Bachelorarbeit

Practical Analysis of Embedded Microcontrollers against Clock Glitching Attacks

Ricardo Gomes da Silva

17. März 2014

Technische Universität Berlin
Fakultät IV
Institut für Softwaretechnik und Theoretische Informatik
Professur Security in Telecommunications

Betreuender Hochschullehrer: Prof. Dr. Jean-Pierre Seifert
Betreuender Mitarbeiter: Bsc. Dmitry Nedospasov

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 17. März 2014

Ricardo Gomes da Silva

Abstract

Clock glitching attacks are one of the different types of hardware fault injections studied nowadays. By glitching the clock, it is possible to change the target's hardware behavior, either by corrupting or simply skipping CPU instructions. Since the software is not prepared to handle a device that has been tampered with, an attacker can exploit such vulnerability and take over the control flow of the program. Multiple attacks can then be performed, such as forcing the device to exiting loops or dump its own memory.

This work applies such technique against AVR microcontrollers by implementing a modular glitcher environment. Such environment allows not only for fine-tuning of attacks, but also a brute-force algorithm for finding the glitching range to be implemented. By executing multiple repeatable experiments, both on handcrafted and compiled code, it demonstrates that such architecture is vulnerable against these attacks by introducing faults that were not expected and cannot be handled by the software. The implications of this regarding the security of the program are discussed in this work. Additionally, a critical analysis of the disadvantages and issues of the approach used and how it can be further improved is provided.

Zusammenfassung

Clock-Glitching-Angriffe sind eine der verschiedenen Arten von Hardware-Fehler-Angriffen, die heutzutage untersucht werden. Das Glitchen der Clock ermöglicht es, das Verhalten der angegriffenen Hardware zu verändern. CPU-Instruktionen können auf diese Weise entweder verändert oder auch komplett übersprungen werden. Da die Software nicht darauf ausgelegt ist, auf Manipulationen an dem Gerät zu reagieren, kann ein Angreifer diese Situation ausnutzen und die Kontrolle über den Programmablauf übernehmen. Viele verschiedene Angriffe können damit durchgeführt werden: Zum Beispiel können Schleifen vorzeitig abgebrochen werden oder das Gerät dazu gebracht werden, den gesamten Speicherinhalt auszugeben.

In der vorliegenden Arbeit werden diese Techniken gegen einen AVR Mikrocontroller angewendet. Hierzu wurde eine modulare Umgebung entwickelt, die das Glitchen der Clock ermöglicht. Diese Umgebung ermöglicht nicht nur eine präzise Einstellung der Angriffsparameter, sondern auch das Ermitteln des genauen Zeitpunkts, an dem der Glitch eingefügt werden soll, durch eine Brute-Force-Suche. Indem sowohl auf angepasstem als auch auf kompiliertem Code verschiedene reproduzierbare Experimente wiederholt werden, zeigt diese Arbeit, dass die untersuchten Architekturen angreifbar durch solche Angriffe sind, in denen unerwartete Fehler eingebracht werden, mit denen die Software nicht umzugehen weiß. Die Auswirkungen, die diese Angriffe auf die Sicherheit der Programme haben, werden in dieser Arbeit diskutiert. Außerdem werden die Eigenschaften und Nachteile dieses konkreten Ansatzes kritisch hinterfragt und der Frage nachgegeangen, wie er verbessert werden kann.

Acknowledgements

I would like to thank both supervisors Dmitry Nedospasov and Prof. Dr. Jean-Pierre Seifert for the help, guidance and lessons on the subject, as well as the long yet highly useful discussions that resulted in my interest for such field of study. Not less important, I would like also to thank my colleague, Juliane Krämer, whose questions regarding this project and her German language skills were very important as motivation to always improve my skills and knowledge. Prof. Dr. Raul Fernando Weber, at my home university, I thank you for the support and patience with all my questions regarding the exchange program, bureaucracy, the life, the universe and everything else.

This thesis would have not have been possible without the exchange program between my home university UFRGS[1] and TU Berlin. I am grateful to my home university, for providing me the required knowledge through its classes in sunny hot days, and TU Berlin, for accepting me as a student and providing the access to the required environment for acquiring knowledge on such field.

I owe my deepest gratitude to my family, friends and colleagues. Without their support and help during the stressful hours of continuous work, either with discussions, studies or distractions during difficult times, nothing would be possible. My good old friend Oggo Petersen, I thank you for the long sleepless hours of gaming before my exams and deadlines. To them, I will be ever in debt.

Ricardo Gomes da Silva

[1] Universidade Federal do Rio Grande do Sul

Contents

List of Figures

1 Introduction

Microcontrollers are extremely popular nowadays, being present in numerous devices and systems. They are designed to be embedded into small systems and interact with the environment, either directly through sensors and acuators, or through other digital systems. The most common example of embedded microcontrollers are *smart cards*, used for numerous scenarios, such as personal identification and access control. On such devices, the software limits the device's interaction with the outside world, such as limiting which how much data can be obtained from it upon request. However, such devices assume that they have not been tampered with, neither that it was previously modified in a manner that the system behaves differently than original (i.e. the system behaves nominally).

Previous studies demonstrated that smart-cards and microcontrollers in general are vulnerable to fault injection attacks, where, by directly injecting a fault through the hardware, it is possible to extract information [KK99, AK96, BGV11, KK99] or bypass security controls on the executing code [And01]. In software attacks, the device is attacked from the software perspective, meaning that the issue is on the code. However, with hardware attacks, the assumption that the hardware is stable and safe is no longer true, since the device can now be controlled from the hardware perspective. Such attacks can be done in precise moments or with specific methods to target different parts of the hardware, such as the CPU's instructions and registers or the device's memory. This allows an attacker not only to change the behavior of the program, but also to extract data or bypass software protections by simply attacking specific regions of the hardware. Since the code does not expect that the hardware is compromised, it continues to execute as normal. This introduces a security issue since the code can now be modified in realtime, allowing an attacker to be able to change the original behavior of the program. The number of possibilities that a fault injection attack opens are not small: it could be used to extract data from the device, bypass security features, skip or repeat loops, and so on.

One of the most common fault injection techniques, called clock glitching, is analysed in this work, where a fault is injected on the clock signal used by the microcontroller. It consists of injecting small faults into the clock signal with the goal of glitching the device's CPU. If successful, the CPU can either skip code instructions, or compute the wrong data, allowing an attacker to change the execution flow of the code.

How clock glitching attacks work and how the target device was prepared and selected are described in Section 2, being followed by the description of the dedicated hardware used on the attack in Sections 3 and 4. Multiple successful experiments were performed, their results described and explained in Section 5. A discussion regarding what was achieved with this work and how it could be further improved is provided in Section 6, followed the conclusion of this work in Section 7.

2 Background

In this section, the necessary background information for understading this work is provided. Initially, fault injection and glitching will be discussed, and later the other devices involved in the experiments.

Glitching

Glitches are result small and transient faults in a system and can be caused both by environment interference or by fault injection experiments. By deliberately creating faults in the system, we disrupt its behavior in a way that might be exploitable for an attacker. Examples of typical targets for glitching are the clock signal [BGV11], the voltage output on the power supply [ABF+03], temperature changes and electromagnetic radiation [CPB+13]. Glitching is also commonly classified as a *non-invasive* fault injection attack since they don't require any modifications on the hardware itself.

In this work, clock glitches are used to attack a microcontroller with the goal to analyse how it reacts against such scenarios. By glitching the clock in precise moments, it is possible to hit one or multiple clock cycles of an instruction. If successful, the attack could lead the microcontroller to a vulnerable state, where an attacker could exploit either by executing another attack, or by already analysing the current state, such as reading leaked data.

A clock glitch consists of increasing the frequency of the circuit clock for a short amount of time (usually one clock cycle), as shown in figure 2.1. Since the circuit cannot always handle the higher clock frequencies properly, it might latch its registers before a new value has arrived to them [BECN+06, KK99], which allows an attacker to take over the control flow of the program.

Microcontrollers

Microcontrollers consits of a processor (CPU) and random-access memory (RAM), as well as additional peripherals and I/O. They are essentially computers programmed to execute code to control or communicate to a device and they are present on innumerous peripherals and products nowadays, making them extremely popular. However, what differentiate them from microprocessors is that they have all of this in one package, together with an embedded non-volatile memory, which allows the code to be executed to be stored directly on the microcontroller. Since their goal is to be embedded into devices or systems, they do not the usually have input and output peripherals as a normal computer. Instead, they have digital pins that can be used for signaling and interfacing other devices.

Even though microcontrollers can be used and applied in multiple scenarios, some have specific features that improve their capabilities or performance in different environments. For example, microcontrollers can have UART (*Universal asynchronous receiver/-transceiver*) ports, which allows easy serial communication with external devices, cryptographic coprocessors, improving the performance of algorithms for encrypting and decrypting data, timers and counters (such as RTC, *Real-time clock*, peripherals), watchdog timers, and others. Such peripherals are considered part of the microcontroller since they are addressable through registers in the executing code, allowing easy access without having to care about the interface between the microcontroller and the peripheral.

In this work, a microcontroller with a single-level pipeline was used. This means that this CPU is able to fetch and decode the next instruction while the previous is executing. Although this increase the resource usage on the hardware level, this improves performance (in terms of MPIS - *million instructions per second*, since it allows the next instruction to be read as soon as the previous has finished, avoiding the extra time for the instruction fetch. Note that, however, instructions that need to operate with the memory have additional executing cycles, as shown on figure 2.2.

Note that, in this work, the microcontroller is also a RISC (*Reduced instruction set computing*) CPU. This means that it is generally able to execute instructions within one clock cycle, although instructions that operate with memory can take additional cycles, as shown on 2.2. This is important since it makes glitching fast instructions (i.e. instructions that take one clock cycle) more difficult, since it is necessary to hit the exact instruction within one clock cycle. The preciseness of a glitcher device in such scenarios is critical, as any delay or clock drift could make it miss the instruction, attacking either the previous or the next one.

Finally, from the point of view of clock glitching, however, having a pipeline has a negative effect. Since the entire CPU uses the same clock signal for each stage of the pipeline, glitching the clock might cause not only the corruption of the current instruction, but also of the next one. Therefore, when glitching the clock, this has to be kept in mind, since usually one might not want to corrupt the next instruction (as it is part of the program logic and could be critical for the glitch to have any effect).

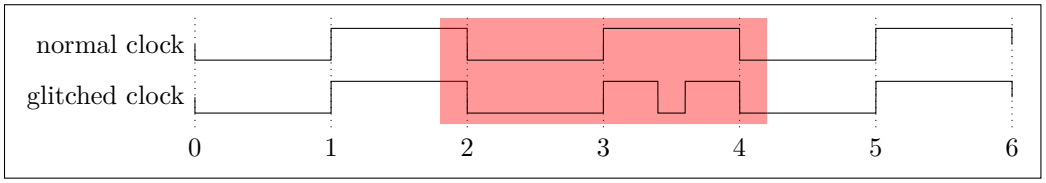

Figure 2.1: Clock glitching example. The highlighted areas indicate glitched clock cycles, where there are multiple rising edges instead of one. Note, that both before and after the glitch the clock is set to its normal frequency, allowing the device to operate normally before and after the attack.

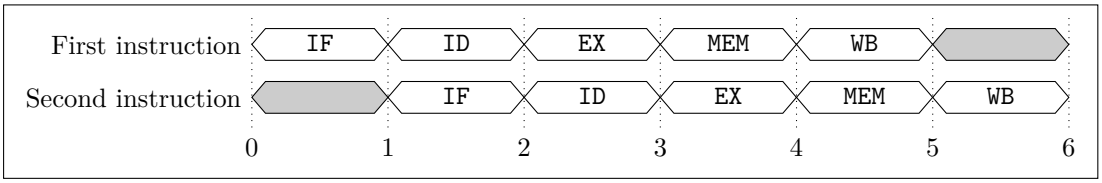

Figure 2.2: This image describes how a single-level pipeline works. The three first stages consist of fetching the instruction from the cache (IF - *Instruction Fetch*, decoding the instruction (ID - *Instruction Decode*) and finally executing it (EX - *Execution*). However, if the instruction needs to operate memory, either for read or write, one extra clock cycle are required for such operation (MEM - *Memory access*). Finally, if the instruction needs to write back its values to registers (e.g. arithmetic and memory load instructions), it is done on the last stage (WB - *Write Back*). [HP11]

3 Setup

This section introduces the equipment used in subsequent experiments, providing also a brief description and specification of their features. How they are integrated to create a glitcher will be further discussed in Section 4.

Microcontroller

There are multiple types of microcontrollers on the market, each one with different features and capabilities. For this work, however, a more generic microcontroller produced by Atmel was selected - the AVR XMEGA A family. These are low power, high performance and peripheral rich 8-bit RISC microcontrollers based on the AVR architecture [Atm12]. For easy development and testing, an *XMEGA-A1 Xplained* was used, which is an evaluation kit provided by Atmel for evaluation of the ATxmega128A1 microcontroller [Atm11] that provides a ready-to-use environment, facilitating the experiments.

By using an evaluation kit instead of the microcontroller directly, there is no need to consider issues caused by an incorrect device setup (such as wrong wiring). The board also contains pin headers, which facilitates creating an interface between the microcontroller and other devices (e.g. AVR programmers), among other devices, such as a light sensor, speakers, buttons and LEDs. Not all of the available features on this board were used in this work, but they are important and useful when preparing the device for the first time, since they provide feedback to troubleshoot any issues.

Die Datenkrake

For clock glitching it is essential to have very precise timing. For this reason, dedicated hardware must be used, and in this work, *Die Datenkrake* was selected for this task. *Die Datenkrake*, or DDK, is an open source security-focused development platform. It consists both an ARM CPU and a FPGA (*Field-programmable gate array*), each one being responsible for a different task in this work. It also contains 8 I/O channels which allows easy and parallel analysis and communication with multiple devices [NS13], as shown on Figure 3.1.

