Duplication-Based Fault Tolerance for RISC-V Embedded Software

Volodymyr Bezsmertnyi[1,2], Jean-Michel Cioranesco[1], and Thomas Eisenbarth[2]

[1] NXP Semiconductors, Beiersdorfstraße 12, 22529 Hamburg, Germany
{volodymyr.bezsmertnyi,jean-michel.cioranesco}@nxp.com
[2] Institute for IT-Security, Ratzeburger Allee 160, 23562 Lübeck, Germany
thomas.eisenbarth@uni-luebeck.de

Disclaimer

This preprint has not undergone peer review (when applicable) or any post-submission improvements or corrections. The Version of Record of this contribution is published in "Computer Security - ESORICS 2024: 29th European Symposium on Research in Computer Security, Bydgoszcz, Poland, September 16-20, 2024, Proceedings, Part IV", and is available online at https://doi.org/10.1007/978-3-031-70903-6_5

Abstract. Embedded devices play critical roles in security and safety, demanding robust protection against fault injection attacks. Among the myriad of fault effects, the instruction skip fault model stands out due to its recurrent manifestation in silicon devices. Furthermore, the continually evolving landscape of hardware attacks facilitates increasingly sophisticated exploits by achieving multiple instruction skips. In this work, we propose an extension of the RISC-V debug specification which enables efficient fault injection testing of the firmware executed on an FPGA-emulated core under a commonly observed instruction skip fault model. We use insights from a fault injection campaign to harden and protect potentially exploitable instructions and propose an assembly level duplication-based approach for software fault tolerance against instruction skip applied to RISC-V architecture. Additionally, we provide a custom debugger implementation which accelerates fault injection campaign by factor of ten. By combining fault injection testing and a generic instruction duplication technique, our methodology can increase fault tolerance of the reference software while having minimal performance loss and code size overhead.

Keywords: risc-v · fault tolerance · fault injection emulation · instruction skip · fpga

1 Introduction

Embedded devices with high security requirements are increasing in areas such as automobiles, medical equipment, financial systems, and critical infrastructure.

One characteristic of such systems is that part of the code being executed, in particular the boot code, is not planned to be updated in the field. Thus, a way of securing an embedded device is to identify vulnerabilities early in the product development life-cycle by checking code against logical as well as physical attacks, since an attacker may have the device in hands. Identifying and addressing security vulnerabilities early on in the design phase can also help reduce costs and time required to patch these issues [15]. Pre-silicon security evaluation can include reviewing architecture and code, testing for compliance with security standards, and conducting penetration testing. Therefore, software/hardware co-testing should be thoroughly carried out before being deployed in its final application.

One attack vector on embedded code is fault injection, where a potential attacker can manipulate the hardware using different means like voltage glitches, laser beaming, electromagnetic pulses, and others [6]. This can include injecting faults into the system's memory, processor, or other components to see how the system responds and if there are any security vulnerabilities that can be exploited as a result. In particular, code and data flow can be perturbed, with the common manifestation being an instruction skipped or not executed. *Instruction skips* make it possible to bypass security measures or execute malicious code on the target. As fault injection techniques are rapidly advancing, leading to the emergence of more complex and sophisticated attacks such as multiple instruction skip attacks [35,30], it is crucial to consider these new types of attack in research and development efforts. In the face of single or multiple instruction skip attacks, the imperative for software fault tolerance becomes apparent. Various software fault tolerance techniques like instruction duplication and control flow integrity checking [28] have to be implemented to fight such attacks. As a consequence, integrating software fault tolerance mechanisms within pre-silicon testing protocols becomes necessary to ensure robustness against these specific threats.

With the advent of the RISC-V architecture, its modular and open-source nature allows customization of hardware components, providing an ideal platform for specialized functionalities. Notably, the openness of the RISC-V architecture enables the integration of hardware acceleration support tailored for pre-silicon fault injection testing. Our objective in this study is to create efficient tools designed to facilitate fault injection processes, enabling the hardening of code against instruction skip faults specific to the RISC-V architecture. To achieve this, we propose an extension for the RISC-V debug specification, featuring the automated skipping of an arbitrary number of instructions at a target location. This advancement accelerates fault injection by reducing the communication load between the host debugger and the debug module on the chip. Furthermore, leveraging insights gained from the fault injection campaign, we present a technique to harden vulnerable instructions by implementing an assembly-level duplication-based countermeasure [23], adapted specifically for the nuances of the RISC-V instruction set. Additionally, we implemented the proposed extension within the debug module of the open-source CORE-V-MCU

System-on-Chip, allowing for a comprehensive evaluation. To highlight the advantage of open-source architecture, we provided own debugger implementation with a custom debugging protocol based on QSPI interface instead of conventional JTAG. This modification allowed to speed up fault injection testing by factor of ten. Finally, the effectiveness of our approach was evaluated through rigorous testing against reference software, showcasing the efficiency of our code-hardening technique. The combination of our technique with fault injection testing demonstrated the absence of silent data corruption errors, all while imposing minimal code size overhead compared to duplicating every relevant instruction. The implementation of our proposed method will be released as open source and available on GitHub[3].

2 Related work

This section highlights the research conducted on fault injection, in particular on physical attacks that exploit hardware vulnerabilities. Additionally, instruction skip fault model is discussed as a fault effect commonly observed in silicon devices. Furthermore, the section delves into software-implemented fault tolerance and control flow integrity techniques as a mean to harden a system against faults. Finally, the section reviews studies which focus on emulating fault injection especially with a help of a debugger.

2.1 Hardware Fault Injection Attacks

Hardware-based fault injection involves introducing errors into the system by physically altering the hardware of the system. A comprehensive survey of distinct fault injection approaches is presented in [38], [19], [6]. In [8], multiple fault injection attacks on microcontroller-based cryptographic algorithm implementations are demonstrated. In [11], the practicality of fault injections is examined through empirical research. A systematic examination of fault injections in Internet-of-Things devices is conducted in [17].

