ORSHIN

D4.1

Report on security audit and testing

Project number	101070008
Project acronym	ORSHIN
Project title	Open-source ReSilient Hardware and software for Internet of thiNgs
Start date of the project	1st October, 2022
Duration	36 months
Call	HORIZON-CL3-2021-CS-01

Deliverable type	Report
Deliverable reference number	CL3-2021-CS-01 / D4.1 / 1.0
Work package contributing to the deliverable	WP4
Due date	Jun 2025 – M33
Actual submission date	30th June 2025

Responsible organisation	NXP
Editor	Jean-Michel Cioranesco, Oliver Diehl, Volodymyr Bezsmertnyi, Ammar Ben Khadra
Dissemination level	PU
Revision	1.0

Abstract	We proposed and developed two concepts of hardware acceleration for SW security testing, one for accelerating simulation of fault injection on Software pre-silicon and the other to support Logical Software testing such as Fuzzing or Symbolic execution.
Keywords	Security Testing, Fault injection, Fuzzing, RISC-V, Debugger, Logical vulnerabilities

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.

Editor

NXP Semiconductors Germany GmbH (NXP)

Contributors (ordered according to beneficiary numbers)

Jean-Michel Cioranesco (NXP)

Oliver Diehl (NXP)

Volodymyr Bezsmertnyi (NXP)

Ammar Ben Khadra (NXP)

Reviewer (ordered according to beneficiary numbers)

Clarisse Ginet (TXP)

Jan Pleskac (TRPC)

Executive Summary

Deliverable D4.1 presents the successful implementation of Work Package 4, focused on pre-silicon security testing of embedded firmware. It comprises of two key tasks:

- **Task 4.1: Fault Injection Emulation**
  Developed a hardware-assisted framework for fault injection on RISC-V systems, featuring a custom debug module and automatic code hardening via instruction duplication.
- **Task 4.2: Logical Vulnerability Testing**
  Introduced the Software Testing Acceleration Module (STAM) for hybrid fuzzing and symbolic execution, enabling efficient tracing and code gadget resolution. This approach significantly improves vulnerability detection and outperforms existing tools.

The prototype, publicly available with full documentation, includes FPGA bitfiles, firmware and driver for the debug connector, fault injection and hybrid fuzzing testing frameworks, a modified OpenOCD, and example test code.

These contributions mark a major advancement in firmware security testing, combining hardware acceleration with intelligent software techniques while the open-source nature of the tools promotes broad adoption and collaboration.

Table of Content

Chapter 1 Introduction 1

Chapter 2 Task 4.1 2

2.1 Task Description 2
2.2 Background on Fault attacks and code hardenings 2
2.2.1 Hardware Fault Injection Attacks 2
2.2.2 Instruction Skip Fault Model 2
2.2.3 Software-Implemented Fault Tolerance 3
2.2.4 Emulated Fault Injection 3
2.3 Concept Proposal 4
2.4 Task Objectives 5
2.5 Design and Implementation 7
2.5.1 Protection by Fault Injection Emulation 7
2.5.2 Debugger-Driven FI Testing 8
2.5.3 Debug Specification Extension 9
2.5.4 Code Hardening Tool 10
2.5.5 Hardware Implementation 11
2.6 Conclusion and Continuation of T4.1 13

Chapter 3 Task 4.2 15

3.1 Task Description 15
3.2 Concept Proposal 15
3.3 Task Objectives 18
3.4 Solution Design 18
3.5 STAM 19
3.6 Tracing Mechanism 21
3.7 Hardware Acceleration 23
3.8 Hybrid fuzzing 24
3.9 Implementation 24
3.10 Evaluation 25
3.10.1 Tracing 25
3.10.2 Hybrid fuzzing 26
3.11 Conclusion and Continuation of T4.2 27

Chapter 4 Summary and Conclusion 29

Chapter 5 List of Abbreviations 30

Chapter 6 Bibliography 31

List of Figures

Figure 1: Classification of Pre-Silicon Physical Attack Testing 5
Figure 2: Design for Hardware/Firmware hybrid FI security testing 6
Figure 3: Workflow of our Fault Injection Framework.....7
Figure 4: Example of Protected Assembly Code 11
Figure 5: Block diagram of the FIM module 12
Figure 6: QSPI pinout of Teensy 4.1.....13
Figure 7 General architecture of a feedback directed fuzzer 16
Figure 8: Design for native system-level software security testing.....18
Figure 9: Setup overview 19
Figure 10: Single Instruction Tracing 22
Figure 11: Branch Tracing Concept 23
Figure 12: Performance of different tracing strategies 26
Figure 13 Edge coverage comparison 27

List of Tables

Table 1: Control and Status Registers of Software Testing Automation Module.....21

Chapter 1 Introduction

This deliverable presents the outcomes of Work Package 4 (WP4) of the ORSHIN project, which aims to enhance the security and resilience of embedded systems through open-source innovation. Task 4.1 and 4.2 of WP4 focus on pre-silicon security testing of firmware, addressing both physical and logical vulnerabilities in RISC-V-based platforms.

As embedded systems become increasingly prevalent in critical applications, ensuring their security at the earliest stages of development is essential. Traditional post-silicon testing methods are often insufficient for detecting subtle or hardware-dependent vulnerabilities. Moreover, the open-source nature of RISC-V hardware introduces new opportunities—and challenges—for scalable, transparent, and collaborative security validation. WP4 responds to these challenges by developing hardware-accelerated, open-source frameworks for fault injection and logical vulnerability testing, enabling comprehensive pre-silicon evaluation of firmware security, while the prototypes resulted from the tasks T4.1 and T4.2 will be made publicly available.

T4.1 introduces a hardware-assisted fault injection testing framework built around a custom Fault Injection Module (FIM). This module, integrated into an FPGA-emulated SoC, enables debugger-driven instruction skip emulation and supports automated code hardening using duplication-based countermeasures tailored to the RISC-V instruction set. The framework leverages a high-speed QSPI interface and custom debug module extensions to achieve efficient and scalable testing.

T4.2 extends this infrastructure with the Software Testing Acceleration Module (STAM), which adds hardware-assisted instruction and branch tracing capabilities. STAM introduces custom debug registers and a hardware-accelerated binary search engine to support high-throughput trace resolution. These enhancements enable a hybrid fuzzing framework that combines feedback-guided fuzzing with symbolic execution, significantly improving code coverage and vulnerability detection.

The report begins with Chapter 1, which introduces the scope, motivation, and objectives of Work Package 4 within the ORSHIN project. Chapter 2 focuses on T4.1, detailing the design and implementation of a hardware-assisted fault injection testing framework. It covers the background on fault models, the rationale for using FPGA-based emulation, the architecture of the FIM, and the development of a code hardening tool tailored to the RISC-V instruction set. Chapter 3 presents T4.2, which extends the FIM into the STAM to support advanced software testing techniques. This chapter describes the tracing mechanisms, hybrid fuzzing methodology, and the integration of symbolic execution for improved vulnerability detection. Chapter 4 summarizes the key results and discusses future directions for research and development. The report concludes with Chapter 5, which provides a list of abbreviations used throughout the document, and Chapter 6, which compiles the references cited in the report.

This deliverable provides a comprehensive technical account of the methodologies, implementations, and performance evaluations conducted in WP4, offering a robust foundation for further research and development in secure open-source embedded systems.

Update: comparing to iD4.1, this deliverable contains reworked Chapter 3, as well as updated executive summary, introduction and conclusion chapters.

Chapter 2 Task 4.1

2.1 Task Description

Task 4.1 Design for Hardware/Firmware hybrid security testing (M03-M33; Task Lead: NXP)

Closed-source hardware in production is not designed for security testing (i.e., limited introspection capability), making the security testing very challenging for traditional security testing methods (e.g., fuzzing, symbolic execution). In this task, we will design and implement novel and custom hardware accelerators for software security testing. The proposed method will enable hardware and software co-testing.

2.2 Background on Fault attacks and code hardenings

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.

2.2.1 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 [4], [5], [6]. In [7], multiple fault injection attacks on microcontroller-based cryptographic algorithm implementations are demonstrated. In [8], the practicality of fault injections is examined through empirical research. A systematic examination of fault injections in Internet-of-Things devices is conducted in [9].

2.2.2 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. The number of instructions being skipped varies depending on the part of the system being corrupted (CPU fetches, pre-fetches, caches, instructions being read from memory lines). Faults in decoding and execution stage of the processor, results in greater variety of faults, that we will add to our simulation engine in a second stage. Here, we list some examples of works that achieved either single or multiple instruction skips through fault injection.

Single instruction skip is a fault effect frequently seen in fault injection testing of many microcontrollers. A recent work [10] 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 [11] showed an exploit where VFI was used to escalate privilege in Linux from user space. Balasch et al. [12] 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 [13] proposed a technique that uses multiple lasers in order to induce multiple single-bit faults in an ARM Cortex-M3. Menu et al. [14] 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. [15] studied EM pulse effects at the ISA (instruction set architecture) level.

Multiple Instruction Skip is 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. Rivière et al. [16] managed to skip up to four consecutive instructions by electromagnetically faulting the instruction cache of an ARM Cortex-M CPU. Blömer et al. [17] utilized multiple clock fault injections for attacking two consecutive instructions. Dutertre et al. [18] were able to skip groups of instructions by laser illumination on an 8-bit non-secure ATmega328P microcontroller. Yuce et al. [19] 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 [20] reported EM-induced skips of up to six consecutive instructions with low repeatability on a RISC-V FPGA implementation.