Due to the necessary timing characteristics of a clock glitcher, the DDK's FPGA was used to generate all the necessary clocks and signals. Although the signal could be generated by toggling an output directly on the ARM CPU, CPU interruptions and parallel tasks might affect its performance, resulting in inconsistent timing. Additionally, the switching frequencies of the I/O are a fraction of what the FPGA is capable of, and, since the glitching clock is faster than the normal one, any interruption on this algorithm could affect the glitcher's precision by creating unexpected delays. Another reason to

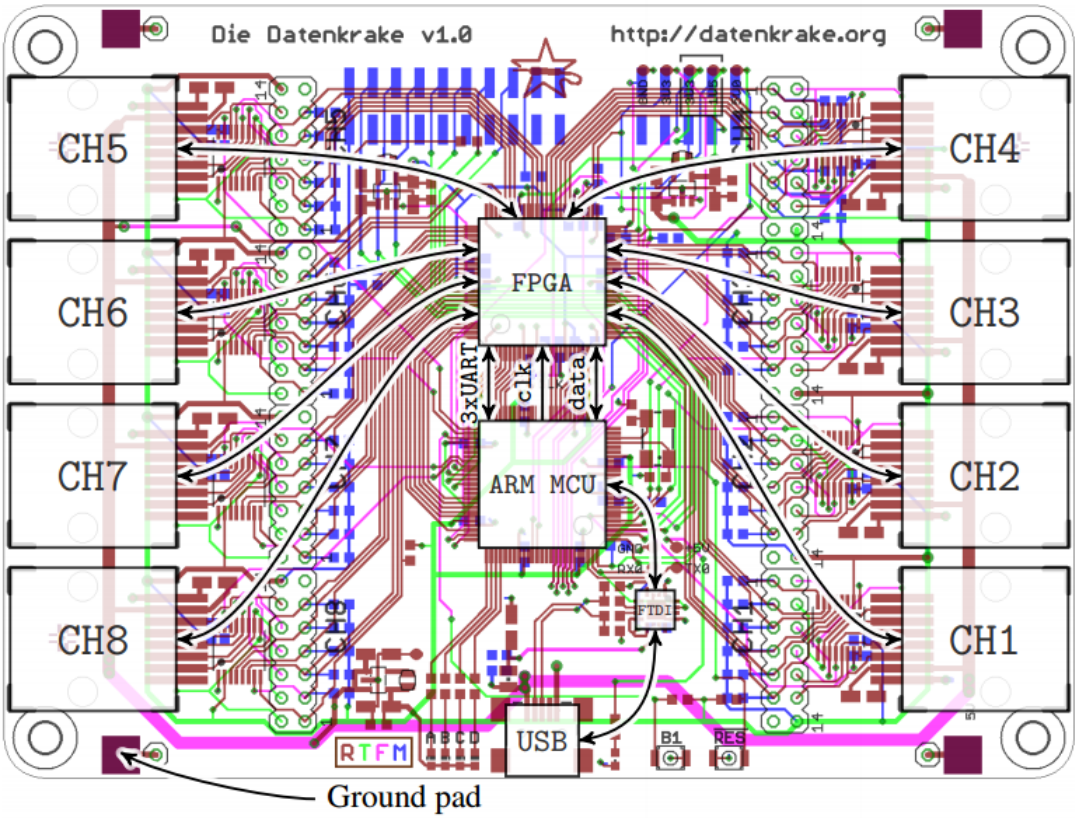

Figure 3.1: *Die Datenkrake* hardware layout (version 1.0). The board provides easy access to GPIO pins through RJ45 connectors, which also provide power. Therefore, they can be used both for data and driving level translation circuits, in case the device does not use a standard I/O interface. [NS13].

use the FPGA instead of the ARM is the fact that the clock has to be synchronized with the target's clock. The overall clock drift between the glitcher and the target would result in imprecise glitch timing. Therefore, to avoid such issues, it is easier to use the FPGA to produce the target's clock, feeding it externally with a controlled and synchronized signal. To generate synchronized clocks, a PLL (*Phase-locked loop*) provided on the FPGA is used to output three different signals (the DDK's internal clock, the target normal clock and the glitching clock), providing the guarantee that all of them have their rising edges synchronized and that there is no delay or drift between them. Finally, to keep both the ARM CPU and the FPGA synchronized, the FPGA is driven by a clock provided from the ARM CPU. By using this setup, it is possible to guarantee no clock drifts between them as well.

The DDK's CPU runs an RTOS (*Real-time operating system*) which is responsible for handling all the less time-critical operations in multiple parallel tasks, such as handling the UART ports provided by the CPU and the CPU-FPGA bus signaling. The DDK software, therefore, runs directly on the board. To avoid requiring external software or hardware to be able to use the DDK, a console is provided on the UART port available through the USB port. This console will be later used in this work to control the glitcher by sending commands to configure and run experiments. The console interface is designed also to be able to handle the FPGA, such as reading and write values to addresses, which will be later converted to Wishbone commands through a bus that connect both devices. Such addresses are decoded and point to each available DDK channel based on their most-significant bits, while the least-significant are the specific Wishbone address to be read or written to. Therefore, it is possible to execute compiled C code on the ARM that directly controls any module on any channel on the FPGA, allowing a high level of automation without requiring the Verilog FPGA code to be rewritten or modified at all. Finally, this can also be considered *hardware/software codesign*, where a product or device hardware and software are developed together with a high level of integration between them.

Wishbone Bus

The DDK FPGA code is divided into multiple independent modules. To allow a stable and easy communication between them, the DDK architecture uses the *Wishbone Bus* for internal communication. This bus is an open-source computer bus that allows interaction between modules without the necessity of each one knowing the internals of the other, allowing a simple and transparent module integration.

For this work, it is not necessary to understand the technical details of the Wishbone bus. However, it is interesting to know that read/write operations can be done in one clock cycle [Ope10], allowing a fast data transfer between modules. It also uses a *strobe* signal to indicate which module is being accessed, while the others are kept idle. Each Wishbone-compatible module has four basic input signals: `adr_i` (address input), `dat_i` (data input), `we_i` (write enable) and `stb_i` (strobe input). This allows another module to specify which address in the module it wants to access, either for read (`we_i` as 0) or write (`we_i` as 1), and which data it wants to send together with the operation.

Once activated (`stb_i` as 1 for one clock cycle), the module can handle the inputs, doing the necessary logic. Note that the module must respond within the next clock cycle, whenever if a complex operation takes one or more cycles to conclude. There are two output signals: `ack_o` (acknowledge) and `dat_o` (data output). The first is used to indicated that the previous operation was valid and acknowledged (i.e. it was a valid operation, such as having a valid `adr_i`), while the second one is to indicate the return data.

Note that, in the DDK, both data and address are 8-bit long. Higher values can be written, as it will be shown further in this work, but the data has to be splitted in 8-bit chunks. Finally, as previously noted, on the DDK the address is used to indicated not only which operation should be performed, but also which channel should perform it, being the four most-significant bits the channel and the four least-significant the operation. In the modules further discussed in this work, however, the addresses will be also called registers, since they usually target specific internal registers in the modules, writing and reading from them instead of representing an operation by themselves.

4 Glitcher development

For analysing the effects of clock glitching against microcontrollers, a complete test environment was created. It consists of three main parts: the glitcher, the microcontroller and an external monitor. This section describes in details not only how such components were designed and implemented, but also how they interface each other for creating the test environment.

Glitcher

The glitcher is the most important part of this work, and for such reason its design is the most sophisticated one. From the point of view of the environment, the glitcher is one single module. However, internally, it's made of three self-contained modules: the core, the main module and the Wishbone interface. This separation was done not only with the goal of keeping the glitcher easy to maintain and modify, but also to keep it as modular as possible. The layout is based on tasks - each module is responsible for its own task and can be used and tested separately.

Glitcher's core

The core of the glitcher is the simplest and lowest-level module in the setup. Its inputs are the normal unglitched clock (`clk_in`), the clock used for glitching (`clk_gl`), an enable signal to activate the module (`en`) and a 8-bit mode of operation (`mode`). The only output is the glitched clock (`clk_out`), which is only changed when the module is enabled (when not, it outputs the same as `clk_in`). The figure 4.1 describes the module.

The core is also a combinational logic circuit. This means that it doesn't depends on its main clock, being the output only a function of the inputs. This is important because the glitcher must be completely time-independent, being able to be enabled and disabled whenever possible.

The output is provided depending on the current value of the mode, being selected by a multiplexer, as shown in figure 4.2. There are five available modes:

- *bypass*: the signal is not altered and `clk_in` is output on `clk_out`.
- *zero*: the output is kept low (logic zero).
- *one*: the output is kept high (logic one).
- *not*: the output is the result of `clk_in` through a NOT gate.
- *clkgl*: the output is the same as `clk_gl`.

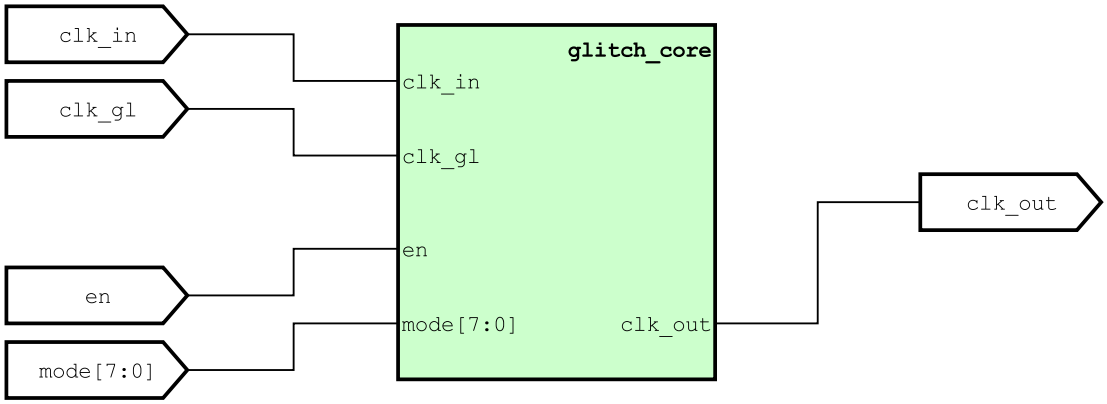

Figure 4.1: Core module of the glitcher. Once activated, this module changes the `clk_out` accordingly, being configured by the `mode[7:0]` input. This module remains modifying the output until it is no longer enabled, being completely independent from any of the clock inputs (i.e. this module works by combinatorial logic).

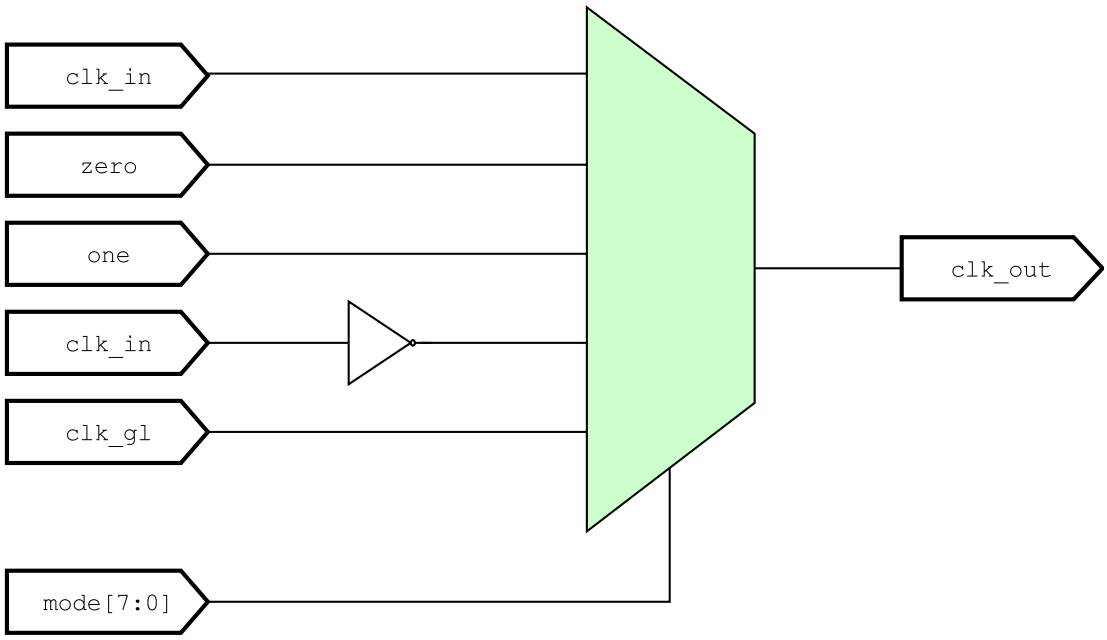

Figure 4.2: Multiplexer inside the core module of the glitcher. This allows easy selection of which source will be output from the core by changing the value provided in `mode[7:0]`. The available sources are the normal clock (`clk_in`) , a logic zero (`zero`), a logic one (`one`), the normal clock through a NOT gate and the glitching clock (`clk_gl`).

In most of cases, only the last mode will be used, since it outputs the clock used for glitching. However, as it will be later described, a bypass might be used to indicate a glitch that does nothing, but that stills count time, which is useful for synchronising the target and the glitcher.

Glitcher's main module

For glitching purposes, the core module is enough, since it provides all the necessary steps for generating a glitched clock signal. However, to interface it, it would be necessary to control the timing of enabling and disabling it. Such timing is critical, since usually a glitch is fast and short, as in a few clock cycles. For such reason, the main module was designed, as shown in 4.3. Its goal is to provide an easier interface to control for how long should the core stay enabled, as well allowing to run glitches in sequence.

For each glitch, there are three values that must be specified: a *delay*, the *width* of the glitch and the *mode*. The last one must be one of the modes available on the core module. The delay is a 16-bit long value that indicates how many clock cycles should the glitcher wait before glitching, while the width is a 8-bit long value that indicates for how many clock cycles should the core be enabled. This allows a precise glitch: after waiting for a few *delay* cycles (without modifying the clock signal), glitch for *width*, returning back to the normal clock after that.

Since in most of the cases more than one glitch might be necessary, a FIFO (*First In First Out*) queue was implemented, where each element is a 32-bit long value that represents the mode (8 bits), the delay (16 bits) and the width (8 bits) of each glitch operation. The main module is responsible for reading this FIFO as soon as it gets full, executing all glitches specified on it. However, before reading the FIFO and executing the glitches, the target must be reset and the glitcher must wait until it boots - tasks that will be further explained in this section and are not relevant for the understanding of this module. The whole process can, therefore, be implemented as a state machine, shown in figure 4.4. It has the following states:

- *idle*: the module is waiting to be activated, which happens when the FIFO is full.
- *wait*: the module is waiting for the target to reset.
- *read*: the module is reading the next entry from the FIFO.
- *delay*: the module is running the delay specified.
- *width*: the module is glitching (the core is enabled during this state).