2.2 Instruction Skip Fault Model

The instruction skip fault model is a commonly studied fault model in the field of computer architecture and digital circuit design. This fault model occurs when one or more consecutive instructions in a program are not executed due to a fault in the hardware or software of the system. Here, we list some examples of works that achieved either single or multiple instruction skips through fault injection.

[3] `https://github.com/orshinAtNXP/`

Single Instruction Skip A single instruction skip is a fault effect frequently seen in fault injection testing of many microcontrollers. A recent work [30] that was presented at Black Hat 2022 utilizes Voltage Fault Injection (VFI) for skipping a single instruction at different points in time in order to defeat ARM TrustZone. The work from [34] showed an exploit where VFI was used to escalate privilege in Linux from user space. Balasch et al. [5] investigated the effects of clock glitches on an 8-bit microcontroller and provided a possible explanation for the observed instruction skip. Colombier et al. in [12] proposed a technique that uses multiple lasers in order to induce multiple single-bit faults in an ARM Cortex-M3. Menu et al. [22] investigated electromagnetic (EM) fault injections and questioned an EM fault model since the authors could skip multiple consecutive instructions with their method. Proy et al. [27] studied EM pulse effects at the ISA (instruction set architecture) level.

Multiple Instruction Skip Less common but still dangerous fault effect is the multiple instruction skip which can be achieved either due to multiple glitches in a row or a single glitch impacting the critical path in the cores instruction pipeline. Riviére et al. [29] managed to skip up to four consecutive instructions by electromagnetically faulting the instruction cache of an ARM Cortex-M CPU. Blömer et al. [10] utilized multiple clock fault injections for attacking two consecutive instructions. Dutertre et al. [13] were able to skip groups of instructions by laser illumination on an 8-bit non-secure ATmega328P microcontroller. Yuce et al. [36] were able to skip multiple instructions stored in the target's pipeline with clock glitches in a 32-bit LEON3 processor on a Xilinx FPGA. The authors of [14] reported EM-induced skips of up to six consecutive instructions with low repeatability on a RISC-V FPGA implementation.

2.3 Software-Implemented Fault Tolerance

Software-Implemented Fault Tolerance (SWIFT) is an approach to improving the reliability of software systems by incorporating fault-tolerant techniques like error-detection and redundancy mechanisms directly into the software code with a goal to harden systems against fault models, particularly instruction skip faults. Moro et al. [23] provided a formal proof showing the efficiency of redundancy-based countermeasures against a single instruction skip. Their countermeasure consists of replacing a non-idempotent instruction with an idempotent one and duplicating it. Replacement schemes were provided for the ARM instruction set, followed by a formal proof of countermeasure efficiency. We adopt this approach for the RISC-V instructions in our code hardening tool. In [24] Moro et al. performed evaluation of two countermeasures by launching physical fault attacks and assessing the impact. Barenghi et al. [7] proposed software countermeasures for cryptographic algorithms including intrusion and fault detection. Barry et al. [9] implemented a LLVM compiler extension which protects against instruction skip attacks. Sharif et al. [33] developed a compiler framework targeting RISC-V processors which hardens code using various fault tolerance

techniques. Schirmeier et al. [32] provided a fault injection framework for detecting vulnerable code by emulating faults with a debugger. Kiaei et al. [21] perform assembly rewriting and lift an x86 binary to an intermediate representation in order to harden vulnerable instructions which they discover by emulating fault injections.

2.4 Emulated Fault Injection

In fault injection emulation, the FPGA is programmed to replicate faults that might occur in the actual hardware, such as electrical or logical faults, to assess how a system or software responds to these faults. A debugger can be used to change the software behavior simulating possible fault effects on the software level. Here we highlight works, where a debugger is used for injection of the faults as it is done in our work. Portela-Garcia et al. [26] utilized the On-Chip Debugger (OCD) to inject faults into a microcontroller that supported JTAG debugging capabilities. Instead of controlling the fault injection campaign from the host, they moved the controlling logic to a separate Systems on Programmable Chip (SoPC) and the host only configures the fault injection campaign via communication with SoPC. Mosdorf et al. [25] injected faults using the GDB debugger and a J-Link debugger via the JTAG interface of an ARM device. Schirmeier et al. [32] provided a fault injection framework for assessing the fault tolerance of a system by emulating faults with a debugger on multiple emulators. Zhang et al. [37] utilized a debugger for the fault injection testing of a real-time operating system. Ahmad et al. [4] developed a fault injection framework based on a debugger for x86 CPUs and used GDB to interrupt the program's execution to inject faults at runtime.

3 Protection by Fault Injection Emulation

In this section, we introduce a methodology for protecting a firmware against instruction skip attacks and provide an overview of the separate steps of the flow for code hardening. We call our approach *Skip Protection by Fault Injection Emulation* (SPFIE) and incorporate it into a framework for fault injection testing on an emulated RISC-V core. The framework can be used to embed continuous security testing into the development process of the software, since it provides an efficient fault injection testing and hardens code with minimal user interaction. Our framework is also capable of skipping an arbitrary number of instructions in the given software, which can be used for identifying vulnerable instructions or code snippets. A user provides a compiled binary that will be executed on the target emulated core and configures the framework to test a list of critical functions. The framework then performs fault injection (FI) testing by executing the binary on the core emulated with an FPGA and produces a list of vulnerable instructions that require additional protection. Our framework also requires access to the source code and build scripts of the software in order to be embedded into the build flow. Having the sources, our code hardening tool

finds and replaces vulnerable instructions with a protected version of the original instruction. Finally, another iteration of the fault injection testing is performed on the hardened code in order to verify the absence of the previously detected vulnerabilities.