2.2.3 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. [21] provided a formal proof showing the efficiency of redundancy-based countermeasures against a single instruction skip. Their countermeasure consists of in 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 [22] Moro et al. performed evaluation of two countermeasures by launching physical fault attacks and assessing the impact. Barenghi et al. [23] proposed software countermeasures for cryptographic algorithms including intrusion and fault detection. Barry et al. [24] implemented a LLVM compiler extension which protects against instruction skip attacks. Sharif et al. [25] developed a compiler framework targeting RISC-V processors which hardens code using various fault tolerance techniques. Schirmeier et al. [26] provided a fault injection framework for detecting vulnerable code by emulating faults with a debugger. Kiaei et al. [27] 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.

2.2.4 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 at 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. [28] utilized the On-Chip Debugger (OCD) to inject faults into a microcontroller that supported JTAG debugging Fault Tolerance for RISC-V 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. [29] injected faults using the GDB debugger and a J-Link debugger via the JTAG interface of an ARM device. Schirmeier et al. [30] 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. [31] utilized a debugger for the fault injection testing of a real-time operating system. Ahmad et al. [32] 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.

2.3 Concept Proposal

Hybrid software-hardware testing offers a comprehensive approach to security assessment by combining the strengths of both software and hardware-based testing methodologies. By integrating software-based techniques such as static and dynamic analysis with hardware-based approaches like fault injection and side-channel analysis, hybrid testing provides a more thorough evaluation of system vulnerabilities.

This approach enables the detection of both logical vulnerabilities in software code and physical vulnerabilities arising from hardware-level weaknesses, offering a more holistic view of system security compared to emulator-based testing.

Hybrid testing facilitates the identification of complex attack vectors that span across software and hardware components, allowing for more effective mitigation strategies to be developed. For example, it can identify vulnerabilities which are timing/hardware dependant and could be overlooked when testing with an emulated virtual environment, which consists of an ISA simulator and C-model of the hardware peripherals such as crypto-accelerators, firewalls, key-management blocks, memory subsystem etc.

Pre-silicon fault injection testing plays a crucial role in ensuring the reliability and security of integrated circuits before they are fabricated, or "taped out," for production. This testing methodology involves intentionally inducing faults or errors in the hardware design to assess its resilience against various fault injection attacks, such as voltage glitches, laser attacks, or electromagnetic interference. Identifying vulnerabilities at the pre-silicon stage is paramount for mitigating the risk of costly production errors and security breaches.

One of the primary reasons, why pre-silicon fault injection testing is critical, is its ability to uncover design flaws and vulnerabilities early in the development process. By subjecting the hardware design to simulated fault injection attacks, designers can identify weak points in the system architecture, logic gates, or memory elements that may be susceptible to exploitation by malicious actors or environmental factors. Addressing these vulnerabilities before tape-out reduces the likelihood of costly design iterations or product recalls during the later stages of production.

Moreover, pre-silicon fault injection testing helps to validate the effectiveness of built-in security mechanisms, such as error detection and correction codes, redundancy schemes, and secure boot mechanisms. By subjecting these mechanisms to simulated fault injection attacks, designers can assess their robustness and identify any potential weaknesses that may compromise the security of the system. Strengthening these security features early in the design phase enhances the overall security posture of the integrated circuit and reduces the risk of post-production security breaches.

Three approaches are possible for pre-silicon fault injection testing, resumed in Figure 1. First approach is fault injection in RTL simulation, which has as advantages to provide a detailed, cycle-accurate representation of the hardware design, to enable precise control over fault injection scenarios and parameters and to allow for comprehensive testing of the entire system, including complex interactions between hardware components. It also has disadvantages as it requires significant computational resources and simulation time, limiting scalability, and it may overlook timing-related issues or non-deterministic behaviour due to simulation abstraction. Additionally matching a physical attack fault model, to several gates that must be flipped is not trivial.

The second approach is fault injection on an Instruction set simulator with hardware peripherals represented as C Models. The advantages of such a representation offer a balance between accuracy and simulation speed, making it suitable for large-scale testing, allow for the integration of software and hardware fault injection techniques, enabling more realistic testing scenarios, and facilitate the reuse of existing software test suites and development tools. It has the disadvantages that it offers limited accuracy compared to full RTL simulation, particularly for timing-sensitive designs, and relies on the accuracy of C models to represent hardware peripherals, which may introduce abstraction errors.

Finally, the third approach, and the one we focus in this task, is to emulate fault injection on an FPGA. It has as advantage to provide a hardware-based testing environment, offering real-time execution and accurate timing, allow for the injection of faults directly into the physical hardware, enabling realistic testing scenarios and facilitate rapid prototyping and iterative testing, reducing development time and cost.

As for the other methodologies, it presents several drawbacks. It requires additional effort and expertise to implement fault injection capabilities on the FPGA, it offers limited scalability for testing large-scale designs or complex systems and it may incur higher upfront costs for FPGA development boards and associated tools.

In summary, each pre-silicon fault injection methodology offers distinct benefits and drawbacks, depending on the specific requirements and constraints of the hardware design and testing objectives. Full RTL simulation provides detailed accuracy but may be resource-intensive, while simulation with C models balances accuracy with speed. FPGA emulation offers real-time testing but requires additional setup and may have scalability limitations. Choosing the most appropriate methodology depends on factors such as design complexity, testing goals, and available resources.

Pre-silicon methodologies	Bring-up effort	Runtime	Coverage in respect to Effort	Missing Modeling	Advantages	Disadvantages
Full RTL Simulation	Low	Very slow	Very High	Analog part	Most accurate modelling of the system	Too slow to offer a high coverage, Licenses, Computing resources
ISA emulation + peripheral models	Very high	Fast (depending on modelling)	Low	Timing information – uArchitectural states – Peripherals – Analog parts	Fast simulation, and enable parallelization of tests	Missing/inaccurate modelling limits coverage and brings false results
FPGA emulation	Mid to High	Fast	Very high	Analog part / physical effects	Enable very high coverage	Has to be ported per platform

Figure 1: Classification of Pre-Silicon Physical Attack Testing

2.4 Task Objectives

The main objective of Task 4.1 is to design and implement a state-of-the-art, open source, hardware accelerator for software FI testing on FPGA emulated system (see Figure 2).

The proposed method would enable hardware and software pre-silicon co-testing of fault injection with various fault models.

We leverage the open source instruction set RISC-V and related open sourced hardware implementations to prototype our solution. The Hardware implementation and Software testing is performed on RV32IMC architecture.

We selected the FPGA emulation platform based on core CV32E40P from OpenHW [1] for our development, which is a mature open source project offering years of code support, and a large community that can possibly become user of our solution. In particular the open source project offers full FPGA emulation, from which we can build our demonstrator, and run software on a custom hardware variation of the SoC.

We named our custom block Fault Injection Module (FIM), This block enables direct instrumentation of the RISCV core by a direct control of the Debug module. This module also implements its own high speed protocol to optimize host communication and enable tracing and large memory snapshotting in future developments of T4.2. The design of the FIM module has been integrated into our demonstrator.

On the host side we develop a SW test framework using FIM and debug module instrumentation to perform emulated fault injection on FPGA. Thanks to the direct control of the debug module, we will have full CPU control with breakpoint capability and memory accesses. The design is kept to a minimal size to limit resource utilization on FPGA. 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, adapted specifically for the nuances of the RISC-V instruction set.

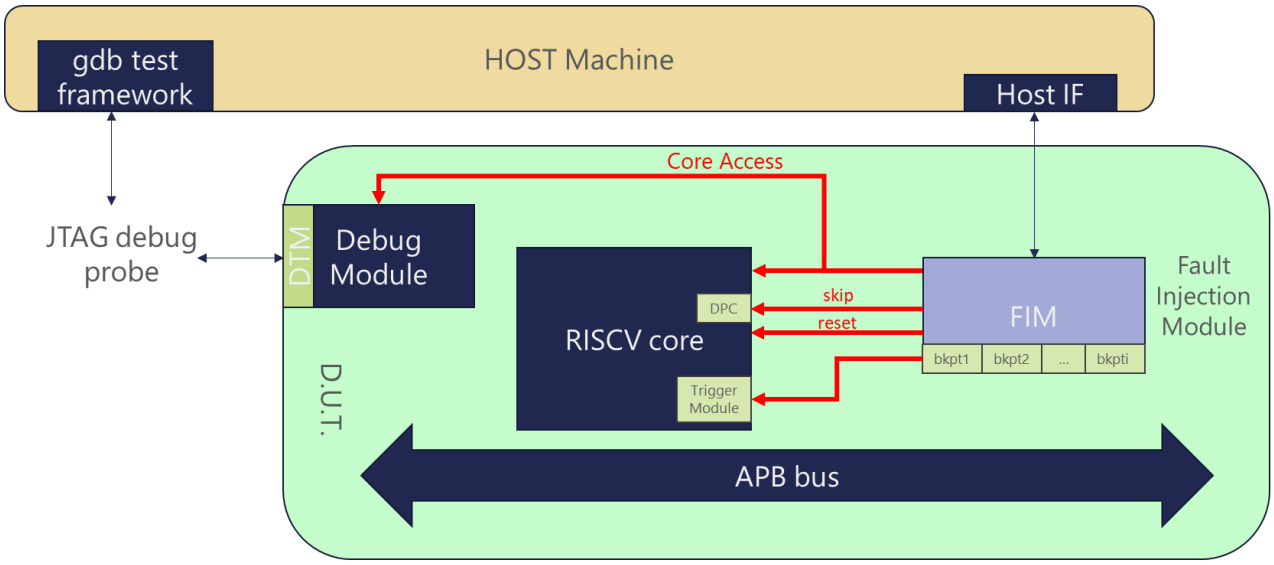


Figure 2: Design for Hardware/Firmware hybrid FI security testing

2.5 Design and Implementation

2.5.1 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 following Figure 3.