The state machine is trivial and sequential, since it just has to keep reading the FIFO until it is empty. After reading the new glitch settings, the module will enter the *delay*, *width* or *read* state, depending on the settings. If the delay duration is zero, the glitcher will glitch straight away (entering the *width* state). If a width was not specified, the glitcher will execute only the delay. This is useful when one wants to wait for a very long time before glitching. Finally, if both delay and width values are zero, then nothing

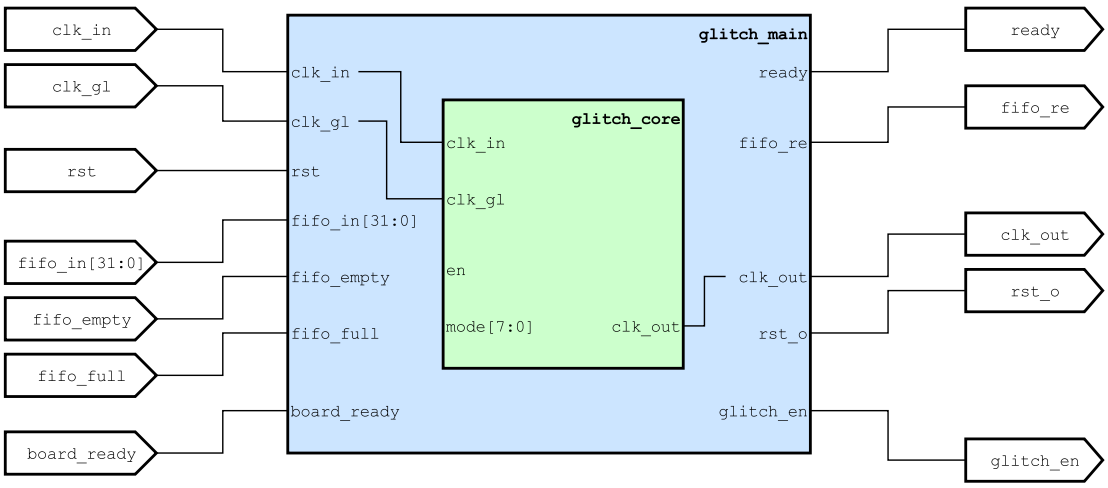

Figure 4.3: Main module of the glitcher. Note that there is an instance of the core module inside the main module. This allows this module to have complete control of the core, being now responsible for enabling and disabling it whenever necessary. The clock signals (`clk_in` and `clk_gl`), however, are wired directly to the core module, being used directly as provided. Wires to handle the reading process of the FIFO (`fifo_in[31:0]`, `fifo_empty`, `fifo_full` and `fifo_re`) are also provided, together with a trigger indicating that the target has finish its boot (`board_ready`).

must be done - it's exactly like a `NOP` CPU instruction. In this case, the module will just go back to the *read* state.

It's important to note that each state transition takes one clock cycle. This adds an internal delay for the glitcher to work, so even when the delay value is set to zero, there will be a small gap with normal clock before glitching. Also, when glitching for one clock cycle, multiple rising edges will be triggered, which affects the glitcher's precision to hit only one instruction. This means that, due to all the internal delays, the glitcher takes a few clock cycles to start and glitches for more clock cycles than it should. During the experiments, however, the glitcher's performance and efficiency were not affected, and, as will be shown in Section 5, it was perfectly possible to glitch the targeted instructions.

This module, just like core module, receives as input both clocks, `clk_in` and `clk_gl`. Those clocks are actually bypassed straight to the core, which is instantiated inside the main module. It also receives a reset signal (`rst`), which is used to reset the whole module, its counters and variables. The other inputs are related to the FIFO (`fifo_in[31:0]`, `fifo_empty` and `fifo_full`), except for the last one, `board_ready`, which is an external signal used to indicated that the target has finished its booting process.

In terms of output, the main module provides outputs the direct result from the core, `clk_out`. It also outputs a ready signal, `ready`, used to indicated that this module is on *idle* state and ready to be used. The `rst_o` output is used for resetting the target, which will be later discussed. Finally, the main module has also a `fifo_re` output,

which is an internal signal used for reading the FIFO module, and a `glitch_en` signal, which is used to indicated that the core is enabled. This last output will be later used for debug purposes.

Glitcher and the Die Datenkrake

The glitcher, as described until now, works as a standalone module and is not integrated to the *Die Datenkrake*. The reason is because internally the DDK uses a Wishbone bus interface to communicate its modules, which is not available yet on the glitcher. Because of that;, it's necessary to create a wrapper around the main module, allowing the glitcher to be used inside the DDK, as shown in figure 4.5.

The first important difference on this module is the presence of three clock signals: `clk_i`, `clk_in` and `clk_gl`. All three clocks are provided by DDK's PLL, which generates 33, 50 and 99 MHz signals. The first clock is the DDK's system clock - this module and any other on the platform has to run with the same clock, which is of 50 MHz. The second is for the target's normal operation - since the target runs at a lower speed than the DDK, such clock must be provided. Finally, the third is the clock used for glitching, which is of a higher frequency than the second. While the system clock runs at 50 MHz, as it is clear in figure 4.5, the clock used for the driving the target is also used as the "system clock" for the main and core modules. The reason is because the glitcher has to be synchronized with the target, otherwise the clock-based internal counters would be out of sync. Therefore, there are two clock domains: the DDK's, at 50 MHz, and the target's, at 33 MHz.

With two clock domains, operations that are clock-dependent between this module and the main won't work properly anymore, since one is running faster than the other. However, the glitcher works by reading a FIFO which contains all the glitches that must be executed in sequence. This is a common technique for synchronizing between different clock domains, since the FIFO module is able to be written using a clock *different* than the one used for reading. This is extremely important in this scenario, since this module can now write to the FIFO at 50 MHz (since it runs at the system clock speed), while the main modulde can read it at 33 MHz (since this is its "system clock", being actually the target's clock).

Note that, on this module, there are no input wires for glitch settings (delay, width or mode), neither a way of directly writing to the FIFO. The reason is that everything now must use the Wishbone interface, including such tasks. Therefore, Wishbone registers were created, and by writing to them by specifying their addresses, it's possible to write to the FIFO. The list of registers is shown on table 4.1. Notice that, since each FIFO element is 32-bit long, it's necessary to use four 8-bit registers to write the values to the FIFO, since this is the width of data, as explained in Section 3. For simplification, when writing the last eight bits (`GLITCH_QUEUE_3`), the module automatically writes to the FIFO the value of all other registers together, forming a 32-bit long FIFO element, as shown in figure 4.6. The last two registers are used only for checking the status of the FIFO, which is used for self-testing, as it will be further discussed.

The glitcher wishbone interface has one extra input, `ch2_in[5:0]`, which corresponds to the input data from DDK's channel 2, and two extra outputs, `ch1_out[5:0]` and

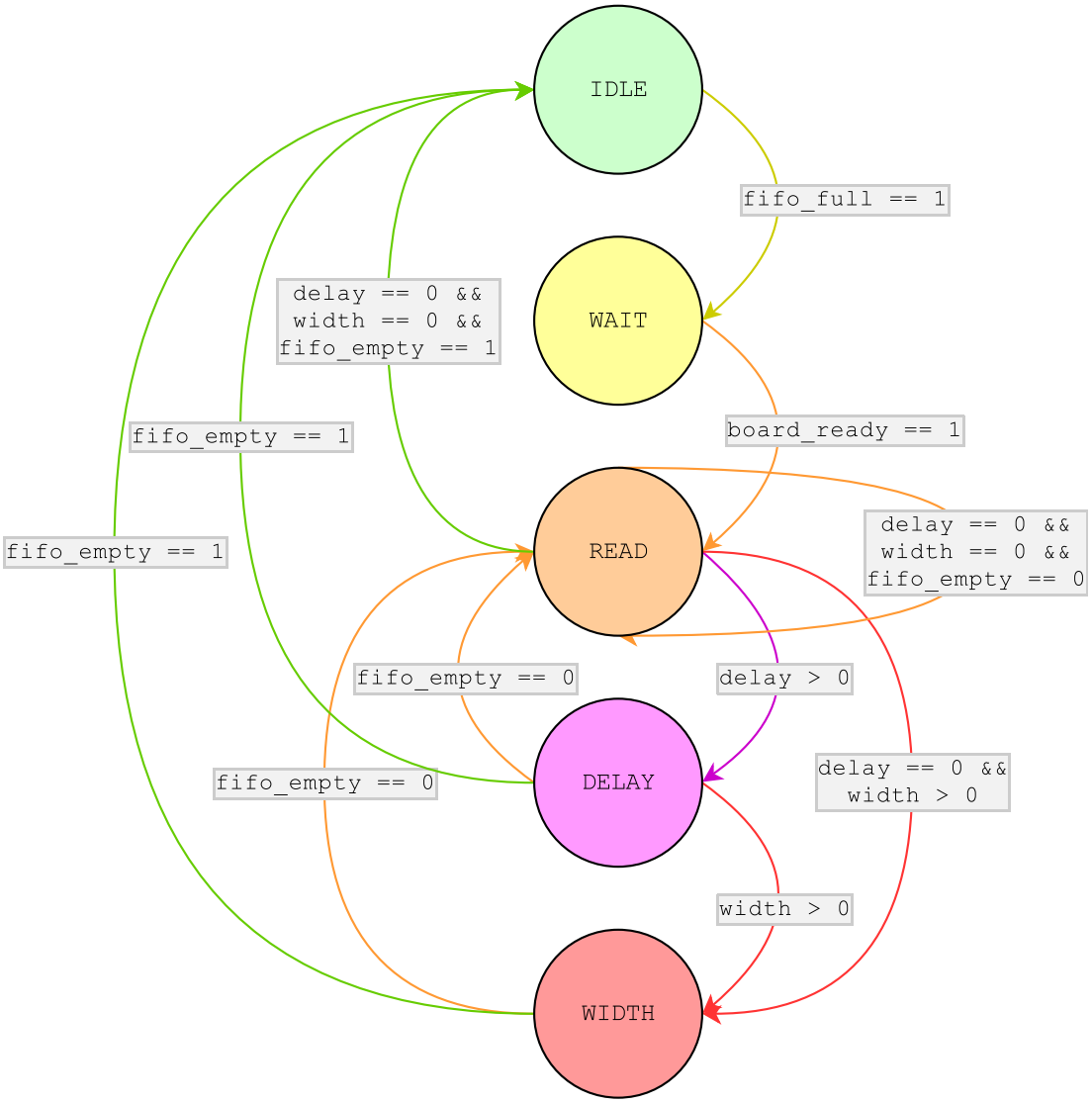

Figure 4.4: State machine of the main module. The glitcher begins on the *idle* state (green), changing to *wait* (yellow) as soon as the FIFO is full. After a trigger from the target indicating that its boot process has completed, the glitcher change to the state *read* (orange), from where it will start reading glitch configurations from the FIFO, alternating between the *read*, *delay* (pink) and *width* (red) states. Note that all four possible combinations are considered in this state machine: a "NOP glitch" (i.e. no delay and no width), a pure delay operation (i.e. no width), a direct glitch (i.e. no delay) and a delayed glitch (i.e. both delay and glitch width are present). After each operation, the next one on the FIFO is fetch, repeating the process. Finally, after the FIFO is empty, the glitcher returns to the *idle* state.

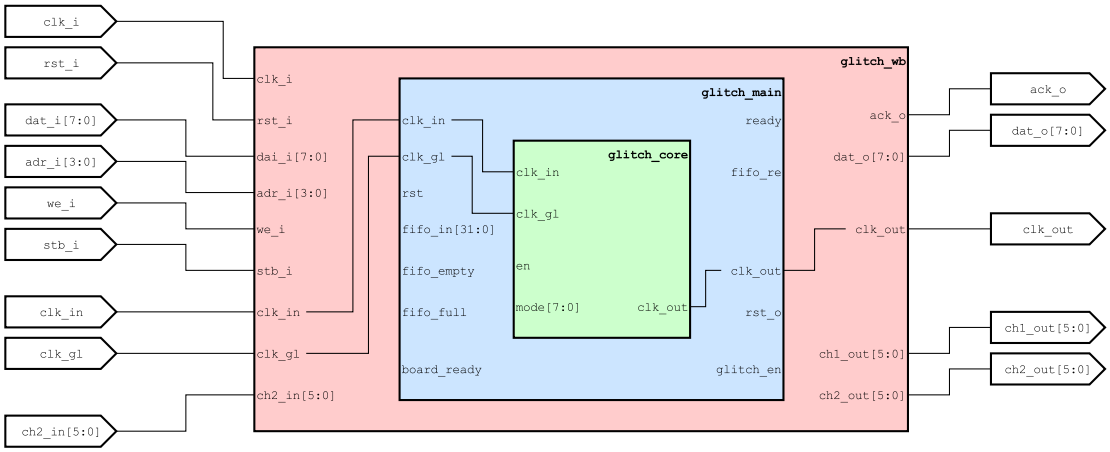

Figure 4.5: Wishbone interface of the glitcher. As shown in the image, this is a wrapper that allows the main module to communicate using the Wishbone protocol. Therefore, all configurations must be properly sent through the bus, being then propagated to the inner modules whenever necessary. Note also the existence of DDK channel input (*ch2_in[5:0]*) and output (*ch1_out[5:0]* and *ch2_out[5:0]*) ports, which will later be used to create an interface with the target, as well as the *clk_out*, exposed to facilitate the debug process.

Register	Description
`GLITCH_QUEUE_0`	Sets bits 7 to 0 of the new element
`GLITCH_QUEUE_1`	Sets bits 15 to 8 of the new element
`GLITCH_QUEUE_2`	Sets bits 23 to 16 of the new element
`GLITCH_QUEUE_3`	Sets bits 31 to 24 of the new element and queue it to the FIFO
`GLITCH_FIFO_EMPTY`	Indicates if the FIFO is empty
`GLITCH_FIFO_FULL`	Indicates if the FIFO is full

Table 4.1: List of Wishbone addressable registers and their functions.

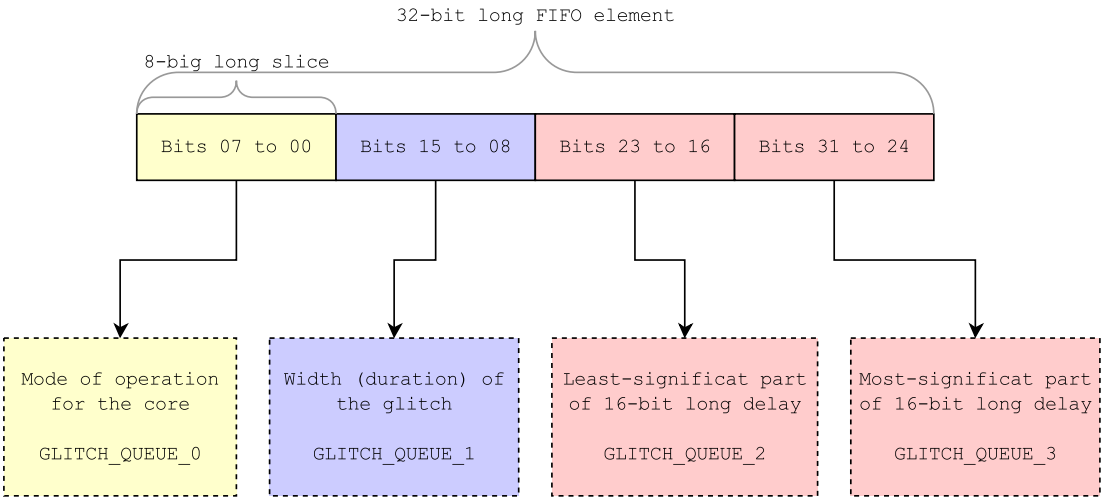

Figure 4.6: Relation between each 8-bit Wishbone register, 32-bit FIFO element showing the meaning of each slice. Note that, since the delay is a 16-bit element, it requires two slices from the 32-bit FIFO entry.