The framework uses fault injection emulation to identify vulnerable instruction addresses and uses the results to patch the assembler language files. To ensure code security on each commit or major source code modification automatically, firmware developers can incorporate this framework into the build flow. The workflow is depicted in the Figure 1. Next, we elaborate on every step of the process:

1. *Generate and Build*: Since our framework hardens the code at the assembly level, the assembler language files need to be generated from the C sources. From the assembler files, we build the initial binary for the fault injection campaign emulated on a FPGA.
2. *FI testing*: In this step, the user performs FI testing by skipping the configured amount of instructions for the given functions to test. The FI campaign results in a list of faulty addresses that, if skipped via a fault injection, can lead to exploitable behavior.
3. *Code Hardening*: Given a list of faulty addresses, our code hardening tool performs a transformation and duplication of the faulty instructions. The patched instructions are then written to the assembler files. Detailed transformations of RISC-V instructions are described in Section 6.
4. *Build and Verify*: In the last step, the final binary is built from the patched assembler files, and another iteration of the FI testing on the final binary can be performed to confirm the absence of vulnerabilities and original functionality of the binary.

By integrating the SPFIE methodology in the build flow of the firmware, the developers can continuously and automatically ensure the security of the code against instruction skip attacks, and a secure version of the binary can be released. By viewing the logs of the framework, the developers can get a direct feedback on the vulnerable instructions. This information can be analyzed in order to gain an understanding of how skipped instructions can impact the code execution. An advantage of this approach is scalability, since increasing the number of available emulators reduces the testing time linearly. The developers can set up additional emulators and uniformly distribute the test addresses across the emulators. Afterwards, the faulty addresses for each emulator instance can be collected and put together for the code hardening. A disadvantage of this approach is that it requires human guidance in form of provided names of the critical functions which are supposed to be tested and hardened.

4 Debugger-Driven FI Testing

This section delves into the specifics of debugger-driven fault injection testing framework, which is employed to skip instruction on an FPGA-emulated

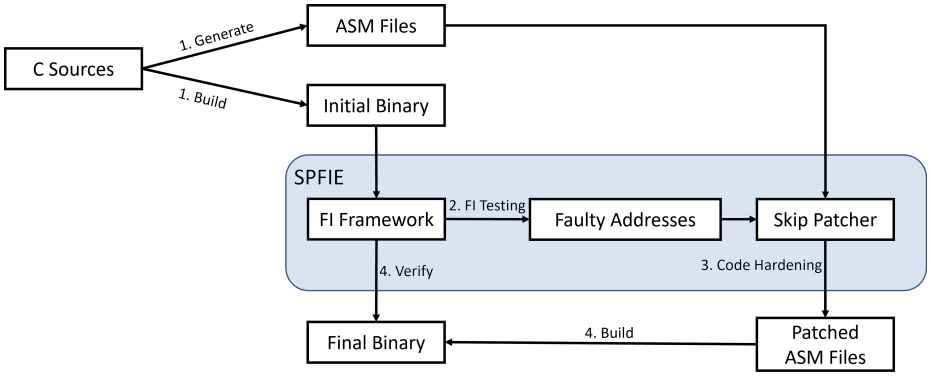

Fig. 1: Schematic overview of the code hardening flow.

system-on-chip (SoC). Here, we describe how the FI campaign is performed and accelerated by a custom debug specification extension.

The reason for opting for an emulation solution is the speed advantage it offers, whereby the code is executed directly on an emulated target device, allowing for full available execution speed. This facilitates the execution of binaries and the injection of faults much faster than simulation-based solutions, enabling us to conduct fault injection testing on large numbers of instructions. For this purpose, an emulation environment needs to be configured to run tests. This includes setting up an FPGA with a synthesized design of the target SoC and establishing the communication to the debug module (DM). With an emulator set up, the user can start the fault injection campaign.

The fault injection testing is controlled by a Fault Injection Controller (FIC) which manages the fault injection campaign by leveraging the debugger and the emulation setup in order to find vulnerable places in the assembly code. The basic idea is to inject faults upon hitting a breakpoint at a target instruction address. The debugger is used to configure special custom registers in the DM(discussed in Section 5) to simulate an instruction skip. By detecting an address, where a fault is supposed to be injected, the DM alters the program counter according to the configuration. Before the FIC starts FI testing, the user evaluates the attackers ability and determines, how many instructions an attacker is potentially able to skip via fault injection into the particular SoC. This mainly depends on the targeted architecture, CPU pipeline stages and the memory subsystem from where instructions are fetched. It is a crucial information for the fault injection campaign and the subsequent code hardening, since our instruction duplication technique introduces fault tolerance to a degree which depends on attackers ability to skip a certain amount of instructions. The user also identifies and provides a list of security critical functions in the binary that have to be tested. For each instruction address in the function-under-test (FUT), the FIC does the following steps:

1. Reset the core to prevent interaction with the core state from the previous executions.
2. Load the executable into the memory.
3. Configure special CSRs in the DM for the automatic instruction skip.
4. Set breakpoint at exception handler.
5. Set breakpoint at last address of main function.
6. Resume the binary execution.

There are 3 possible outcomes of a single test run: the execution can time out, hit the breakpoint at the exception handler or successfully execute the program and hit the breakpoint at the end of the main function. The timed out runs might need further investigation by the user. Execution of the exception handler is an indication of detected fault injection, since the program didn't complete its execution. If the program was executed successfully, that means the fault injection was not detected and silent data corruption might have happened. So, at the end of the fault injection campaign, the FIC invokes the code hardening routine and provides to it the list with faulty addresses for analysis.

