that the system is running unperturbed. When flash access is blocked, the CPU fails to fetch any further instructions and the flashing of the LED stops.

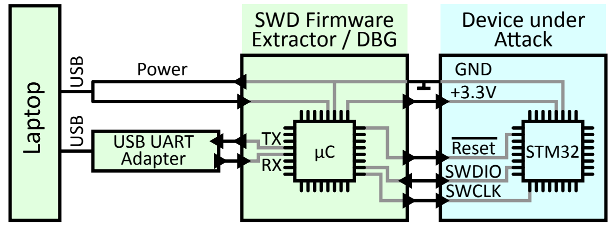

Figure 6: Schematic of the SWD experiments and Debug Interface Exploitation

The experiment shows, that the protection mechanism is not triggered by all SWD requests but only if a system bus (e.g., AHB-Lite [1]) is accessed. Accessing only SWD interface-internal registers has no consequences and the system remains running. But as soon as the debugger uses the system bus to access any other module, such as peripherals, SRAM or flash, debug mode is entered and the flash becomes locked down.

The debugger triggers a transfer on the system bus by reading from or writing to the SWD AHB Access Port (AHB-AP) Data Read/Write register. A closer analysis reveals, that the AHB transaction and therefore the lockdown is triggered by the last rising SWCLK edge of the corresponding SWD packet transmission.

3.3.2 Discovery and Analysis of the Vulnerability

In order to analyze the flash protection logic behavior, the amount of SWD communication is reduced to its essentials. Most SWD requests are removed which leads to a minimal configuration, where the SWD interface is fully initialized, but the flash protection has not been triggered, yet. In the next step, a bus access is triggered by a read request from a flash address. This is expected to trigger the flash protection logic. This happens in fact—but in rare cases, some data ends up in the read buffer. To our surprise, this is data from the actually read-protected flash memory.

If at all, this deviation from the specification is observed only on the *first* bus read access. Any subsequent read attempts always fail and return an SWD ERROR response. To repeat the experiment, the flash protection logic must be reset by a power-cycle.

The read access shows to be only randomly successful. As this is typical for a timing-based race condition, we tinker around with the bus load and timings to investigate this issue further. Therefore we implement different firmwares using Assembly language, one containing mostly no-operations (NOPs) and another one featuring long sequences of store (STR) instructions. Altogether, the bus load varies from roughly 50 % for the NOP firmware to up to 100 % for the STR firmware. The access succeeds in every case while running the NOP firmware, but the access always fails for the STR-sequences firmware. This shows that the issue is bus load related.

A look into the Cortex-M0 specification [3] supports the conjecture of a timing issue. CPU instruction and data fetches have priority over debug access during arbitration. Thus, the debugger has to wait for a free cycle on the bus to place the request. If the debugger gains instant access to the bus, the access takes place before the flash protection locking becomes triggered—thus, a typical race condition occurs. In the other case, when the debugger's access is delayed by an ongoing CPU access, the protection logic wins the race and the flash module will reject the debugger's later-arriving read attempt.

We investigate this further by linearly adding bus load by increasing the number of wait states of the flash module. At zero wait states, a read access takes two cycles on the AHB, one for the address phase and one for the data phase [1]. Wait state cycles are inserted before the data phase. Taking the NOP firmware as a basis, we observe, that adding one wait cycle will cause one out of three read accesses to fail. By gradually increasing the wait states w up to the maximum of seven, we observed, that the read success probability p_s adheres to the following term:

$$p_s = 1 - \frac{w}{2+w} = \frac{2}{2+w}$$

The rearranged term at the right hand side of the equation allows us to give an educated guess about the internal timing issue. The denominator $(2+w)$ represents the total number of bus cycles for the AHB flash access. Therefore the numerator 2 is the number of *vulnerable* bus cycles. Hence, we come to the conclusion, that the flash protection logic is triggered two cycles too late.

By using the STR firmware we are able to show, that the flash module itself manages the access restrictions. The firmware rapidly toggles an LED using the bus, thus bus load will cause a measurable timing jitter. At a failed access to the flash, the bus is blocked for three clock cycles in total, which matches the duration of an error response [1] on the AHB. Thus, the access is actively denied by the flash module. Furthermore, this shows, that the protection logic makes the decision about the access directly after the data phase as no wait states were ever observed during a failed access.

The reason for the protection delay of two clock cycles lies in the chip's hardware design, but the exact location remains unclear. One possible cause may be an incorrect implementation of clock-domain crossing between the debug clock domain [2] and the remaining logic.