`ch2_out[5:0]`, which correspond to the output data from DDK's channels 1 and 2. Note that, each pin from a DDK channel is bidirectional and can be configured in runtime, since it uses tristate logic to select if a wire is connected as input or output. Therefore, not all wires from `ch2_in[5:0]` are used for input, just like not all wires from `ch2_out[5:0]` are used for output, being them manually adjusted on the channel configuration inside the DDK top module (which is not part of the scope of this work). A complete description of both channels and the signals indicating the ones used for input and the ones used for output is provided on table 4.2.

With a few exceptions, most of the signals are output by the DDK on those channels are used for debugging purposes. This was necessary during the development of the glitcher to be able to detect failures in the modules, missing signals and crashes. However, as it will be further discussed in Section 6, such debug pins can now be removed, reducing the number of channels necessary for the glitcher. Finally, further in this section it will be explained how such pins are used to create the interface between the DDK and the target, allowing the former to control, glitch and receive feedback from the later.

The DDK software was also modified to allow an easier control of the glitcher. As previously mentioned, the code running on the ARM processor interfaces with the the FPGA, and, through that, it is possible to use the ARM to control modules on the FPGA side of the board. The glitcher so far has been developed only on the latter, meaning that there is no interface with the external world besides the DDK framework. The original DDK code allows reading and writing into addresses that point to Wishbone registers into each channel. However, for facilitating writing elements into the FIFO, checking the glitcher status and performing tests, custom functions were developed. Such functions use the original DDK framework functions regarding interfacing the FPGA, but they offer a better user interface for one controlling the DDK through the console, allowing easier and faster tests and debug of the whole glitcher setup.

Target

The AVR microcontroller is the target in this work, and since it has to interact with the other components of the test environment, some modifications are necessary. Even though most modifications are only in software, they are necessary to activate hardware features necessary both for the experiments and the interface between the target and the DDK.

Target setup

The microcontroller as target used in this work is able not only to run with an external clock, but also from an internal RC oscillator, which is the default. For this reason, the first step is to change the clock source to `XOSC`, which is the external one. Such external clock can be provided by different devices, such as a quartz crystal oscillator (commonly used to provide a more accurate clock source to the microcontroller) or an external oscillator (e.g. the glitcher's clock output generate on the DDK). This has to be first operation, since if the external clock is not working, there's no point booting

Channel	Pin	Direction	Name	Description
1	0	*out*	`clk_out`	Clock output from the core
1	1	*out*	`clk_in`	Clock input from the core
1	2	*out*	-	*not used*
1	3	*out*	`ready`	Main module `ready` signal
1	4	*out*	`glitch_en`	Indicates the *width* state on the main module
1	5	*out*	`delay_en`	Indicates the *delay* state on the main module
2	0	*out*	`rst_o`	Reset output, used for reseting the target
2	1	*in*	`en_i`	Indicates that the board is ready (input from the target)
2	2	*out*	`board_ready`	Indicates that the board is ready (same as `en_i`)

Table 4.2: Description of DDK's channels 1 and 2 and its relation to the glitcher.

the microcontroller. Note that, even though the microcontroller has a PLL capable of multiplying the clock source frequency, it was kept disabled during the experiments, so that nothing would interfere in the clock provided by the DDK (which could affect the glitching).

To allow communication between the DDK and the target, the microcontroller's GPIO (*General Purpose Input/Output*) as configured accordingly. The GPIO consists of multiple pins on the board that can be toggled between logic zero and logic one, which is good for signaling. Note that, this is bidirectional: in case the DDK wants to signal something to the target, it would be also possible. However, in this work, only communication in the opposite direction (from the target to the DDK) was used.

The second method of communication is by utilizying an USART (*Universal Synchronous/Asynchronous Receiver/Transmitter*) port on the board. An USART port works by serial communication through TX (transmitter) and RX (receiver) pins. This can be used for transmitting larger chunks of data, such as strings. It is also easier to use than the GPIO in this setup, since the GPIO has to be read through the DDK, while the USART can be read directly on an external monitor.

Note that, all of this changes are done by software, being only necessary to add the code for such tasks. This allows easy changes on the settings, requiring only the microcontroller to be reprogrammed. The rest of the code, however, can be used as usual, without requiring extensive modifications.

Target-DDK interface

After the glitcher's FIFO is full, the glitcher's main module remains in *wait* state until it is triggered by the I/O of the target device, indicating that the target is ready. While changing to such state, the glitcher sends an external signal of reset, which is connected

directly to the target. The target then, is kept at reset for 255 clock cycles, giving the microcontroller sufficient time for the reset signal to propagate to all registers of the device. After the reset is finished, the microcontroller starts its normal boot process, which can take a random number of clock cycles.

It's important to note that, although the code doesn't change during resets, and that the AVR architecture is simple, the amount of time required for the target to boot varies. One of the possible reasons is the fact that the external clock requires a synchronisation step, which can depend on various factors, such as the environment, interference and wiring. This process can take oscillate the number of clock cycles necessary for booting the target in the factor of thousands of cycles, making turning impossible to know exactly how long it will take.

The trivial solution for detecting the boot is to make the target trigger the attacker, indicating that the boot was successful and that it is ready to continue. In this environment, after the microcontroller has finished booting, it sends a signal to the DDK, which receives on the `en_i` wire on channel 2. Finally, this signal is then processed through an edge detector, since we want to detect when it is toggled. This allows the main module of the glitcher to leave the *wait* state and go to *read*, where it starts reading the FIFO and executing the glitches according to the order that they were queued.

External monitoring

The necessary delay and glitch duration depends entirely on the code and the target instruction. Therefore, testing in sequence multiple combinations is highly desirable, since it allows extensive tests without human interaction. The solution for this was to create an script that not only generates the combination of all possible glitches inside a range and tests them, but also verifies each one if it was a proper glitch or not.

The first step is to generate a set of glitches to test. This can be done based on any criteria, but usually brute force is easier and more effective. The list can be formed both by single glitches (one glitch and then only bypass glitches - glitches formed by no delay and no width) and multiple glitches (more than one glitch followed by bypasses). Such combinations are usually created inside a limited range, since both the code and the number of clock cycles of each instruction are usually know. It's important to note that the number of combinations affects the duration of the experiment - dozens of thousands can take more than one day to complete. Each combination should be reproducible, which means that the same result must be obtained multiple times. Because of that, it's recommended to run each combination multiple times, definying an accuracy based on the stability of the glitch.

For each test, the commands for queueing the FIFO are passed to the DDK, which then executes the whole process by itself, including the target reset. The result is then checked by validation functions, which return whether or not the glitch was successful. Note that this step depends entirely on the code. Some of the experiments describe on Section 5 utilize the GPIO pins to activate a bit on one of the unused DDK channels, which then can be read and checked. However, the last experiment of this work used

the USART to check the validity of the glitch by reading the value received on the serial line and analysing it.

Finally, a *dummy run* (i.e. normal execution without glitching) is done on the target between glitches. The goal of such execution is to avoid one glitch to interfere with the next one. Since after the glitch is complete the target is reset, a normal execution would allow the detection of crashes on the microcontroller, which could not be detected when already glitching the device (i.e. it would not be possible to differentiate between a successful glitch or a glitch caused by an internal crash). Even though not critical, it is a good pratice to add this extra execution to avoid false positive results when running different glitch combinations in sequence. Note that, this process is also done within a predefined interval in a self-test procedure which verifies if either the DDK or the target crashed or is having any issues.

The self-test procedure for the DDK is simple. By running a dummy execution on the target and verifying the FIFO status, it's possible to detect possible wiring faults or crashes on the state machine. After running a dummy execution, the FIFO should be empty, since the DDK executed the whole glitch process. If not, it means that the DDK did not receive the signal indicating a target boot. Most of the cases, a simple reboot of all devices is sufficient to fix the problem. The target self-test is also simple, being necessary only to run a dummy execution and the glitch validation function. The function must return that the glitch is not valid, since no glitch was executed. If the opposite happens, either the validation function or the microcontroller is not working properly.

The script is also responsible for logging each test, its result and accuracy. This information can be later analysed and used to create another attack, such as a second glitch after the first. This then can be repeated, creating sequential glitches as long as necessary for attacking the target. Such scenario is demonstrated in Section 5, where the `strcpy` C function is sequentially glitched based on the results of previous glitches.

5 Results

This work analysed three scenarios against glitching: unconditional loops, conditional loops and the `strcpy` function from the standard C library. The first two were selected with the goal of demonstrating that loops can be glitched once their branch instructions are corrupted. The last one is done to demonstrate that C code can also be glitched, and also to demonstrate a real-world scenario where a simple function allows leakage of data once glitched.

Unconditional loops

For the first experiment, it was easier to glitch unconditional loops without code inside (infinite loops). For that, both jumps relative jumps instructions were studied and tested. For a more realistic analysis, code with multiple loops and code inside them were also experimented with.

Jumps

A jump instruction is used to change the current position of the program counter, pointing it to somewhere else in the memory. It consists of loading the specified value into the internal CPU register `PC`, changing the control flow of the code. The instruction `JMP` does this task on AVR microcontrollers, and it takes three clock cycles [Atm98]. It is important to note that this instruction is longer than other instructions: while most of the instructions on the AVR architecture are 16-bit long, this one requires 32 bits. The reason why instruction requires twice as many bits is because it holds a full 22-bit memory address on it, which does not fit inside one single 16-bit instruction. Therefore, for this instruction to be execute, it requires two clock cycles for fetching the full address (6 most-significant bits on the first cycle and the 16 least-significant bits on the second) and one extra clock cycle for loading it into the program counter [Atm10].

For glitching this instruction, an infinite loop using it was created, as shown in code 5.1 at lines 1 and 2. Using an infinite loop `breakme` allows the glitcher to be able to hit the instruction whenever possible, since the CPU will always be running the `JMP`. When glitched properly, the instruction is skipped, taking the execution flow to the second loop, `glitched`, located between lines 4 and 7. This second loop is responsable for keeping the GPIO pin 1 low, allowing the DDK to know that this was a valid glitch. Finally, since the `JMP` instruction takes three clock cycles, and there is no inside the loop, the glitcher will always hit it. By analyzing the graph 5.1, it is clear that glitching for two clock cycles is enough to glitch the instruction, which results in skipping it.

A more precise and useful analyzis of glitching the `JMP` can be done by introducing a second loop with irrelevant code inside, as shown in code 5.2. In this second code

```
1 breakme:
2     jmp breakme

4 glitched:
5     ldi r24, 0b11111100
6     sts PORTD_OUTSET, r24
7     rjmp glitched
```

Code 5.1: Unconditional JMP loop code example. This is an infinite loop - the CPU will always execute the same instruction until it is glitched. By skipping this instruction, it is possible to reach the loop that signals a successful glitch.

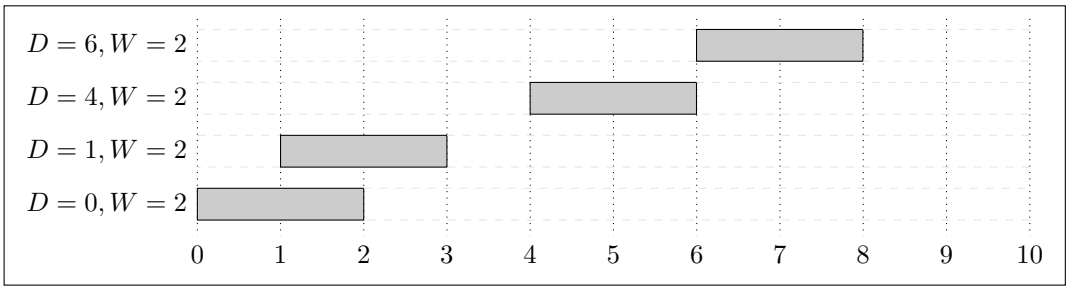

Figure 5.1: Timing diagram of unconditional JMP loop glitching tests. The glitch parameters are represented by D (delay in clock cycles) and W (width in clock cycles), being executed in different runs (i.e. there is a full target reset between each glitch). The darker blocks in the timeline represent the clock cycles that, by enabling the glitcher, a successful glitch happened.

```
1  breakme:
2      jmp breakme

4  second:
5      ldi r24, 0x00
6      sts PORTE_OUTSET, r24
7      ldi r24, 0xFF
8      sts PORTE_OUTSET, r24

10     jmp second

12 glitched:
13     ldi r24, 0b11111100
14     sts PORTD_OUTSET, r24
15     rjmp glitched
```

Code 5.2: Unconditional double JMP loop code example. In this example two glitches must be performed to be able to exit both infinite loops. The second loop has code to toggle the LEDs on the XMEGA evaluation kit, meaning that the CPU won't be always executing the instruction targeted. Therefore, a precise delay before the second glitch is required. Finally, as the previous example, after all glitches, an infinite loop triggers the DDK indicating a successful glitch.

snippet, the first loop `breakme` presented before still exists, but then the code hits a second loop, `second`, which has code inside. Although the code inside this new loop is related to the LEDs connected to the microcontroller board, understanding it is not necessary.

The timing diagram for the glitches of the code 5.2, shown in the figure 5.2, shows a clear gap in the timeline where no successful glitches were achieved. In this gap, the instructions for blinking the LEDs are running, and since glitching them will not force the flow out of the current loop, they are not relevant. It is also possible to see that glitching in sequence also force second loop to exit, which is due to the fact that the glitcher is not precise enough and will hit multiple instructions, and also due to internal delays. Therefore, there is a chance that the glitcher is hitting the second `JMP` instruction, even though the second delay is set to zero.