5 Debug Specification Extension

To accelerate the fault injection campaign by minimizing host-to-target communication, we propose a modification to the on-chip debug module. This enhancement allows for more efficient instruction skipping. The openness of the RISC-V ecosystem grants access to the debug specification, offering room for custom debug features. Controlling the debug module involves manipulating its internal Control and Status Registers (CSRs), which include 16 reserved registers designated for custom functionalities. By detailing our method at the debug specification level, we ensure its independence from specific debug module implementations, ensuring a level of portability across diverse RISC-V system designs. In the following, we outline the specifications of three custom registers, explaining their function in skipping an arbitrary number of instructions at runtime. This method optimizes the FI process, contributing to enhanced efficiency while maintaining adaptability across varying system architectures.

The custom debug registers designed for instruction skipping are as follows:

- `fi_address`: This register stores the address of the target instruction where a fault is to be injected during a single test. Upon setting the `fi_address`, the DM sets a hardware breakpoint at the address in the `fi_address` register to be able to skip the target instruction before its is executed.
- `hit_count`: Within this register resides a numerical value indicating the number of times the target instruction must be executed before the fault injection is triggered. The DM should decrease the `hit_count` value by 1 every time the `fi_address` is encountered. Finally, if `hit_count` value is 0, the fault is injected and the hardware breakpoint at `fi_address` is removed.
- `pc_delta`: Contained in this register is a two's complement number that dictates the program counter's advancement when the address specified in

`fi_address` is encountered at least `hit_count` times. This value determines the shift in the program counter upon meeting the specified conditions.

As one can see, using this construction, we can also skip multiple consecutive instructions as well by setting the `pc_delta` register accordingly. It is also possible to simulate more advanced fault models such jump to an arbitrary address, which can be useful in some cases, like for testing unexpected control flow violations.

6 Code Hardening Tool

The Code Hardening Tool is invoked after the FI testing is completed. It gets the list of faulty addresses and the number of skipped instructions in the FI campaign, and its goal is to patch the faulty addresses in the assembler files by duplicating them. So, for each faulty address we need to find the corresponding group of assembly instructions in the sources and replace it with a duplicated sequence of *idempotent* instructions. An idempotent instruction is an instruction that can be executed multiple times without changing the result beyond the first execution. In other words, the effect of the instruction remains the same no matter how many times it is executed. Such instructions are useful for the fault-tolerant replacement sequences that we propose, similar to the countermeasure by Moro et al. [23]. If every instruction in such a sequence is duplicated more times than an attacker is able to skip, then every instruction in the sequence is executed at least once, and the execution of the duplicated idempotent instruction sequence does not lead to side effects that might change the result of the program's execution.

We define five instruction classes for the RISC-V IMC instruction set: idempotent, separable, pseudo-instructions, compressed, and special instructions.

Idempotent instructions can be duplicated without any transformations. These include store and branching instructions as well as load and arithmetic instructions where every source operand differs from the destination operand. The CHT can duplicate such instructions directly without replacing them.

Separable instructions are arithmetic operations where one of the source operands is simultaneously the destination operand. Such instructions cannot be duplicated right away and need to be replaced using an extra register. The extra register needs to be free, meaning it should not have been used in the calculations before. Some example transformations of arithmetic and load instructions are provided in the Table 1.

Pseudo-instructions in RISC-V are assembler directives that are not part of the official RISC-V instruction set but are provided by the assembler to make it easier for programmers to write code. Pseudo-instructions are translated by the assembler into one or more actual RISC-V instructions. When the CHT encounters such an instruction, it rewrites it using special, idempotent, and separable instructions. Afterwards, every instruction in the resulting sequence will be replaced by an idempotent one. The examples of some pseudo-instruction transformations are presented in Table 2.

Example instruction	Description	Transformation
`addi a0,a0,1`	Increments `a0` register by one.	`mv rx,a0` `addi a0,rx,1`
`lw a0,8(a0)`	Loads a 32-bit word from memory at the address `(a0 + 8)` and stores the result in register `a0`.	`lw rx, 8(a0)` `mv a0, rx`

Table 1: A table with some examples for replacement sequences for the separable RISC-V instructions. The register `rx` represents a free register used to temporarily store a value.

Example instruction	Description	Transformation

`la a0,0xdeadbeef`	Loads an immediate 32-bit value. Expands into:
	`lui a0,0xdeadc`
	`addi a0,a0,0xeef`
		`lui rx,0xdeadc`
		`addi a0,rx,0xeef`

`call fn`	Calls a subroutine by storing the address of the next instruction in `ra` register and jumping to the label `fn`. Expands into the special instruction `jal ra,fn`.	`lui rx, %hi(fn_ret)`
		`addi ra,rx,%lo(fn_ret)`
		`j fn`
		`fn_ret:`

Table 2: A table with some examples for replacement sequences for the RISC-V pseudo-instructions. The register `rx` represents a free register used to temporarily store a value.

Compressed instructions are a subset of the RISC-V instruction set that uses 16-bit instructions instead of the standard 32-bit instructions. The compressed instruction set uses the same instruction formats as the standard instruction set, but with shorter opcodes and fewer operands. The compressed instructions will be "decompressed" by the CHT. The decompression process involves looking up the underlying instruction and multiplying an immediate value by a factor depending on the instruction. If the decompressed instruction is a separable or a special instruction, it will be transformed into an idempotent instruction accordingly. Example transformations of some compressed instructions are presented in the Table 3.

Special. Three special instructions in the standard set, namely `jal`, `jalr`, and `auipc`, are generally not idempotent depending on operands. These instructions, commonly used for jumps and subroutine calls, require transformation sequences that always rely on label-based offsets within assembler files. This requirement arises because these instructions either use or alter the program counter, and introducing new instructions into the assembler files can affect their behavior.