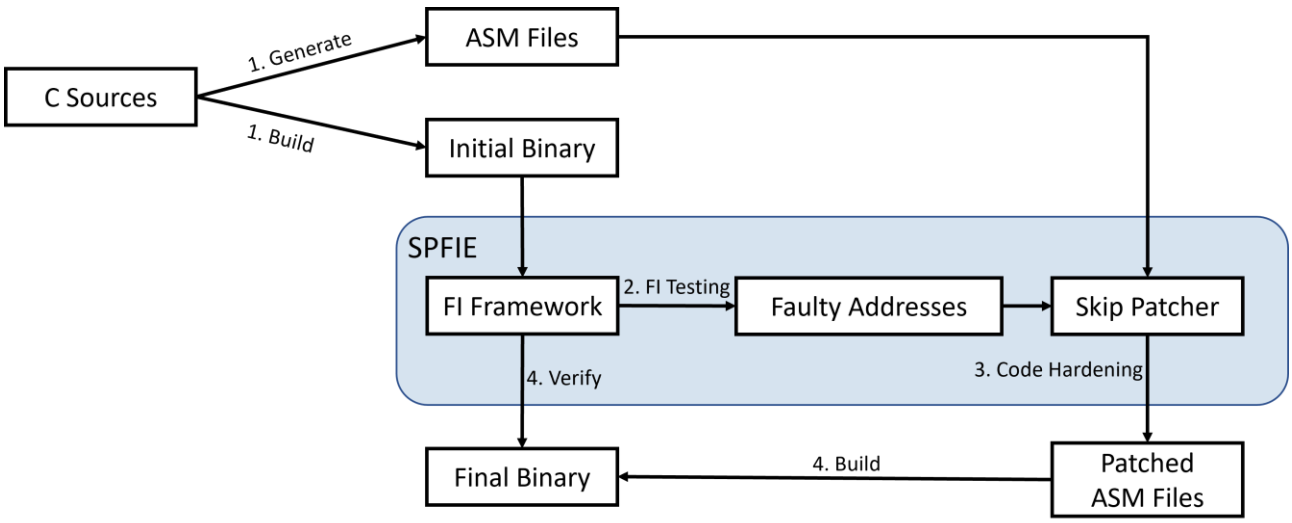


Figure 3: Workflow of our Fault Injection Framework

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 a later section.
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.

2.5.2 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 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 a large number 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 the next section) 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 registers 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 timeouts 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.

2.5.3 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 includes 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:

1. **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 it is executed.
2. **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.
3. **pc_delta**: Contained in this register is a signed integer 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 as jump to an arbitrary address, which can be useful in some cases, like for testing unexpected control flow violations.

2.5.4 Code Hardening Tool

The Code Hardening Tool (CHT) 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. 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.

- **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.

- **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.

- **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.

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 following Figure 4.

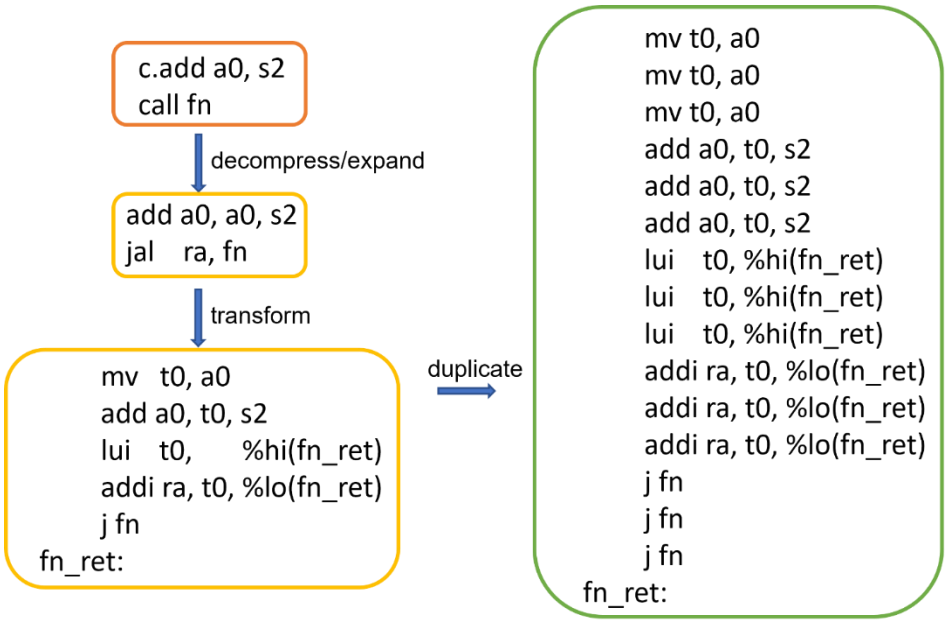


Figure 4: Example of Protected Assembly Code

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***. After 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.

2.5.5 Hardware Implementation

Our solution comprises of three distinct parts; The hardware module integrated in the SoC, which interfaces with the debug module internally and externally to a master controller using a QSPI custom interface, the controller which interfaces our host machine via USB2 and our embedded module using QSPI interface and the Host test framework which controls the overall testing and execution on the target.

The host controller can be for example a computer, which communicates to the DUT via a USB-to-SPI bridge implemented for example on a Teensy 4.1 [2] board.

The host sets the hardware breakpoint. The CPU executes normally until it hits the breakpoint. The CPU executes the debug code (in the debug ROM).

The debug code must contain the instruction to jump into FIM code. The CPU performs the programmed skip and continues the execution. The host can be programmed to automatically emulate and monitor different scenarios (i.e. different locations of the breakpoint and different skip mechanics).

Fault Injection Module (FIM)

The FIM module, integrated in an SoC, can be used to emulate fault injection.
This component is especially useful to perform fault injection analysis at highspeed on FPGA, to develop countermeasures pre-silicon.
The basic functionality offered by the FIM can be described as a high-speed interface to the CPU's debug module (DM), coupled with a tiny SRAM which extends the debug ROM included in the DM. By leveraging the functionalities offered by the DM, fault injection can be emulated by setting a hardware breakpoint and modifying the CPU's program counter (PC) after reaching the breakpoint.
An alternative approach could have been the implementation of something like an "hardware OpenOCD" This would have surely granted faster execution speed (since it would have required less messages exchanged between the host and the DUT), but would have required the implementation of a more complex and harder to maintain FIM module.
The selected architecture, on the contrary, has been developed quickly and can support different test strategies, which can be easily implemented on the host controller.

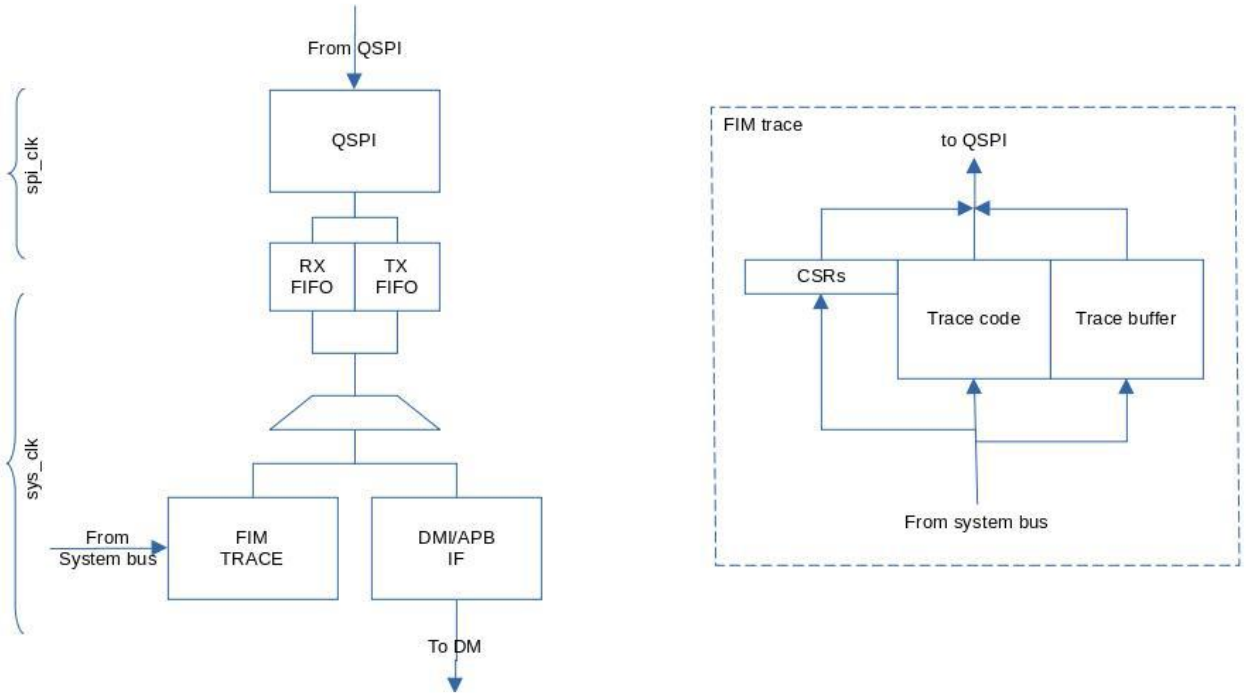


Figure 5: Block diagram of the FIM module

Debug Controller

We based our controller on the Arduino based Teensy board [2]. The Teensy board runs a communication dispatcher, where it monitors incoming transactions from slave interface. It is connected to the host computer with a full speed USB2 and receives serial commands. The incoming commands to the FIM can be packed in multiple individual commands to the DMI. Commands are unpacked and transferred to the slave interface, a custom QSPI protocol. The QSPI interface is implemented as software code, but instrumenting a single IO register bank of the Teensy, which gives it a hardware QSPI behaviour. Moreover the Teensy controller is clocked at 520MHz core frequency, which allows high throughput to the DUT. An optimized interposer to the FPGA could allow us to raise the CPU frequency, increasing the throughput even further. The Teensy can be

overclocked to 1GhZ, so the timing between the QSPI lines in the interconnection to the FPGA are more of an issue, due to potential data losses.