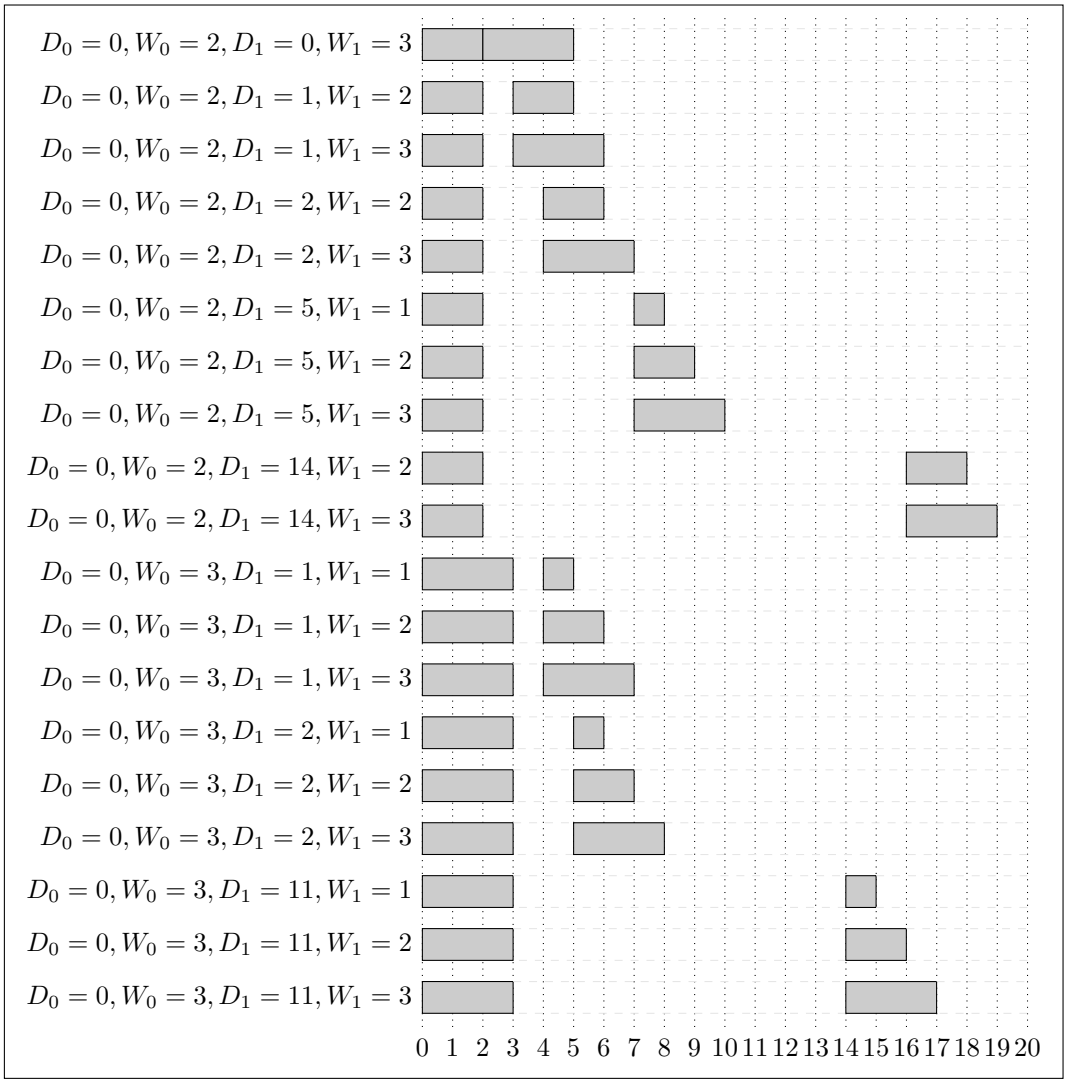

Figure 5.2: Timing diagram of unconditional JMP double loop glitching tests. In this graph, D_0 and W_0 correspond to the delay and width of the first glitch, while D_1 and W_1 correspond to the delay and width of the second glitch, with all units in clock cycles. The gaps between glitches indicate that the loop was executed at least once, since glitching the LED toggling instructions have no effect in the loop itself.

Relative jumps

Another type of unconditional branch is the *relative jump*, which is represented by the instruction `RJMP`. This instruction works by changing the flow of the program to the address resulting of the sum between the current program counter address and the specified value in the instruction, plus one. This instruction also takes two clock cycles instead of three of the `JMP` [Atm98], since here the address is 12-bit long (while the opcode itself occupies four bits), instead of 22-bit as the previous one. Since this is a relative and not a direct branch, its argument consist of an offset from the current location, which is considerably smaller than the full address. Therefore, one clock cycle is necessary for executing the arithmetic operation for calculating the new address, while another is required for effectly loading it into the register [Atm10].

The code snippet 5.3 shows how the `RJMP` was initially tested. The concept is the same as the previous instruction - an infinite loop that forces the instruction to be glitched at any moment. Just like with the `JMP`, the `RJMP` can be glitched, which causes the CPU to skip the instruction, allowing it to get out of the `breakme` loop and hit the second loop, `glitched`. This second loop is then used to allow the DDK to detect the glitch. However, as shown in 5.3, the `RJMP` can also be glitched within one to three clock cycles, while the `JMP` requires exactly two clock cycles of glitching to be always skipped.

Just like the `JMP`, to have a more precise analysis of the `RJMP` instruction, a second loop `second` with code inside of it was introduced, as shown in code 5.4. By testing all possible combinations inside a finite range, the timing graph 5.4 shows both glitches being executed, being the first glitch responsable for the `breakme` loop and the second for the `second`. It is clear on the graph that the second glitch can be execute either right away (since the internal delays of the glitcher are enough to get through the contents of the second loop) or a while after, which means that during the execution of the loop itself no glitch combinations are useful.

```
1  breakme:
2      rjmp breakme

4  glitched:
5      ldi r24, 0b11111100
6      sts PORTD_OUTSET, r24

8      rjmp glitched
```

Code 5.3: Unconditional RJMP loop code example. This code has the same behavior as the code 5.1 - an infinite loop forcing the CPU to execute always the same instruction. After the infinite loop is glitched, the DDK is triggered indicating a successful glitch.

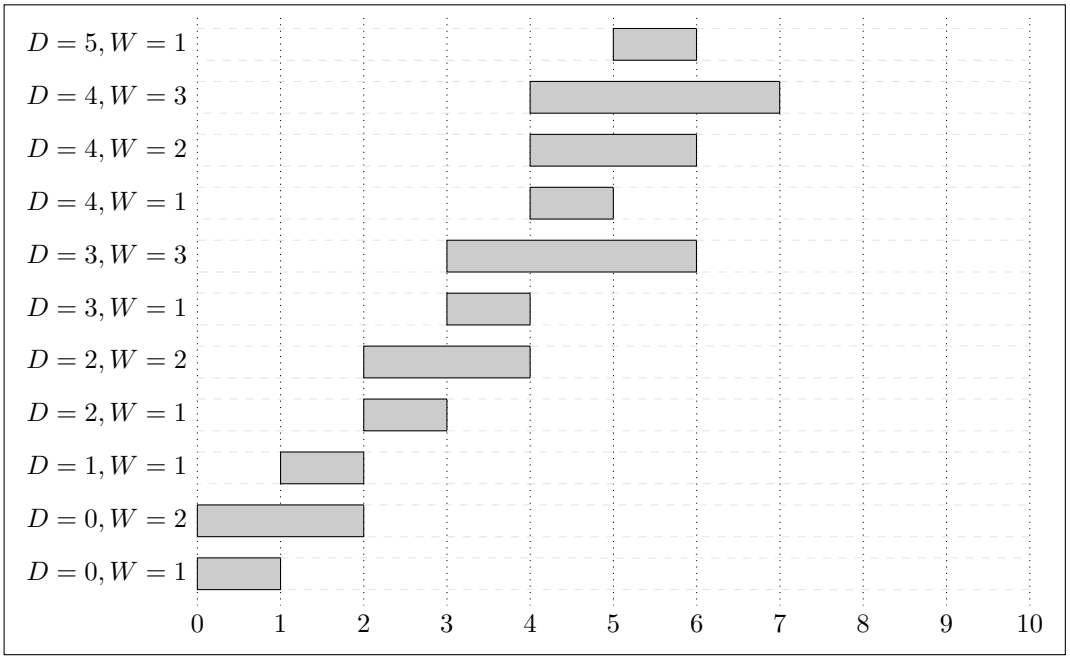

Figure 5.3: Timing diagram of unconditional RJMP loop glitching tests. In this graph, D corresponds to the delay (in clock cycles), while W corresponds to the width (also in clock cycles). Glitching on almost every clock cycle is possible since there are no instructions inside this loop (i.e. this instruction is always being executed on the CPU). Not less important, the graph shows that multiple glitch widths are valid and have the same effect on the instruction.

```
1  breakme:
2      rjmp breakme

4  second:
5      ldi r24, 0x00
6      sts PORTE_OUTSET, r24
7      ldi r24, 0xFF
8      sts PORTE_OUTSET, r24

10     rjmp second

12 glitched:
13     ldi r24, 0b11111100
14     sts PORTD_OUTSET, r24

16     rjmp glitched
```

Code 5.4: Unconditional RJMP double loop code example. This code has the same behavior as the code 5.2 - after exiting both loops the DDK is triggered indicating a successful loop. Not less important, the contents of the second loop force a precise delay to be used, since the instruction is not always being executed on the CPU.

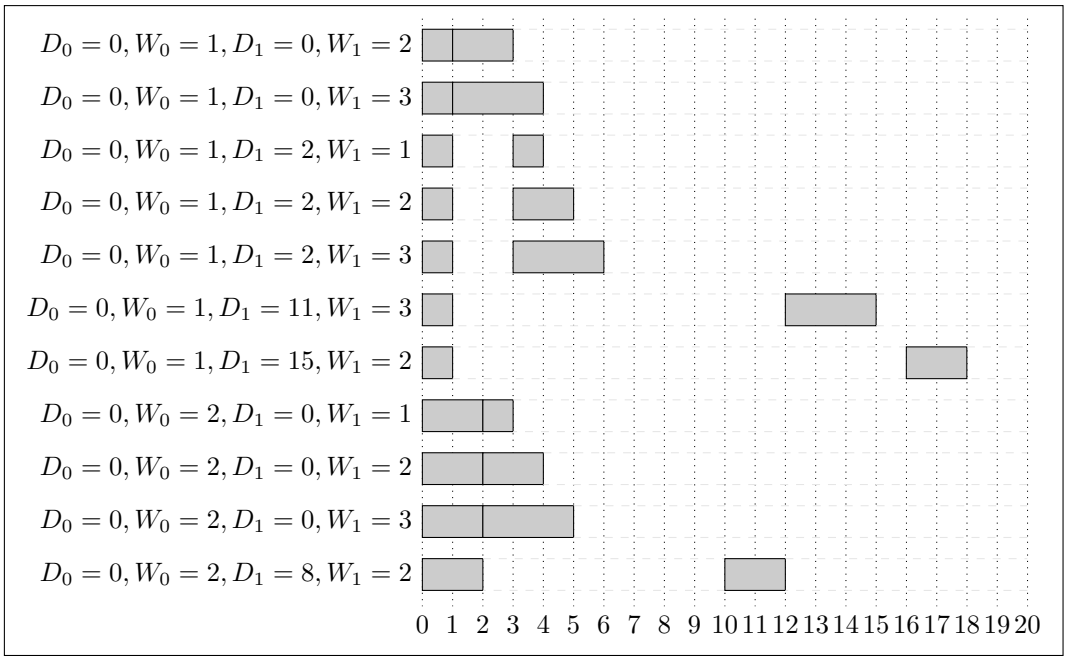

Figure 5.4: Timing diagram of unconditional RJMP double loop glitching tests. In this graph, D_0 and W_0 correspond to the delay and width of the first glitch, while D_1 and W_1 correspond to the delay and width of the second glitch, with all units in clock cycles. Note that the second glitch can either be executed directly after the first, passing the second loop once, or a few clock cycles later, passing the second loop more than once. This means that a glitch in the middle of the loop is not successful, since it glitches the instructions not related to the loop itself, but with the LED toggling.

Conditional loops

Most of the loops in codes are conditional loops and not infinite. This means they depend on some condition, which can be virtually anything - from a value of a counter variable to an external input from the user. Because of that, two types of conditional branches were tested: *branch if equal* and *branch if not equal*. Those are easily found when analyzing code compiled by the AVR-GCC toolchain, and for such reason they were selected among the multiple types of branch instructions available on the AVR microcontroller CPU.

Branch if equal

Branch if equal is a type of conditional branch used to change the current flow of the program if the compared values are equal. Internally, this is represented by the resulting value of the subtraction of both values - if zero, the branch is executed, otherwise it is skipped. For this reason this instruction is also know as *branch if zero*.

On AVR microcontrollers, this type of branch is represented by the instruction `BREQ` and takes from one to two clock cycles [Atm98]. The first clock cycle is for checking the internal CPU flag `zero`, which indicates if the last operation returned zero. The second clock cycle only exists if the result of the previous step is *true* (i.e. the zero flag is active) and consits of changing the current flow of the program by changing the program counter accordingly.

For testing this instruction, the code 5.5 was used. Just like the initial tests with unconditional loops, this is an infinite loop `breakme` with no code inside, except with the `CPI` instruction, which is responsible for activating the CPU flag `zero` by comparing the register `r24` with the immediate value `0xFF`. The register was previously filled with this value, which allows the comparison to be *true*.

As expected, the results are similar to the ones achieved with `JMP` and `RJMP`: it is consistent over time, as shown in the figure 5.5. This means that it does not matter when the glitch happens, as long as it is done within two clock cycles. Note that, in this scenario, it is possible to glitch the `CPI` instruction as well. However, this does not affect the results negatively since, by slowing increasing the delay, it is possible to skip this instruction (which takes one clock cycle [Atm98]).

A pratical example of a loop with `BREQ` is shown in the C code 5.6. Once compiled, the resulting output is similar to the code 5.5. This is important because it shows that the instruction is used even on very simple code snippets.

Just like the previous tests, code with two loops were also tested with `BREQ`. The code 5.7 behaves exactly like the previous examples, but it uses conditional loops to create an infinite loop (since the condition is always satistifed). Note that the last loop, `glitched`, does not need to use the `BREQ` instruction since it will not be glitched and can use a simple relative jump.

Just like the previous double glitch experiments, after glitching the first loop, it is only a matter of finding the perfect timing to glitch the second one. As shown in the figure 5.6, there are multiple combinations that will force both loops to exit.

```
1  breakme:
2      cpi r24, 0xFF
3      breq breakme
```

Code 5.5: Conditional BREQ loop code example. This code compares the register `r24` (whose value was previously set) to a constant and then repeats the loop if they are equal. Note that, even though the compare instruction can also be glitched, by increasing the delay this will not affect the experiment results.