Example instruction	Description	Transformation
`c.addi16sp 8`	Adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer, where the immediate is scaled to represent multiples of 16. Expands into a separable instruction `addi sp,sp,16*8`.	`mv rx,sp` `addi sp,rx,128`
`c.add sp,a0`	Adds the values in registers `sp` and `a0` and writes the result to register `sp`. Expands into a separable instruction `add sp,sp,a0`.	`mv rx,sp` `add sp,rx,a0`

Table 3: A table with some examples for replacement sequences for the compressed RISC-V instructions. The register `rx` represents a free register used to temporarily store a value.

The replacement sequences are detailed in Table 4. Transforming `jal` involves establishing a *return label* after the sequence's final instruction. This label stores the subsequent instruction's address after `jal` execution. Utilizing `la` to load the return label's address and `j` to jump to the target label ensures a safe jump with the correct destination register address.

Similarly, the `jalr` transformation entails loading the return label's address, storing the incremented source register value in a temporary register, and executing a jump to the temporary register's stored value. It is crucial that the return label points to the instruction *after the entire* duplicated sequence.

For `auipc`, requiring a label after the sequence, we initialize temporary registers with the label's address and immediate value using `lui`. Adding these temporary registers results in a PC-relative address stored in the destination register, ensuring the transformation relies solely on idempotent instructions.

In order to harden an instruction, the CHT expands the target instruction if it is a pseudo-instruction or decompresses it if it is a compressed instruction according to the instruction set specification. Each instruction in the resulting sequence will be then transformed and duplicated after the transformations. By duplicating each instruction in a sequence more times than the attacker can skip, we ensure that every instruction in the sequence will be executed at least once, and an attacker needs to be able to skip more instructions for a successful attack.

An example of the protection process is depicted in the Figure 2. We start by having a vulnerable group of two consecutive instructions: a compressed instruction `c.add a0,s2` and a pseudo-instruction `call fn`. After the first step, the compressed instructions expand into a separable instruction `add a0,a0,s2` and the pseudo-instruction expands into the special instruction `jal ra,fn`. Af-

Instruction	Description	Transformation

`jal rd,offset`	Puts the address of the next instruction into `rd` register and adds `offset` to the program counter.	`lui rx,%hi(ret_lbl)`
		`addi rd,rx,%lo(ret_lbl)`
		`j offset`
		`ret_lbl:`

`jalr rd,rs1,offset`	Puts the address of the next instruction into `rd` register, adds an offset to the value of the `rs1` register, and sets the program counter to the resulted value.	`lui rx,%hi(ret_lbl)`
		`addi rd,rx,%lo(ret_lbl)`
		`addi rx,rs1,offset`
		`jr rx`
		`ret_lbl:`

`auipc rd,offset`	Adds an immediate 20-bit value `offset` to the upper 20 bits of the current program counter, filling lower 12 bits with zeros, and stores the result in the `rd` register.	`lui rx2,%hi(pc_lbl)`
		`addi rx1,rx2,%lo(pc_lbl)`
		`lui rx2,offset`
		`pc_lbl:`
		`add rd,rx1,rx2`

Table 4: A table with replacement sequences for the special RISC-V instructions. The registers `rx`, `rx1` and `rx2` represent free registers used to temporarily store values, `imm` and `offset` represent operands with any immediate value based on a label within the code.

ter the applied transformation step, a return label is introduced, and the free temporary register `t0` is used in the transformation of the separable and the special instruction. The instructions in the transformed sequence will be duplicated three times because the original group size was 2. Finally, the original instructions in the assembler files will be replaced by the fault-tolerant version.

7 Implementation

As emulation platform we selected the open-source 32-bit microcontroller unit (MCU) *CORE-V-MCU* [31]. It features the open-source 4-stage, in-order 32-bit RISC-V core *CV32E40p* [18], operating at a core frequency of 20 MHz. We customized the debug module of the emulated design according to specification from Section 5 for a faster instruction skip, owing to our access to the source code of the emulated SoC. Highlighting the advantages of the RISC-V architecture's open-source nature, we developed a custom debugger. This debugger communicates Debug Module Interface (DMI) commands over a custom QSPI protocol. The emulation setup's schematic overview is depicted in Figure 3. Two debugging processes operate on the host: the conventional OpenOCD [2] and our custom debugger. OpenOCD communicates with the HS-2 Debugger [1] via USB, and HS-2 utilizes JTAG transport layer to interact with the SoC's debug module. Simultaneously, our custom debugger communicates with a Teensy 4.1 Arduino microcontroller [3]. The Teensy is programmed to function as a QSPI master, receiving DMI commands from the host over USB and transmitting

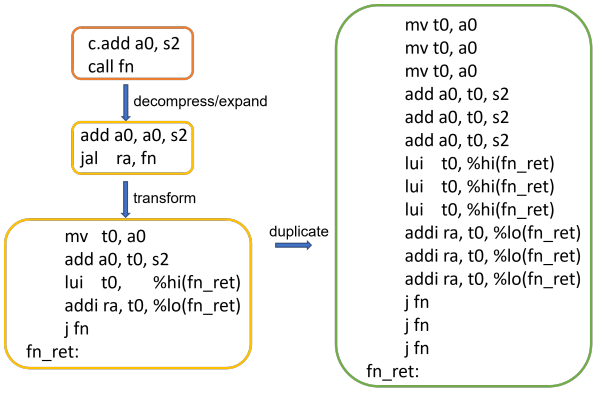

Fig. 2: An example of protecting a group of vulnerable instructions consisting of a compressed and a pseudo-instruction.

them to the SoC using a software-based QSPI implementation. Essentially, we supplemented the SoC with a hardware multiplexer responsible for managing JTAG and QSPI slave communication. The multiplexer translates debugging commands into general Debug Module Interface commands, adhering to the RISC-V Debug Specification [16]. IN this particular design, DM implementation operates on an execution-based principle. This means that when the core enters debug mode, the code in the DM's ROM, referred to as the "park loop," is executed. We extended the DM registers and ROM code in line with our custom extension's specifications to allow automated program counter modification upon breakpoint. This fully automates the process of emulating an instruction skip on the chip without instrumenting the code being executed. To enable high-speed debugging and accommodate the use of two different debugger implementations, we adjusted the SoC's pinout, introducing 10 additional pins to integrate the QSPI interface.