Figure 6: QSPI pinout of Teensy 4.1

2.6 Conclusion and Continuation of T4.1

To accelerate debugger-driven fault injection testing, we designed an extension compliant with RISC-V debug specification. Our methodology 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 duplication while maintaining an equivalent level of security. This approach ensures enhanced fault tolerance while minimizing the associated resource demands.

This project opened some potential to investigate the topic further and to improve the current state of the art. One of the big questions to examine is the possibility of skipping multiple consecutive instructions. Different combinations of techniques and fault injection methods with various parameters can be explored to physically attack hardware that can lead to multiple instruction skips. With the advancements in hardware attacks, our approach would gain even more importance. The fault injection testing can be accelerated if the communication to the tested device is minimal. For this, the controlling logic can be moved to a specific device, which would control the target device, load the code and inject faults. The open-source platforms would facilitate such developments due to their ability to modify open hardware designs. The current approach could also be extended to various fault models. The new fault models would then be implemented in the debug module, and some additional hardware extensions would need to be done. One of the limitations of our approach is that developers need to configure our fault injection framework what functions to test. So, an interesting direction to look into would be an automatic way to detect critical functions in the code.

Another part of our solution that requires developer intervention is timeout handling. To help developers tackle this problem, efficient tracing needed to be implemented, as well as some automatic tooling that helps developers understand if the timeout needs to be investigated further. A way to improve our code hardening tool would be to implement it as a compiler extension. Overall, our approach to improve fault tolerance of the code showed its efficiency and can help developers secure critical parts of the embedded software.

Chapter 3 Task 4.2

3.1 Task Description

Task 4.2 Advanced Security Testing of Mixed Source Firmware Programs (M06-M33; Task Lead: NXP)

A complementary approach to hardware support for security testing, is a compiler support to enable symbolic execution by instrumenting the code during compilation (SymCC). However, another challenge is that although the hardware will be open, the software may not be fully open-source (i.e., a proprietary Wi-Fi or Bluetooth software stack). In this task, we propose to develop a toolchain based on Inception, to lift closed-source software to a more abstract representation so that SymCC could provide support for testing firmware programs mixing different level of semantics (e.g., assembly mixed with C/C++). This compiler extension could rely on hardware support developed in T4.1.

3.2 Concept Proposal

Early experiments with Inception and SymCC have shown that the lifting of binaries, while it allows to lift closed-source, comes with several drawbacks:

- Considerable engineering effort to precisely lift the instruction set of the RISC-V architecture.
- Considerable engineering effort of rehosting the firmware because of the re-modelling of the required peripherals. This task is in particular a reoccurring one for each firmware which poses an additional drawback of this approach.
- Inception and SymCC are incompatible with the tools developed in T4.1 which does not allow to leverage them for native system-level security testing on FPGAs.

In contrast to Inception and SymCC, the hybrid testing approach involving test input mutation with fuzzing library and smart input generation using symbolic engine has provided much more promising results with less engineering effort to implement and maintain the tools. We discuss these points in more detail in the following sections.

Feedback-guided Fuzzing

Feedback-guided fuzzing (also called graybox fuzzing) has emerged in recent years as an effective technique for automatically detecting bugs and security vulnerabilities in software. Modern fuzzers such as LibFuzzer [35] and AFL++ [34] are used regularly by security engineers in many organizations. For example, the OSS-Fuzz [39] project regularly fuzzes hundreds of open-source applications using thousands of CPU cores.

Figure 7 depicts the general architecture of a feedback-directed fuzzer. The fuzzer consists of four key functions:

- Input generation: the fuzzer needs to continuously feed the software-under test (SUT) with effective inputs to explore new code paths. Inputs could be generated in several ways. For example, byte-level mutations of previous inputs have proved to be fast and reasonably effective. However, detecting deeper bugs would most likely require structure-aware input generation.
- Feedback evaluation: A graybox fuzzer depends on evaluating a feedback signal to guide it towards new and interesting code paths. Upon finding a new path, the fuzzer would queue the current input to be used later for input generation. AFL has pioneered using edge-level coverage as a feedback signal. This feedback signal is fast to evaluate, generic, yet effective

in practice. Other fuzzers like LibFuzzer have built upon similar techniques but added more sophisticated feedback signals like comparison tracing.

- Bug detection: the ultimate goal of a fuzzer is to detect security-relevant bugs like out-of-bound memory accesses and undefined behaviors. To this end, fuzzers usually rely on compiler-instrumentation to insert code that can check bug conditions at run-time. The popular compilers `gcc` and `clang` provide special compiler flags to enable various bug sanitizers like AddressSanitizer (ASan) [37]. Sanitizers not only enable early detection of bugs, but they also provide detailed backtraces that are essential for root cause analysis.
- Scheduling: a fuzzer might discover many interesting inputs during a fuzz session. It will gradually accumulate them in a seed queue. Expectedly, a scheduling problem would arise here as the fuzzer needs to prioritize its seeds and select the next input to be mutated. This scheduling problem has been studied in several works [40]. However, the expected gain from improved scheduling is not high. It can be in the order of 2% higher coverage for a sophisticated algorithm like in FSE [40]. Therefore, we do not consider scheduling further in our project.

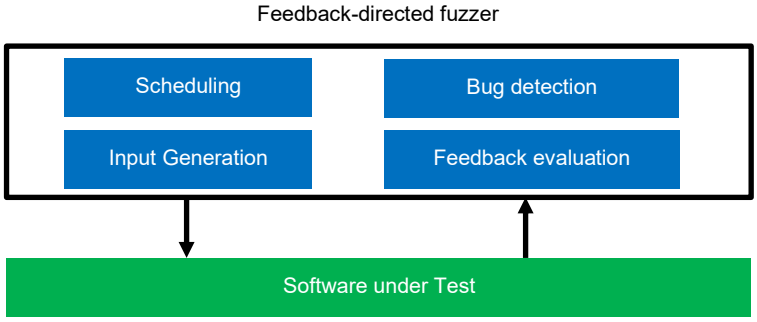


Figure 7 General architecture of a feedback directed fuzzer

Concolic execution

Concolic execution, a blend of concrete and symbolic execution, is a software testing technique designed to enhance the thoroughness of program analysis. By combining the real-world execution of a program with specific inputs (concrete execution) and the theoretical analysis of the program using symbolic inputs (symbolic execution), concolic execution aims to explore various execution paths within the software.

The approach begins with concrete execution, where the program runs with actual inputs, allowing testers to observe real behavior and outcomes. Simultaneously, symbolic execution uses symbolic inputs to represent a range of possible values, enabling the exploration of multiple paths at once. As the program executes, constraints for each path are generated and solved to produce new inputs that drive the program down unexplored paths. This systematic exploration helps in identifying bugs and vulnerabilities that might be missed by traditional testing methods.

Hybrid testing approach

Fuzzing, while effective in identifying crashes and simple bugs, faces several challenges such as limited code coverage, high resource consumption, handling complex inputs, and missing subtle issues like logic errors. Concolic execution addresses these limitations by combining concrete and symbolic execution. This hybrid approach systematically explores different execution paths, leading to higher code coverage. It leverages symbolic execution to understand and to explore the internal logic of the program, making it easier to reproduce and to analyze results. By generating meaningful inputs that explore new paths, concolic execution optimizes resource usage and reduces redundant tests. It can handle complex input formats by generating constraints for symbolic variables, which

are then solved to produce valid inputs that can lead to a new execution path. Concolic execution precisely identifies the conditions under which bugs occur, including subtle logic errors and vulnerabilities that might be missed by fuzzing alone.

Firmware Testing

We consider in this project the testing of firmware, which is the software directly interfacing with the hardware. Several challenges arise in such settings compared to regular application software. First, the computing resources available for executing firmware are quite limited. This means that it is typically not possible to instrument firmware for bug detection and feedback evaluation. For example, ASan requires at least double the amount of memory for its shadow memory implementation. Similarly, running sophisticated input generation algorithms requires more computing resources than what is typically available on microcontrollers. Moreover, microcontrollers usually have a diverse set of peripherals and hardware modules that provide input to the firmware. Such diverse inputs are difficult to accurately emulate by software emulator alone.

The scientific community has approached the above challenges by following three main approaches that sometimes overlap:

- **Recompilation**: In this approach, the source code of the firmware is instrumented and recompiled using a mainstream compiler. For example, one could compile the firmware using clang while enabling ASan. Then, it could be tested using AFL++ on a standard x86-64 machine. This approach is fraught with perils though. It is likely that the firmware cannot be recompiled in the first place due to an incompatibility in compiler options or inline assembly. Even after recompilation, the security engineer will probably need to stub various hardware dependencies before being able of running the firmware. The trouble won't be over yet as the end binary can be significantly different from the original one. This could cause the fuzzer to produce a large number of false positives (and false negatives). As a consequence, recompilation could still be worth pursuing, but only for library code or firmware components with low hardware dependencies.
- **Rehosting**: it is possible to test firmware by running it inside a virtual execution environment (VEE) using a full-system emulator like QEMU [38]. This setup requires an initial investment in developing accurate hardware models for microcontroller peripherals. Consequently, the amount of work required for setting up a VEE varies significantly depending on the target hardware. In rehosting-based testing, the binary typically runs without instrumentation. Instead, the VEE can be hooked to implement dynamic feedback evaluation and bug detection. There has been a surging interest recently in rehosting and fast development of VEEs like HALucinator [41] and others.
- **Native execution**: testing the firmware directly on the microcontroller provides several benefits. First, it avoids the need for developing custom hardware models. That is, testing can commence directly on the target. Additionally, it provides high fidelity that could be necessary to detect timing issues and race conditions. Such fidelity is difficult to achieve in VEEs. However, native testing comes with its own challenges. It is difficult to instrument firmware on the target hardware. Therefore, bug detection and feedback evaluation should be executed on an external host that communicates with the target. Additionally, debuggability is often limited which makes it challenging to implement root cause analysis and crash detection.