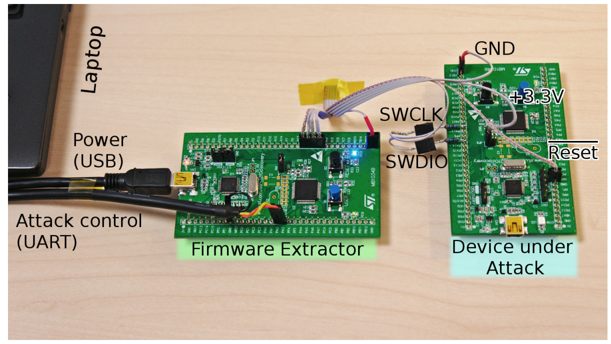

Figure 7: Setup for PoCfirmware extraction out of a device using RDP Level 1

Altogether, these experiments show, that there is a major hardware issue, that annuls the chip's content protection mechanism.

3.3.3 Proof-of-Concept: Code Extraction

This PoC demonstrates, that not only a single access can be performed, but the whole firmware can be extracted from a microcontroller, using the aforementioned weakness. The setup is shown in Figure 7, which is the practical realization of Figure 6. The Firmware Extractor reads out the firmware of the device under attack which is set to RDP Level 1. The PoC employs two STM32F0 Discovery boards with STM32F051R8T6 microcontrollers.

In the PoC, the laptop controls the Firmware Extractor using its UART interface. This interface configures parameters, e.g., the read source address and the data length to be extracted. During readout, the data is sent to the laptop using the UART interface.

The attack is implemented in a straightforward manner with one extension. As the read access works only once before the system is locked up, the Firmware Extractor performs a power cycle on the device under attack after each readout. Thus, the following steps are executed:

1. **System Reset:** Initially, a power cycle is applied to reset the system and flash protection logic.

2. **Debugger interface initialization:** The steps, described in the datasheets [15, 2], are followed. First, the SWD interface is reset by applying the reset pattern onto SWDIO and SWCLK. Secondly, the debugger reads the IDCODE from the SWD Debug Port (SWD-DP). Thirdly, the debugger sets the System power-up request (CSYSPWRUPREQ) and Debug power-up request (CDBGPWRUPREQ) in the CTRL/STAT register of the SWD-DP to fully initialize the debug interface.

3. **Set the access width to 32 bit:** Although optional, switching into 32-bit mode is recommended, as a full word can be extracted in each step. Therefore, the size field in the AHB Access Port (AHB-AP) Control/Status Word Register has to be set to `0x02`.

4. **Set flash source address:** The address to be read is written into the AHB-AP Transfer Address Register.

5. **Trigger the flash read bus access:** The debugger performs a read access from the AHB-AP Data Read/Write register. This triggers the AHB transaction that reads the flash memory.

6. **Read the extracted data:** The result of the previous access is read from the DP Read Buffer register. On success, if flash access was granted, an SWD OK response is returned. On failure, SWD ERROR is returned. In the case of SWD OK, the word was correctly extracted from flash and is sent to the Laptop.

7. **Iterate:** Upon success, this procedure is restarted with $(\text{address} + 4)$, in order to read the next word from flash. On failure, the address is not incremented and the read access is retried.

The PoC was successfully tested on STM32F051R8T6 and STM32F030R8T6. Due to the very similar chip functionality, we expect, that the whole STM32F0 series is affected. Nevertheless, experiments on a few samples indicate, that other *series* might not be vulnerable.

The PoC extracts firmware at 45 bytes per second on average. The largest STM32F0 microcontrollers with up to 256 kByte of flash memory can be read out completely in less than two hours, thus rendering the attack practically feasible.

3.3.4 Consideration of Countermeasures

The attack can be aggravated by relinquishing the use of RDP Level 1 and setting the device to RDP Level 2, since this disables the debug interface. Nevertheless, this does not fully resolve the issue, because the protection level is non-permanent, as shown in Section 3.2. In comparison to the CBS attack, the requirements put on the security downgrade are significantly relaxed. While the firmware must remain intact and executable for CBS, the debug interface exploit works also on damaged firmware. Thus correctly placing the mask for the security downgrade is less demanding. Even if data errors occur, they are usually limited to a small range of addresses, thus destroying cryptographic material or other IP is rather unlikely. Hence, there is no strong protection, especially if combined attacks are also considered.

4 Conclusion and Outlook

We presented three major security issues in the STM32F0 series that leverage firmware content protection. We were able to develop all weaknesses to vulnerabilities by demonstrating their practical exploitability. The demonstration was conducted with the aid of three PoCs, one for each vulnerability. They demonstrate the practical relevance of our work and present novel methods for attacks and reverse-engineering.