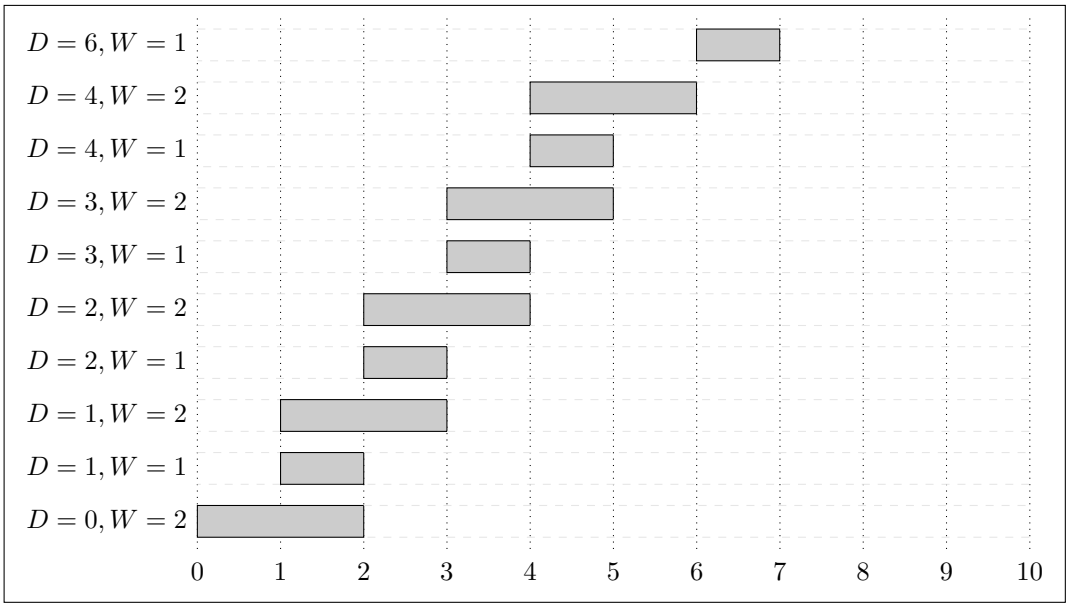

Figure 5.5: Timing diagram of conditional BREQ loop glitching tests. In this graph, D corresponds to the delay (in clock cycles), while W corresponds to the width of the glitch (also in clock cycles). As the graph shows, it is possible to glitch this instruction with different combinations of width. Also, by increasing the delay, the comparison is being skipped, allowing the glitcher to hit the branch instruction.

```
1  volatile unsigned int i = 255;

3  while (i == 255) {
4      // Do something.
5  }
```

Code 5.6: Example of BREQ loop in C code. This code, once compiled, will result into a loop that uses the `BREQ` instruction on it.

```
1  breakme:
2      cpi r24, 0xFF
3      breq breakme

5  second:
6      ldi r24, 0x00
7      sts PORTE_OUTSET, r24
8      ldi r24, 0xFF
9      sts PORTE_OUTSET, r24

11     cpi r24, 0xFF
12     breq second

14 glitched:
15     ldi r24, 0b11111100
16     sts PORTD_OUTSET, r24

18     rjmp second
```

Code 5.7: Conditional BREQ double loop code example. Note that this code behaves exactly as the previous examples, except that the first loop will not always execute the same instruction due to the presence of the comparison instruction. After glitching both loops (and exiting them), the target triggers the DDK indicating a successful glitch.

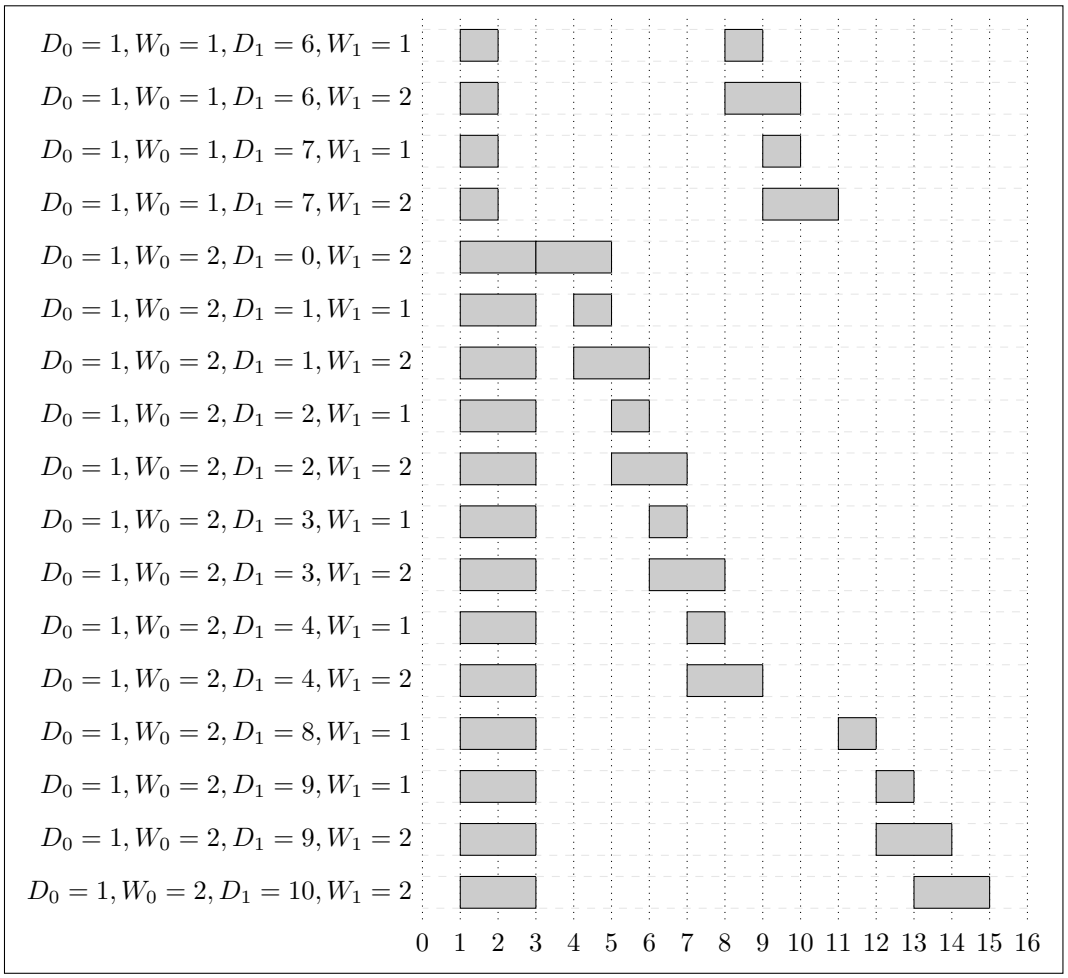

Figure 5.6: Timing diagram of conditional BREQ double loop glitching tests. In this graph, D_0 and W_0 correspond to the delay and width of the first glitch, while D_1 and W_1 correspond to the delay and width of the second glitch, with all units in clock cycles. The gaps between glitches indicate that the loop was executed at least once, since glitching the LED toggling instructions have no effect in the loop itself. Also, since the `CPI` instruction takes one clock cycle [Atm98], increasing the delay by one cycle allows the glitcher to skip it and hit the `BREQ`.

Branch if not equal

The second conditional branch instruction tested was the *branch if not equal*, represented by `BRNE`. This instruction behaves exactly like the previous one, except it branches if the `zero` flag is *not* active. For this reason, it is also known as *branch if not zero*. This branch instruction takes two clock cycles to be executed for the same reasons as the `BREQ`: the first clock cycle is required for checking the flag, while the second one is for executing the branch itself.

The instruction was tested just like the `BREQ`, as shown in code 5.8. Note that the register `r24` was previously loaded with a value diferent than `0xFF`, which allows the `CPI` (*compare with immediate*) instruction to properly set the `zero` flag as not active. Finally, the `CPI` instruction can be ignored, since multiple experiments with the increasing delay setting will force it to be skipped.

The figure 5.7 shows in which moments the glitch was possible. As it is clearly visible, it repeats itself over time, since it is running an infinite loop. By increasing the delay, it would be possible to skip the `CPI` instruction and glitch the `BRNE` itself, forcing it to glitch the correct instruction, even when the initial timing (no delay) is off.

The importance of this instruction is visible when optimized C code is disassembled. The code 5.6 previously demonstrated uses the `BREQ` instruction when compiled without any optimization. However, in embedded systems optimization is a common pratice, since the size of the binary and the performance are critical as the storage capacity and processing power are limited. When compiled with optimizations enabled, this code uses `BRNE` instead of `BREQ`, and through empirically analysing the disassembled code of different types of loop (`while` and `for`), it is clear that the compiler prioritizes the use of this new instruction instead of the normal `BREQ`.

A double glitch of the `BRNE` was also tested with the code 5.9. Just like the previous experiments, it is possible to exit both loops (`breakme` and `second`) as long as the timing is accurate enough, as shown in the figure 5.8.

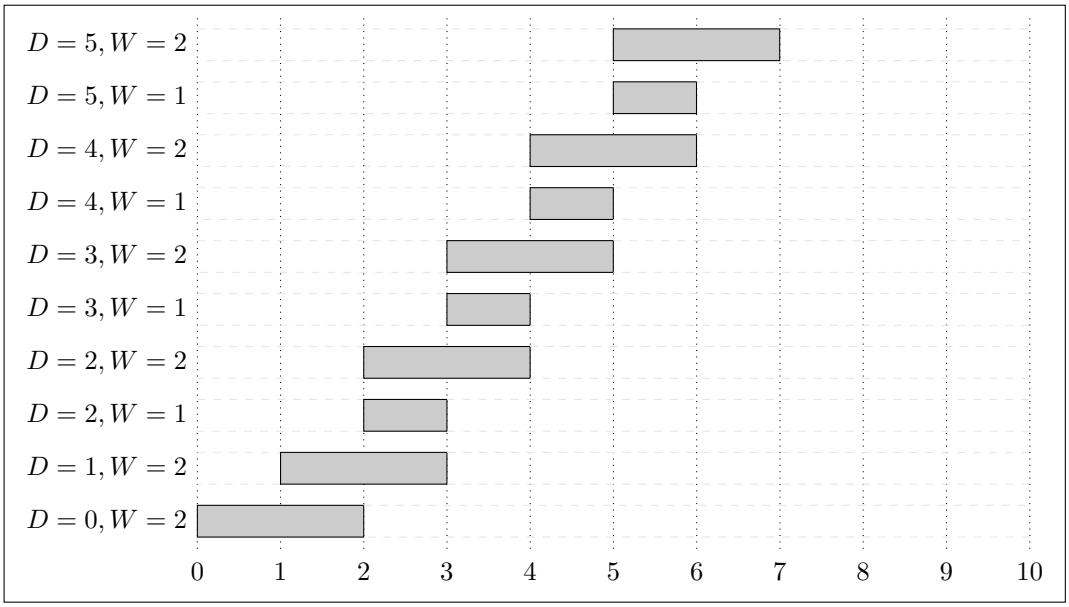

Figure 5.7: Timing diagram of conditional BRNE loop glitching tests. In this graph, D corresponds to the delay (in clock cycles), while W corresponds to the width of the glitch (also in clock cycles). It is possible to glitch this instruction with different combinations of width, as shown in the graph. Not less important, by increasing the delay, the comparison is being skipped, allowing the glitcher to hit the branch instruction.

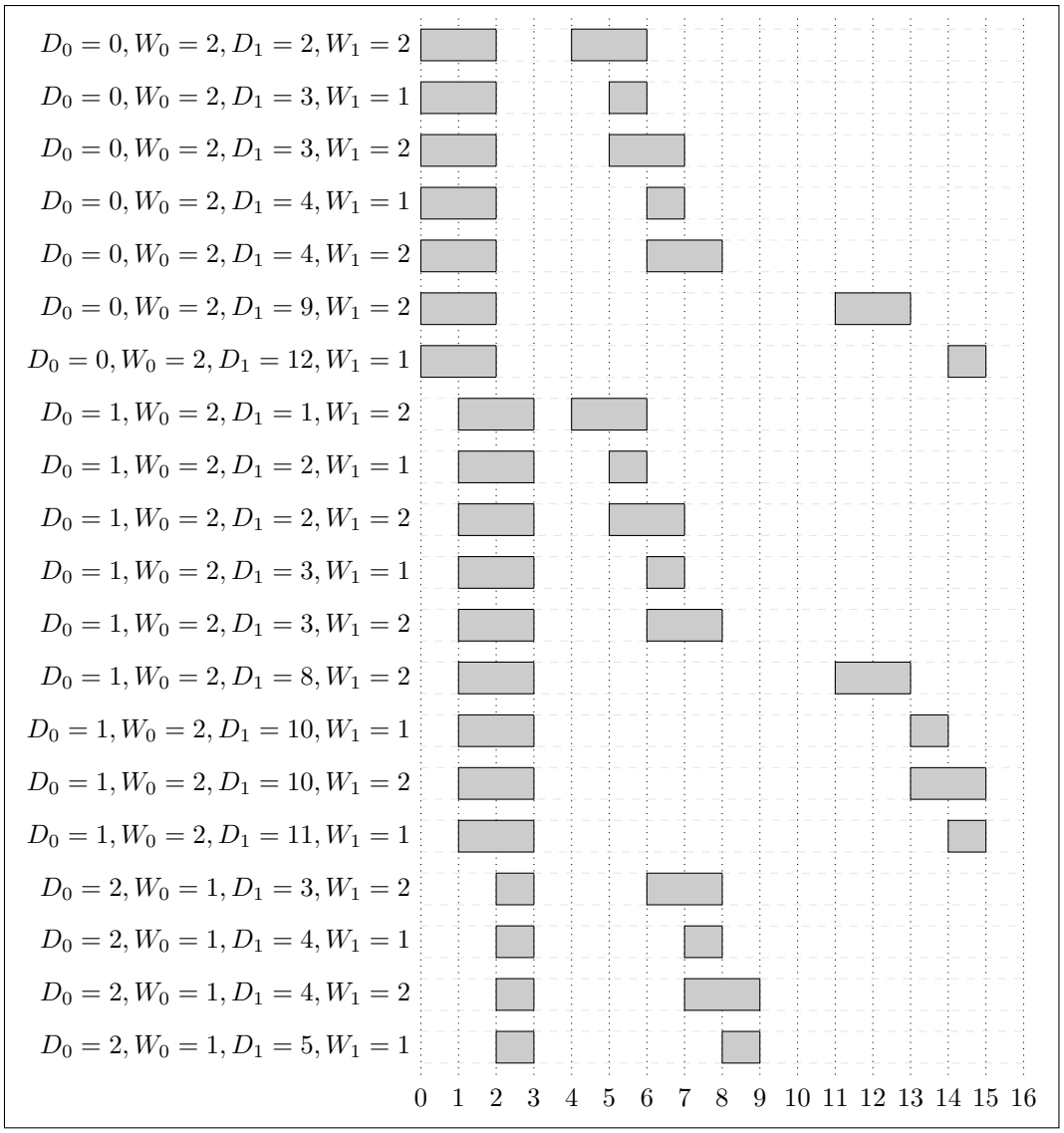

Figure 5.8: Timing diagram of conditional BRNE double loop glitching tests. In this graph, D_0 and W_0 correspond to the delay and width of the first glitch, while D_1 and W_1 correspond to the delay and width of the second glitch, with all units in clock cycles. The gaps between glitches indicate that the loop was executed at least once, since glitching the LED toggling instructions have no effect in the loop itself.

```
1  breakme:
2      cpi r24, 0xEE
3      brne breakme
```

Code 5.8: Conditional BRNE loop code example. This code behaves exactly as the code 5.5, where an infinite loop can be glitched even though the comparison instruction is present before the branch.

```
1  breakme:
2      cpi r24, 0xEE
3      brne breakme

5  second:
6      ldi r24, 0x00
7      sts PORTE_OUTSET, r24
8      ldi r24, 0xFF
9      sts PORTE_OUTSET, r24

11     cpi r24, 0xEE
12     brne second

14 glitched:
15     ldi r24, 0b11111100
16     sts PORTD_OUTSET, r24

18     rjmp glitched
```

Code 5.9: Conditional BRNE double loop code example. Note that this code behaves exactly as the previous examples, including the fact that the first loop will not always execute the same instruction due to the presence of the comparison instruction. After glitching both loops (and exiting them), the target triggers the DDK indicating a successful glitch.