3.3 Task Objectives

The aim of this task is to enable hardware support for logical vulnerability SW testing by instrumenting code during compile- or runtime. The code instrumentation can accelerate SW testing methodologies like feedback-guided fuzzing by efficient analysis and reporting of the code being tested (code coverage, vulnerability detection). It will extend the FIM developed in the T4.1 to provide features for tracing capabilities required for software security testing via hybrid fuzzing approach where fuzzing is combined with concolic execution to produce more coverage. This extended version of the FIM is called SW Testing Acceleration Module (STAM) throughout this chapter (see Figure 8).

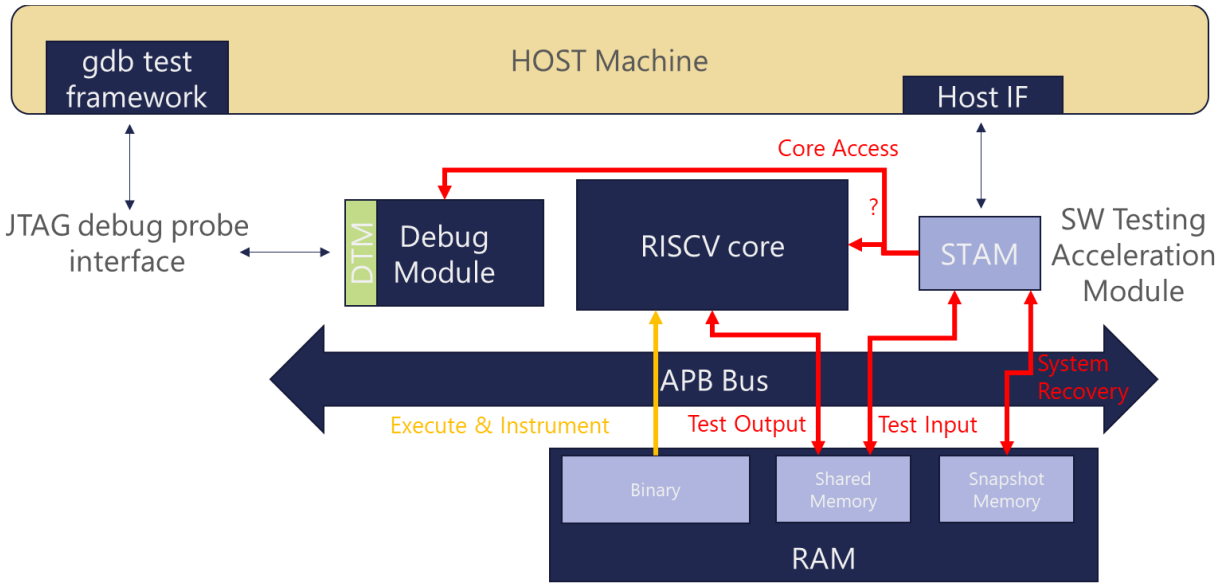


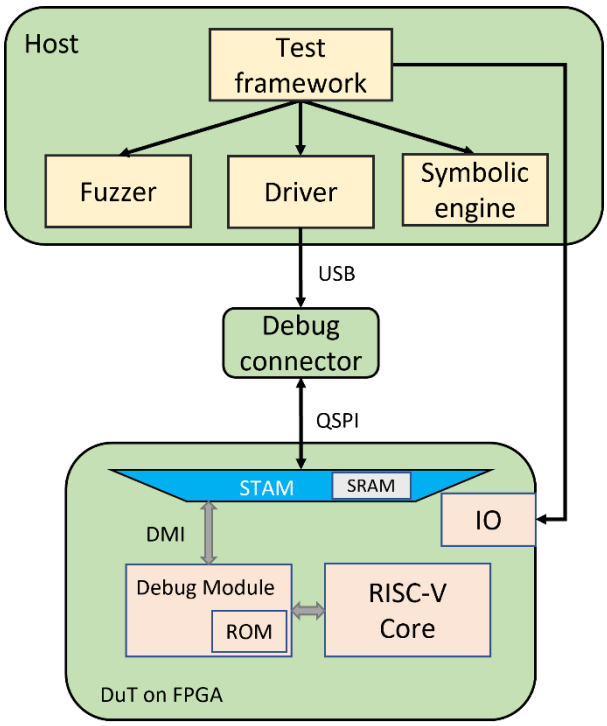
Figure 8: Design for native system-level software security testing

3.4 Solution Design

Given all the arguments from above we focus on developing a hardware module STAM to extend an existing debug infrastructure which will assist in our hybrid testing approach. One of the main problems in firmware testing is the limited observability. Tracing can significantly enhance coverage-guided fuzzing and concolic execution in firmware testing scenarios where targets don't provide proper feedback into program execution flow.

For coverage-guided fuzzing, tracing helps efficiently track code coverage by identifying which parts of the firmware are exercised during fuzzing. This ensures that the fuzzer focuses on unexplored paths, improving the overall effectiveness of the fuzzing process. Additionally, leveraging on-target tracing minimizes the overhead associated with traditional instrumentation, allowing for faster execution of test cases and more efficient fuzzing cycles. Tracing also provides detailed insights into the execution flow, enabling the identification of subtle bugs that might be missed by other testing methods, which is particularly useful for detecting security vulnerabilities.

In the context of concolic execution, tracing facilitates the generation of new test cases based on the observed execution paths, systematically covering different code branches and improving test coverage. By providing real-time data on the execution flow, tracing helps in efficiently solving constraints during symbolic execution, accelerating the process of finding feasible paths and potential bugs.



As one can see lightweight tracing capabilities are essential for automated firmware testing. So we took our setup from T4.1 and modified our target SoC to facilitate instruction flow tracking using our high-speed debugger. Furthermore we incorporated a fuzzing library and symbolic engine in our test framework. The combination of these techniques yields us a necessary foundation for automated testing of firmware on an emulated SoC in pre-silicon phase.

The high-level overview of the solution for automated testing of embedded RISC-V software is presented in the Figure 9. The hardware setup is basically the same as in T4.1. It involves 3 main hardware pieces: a host computer which drives the testing, our debug connector which transmits debug commands at high-speed and the DUT emulated on FPGA. Now we provide the functional description of the components in the figure:

- **Test framework** is the software on the host that controls the testing process. It uses fuzzer and symbolic engine for input generation and debug driver to control and trace the execution of the firmware. The inputs are supplied to the target via serial communication channel.
- **Fuzzer** generates a series of bytes through mutating based on input corpus and coverage provided by the framework.
- **Driver library** is an interface for controlling the target FPGA via the debug connector. It communicates debug commands via USB serial interface which allows high throughput and enables high-speed debugging.
- **Symbolic engine** analyzes the tested binary and symbolically executes it based on executed basic blocks. The engine is also capable of building and resolving constraints for new paths from which an input for an unvisited branch can be derived.
- **Debug connector** is a microcontroller which converts debug commands taken from serial USB into QSPI signals that FIM can process.
- **STAM** (former FIM) relays debug commands from the host to the debug module and handles commands related to our custom DMI registers. It features its own portion of SRAM for storing the additional code that handles actions which are executed once a breakpoint is encountered.
- **Debug Module (DM)** receives DMI commands and controls the core accordingly. The DM is execution-based. This means that when the core enters debug mode, the code in the DM's ROM, referred to as the "park loop", is executed. This loop is supposed to wait for and execute debug operations while the core is halted.

3.5 STAM

In order to gain more observability into the firmware execution we propose a modification of the DM in form of extended register set. The STAM is designed to provide a flexible way of inspecting or modifying system state on target at preset breakpoints. The general idea of our approach is to set breakpoints (either hardware or software via ebreak instruction) and when hit, handle them automatically on target using user provided code. For this, we integrate SRAM for instrumentation

code and another communication channel for transmitting relevant data to the host via a specific DM register.

The official DMI specification allows for Custom Debug Module Registers (0x70-0x7f). In order to enable efficient testing we propose to introduce the following custom registers:

Name	Description	Permissions	Memory-mapped
STAM_CSR	bit0: enable user redirect (R/W) bit1: trace buffer full (R) bit2: unused bit3: trace buffer empty (R) bit31~26: a double exponential mapping from bit position to buffer size (R)	R/W	Yes
TRACE_READ	Returns the next data word (4 byte) in the trace ring buffer	W	No
TRACE_LOAD	Indicates the amount of words in the trace buffer	R	Yes
BS_START	Pointer to the start of the memory region within STAM SRAM used to store data	R/W	Yes
BS_END	Pointer to the end of the memory region within STAM SRAM used to store data	R/W	Yes
BS_VALUE	On write uses binary search algorithm as hardware accelerator to search for a given value within specified range of the memory. The data in memory should be laid out in ascending order.	R/W	Yes
BS_RESULT_ADDR	0x0 if search failed. Otherwise memory address of the requested value.	R	Yes
BS_RESULT_IND	-1: data not found -2: search start address outside SRAM -3: search end address outside SRAM -4: start address bigger than end address =>0: index of the found word	R	Yes
FI_COUNT	Contains the number of times the target instruction must be executed before the fault injection is triggered. The DM should decrease the FI_COUNT value by 1 every time the FI_LOC is encountered. Finally, if FI_COUNT value is 0, the fault is injected and the hardware breakpoint at FI_LOC is removed.	R/W	Yes

Name	Description	Permissions	Memory-mapped
FI_LOC	Contains the address of the target instruction where a fault is to be injected during a single test	R/W	Yes
FI_VALUE	Contains the address that dictates the program counter's advancement when the address specified in FI_LOC is encountered at least FI_COUNT times. This value determines the shift in the program counter upon meeting the specified conditions.	R/W	Yes

Table 1: Control and Status Registers of Software Testing Automation Module

Some of the above registers should be implemented as memory-mapped such that user provided assembly code could interact with hardware.