8 Evaluation

In this section, we conduct an assessment of different debugger implementations and analyze the impact of our hardening method on several key metrics, namely the overall testing time, program runtime, and code size. To evaluate our approach, we opted to employ well-established cryptographic algorithms obtained from the MiBench benchmark suite [20]. These algorithms encompass popular cryptographic functions such as AES-128, SHA-256, and Blowfish-CFB64. Throughout our experiments, we focused on a scenario where the SoC received an input that comprehensively exercises the execution of all instructions within the functions under test. Our evaluation leveraged the previously described setup, allowing us to rigorously assess the performance of our implementation. The experiments were conducted on a host machine running Windows 10, equipped

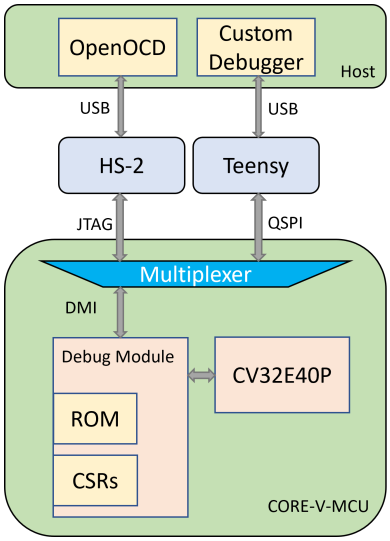

Fig. 3: Schematic overview of the emulation setup.

with an Intel Core i5-1145G7 processor operating at clock speed of 2.6 GHz and 16 GB of RAM.

To begin, we assess the influence of debugger selection and the utilization of our custom DM extension on fault injection testing time. Our initial focus involves conducting a single instruction skip fault injection campaign specifically targeting cryptographic calculation functions, while timeout was configured to 3 seconds. During these evaluations, we compared the outputs of a single run against an established golden truth value to detect Silent Data Corruption (SDC) upon completion of the main function. Thus, we consider instructions as vulnerable if skipping them resulted SDC, indicating a necessity for hardening measures.

Furthermore, to gauge the impact of our DM extension, we implemented debugger-driven fault injection testing (as detailed in Section 4) without deploying the DM extension. The only difference in this approach occurred at step 3, where instead of configuring extended CSRs, we set a breakpoint at the fault address and adjusted the program counter by the byte count of the instruction to skip once the breakpoint was hit. The individual test run durations are detailed in Table 5, while the outcomes of the fault injection campaign are summarized in Table 6. Notably, the Blowfish test program stood out, with almost half of its instructions identified as vulnerable. Upon manual inspection of the assembler code, we observed heavy reliance on loop unrolling within the algorithm, resulting in multiple instances of vulnerable instructions.

An evident outcome is the notable reduction in test time facilitated by our custom DM extension, observed across both debugger implementations. This improvement stems from a reduction in communication overhead compared to the pure debugger solution. The hardware-driven fault injection circumvents the

	Not using extension	Using extension
OpenOCD	1.55s (1x)	1.21s (1.28x)
Custom debugger	0.24s (6.45x)	0.14s (11.07x)

Table 5: Average single test run duration of the FI campaign performed by two different debugger implementations and using different fault injection methods. The numbers in parentheses represent the speedup factor compared to the baseline, which is FI campaign performed by OpenOCD not using extension.

	Total instructions tested	Number of vulnerable instructions	Timeout events
AES	3512	54	36
SHA	520	69	14
Blowfish	1241	564	6

Table 6: Results of fault injection campaign.

need for debugger notifications, eliminating delays within a single run. Hence, we observed a distinct at least 21% decrease in single run duration with the enhanced fault injection method compared to the pure debugger solution.

Moreover, exchanging the debug transport module protocol (originally JTAG) for a more efficient protocol (QSPI) led to a substantial increase in communication speed, resulting in a significant reduction in overall testing time by several orders of magnitude. To underscore this difference, we conducted an additional experiment to measure the speed of read/write operations to memory. The measured speed for memory operations, regardless of read or write, amounted to 0.13 MB/s for the OpenOCD/HS-2 combination and surged to 2.45 MB/s for the Teensy implementation. This enhancement, approximately 18 times faster than the conventional debug toolchain, underscores the efficiency gain achieved through the protocol switch.

Our subsequent evaluation analyzed the hardening of vulnerable instructions and validating that their skipping no longer leads to any occurrences of SDCs, which serves as evidence of the efficacy of our hardening strategy. In addition, we conducted a comparative analysis to assess the impact of hardening on both runtime and code size, detailed in Tables 7 and 8 respectively. When discussing the impact on code size, a program is classified as partially hardened if only the vulnerable instructions were duplicated. On the other hand, a fully hardened program involves duplicating every instruction within a tested function. Our observations reveal that code size experiences minimal overhead when subjected to fault injection testing before hardening implementation. However, fully hardened functions result in at least a twofold increase in code size compared to its pre-hardened state. Furthermore, our experiment data underscores that program runtime is less affected in the partially hardened scenario. This outcome can be attributed to the selective duplication of only the necessary instructions in the partially hardened variant.