The strcpy function

To simulate a more realistic scenario, a C code was also glitched. This code uses the `strcpy` function, which is responsible for copying a *null-terminated* string from one point of the memory to another. This functions works by checking the current byte being copied, and since strings in C end with a *null* byte, it stops when one is found. The goal of this experiment is to glitch on the exact moment where the function should stop copying (i.e. when the null byte is found), forcing it to copy more bytes than the original string. As result, the function would copy also variables around it in the memory, which could reveal hidden or secret information on the code (such as encryption keys, paths or memory addresses).

In the following sections it will be explained the attacked code, as well the goal of this attack. Limitations regarding what can be achieved with the current glitcher implementation when attacking the `strcpy` function are also discussed, followed by an explanation showing how to overcome them. Finally, the glitching results will be then described, showing what can be achieved with single or multiple glitches attacks.

The exploitable code

The code 5.10 shows a simplified version of the attacked code. The original code has calls to enable the external clock and UART and to synchronise with the DDK, but those parts of the code are not relevant for this analysis. It consists of seven global variables (lines 1 to 7), with the one right in the middle (line 4) the string that will be copied to a local buffer. The program begins by defining a local variable, `buffer`, 256 bytes long, which will be used to store the copied string. After that, the `strcpy` is called (line 13), passing as first argument the destination buffer (`buffer`) and the origin string (`foobar`). Later, `buffer` is printed using `fputs` (line 14) and the codes enter an infinite loop of `NOP` instructions (line 16).

The goal of this attack is to glitch the `strcpy` function exactly when the null byte that terminates the `foobar` string is being checked. By doing that, the next string in the memory will be copied, until the function hits another null byte. Once printed over the UART port, it would be possible to see multiple strings (i.e. the original `foobar` string and one of the secret strings around it).

strcpy internals and padding

Part of the disassembled code of the `strcpy` function is shown in code 5.11. The part here discussed is the relevant one, since it contains the actual loop that copies the string.

It works by loading into the register `r0` the current byte pointed by the special register `Z` and then storing it into the address pointed by the special register `X`. Both registers are incremented after being used, which means that the pointer advances in the memory. Finally, `strcpy` checks if the byte is null by running it through an `AND` instruction (which sets the `zero` flag if the value is zero) and looping if not by using the `BRNE` instruction.

Since the goal is to copy even when a null byte is found, glitching the loop (`BRNE`) is not useful. As previously shown in the experiments, glitching this instruction results in

```
1  char secret0[] = "000000";
2  char secret1[] = "111111";
3  char secret2[] = "222222";
4  char foobar[]  = "foobar";
5  char secret3[] = "333333";
6  char secret4[] = "444444";
7  char secret5[] = "555555";

9  int main()
10 {
11     const char buffer[256] = {0};

13     strcpy(buffer, foobar);
14     fputs(buffer, stdout);

16     while (1) { asm volatile("nop\n"); }
17 }
```

Code 5.10: Exploitable C code calling the strcpy function. This code copies a global string into a local buffer and then prints it. After that, the CPU is held at an infinite loop.

```
1  loop:
2      ld r0, Z+
3      st X+, r0
4      and r0, r0

6      brne loop
```

Code 5.11: Partial disassembled code of the strcpy function. Note the presence of the load instruction (`ld`) with a post-increment for reading one byte from the source string and the store instruction (`st`) with a post-increment for writing one byte to the destination buffer. After that, the byte is verified in case of zero with an `AND` instruction to detect an end of string.

```
1  char* strcpy(char *dest, const char *source)
2  {
3      asm("nop\n"); asm("nop\n");
4      asm("movw r30, r22\n");
5      asm("nop\n"); asm("nop\n");
6      asm("movw r26, r24\n");
7      asm("nop\n"); asm("nop\n");
8
9      asm("strcpy_loop:\n");
10     asm("ld r0, Z+\n");
11     asm("nop\n"); asm("nop\n");
12     asm("st X+, r0\n");
13     asm("nop\n"); asm("nop\n");
14     asm("and r0, r0\n");
15     asm("nop\n"); asm("nop\n");
16     asm("brne strcpy_loop\n");
17     asm("nop\n"); asm("nop\n");
18 }
```

Code 5.12: Padded version of strcpy inserted as C code. Note that between each instruction a `NOP` was placed to avoid pipeline issues.

skipping it and not forcing it to loop. Therefore, the points where one might want to glitch are the `AND` instruction (to glitch the comparison) or the `LD` instruction (to read a different byte from the memory). By attacking any of them, it would be possible to not set the `zero` flag, which would force the loop to continue even when `r0` is null.

Unfortunately, as previously noted, the glitcher is not precise enough to glitch fast instructions such as `AND` (which takes one clock cycle [Atm98]). Glitching the `LD` would be hard as well, since it takes two clock cycles, and glitching it might damage the instructions around it for the same reason. Therefore, for simplifying the attack, the `strcpy` was padded between every instruction with two `NOP` instructions. This allows the glitcher to hit one specific instruction without disrupting another one (so hitting the `AND` would not damage the fetching and decoding of the `BRNE` in the pipeline, for example). By inserting the code 5.12 into the exploitable C code, the program is forced to use our padded version of the function, which can be more easily glitched.

Results

Extensive tests were done for finding the timing to glitch this code. By specifying a long range of delay values, it is possible to get the whole original string to be copied before glitching. Initially only one secret was captured, but by reproducing the attack and trying to locate the timing of a second and third glitch, it is possible to copy multiple strings around the original one.

Capturing the first secret

This is the first attack done and its goal is to force the `strcpy` to copy not only the `foobar` string, but one of the secrets around it. Since we want the whole original string to be copied first, the glitch must be precisely done right when the null-byte is being read or tested. This means that, unlike the previous experiments, there are no multiple ranges of possible glitch settings, but a small one which is the window of glitching that will hit the null byte, as shown in the figure 5.9.

Using a script to monitor the output from the target's UART, it is possible to see both strings, as shown on log 5.13. The tests demonstrated on it are equivalent to the both small windows of glitching represented in graph 5.9. As shown in the log, first the original string is copied (`66 6F 6F 62 61 72`, `"foobar"`), then a bogus byte (`0x10`) and finally the next string in the memory (`32 32 32 32 32 32`, `"222222"`), which correspond to the `secret2` string. The corrupted byte in the middle is present since it is most likely that the `LD` instruction was glitched. By glitching the load of the value into `r0`, a wrong value would be stored in the destination and later compared, allowing the secret string to be copied.

Capturing other secrets

Once the first glitch has been done successfully, it is trivial to reproduce it for the second string. This means that a double glitch can be done, which will force the program to reveal the next string in the memory. By trying a range of combinations, just like the previous glitch, it is possible to find the small window where the null byte is being processed and glitch on that moment. Both glitches are shown in the figure 5.10 and correspond to the monitor log 5.14.

Note that, on the log 5.14, right after the result obtained in the previous glitch there is a third string (`31 31 31 31 31 31`, `"111111"`), which corresponds to the `secret1` variable. As separator, it is still the byte `0x10`, which shows clearly that this second glitch is very stable. Finally, to capture another string in the memory, it would be necessary only to introduce a third glitch. This is indeed possible, as shown in the timing diagram of the figure 5.11 and log 5.15.

In this last experiment, the third string (`30 30 30 30 30 30`, `"000000"` corresponds to the `secret0` variable in the memory, which shows that multiple glitches are possible. The number of maximum glitches allowed is virtually infinite, as long as it does not damage other instructions that might corrupt the original program's flow.

It is important to note that after a while unstable results might be output on the target's UART. For example, on log 5.15, multiple strings of different lengths were printed, oscillating between 25 and 27 bytes long. This indicates that the null byte is not being stored on the destination buffer, which is a glitch on the `ST` (*store*) instruction. Also, by analysing the strings, it is possible to see a different byte on the second and third result, a `0x50` (ASCII character '`P`'), which indicates that the byte is still corrupted, but in a different way. However, since the goal is to obtain multiple strings around `foobar`, the byte that replaces the null byte of each string, or the fact that it might not even be copied, is not relevant, as long as the other strings are intact.

Another important detail is the the glitch delay between each attack. The first glitch requires a delay of 94 clock cycles, while the second and third of 80 clock cycles. What happened is that the first glitch takes a little more time since it has to consider the function call and its prelude, while the others do not need, since they never left the `strcpy` loop.

```
1 + Running test 189 of 1502
2 ? Got something from target, len = 13 [66 6F 6F 62 61 72
3   10 32 32 32 32 32 32] [foobar 222222]
4 + Accuracy: 100.0%

6 + Running test 190 of 1502
7 ? Got something from target, len = 13 [66 6F 6F 62 61 72
8   10 32 32 32 32 32 32] [foobar 222222]
9 + Accuracy: 100.0%
```

Code 5.13: Output from monitor script showing the first secret string captured. Note that the byte replacing the null byte is `0x10`. This means that a wrong value was copied to the destination string, indicating that either the load or the store instructions were glitched.

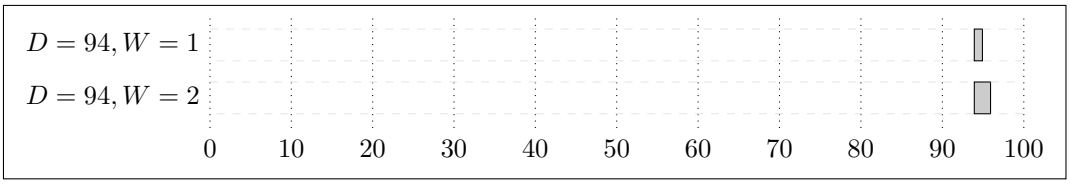

Figure 5.9: Timing diagram of the first glitch on the strcpy function. Note that there are not many possibilities of glitching the code - only on one specific moment it is possible to attack the target. This moment represents the end of the string, which is the goal of the attack.

```
+ Running test 1777 of 4444
? Got something from target, len = 20 [66 6F 6F 62 61 72
  10 32 32 32 32 32 32 10 31 31 31 31 31 31] [foobar 222
  222 111111]
+ Accuracy: 100.0%

+ Running test 1778 of 4444
? Got something from target, len = 20 [66 6F 6F 62 61 72
  10 32 32 32 32 32 32 10 31 31 31 31 31 31] [foobar 222
  222 111111]
+ Accuracy: 100.0%
```

Code 5.14: Output from monitor script showing two secret strings captured. Note that all strings are separated by a constant value (`0x10`), which can be interpreted as a very precise and reliable glitch.

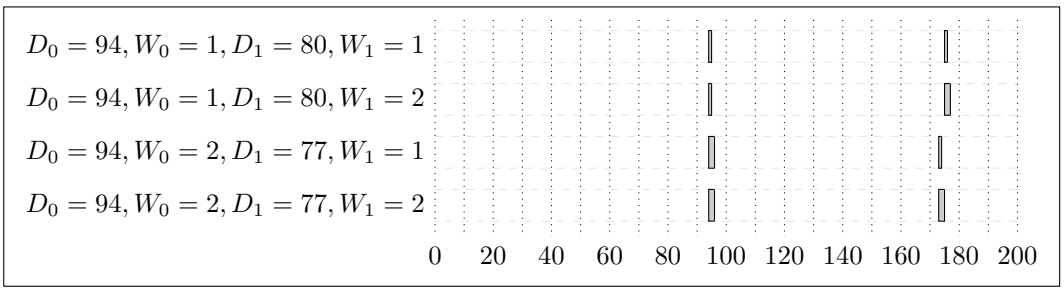

Figure 5.10: Timing diagram of the two glitches on the strcpy function. Note that the first glitch remains at the same position (since it was targeted for hitting the end of the first string), while the second glitch oscillates a few clock cycles. This allows it to hit different instructions on the execution, both causing the same effect.

```
1  + Running test 81 of 101
2
3  ? Got something from target, len = 25 [66 6F 6F 62 61 72
4    50 32 32 32 32 32 32 31 31 31 31 31 31 30 30 30 30 30
5    30] [foobar 2222221111111000000]
6
7  ? Got something from target, len = 27 [66 6F 6F 62 61 72
8    50 32 32 32 32 32 32 10 31 31 31 31 31 31 10 30 30 30
9    30 30 30] [foobarP222222 111111 000000]
10
11 ? Got something from target, len = 25 [66 6F 6F 62 61 72
12   50 32 32 32 32 32 32 31 31 31 31 31 31 30 30 30 30 30
13   30] [foobarP2222221111111000000]
14
15 ? Got something from target, len = 26 [66 6F 6F 62 61 72
16   32 32 32 32 32 32 10 31 31 31 31 31 31 10 30 30 30 30
17   30 30] [foobar222222 111111 000000]
18
19 + Accuracy: 100.0%
```

Code 5.15: Output from monitor script showing three secret strings captured. Note that the separator oscillates this time, while on the previous experiments it was a constant value (`0x10`).

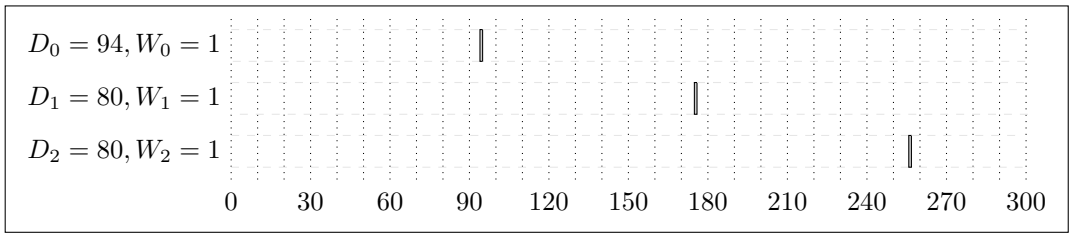

Figure 5.11: Timing diagram of the three glitches on the strcpy function. In this graph each line represent a step of the glitcher instead of a whole glitch combination by itself. Note that only a subset of each previous glitch can be used in this attack. The reason is because it has to be very precise, and any damage to the instructions nearby would cause the third glitch to fail.