3.6 Tracing Mechanism

When the core is in a debug mode, because it was halted by a debugger or a breakpoint is hit, STAM checks if redirect is enabled. In this case it points the core to execute the tracing code in STAM SRAM which was written there by the debugger on the host. The tracing code may inspect or change the system state and eventually communicate traced information to the outside by writing to a specific memory location within SRAM which populates the trace buffer. The contents of the buffer can be read out asynchronously by the debugger via TRACE_READ register.

There are two modes implemented for tracing, (1) the single instruction trace mode and (2) the branch trace mode. Both modes will be described in detail in the following two subsections.

Single Instruction Tracing Mode

The single instruction trace mode allows to trace the code execution by single-stepping through the code. The tracing can be activated by placing a simple code gadget into STAM memory and instructing the core to step through executed code. The single tracing steps are depicted in Figure 10 and the following description refers to each step in the diagram.

- Step 1: The STAM is in control of the core until the tracing is disabled by the host. It steps through the code and with every issued step-command.
- Step 2: On each step the core enters debug mode and the STAM code is executed which stores the context. The trace code collects execution state information like the current program counter (PC) in the trace buffer and eventually restores the context and exits the debug mode.
- Step 3: The core continues the execution of the next instruction which will in turn start the same process again. If the trace code detects that tracing is inactive or the buffer is full, STAM would wait for the RESUME command from the host or until the buffer got emptied.

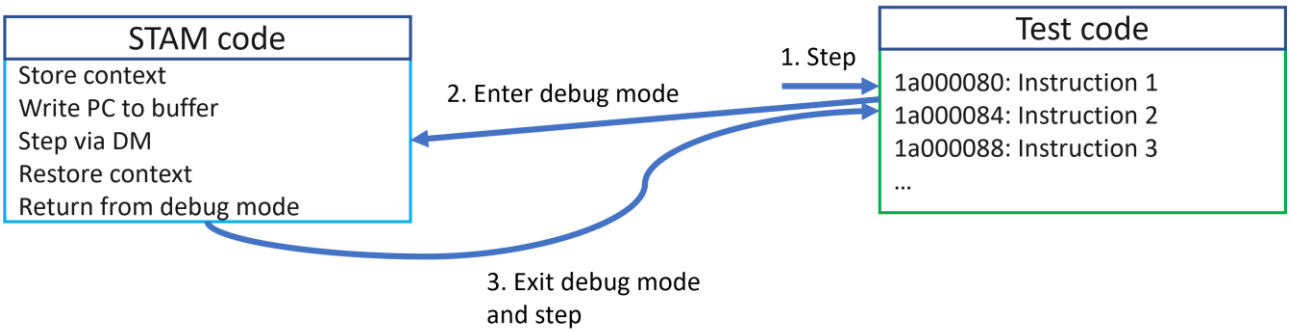


Figure 10: Single Instruction Tracing

Branch Tracing Mode

The Branch Tracing Mode is a lightweight, high-performance tracing technique designed to capture control flow transitions at runtime with minimal overhead. Unlike the Single Instruction Trace Mode, which steps through every instruction, this mode focuses exclusively on control flow instructions, making it particularly well-suited for edge-based feedback-guided fuzzing. In this context, detecting new transitions between basic blocks is essential for steering input mutations and exploring deeper execution paths.

Branch tracing operates by halting the processor at each indirect jump instruction and recording the jump destination. To enable this, each indirect jump in the firmware binary is replaced with a software breakpoint (e.g., an ebreak instruction). Since the original instruction is no longer present, a corresponding tracing gadget is generated for each replaced instruction. These gadgets simulate the original behavior, compute the new program counter (PC), and write the result to the trace buffer. A jump table is also generated, mapping each breakpoint address to its corresponding gadget. This enables efficient redirection during runtime.

In Figure 11, the required steps for the branch tracing concept are depicted.

- Step 1: The debugger loads the instrumented binary, tracing gadgets, and jump table into the target system. It then sets the redirect bit in the STAM control and status register (CSR) and starts program execution.
- Step 2: When an instrumented indirect jump is encountered, the core hits the inserted breakpoint and enters debug mode. STAM detects this event, saves the current execution context, and uses a hardware-accelerated binary search to locate the appropriate gadget in the jump table based on the breakpoint address.
- Step 3: Control is transferred to the selected gadget, which simulates the original instruction, computes the jump target, and writes the destination address to the trace buffer. Meanwhile, the debugger asynchronously reads the buffer contents.
- Step 4: After the trace data is recorded, STAM restores the saved context and exits debug mode. Execution resumes at the newly computed PC, continuing the program flow.

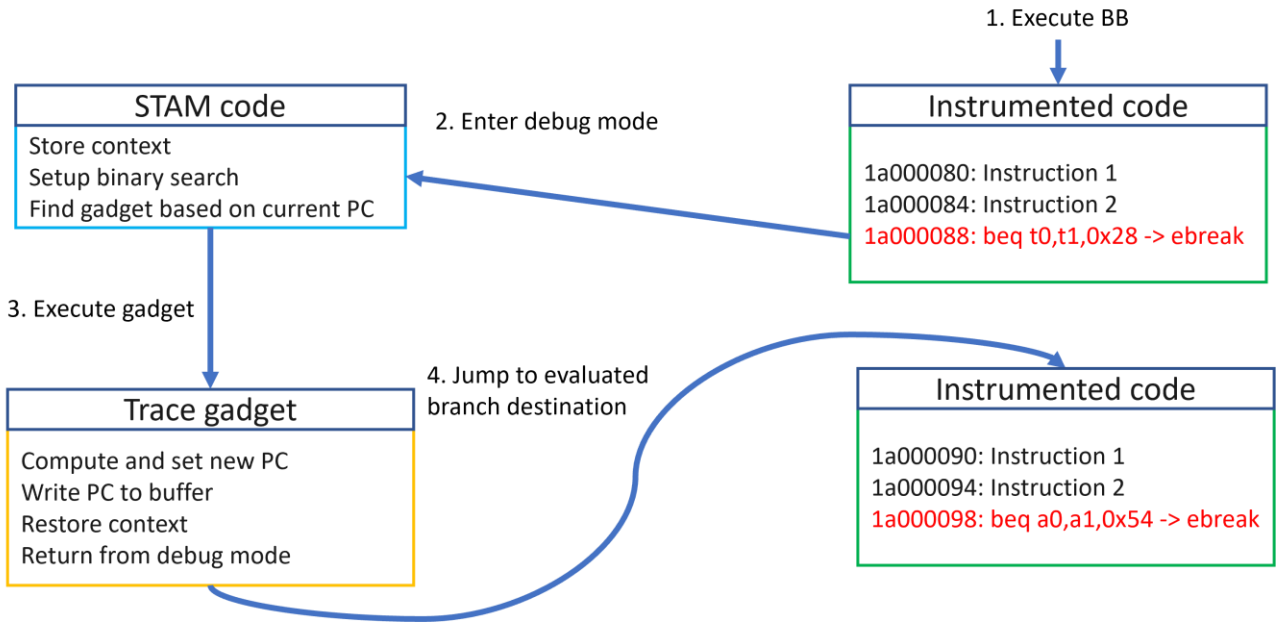


Figure 11: Branch Tracing Concept

3.7 Hardware Acceleration

In real-world firmware, the number of instrumented indirect jump instructions can reach into the thousands. Efficiently mapping each triggered breakpoint to its corresponding tracing gadget is therefore critical for maintaining high execution throughput during branch tracing. To address this, we implemented a hardware-accelerated binary search mechanism within STAM.

A naïve linear search through the jump table would introduce significant latency, especially as the number of breakpoints increases. To mitigate this, we leveraged a binary search algorithm implemented in hardware, which significantly reduces lookup time and enables near-constant-time gadget resolution.

The hardware-accelerated binary search operates over a sorted jump table stored in memory. The process is configured and executed as follows:

- Jump Table Preparation: All breakpoint addresses are sorted in ascending order and stored contiguously in memory. Each entry corresponds to a tracing gadget.
- Configuration via Memory-Mapped Registers: The gadget specifies the memory range of the jump table using the following memory-mapped control registers:
  - BS_START: Start address of the jump table.
  - BS_END: End address of the jump table.
  - BS_VALUE: The address of the current breakpoint to be resolved.
- Search Execution and Result Retrieval: Once configured, the hardware performs a binary search and writes the result to the BS_RESULT_IND register, which contains the index of the matching entry in the jump table.
- Gadget Address Calculation: Since all tracing gadgets are of uniform size and laid out sequentially in memory, the address of the target gadget can be computed as:
  Gadget Address=GADGET_BASE+(BS_RESULT_IND×GADGET_SIZE)

  where GADGET_BASE is the start address of the gadget region.

To ensure correctness and performance, the following layout constraints must be met: the jump table must be sorted by breakpoint address and all gadgets must be of equal size and placed contiguously

in memory. This design enables fast and deterministic gadget resolution, which is essential for maintaining the responsiveness of the tracing system under high-frequency breakpoint events.

3.8 Hybrid fuzzing

Fuzzing has proven to be an effective technique for testing software and uncovering vulnerabilities, primarily due to its ability to generate large volumes of mutated inputs and achieve high test throughput. However, traditional fuzzing - especially in its graybox form - has limited insight into the internal logic of the software under test. It typically relies on coverage feedback, which can be insufficient when execution is gated by complex conditions, such as checks for magic values or deep nested branches. In such cases, the fuzzer may spend a significant amount of time generating inputs that fail to make meaningful progress.

To overcome this limitation, we adopt a hybrid fuzzing approach that combines fuzzing with concolic execution. This integration allows us to intelligently guide input generation when the fuzzer becomes stuck and fails to discover new coverage. Concolic execution, while powerful, is computationally expensive and not suitable for continuous use. Therefore, we invoke it selectively, only when the fuzzer plateaus.