	Original size	Fully hardened size	Partially hardened size
AES	3512 (100%)	8340 (237%)	3584 (102%)
SHA	520 (100%)	1102 (211%)	602 (115%)
Blowfish	1241 (100%)	2704 (218%)	1953 (157%)

Table 7: Comparison of instruction number for original and hardened binaries. The numbers in parentheses represent the percentage increase compared to the original size.

	Original runtime(ms)	Fully hardened runtime(ms)	Partially hardened runtime(ms)
AES	24.3 (100%)	65.6 (270%)	27.0 (111%)
SHA	21.1 (100%)	41.2 (195%)	23.4 (110%)
Blowfish	18.5 (100%)	40.5 (219%)	25.6 (121%)

Table 8: Comparison of average runtime performance for original and hardened binaries. The numbers in parentheses to represent the percentage increase in runtime compared to the original runtime.

9 Conclusion

In this work, we dove into the topic of the fault injection emulation, to enable security assessment during the code development phase. It is desired for some critical code parts in embedded software to be as fault-tolerant and sound as possible. Via fault injection emulation, developers can test their system in a fast and efficient way in the pre-silicon phase. One particularly relevant fault model is the instruction skip model. If an attacker can force a device to skip one or more instructions via physical fault injection, this can lead to a bypass of crucial security checks and countermeasures. Upcoming open-source technologies like the RISC-V architecture make it possible to design and tailor systems for specific needs like fault injection testing. To accelerate debugger-driven fault injection testing, we designed an extension compliant with RISC-V debug specification. Our proposal involves modifying the debug module within an SoC to facilitate automatic skipping of an arbitrary number of instructions at a breakpoint. Through the integration of special CSRs and debug module modifications, we achieved a reduction in the duration of a single test run compared to using a pure debugger solution. Additionally, by replacing the conventional JTAG debug transport protocol with a custom QSPI interface, we witnessed a remarkable improvement in communication speed, enhancing the performance of the debugging toolchain significantly. As a strategy to counter fault injection attacks, we introduced a duplication-based code hardening technique adopted for the RISC-V instruction set to improve fault tolerance of test binaries. By fault injection testing and patching only vulnerable parts of code, we were able to completely prevent fault effects within the assumed threat model. Notably, the partial code hardening introduced less size and runtime overhead compared to full code du-

plication while maintaining an equivalent level of security. This approach ensures enhanced fault tolerance while minimizing the associated resource demands.

Acknowledgments

Funded by the European Union under grant agreement no. 101070008. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

References

1. JTAG-HS2 programming cable. https://digilent.com/shop/jtag-hs2-programming-cable/, accessed on 2023-12-01
2. Open on-chip debugger. https://openocd.org/, accessed on 2023-12-01
3. Teensy® 4.1 development board. https://www.pjrc.com/store/teensy41.html, accessed on 2024-01-04
4. Ahmad, H.A.h., Sedaghat, Y., Moradiyan, M.: LDSFI: a lightweight dynamic software-based fault injection. In: 2019 9th International Conference on Computer and Knowledge Engineering (ICCKE). pp. 207–213 (2019). https://doi.org/10.1109/ICCKE48569.2019.8964875, ISSN: 2643-279X
5. Balasch, J., Gierlichs, B., Verbauwhede, I.: An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In: 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography. pp. 105–114. https://doi.org/10.1109/FDTC.2011.9
6. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE **94**(2), 370–382 (2006). https://doi.org/10.1109/JPROC.2005.862424, http://ieeexplore.ieee.org/document/1580506/
7. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures **100**(11), 3056–3076. https://doi.org/10.1109/JPROC.2012.2188769, conference Name: Proceedings of the IEEE
8. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. Proceedings of the IEEE **100**(11), 3056–3076 (2012). https://doi.org/10.1109/JPROC.2012.2188769
9. Barry, T., Couroussé, D., Robisson, B.: Compilation of a countermeasure against instruction-skip fault attacks. In: Proceedings of the Third Workshop on Cryptography and Security in Computing Systems. pp. 1–6. ACM. https://doi.org/10.1145/2858930.2858931, https://dl.acm.org/doi/10.1145/2858930.2858931
10. Blömer, J., Silva, R.G.d., Günther, P., Krämer, J., Seifert, J.P.: A practical second-order fault attack against a real-world pairing implementation, https://eprint.iacr.org/undefined/undefined
11. Breier, J., Hou, X.: How practical are fault injection attacks, really? **10**, 113122–113130. https://doi.org/10.1109/ACCESS.2022.3217212, conference Name: IEEE Access