6 Discussion

As shown in this work, clock glitching can corrupt instructions on AVR microcontrollers in a predictable and repeatable fashion. The design of the implemented glitcher in this work was outlined in Section 4. Experiments not only with unconditional and conditional loops were done, but also with compiled C showing that the code does not necessarily have to be designed specifically for being attacked. As shown in Section 5, even though the glitcher has internal delays, attacks can be done.

By controlling the clock, the CPU can be directly attacked, allowing an attacker to compromise the entire system, since the CPU requires its clock to work (i.e. if the clock is stopped, the CPU is halted). It is possible then to glitch in specific moments, targeting specific instructions on the executing code, as it was done in this work. Considering that the code running on the microcontroller is previously known by the attacker, finding an attack vector can be easily done. By adjusting the glitcher to hit the necessary instructions, an attack could change the original execution flow of the program, allowing the attacker to skip or repeat loops, calls or simply corrupt instructions in a way that wrong values are loaded or calculated. As shown in this work, functions like the `strcpy` are vulnerable to such attacks, allowing data to be extracted from the memory. Such techniques have been previously proposed by other as well as an attack vector for *smart cards* [KK99, AK96], where by glitching the loop it is possible to dump the whole memory. When working with an unknown environment, where the attacker does not have previous access to the executing code, glitching the precise instruction can be considered more difficult. However, in this work most of the attacks were done by brute-force - a process that can take from hours to days of execution. Although slower, brute-force attacks can be used when glitching the clock signal to find the exact parameters that will glitch interesting instructions.

An attacker could also use heuristic methods to find out such parameters, accelerating the search [CPB+13]. To determine if a glitch is interesting or not in a case of dumping data after the string, the exposed information can be analysed and checked regarding its contents if machine code, internal keys or any other relevant information was extracted. Note that, when binary information is dumped (like machine code), functions like `strcpy` might get stuck on the null byte, since it represents the end of an string. Although it is possible to circumvent this issue by continuously glitching, this is not always trivial, since, as shown in Section 5, different values can be loaded on each glitch. This would require an extra analysis to find out if that byte was actually a null byte or another extract one from the memory.

It is important to note that, since the target in this work was an AVR microcontroller with a RISC CPU, the relatively simple pipeline used by the AVR architecture must be considered as well. By fetching instructions while executing others, a glitch on one instruction could corrupt the next one, since both execute on the same the clock cycle.

The typical approach to address this issue is to insert a `NOP` between each instruction from the original code, as done with the `strcpy` function in this work. In case the next instruction gets corrupted due to the clock glitch, no original code instruction is damaged. However, this solution is not always available, since the code is not always modifiable (or even known). Even though the pipeline was a known issue from the beginning, during this work it was not consired on the initial experiments, only on the last one. By analysing the instructions available to the AVR architecture, it is clear that some instructions take more than one clock cycle [Atm98], even though this is a RISC architecture [Atm12]. This is due some instructions either require bigger operands, such as the `JMP` instruction (where the operand does not fit inside the whole instruction), or require a new instruction to be fetch, such as branch instructions (where the first instruction after the branch has completed has to be fetch before being executed). This is also interesting because, by having multiple fetches, it is possible to glitch only a part of the instruction, corrupting only part of it (e.g. its operands). This would make the instruction behave erractly, which could lead into a valid scenario where an attack is possible (such as jumping to the wrong address in the memory, for example). Finally, note that, since a clock glitch forces the target to run outside its normal operation range, the behavior is unknown and unpredictable. Therefore, the chance that, by tweaking the glitch (such as increasing the frequency or changing the wave parameters), the next instruction does not get corrupted is not completely excluded, even though it did not happen during this work.

This work can be expanded for any RISC CPU that is vulnerable to a clock glitching attack, and in such scenarios, the pipeline would most likely still be present and interfere with the glitches. However, if the target was a CISC (*Complex instruction set computing*) CPU, it would be, in theory, possible to hit parts of the execution of the instruction, since CISC CPU instructions can take more than one clock cycle. Note that such instructions perform multiple single-clock low-level operations, thus allowing an attacker to glitch a specific operation within the same instruction. In instructions with both data read and write into the memory, for example, it would be possible to glitch only the read, forcing the CPU to not read the correct data from the memory, or the write, forcing the CPU to not write the correct data to the memory. However, modern CISC architectures are based on RISC-like micro-operations. Hence, many many of the observations for RISC architectures will apply to micro-operations as well.

In this work the target was running without its internal PLL (i.e. the clock received from the external source is the clock used internally). However, if activated, the PLL could interfere on a single clock glitch, since the glitch would pass through it first. Even though longer glitches (i.e. multiple clock cycles as width) might be able to pass through it, but there is no guarantee about its repeatability and accuracy. Not less important, the glitcher could be used to glitch other clock sources without providing it directly to the target. For example, by grounding a quartz crystal oscillator, it would be possible to force it to a logic zero. This allows the attacker to induce values on the clock signal, generating clock cycles that are at a higher frequency than the oscillator produces. The concept of such attack is the same as the one demonstrated in this work, except that there is the chance of a clock drift between different clock domains, since not all of them are generated by the DDK's PLL. Finally, the same principle can be applied by taking

an external clock and passing it through the glitcher instead of generating it internally, since a glitch can be generated by simply feeding a logic zero instead of the original signal as output for short amounts of time.

Future work

One of the biggest limitations of the glitcher developed in this work is regarding its delay after the target has booted. Due to the state switching in the main module (which takes one clock cycle per switch), there is a delay between the moment when the target finish its boot and the moment when the glitcher starts the glitching process. Such issue happens between the states *wait* and *delay*, since it has to first read the next glitch setting from the FIFO in the state *read*. One solution would be to create an extra module responsible for handling the FIFO reading. This module would have its own state machine and would take care of reading the FIFO and proving the next glitch settings straight to the main module. By removing the reading process from the main module, the delay would disappear, since it would be possible to go straight to the states *delay* and *glitch*. Not less important, the delay that happens between glitches (since it needs to go through *read* again) would also be removed, allowing the glitcher to execute a sequence of glitches without any delays in between.

As previously explained in Section 2, a glitch would consist of having more than one rising edge during the period of one clock cycle. Since the glitching clock (99 MHz) is three times faster than the normal clock (33 MHz), by simply switching clocks the output has three rising edges for glitching. This bevahior, however, is undesired, since it could behave as a double glitch. By having three rising edges instead of two, a glitch could force, for example, the CPU to skip the instructions in two clock cycles instead of only one. This reduces the preciseness of the glitcher, making it harder to glitch fast instructions (such as `AND`, which takes only one clock cycle). Such problem is demonstrated on the figure 6.1. The higlighted area in the figure indicates the moment where the clock is being glitched: it starts on one rising edge and it ends on the next one, having three rising edges between them (including the first and excluding the last). The ideal scenario is also shown as the last signal in the figure, where only two rising edges are present during the period of one clock cycle, being the last rising edge in the highlighted area already from the next clock cycle. One solution for such issue is to use 66 MHz instead of 99 MHz as glitching clock. However, during the development of this work, 66 MHz proved to be unstable and unreliable for glitching, since the microcontroller was able to still compute some instructions properly with at this frequency. Therefore, the best solution would be to create another module, responsible only for generating the ideal signal as shown on the figure. The output of this module would then be used as input on the core as glitching clock, giving the desired behavior for the glitcher.

Another limitation of the glitcher is the interface between the DDK and the external monitoring script. This interfaces uses the DDK's internal UART, which is provided directly by the USB port. However, the current implementation has performance issues, which makes the writing and reading speeds of the script slower. The current implementation forces the script to have a delay of about 80ms between each character when

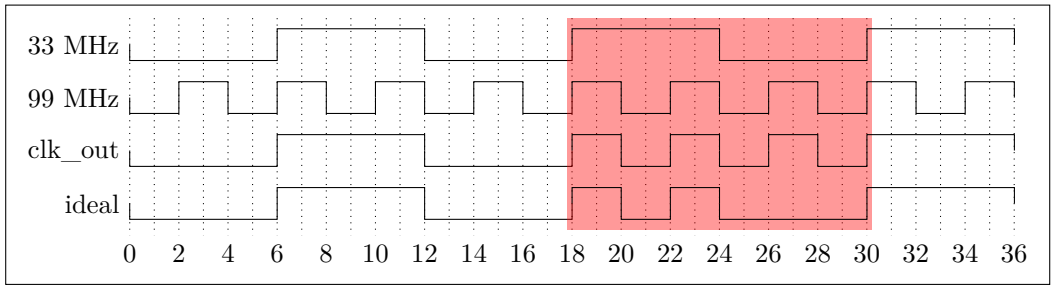

Figure 6.1: Timing diagram depicting the limitations of the current glitcher implementation. The highlighted area indicates the rising edges of the first clock cycle (where the glitcher was enabled), which is counted as part of the glitch, and the second one (where the glitcher was disabled), which is not considered since it is already another normal clock cycle at the same instant. Three fast rising edges (i.e. used for glitching) are present as output of the current glitcher, while ideally only two should be output.

writing to the serial line. By fixing the DDK's UART implementation, this delay could be reduced or removed, allowing the script to send commands faster to the DDK. This would not improve the speed of one isolated test, since the glitcher is not modified, but it would increase the overall speed when running brute-force attacks to try and find a range of settings for glitching. Another solution would be removing completely the external script. This would require an implementation of a USART port on the DDK only for reading the target's output, which then would have to be properly stored. However, since the goal is not to read the full target's output, but to receive a feedback information, reading and storing only one byte would be sufficient. This byte could have multiple values, each one representing what happened (glitch unsuccessful, glitch successful or unknown error), or even data that can later be used for analysis. Both solutions would increase the overall speed of running multiple attacks, being the second one considerably faster than the first, since it would not require any serial communication with the external world at all (except for logging and initial setup, such as giving the full range that should be tested).

One simple improvement for the glitcher would be using proper cabling when connecting the DDK to the target. Depending on which type of wire is being use, it is possible that noise, as well as the lack of proper termination of the wires, can result in interference and ringing on the glitch signal. Therefore, by improving the cabling it is possible to increase the quality of the clock signal that arrives on the target, since the clock will arrive with less distortion. Finally, it is important to note that the whole glitcher can be reduced to only one channel on the DDK by removing the debug pins. By removing unused and unnecessary pins, the glitcher would need only the `clk_out` (clock output), `rst_o` (reset output) and `en_i` (target ready) pins to work. The other three remaining pins can then be used for feedback from the target, either by directly changing the values on them, or by using a USART. Such improvement is interesting since, by freeing the other channels, they can also be used for glitching. This would allow a parallel glitching framework to be developed, which can be later used to glitch

up to eight targets at the same time, being each target connected to one of separate DDK channel. Such framework would reduce the required time for brute-force attacks, allowing bigger ranges and more complex target algorithms (i.e. slower algorithms that take more clock cycles to compute) to be experimented with.

7 Conclusion

In this work, an analysis of embedded microcontrollers against clock glitching attacks was performed. For this, the AVR RISC architecture of an XMEGA microcontroller was attacked by a glitcher developed in the *Die Datenkrake* platform. The microcontroller, located on an Atmel evaluation board, was attacked by providing an external clock to it. The clock signal induced faults in the system, which were then analysed and explained in this work.

The implemented glitcher is completely modular and fully expandable, allowing future work to modify it for specific scenarios or even improve it. The glitcher design itself utilizes software hardware co-design, i.e. the glitcher was realized using software as well as a dedicated hardware design. The hardware, running on the DDK's FPGA, is responsible for generating the required synchronized clock signals and handling all hardware related operations (e.g. synchronization triggers). The software, running on the DDK's ARM CPU, is responsible for interfacing such hardware modules for configuration, fine adjustments and feedback information retrieval.

Multiple experiments were performed against the target. The initial target codes used in such attacks consisted of handcrafted unconditional loops, as a proof of concept to demonstrate that glitching on this architecture is possible. However, since in most scenarios conditional loops are also present, instructions for such branches were also considered with. Both sets of instructions showed that, when glitched, it is possible for an attacker to force the CPU to skip those instructions, which resulted in exiting the loop in those programs. Finally, to demonstrate that the code does not need to be specially designed for being attacked, the `strcpy` function from the C standard library was targeted, allowing the attacker to bypass the end of strings in the memory and dump more data than what was originally designed in the program.

As previously noted, the environment used for the experiments in this work is completely modular. Different targets of different architectures can be attacked, requiring only the new device to be setup accordingly. By changing either the glitcher or the external monitor script, it is possible to introduce better and faster techniques for finding glitching ranges (i.e. ranges where the clock glitch must happen to induce an exploitable fault) without requiring extensive modifications. Not less important, the whole glitcher can be improved regarding its preciseness and accuracy, allowing better glitches in more complex scenarios or programs to be performed, see Section 6.

This work successfully demonstrated that the AVR platform is vulnerable to clock glitching attacks, and that the executing program can be exploited without detection. By skipping instructions, it is not only possible to bypass validations and verifications, but also to recover data stored on the device. Despite the issues regarding the glitcher's preciseness, it was still possible perform accurate and repeatable attacks against the

target. Finally, the modular design of the glitcher makes it possible do adapt the design to attack other targets as well.

Bibliography

[ABF+03] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault attacks on rsa with crt: Concrete results and practical countermeasures. In BurtonS. Kaliski, çetinK. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 260–275. Springer Berlin Heidelberg, 2003. 2

[AK96] Ross Anderson and Markus Kuhn. Tamper resistance – a cautionary note. In *IN PROCEEDINGS OF THE SECOND USENIX WORKSHOP ON ELECTRONIC COMMERCE*, pages 1–11, 1996. 1, 45

[And01] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2001. 1

[Atm98] Atmel. Avr assembler user guide, 1998. 21, 25, 29, 32, 39, 46

[Atm10] Atmel. 8-bit avr instruction set, 2010. 21, 25

[Atm11] Atmel. Atmel avr1924: Xmega-a1 xplained hardware user's guide, 2011. 5

[Atm12] Atmel. Xmega a manual, 2012. 5, 46

[BECN+06] H. Bar-El, H. Choukri, D. Naccache, Michael Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, Feb 2006. 2

[BGV11] J. Balasch, B. Gierlichs, and I. Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pages 105–114, Sept 2011. 1, 2

[CPB+13] R. Carpi, S. Picek, L. Batina, F. Menarini, D. Jakobovic, and M. Golub. Glitch it if you can: parameter search strategies for successful fault injection. 2013. 2, 45

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. 4

[KK99] Oliver Kömmerling and Markus G. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, WOST'99, pages 2–2, Berkeley, CA, USA, 1999. USENIX Association. 1, 2, 45

[NS13] Dmitry Nedospasov and Thorsten Schroder. Introducing die datenkrake: Programmable logic for hardware security analysis. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2013. USENIX. 5, 6

[Ope10] OpenCores. Wishbone b4 - wishbone system-on-chip (soc) interconnection architecture for portable ip cores, 2010. 7