Our branch tracing mechanism plays a key role in this process. When the fuzzer fails to make progress, we extract a concrete execution trace and feed it into the symbolic engine. This trace provides a precise path through the firmware, enabling the symbolic engine to efficiently compute path constraints. Using this information, the engine can suggest new inputs that are likely to steer execution toward unexplored branches.

To support this, we capture an initial snapshot of the system state, including all general-purpose registers (GPRs) and readable memory regions, excluding peripheral-mapped areas. This snapshot is used to initialize the symbolic engine before each symbolic execution run. The user is responsible for marking relevant memory locations or registers as symbolic, which defines the input space for constraint solving.

During symbolic execution, we randomly select an unvisited branch from the trace and instruct the engine to generate an input that would cause the program to take that path. Once a candidate input is computed, it is delivered to the firmware via a serial interface. If the execution of this input results in the invocation of an exception handler, we classify it as a crash.

This hybrid approach allows us to combine the speed and scalability of fuzzing with the precision of symbolic reasoning on a concrete execution, significantly improving the depth and efficiency of firmware testing.

3.9 Implementation

The prototype implementation of our system builds upon the platform developed in Task 4.1. We extended the existing infrastructure by integrating the Software Testing Acceleration Module (STAM), which introduces new DM registers to support the custom features described in previous sections.

To support trace data collection and gadget execution, we connected 128 KB of SRAM to STAM. This memory is fully addressable and accessible by the processor core, and is used for accessing the trace buffer, tracing routines, and instrumentation gadgets. We also modified the existing debug module to interoperate with STAM. Specifically, the DM ROM code was extended to check the

redirect bit and the status of the trace buffer. If redirection is enabled and the buffer is not full, execution is redirected to the STAM SRAM region.

On the software side, we developed a Python-based driver library that enables the Teensy microcontroller to act as a debug connector. This library supports both single instruction and branch tracing modes, and provides a flexible interface for interacting with the STAM hardware.

To handle branching and indirect control flow instructions such as jalr and jr, we implemented a set of tracing gadgets that simulate the original instruction behavior and record the computed jump targets. These gadgets are used in the binary instrumentation process to enable branch tracing.

For broader applicability and to facilitate adoption by the research community, we integrated Teensy support into OpenOCD[42], allowing our enhanced debug infrastructure to be used with standard tooling.

Our hybrid fuzzing methodology is implemented using *libFuzzer* for input mutation and *angr* [36] for symbolic execution of binaries. These tools are orchestrated to work in tandem with the STAM tracing infrastructure, enabling efficient feedback-driven testing of firmware in a pre-silicon environment.

3.10 Evaluation

3.10.1 Tracing

To evaluate the performance of our tracing strategies and demonstrate the effectiveness of our hardware modifications, we conducted a benchmarking campaign using a firmware application that performs SHA hashing across multiple rounds. Since standard tools like OpenOCD do not support tracing of indirect jumps, we use instruction coverage per second (i/s) as our primary performance metric. This metric reflects the number of application instructions executed per second while tracing is active.

Our evaluation begins with a baseline measurement using standard debugging toolchain including OpenOCD in combination with Digilent HS-2 as debug connector, which lacks native support for tracing and especially for indirect jumps. Using basic step functionality of the debugger, tracing performance was limited to approximately 60 i/s. This serves as a reference point for comparing the impact of our enhancements.

We then tested a stepping-based approach using our custom high-speed debugger, which communicates via a Teensy-based debug connector over a QSPI interface. Without any additional acceleration features, this setup achieved a tracing speed of 11,000 i/s, representing a 180× improvement over OpenOCD.

Building on this, we enabled Single Instruction Tracing Mode using a stepping gadget placed in STAM SRAM. This configuration significantly boosted performance to 83,000 i/s, a 7.5× increase over the previous setup and a 1,300× improvement over the OpenOCD baseline.

Next, we evaluated Branch Tracing Mode, where each indirect jump is instrumented with a dedicated gadget. Initially, we implemented a software-based binary search in assembly to map breakpoints to their corresponding gadgets. This approach yielded a tracing speed of approximately 380,000 i/s, which is 4.5× faster than single-instruction gadget tracing and 6,300× faster than OpenOCD.

To further optimize performance, we replaced the software binary search with a hardware-accelerated binary search engine, controlled via memory-mapped registers. This significantly reduced the number of instructions executed per breakpoint and offloaded the search logic from the core to dedicated hardware. As a result, we achieved a peak tracing performance of 700,000 i/s,

which is 85% faster than the software binary search and an impressive 11,700× improvement over the OpenOCD baseline.

It is important to note that in our test case, all tracing gadgets and the jump table fit within the available STAM SRAM. For larger firmware binaries, this may not be feasible, and dynamic reloading of gadgets and jump tables would be required, potentially reducing performance due to the overhead of memory transfers.

Figure 12 illustrates the performance of each tracing strategy and highlights the incremental benefits of our acceleration techniques. These results demonstrate that our approach - combining a high-speed debug interface with memory-resident tracing gadgets and hardware-assisted search - offers substantial performance gains. Depending on the available FPGA resources and SoC design, analysts can choose from a range of tracing configurations to balance performance and resource usage. These tracing capabilities form the foundation for the hybrid fuzzing methodology described in the next section.

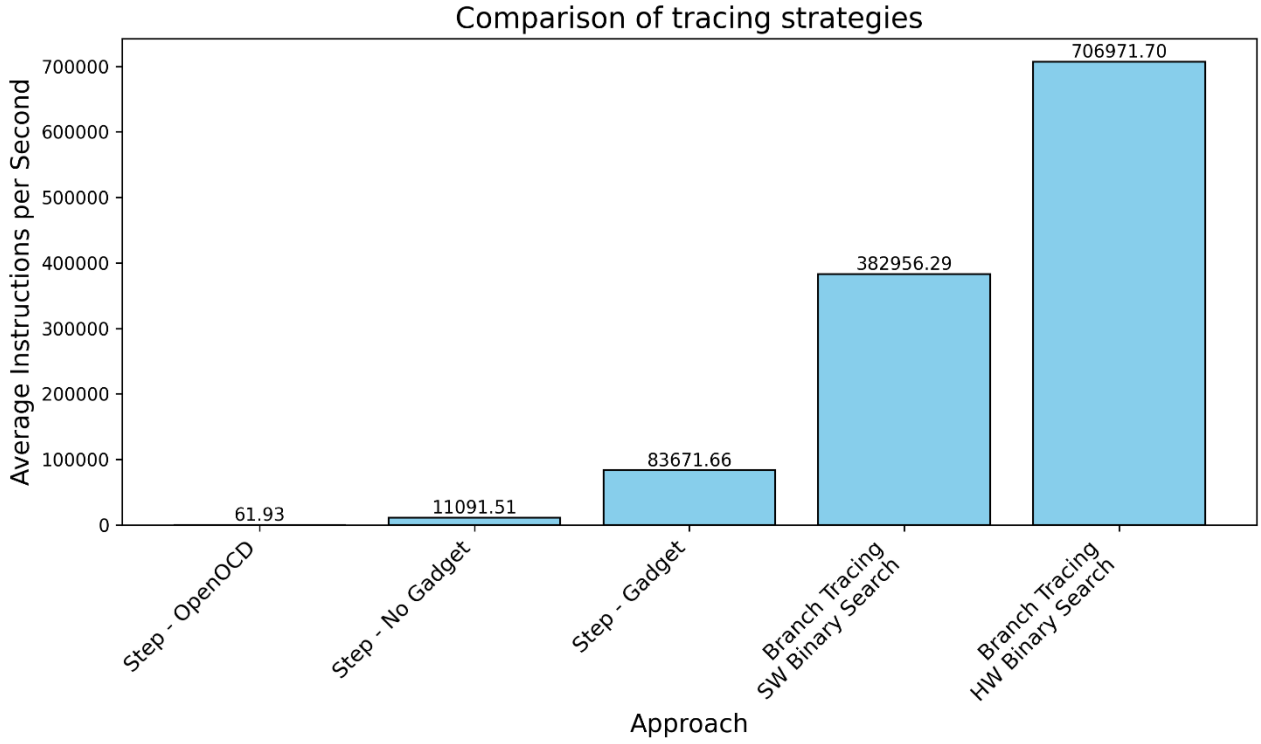


Figure 12: Performance of different tracing strategies

3.10.2 Hybrid fuzzing

To evaluate the effectiveness of our hybrid fuzzing approach, we developed a custom test program that communicates over UART using a lightweight protocol. The firmware was instrumented with artificial vulnerabilities, such as buffer overflows, which trigger a crash and invoke the exception handler when exploited. This setup allowed us to assess the ability of our system to discover real faults through guided input generation.

We compared our approach against GDBFuzz[33], a state-of-the-art graybox fuzzer for embedded systems that relies on hardware breakpoints to construct a basic block coverage map. While GDBFuzz operates at the basic block level, our system tracks edge coverage in form of sequence

of taken branches, which provides finer-grained feedback and enables more precise guidance for input mutation.

To ensure a fair comparison, we evaluated both tools over the same firmware and runtime conditions. For each input generated by GDBFuzz that resulted in new basic block discovery, we re-executed the firmware using our tracing infrastructure to compute the corresponding edge coverage. This allowed us to normalize the results and compare the tools based on the same metric.

As shown in Figure 13 Edge coverage comparison, our hybrid fuzzer significantly outperforms GDBFuzz in both coverage depth and discovery speed. Within the first minute of execution, our tool covered 55 unique edges, whereas GDBFuzz required approximately 40 minutes to reach the same level. Over the full 10-hour campaign, GDBFuzz failed to match the total edge coverage achieved by our hybrid approach.