12. Colombier, B., Grandamme, P., Vernay, J., Chanavat, E., Bossuet, L., de Laulanié, L., Chassagne, B.: Multi-spot laser fault injection setup: New possibilities for fault injection attacks. In: Grosso, V., Pöppelmann, T. (eds.) Smart Card Research and Advanced Applications, vol. 13173, pp. 151–166. Springer International Publishing. https://doi.org/10.1007/978-3-030-97348-3_9, https://link.springer.com/10.1007/978-3-030-97348-3_9, series Title: Lecture Notes in Computer Science
13. Dutertre, J.M., Riom, T., Potin, O., Rigaud, J.B.: Experimental analysis of the laser-induced instruction skip fault model. In: Askarov, A., Hansen, R.R., Rafnsson, W. (eds.) Secure IT Systems, vol. 11875, pp. 221–237. Springer International Publishing. https://doi.org/10.1007/978-3-030-35055-0_14, http://link.springer.com/10.1007/978-3-030-35055-0_14, series Title: Lecture Notes in Computer Science
14. Elmohr, M.A.: Embedded systems security: On EM fault injection on RISC-v and BR/TBR PUF design on FPGA
15. Farooq, U., Mehrez, H.: Pre-silicon verification using multi-FPGA platforms: A review **37**(1), 7–24. https://doi.org/10.1007/s10836-021-05929-1, https://link.springer.com/10.1007/s10836-021-05929-1
16. Foundation, R.V.: RISC-V Debug Specification. Specification 0.13.2, RISC-V Foundation (2019), https://riscv.org/specifications/debug-specification/
17. Gangolli, A., Mahmoud, Q.H., Azim, A.: A systematic review of fault injection attacks on IoT systems **11**(13), 2023. https://doi.org/10.3390/electronics11132023, https://www.mdpi.com/2079-9292/11/13/2023
18. Gautschi, M., Schiavone, P.D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gurkaynak, F., Benini, L.: Near-threshold RISC-v core with DSP extensions for scalable IoT endpoint devices (2017). https://doi.org/10.1109/TVLSI.2017.2654506, https://ieeexplore.ieee.org/document/7864441
19. Giraud, C., Thiebeauld, H.: A survey on fault attacks. International Federation for Information Processing Digital Library; Smart Card Research and Advanced Applications VI; **153** (2004). https://doi.org/10.1007/1-4020-8147-2_11
20. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: Mibench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538). pp. 3–14 (2001). https://doi.org/10.1109/WWC.2001.990739
21. Kiaei, P., Breunesse, C.B., Ahmadi, M., Schaumont, P., Woudenberg, J.v.: Rewrite to reinforce: Rewriting the binary to apply countermeasures against fault injection. In: 2021 58th ACM/IEEE Design Automation Conference (DAC). pp. 319–324 (2021). https://doi.org/10.1109/DAC18074.2021.9586278
22. Menu, A., Dutertre, J.M., Potin, O., Rigaud, J.B., Danger, J.L.: Experimental analysis of the electromagnetic instruction skip fault model. In: 2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS). pp. 1–7. https://doi.org/10.1109/DTIS48698.2020.9081261
23. Moro, N., Heydemann, K., Encrenaz, E., Robisson, B.: Formal verification of a software countermeasure against instruction skip attacks **4**(3), 145–156. https://doi.org/10.1007/s13389-014-0077-7, http://link.springer.com/10.1007/s13389-014-0077-7
24. Moro, N., Heydemann, K., Dehbaoui, A., Robisson, B., Encrenaz, E.: Experimental evaluation of two software countermeasures against fault attacks. In: 2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST). pp. 112–117. https://doi.org/10.1109/HST.2014.6855580

25. MOSDORF, M., SOSNOWSKI, J.: Fault injection in embedded systems using gnu debugger (2011)
26. Portela-García, M., López-Ongil, C., Garcia Valderas, M.G., Entrena, L.: Fault injection in modern microprocessors using on-chip debugging infrastructures. IEEE Transactions on Dependable and Secure Computing **8**(2), 308–314 (2011). https://doi.org/10.1109/TDSC.2010.50
27. Proy, J., Heydemann, K., Majéric, F., Cohen, A., Berzati, A.: Studying EM pulse effects on superscalar microarchitectures at ISA level, `http://arxiv.org/abs/1903.02623`
28. Reis, G., Chang, J., Vachharajani, N., Rangan, R., August, D.: SWIFT: Software implemented fault tolerance. In: International Symposium on Code Generation and Optimization. pp. 243–254. IEEE. https://doi.org/10.1109/CGO.2005.34, `http://ieeexplore.ieee.org/document/1402092/`
29. Rivière, L., Najm, Z., Rauzy, P., Danger, J.L., Bringer, J., Sauvage, L.: High precision fault injections on the instruction cache of ARMv7-m architectures, `https://eprint.iacr.org/undefined/undefined`
30. Saß, M., Mitev, R., Sadeghi, A.R.: Oops..! i glitched it again! how to multi-glitch the glitching-protections on ARM TrustZone-m, `http://arxiv.org/abs/2302.06932`
31. Schiavone, P.D., Rossi, D., Di Mauro, A., Gurkaynak, F., Saxe, T., Wang, M., Yap, K.C., Benini, L.: Arnold: An eFPGA-augmented RISC-v SoC for flexible and low-power IoT end nodes . https://doi.org/10.1109/TVLSI.2021.3058162, `https://ieeexplore.ieee.org/document/9369856`
32. Schirmeier, H., Hoffmann, M., Dietrich, C., Lenz, M., Lohmann, D., Spinczyk, O.: FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In: 2015 11th European Dependable Computing Conference (EDCC). pp. 245–255 (2015). https://doi.org/10.1109/EDCC.2015.28
33. Sharif, U., Mueller-Gritschneder, D., Schlichtmann, U.: COMPAS: Compiler-assisted software-implemented hardware fault tolerance for RISC-v. In: 2022 11th Mediterranean Conference on Embedded Computing (MECO). pp. 1–4. https://doi.org/10.1109/MECO55406.2022.9797144, ISSN: 2637-9511
34. Timmers, N., Mune, C.: Escalating privileges in linux using voltage fault injection. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). pp. 1–8 (2017). https://doi.org/10.1109/FDTC.2017.16
35. Witteman, M.: Security highlight: Multi-fault attacks are practical (Apr 2023), `https://www.riscure.com/security-highlight-multi-fault-attacks-are-practical/`
36. Yuce, B., Ghalaty, N.F., Santapuri, H., Deshpande, C., Patrick, C., Schaumont, P.: Software fault resistance is futile: Effective single-glitch attacks. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). pp. 47–58. https://doi.org/10.1109/FDTC.2016.21
37. Zhang, Y., Liu, B., Zhou, Q.: A dynamic software binary fault injection system for real-time embedded software. In: The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety. pp. 676–680 (2011). https://doi.org/10.1109/ICRMS.2011.5979375
38. Ziade, H., Ayoubi, R., Velazco, R.: A survey on fault injection techniques **1**(2) (2004)