First, the readability of SRAM in RDP Level 1 emerges to a practical threat, as it often allows the reading of cryptographic material up to the whole firmware. Secondly, the stronger RDP Level 2 incorporates a design error, which significantly weakens the security concept and enables a security downgrade. Thirdly, the SWD hardware implementation contains a race-condition in RDP Level 1 that completely annuls device security and exposes the firmware to the attacker.

Altogether, these vulnerabilities completely break security for RDP Level 1 and alarmingly weaken security for RDP Level 2. Since only low-cost lab equipment is required, the impact increases. Especially the combination of vulnerabilities, e.g., downgrading security from RDP Level 2 and using the SWD exploit to read the firmware pushes the complexity of attacks down to the hobbyist-level. Cold-Boot Stepping and the security downgrade might also work on other STM32 chip series, as they are based on the same concept. Thus, the impact of this analysis is severe and might be expendable to more systems.

All in all, the presented results need to be taken into consideration when an STM32 microcontroller has to be decently secured. We plan to extend our analysis to more systems, to improve the attacks, and to dig deeper into the STM32 internals.

5 Acknowledgments

The authors would like to thank Robert Specht and Tolga Sel for preceding work concerning STM32 flash security. Special thanks to Michael Hani for supplying decommissioned but helpful lab equipment for our experiments.

6 Availability

We provide supplementary material under the MIT license at `https://science.obermaier-johannes.de`.

The material includes all required scripts, ELF, and source code files for the Cold-Boot Stepping and Debug Interface Exploit PoCs, as well as sample firmware to be attacked.

References

[1] ARM LIMITED. *AMBA 3 AHB-Lite Protocol Specification v1.0*, 2006.

[2] ARM LIMITED. *CoreSight Components Technical Reference Manual*, 2009.

[3] ARM LIMITED. *Cortex-M0 Technical Reference Manual*, 2009.

[4] BAR-EL, H., CHOUKRI, H., NACCACHE, D., TUNSTALL, M., AND WHELAN, C. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE 94*, 2 (2006), 370–382.

[5] BEZ, R., CAMERLENGHI, E., MODELLI, A., AND VISCONTI, A. Introduction to flash memory. *Proceedings of the IEEE 91*, 4 (2003), 489–502.

[6] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM 52*, 5 (2009), 91–98.

[7] IGNAT, V. Patent and product piracy. *IOP Conference Series: Materials Science and Engineering 147*, 1 (2016), 012105.

[8] KAHNG, D., AND SZE, S. M. A floating gate and its application to memory devices. *The Bell System Technical Journal 46*, 6 (July 1967), 1288–1295.

[9] KOTHANDARAMAN, C., IYER, S. K., AND IYER, S. S. Electrically programmable fuse (efuse) using electromigration in silicides. *IEEE Electron Device Letters 23*, 9 (Sept 2002), 523–525.

[10] OBERMAIER, J. OpenOCD Patch: flash/nor/stm32f1x: Added RDP Level 2 support. `http://openocd.zylin.com/4111`, 2017. [Accessed 01-May-2017].

[11] SKOROBOGATOV, S. Flash memory 'bumping' attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2010), Springer, pp. 158–172.

[12] SKOROBOGATOV, S., AND WOODS, C. *Breakthrough Silicon Scanning Discovers Backdoor in Military Chip*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 23–40.

[13] SKOROBOGATOV, S. P., AND ANDERSON, R. J. Optical fault induction attacks. In *International workshop on cryptographic hardware and embedded systems* (2002), Springer, pp. 2–12.

[14] STMICROELECTRONICS. *Datasheet STM32F303xB STM32F303xC*, May 2016. Rev 13.

[15] STMICROELECTRONICS. *RM0091 Reference manual, STM32F0x1/ STM32F0x2/ STM32F0x8 advanced ARM®-based 32-bit MCUs*, January 2017. Rev. 9.

[16] STROBEL, D., DRIESSEN, B., KASPER, T., LEANDER, G., OSWALD, D., SCHELLENBERG, F., AND PAAR, C. *Fuming Acid and Cryptanalysis: Handy Tools for Overcoming a Digital Locking and Access Control System*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 147–164.

[17] VARIOUS AUTHORS. Mikrocontroller.net: Meteotime Crypt. `https://www.mikrocontroller.net/topic/220947`, 2011. [In German; accessed 01-May-2017].

[18] WARREN, H. S. *Hacker's delight*. Pearson Education, 2013.