These results highlight the advantage of combining instruction-level tracing with symbolic execution. By leveraging concrete execution traces and selectively invoking the symbolic engine, our system is able to overcome input-dependent bottlenecks - such as magic number checks - and explore deeper execution paths more efficiently. This leads to faster discovery of vulnerabilities and more comprehensive firmware testing within the same time constraints. Ultimately, our enhancements to the debug module and tracing infrastructure enable a powerful pre-silicon testing methodology that surpasses traditional graybox fuzzing in both precision and performance.

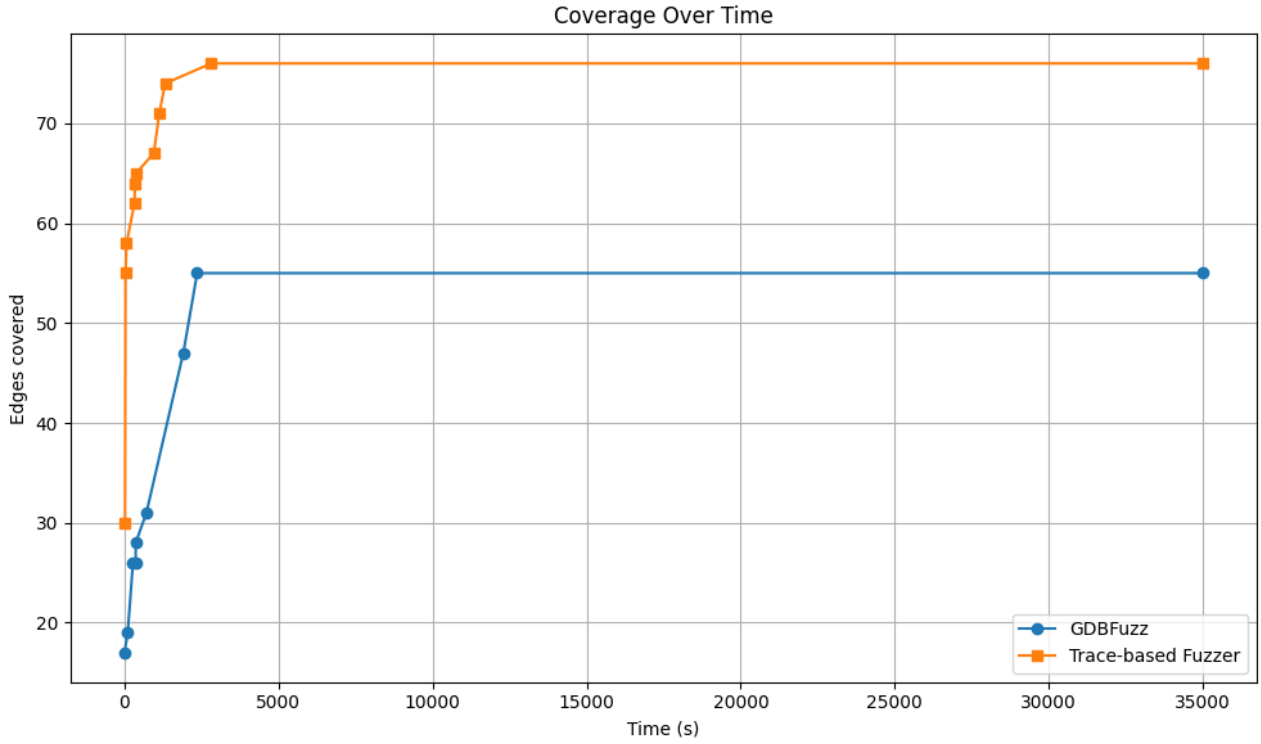


Figure 13 Edge coverage comparison

3.11 Conclusion and Continuation of T4.2

In this task, we presented a novel approach to accelerating automated firmware testing in pre-silicon environments by combining hardware-assisted tracing using an on-chip debug module with hybrid fuzzing techniques. Our system, built around the Software Testing Acceleration Module (STAM), introduces a set of custom debug features that enable efficient instruction flow tracing through both single-instruction and branch-level modes. These tracing capabilities are tightly integrated with a

high-speed debug interface and supported by a hardware-accelerated binary search engine, significantly improving trace resolution performance.

We demonstrated that our tracing infrastructure achieves up to 700,000 instructions per second - an improvement of over 11,000× compared to conventional OpenOCD-based debugging. This performance gain is critical for enabling scalable and responsive feedback mechanisms in fuzzing workflows.

To further enhance test coverage, we implemented a hybrid fuzzing strategy that combines fast input mutation with targeted symbolic execution. By leveraging concrete execution traces, our system can guide the symbolic engine to explore previously unreachable paths, resulting in faster and deeper coverage. In comparative evaluations, our hybrid fuzzer outperformed state-of-the-art graybox fuzzers such as GDBFuzz, achieving significantly higher edge coverage in a fraction of the time.

There are multiple ways to improve the efficiency of hybrid fuzzer as future work. Silent data corruption detection is beneficial for discovering more memory related vulnerabilities. So the techniques like memory tagging or some sort of memory access control in hardware or software can be implemented. One promising direction is the automation of symbolic variable selection to reduce manual effort and improve scalability across diverse firmware targets. Another direction in improving concolic execution is a development of more sophisticated logic for path exploration and interrupt modelling. Finally, exploring adaptive fuzzing strategies that dynamically balance between fuzzing and symbolic execution based on runtime feedback could lead to even more efficient path exploration and vulnerability discovery.

Overall, our approach demonstrates that integrating hardware acceleration with intelligent software testing strategies can dramatically improve the efficiency and effectiveness of firmware validation. The modularity and openness of our implementation, including integration with OpenOCD and support for standard tools like libFuzzer and angr, make it a practical and extensible solution for the research and embedded systems communities. These contributions pave the way for more robust and scalable pre-silicon testing methodologies capable of uncovering complex bugs and vulnerabilities early in the development lifecycle of open-source hardware based on RISC-V.

Chapter 4 Summary and Conclusion

This deliverable D4.1 documents the successful implementation of Task 4.1 and 4.2 in Work Package 4 within the ORSHIN project, focusing on pre-silicon security testing of embedded firmware. The work is structured into two tasks: T4.1, which addresses physical fault injection testing, and T4.2, which targets logical vulnerability testing through hybrid fuzzing and symbolic execution.

In T4.1, we developed a novel hardware-assisted framework for fault injection testing on RISC-V-based systems. This includes the design and implementation of the Fault Injection Module, a high-speed debug interface integrated into an FPGA-emulated SoC. The framework supports debugger-driven FI testing, enhanced by custom debug module extensions that simulate instruction skip attacks. A code hardening approach was also introduced, enabling automatic identification and patching of vulnerable instructions using duplication-based countermeasures tailored to the RISC-V instruction set.

In T4.2, we extended the FIM into the Software Testing Acceleration Module (STAM), enabling advanced software testing techniques. This includes hardware-assisted instruction and branch tracing, a high-speed debug interface, and a hardware-accelerated binary search engine for efficient gadget resolution. These capabilities support a hybrid fuzzing framework that combines input mutation with symbolic execution, significantly improving code coverage and vulnerability detection. Our evaluations demonstrated substantial performance gains and superior coverage compared to existing tools like GDBFuzz.

We publicly provide our prototype and a detailed reproduction guide as part of the D4.2 demonstrator. The prototype includes a system-on-chip bitfile with an enhanced debug module for FPGA emulation, Teensy firmware for custom communication, a high-speed debugging driver, and a fault injection framework. It supports both a fast debugger and standard tools like GDB and OpenOCD, including a modified OpenOCD for our Teensy connector. Also included are a hybrid fuzzing tool using STAM for firmware testing and example test code for our platform.

Together, these contributions represent a significant advancement in the field of pre-silicon firmware security testing. The integration of hardware acceleration with intelligent software testing strategies has proven to be highly effective in identifying and mitigating vulnerabilities in embedded systems. The open-source nature of the tools ensures broad accessibility and encourages adoption by the wider research and development community.

Chapter 5 List of Abbreviations

Abbreviation	Translation
FIM	Fault injection Module
CPU	Central Processing Unit
JTAG	Joined Test Action Group
FI	Fault injection
SW	Software
HW	Hardware
IF	Interface
BKPT	Breakpoint
STAM	Software Testing Acceleration Module
DM	Debug Module
DMI	Debug Module Interface
QSPI	Quad Serial Peripheral Interface
CSR	Control and Status Register
GPR	General Purpose Register
RTL	Register Transfer Level
FPGA	Field Programmable Gate Array
SoC	System on Chip
SuT	Software under Test
DuT	Design under Test
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver Transmitter
ROM	Read-Only Memory
SHA	Secure Hash Algorithm

Chapter 6 Bibliography

[1] OpenHW – CV32E40P Manual - https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/intro.html

[2] PJRC – Teensy 4.1 Development Board - https://www.pjrc.com/store/teensy41.html

[3] RISC-V External Debug Support Version 0.13.2 - https://riscv.org/wp-content/uploads/2019/03/riscv-debug-release.pdf

[4] 38. Ziade, H., Ayoubi, R., Velazco, R.: A survey on fault injection techniques (2004)

[5] 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

[6] 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).

[7] 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

[8] 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

[9] 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

[10] Sas, 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

[11] 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

[12] 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

[13] 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 , https://link.springer.com/10.1007/978-3-030-97348-3 , series Title: Lecture Notes in Computer Science

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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, http://link.springer.com/10.1007/978-3-030-35055-0 , series Title: Lecture Notes in Computer Science

[19] 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

[20] Elmohr, M.A.: Embedded systems security: On EM fault injection on RISC-v and BR/TBR PUF design on FPGA

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] Portela-Garc´ıa, M., L´opez-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

[29] MOSDORF, M., SOSNOWSKI, J.: Fault injection in embedded systems using gnu debugger (2011)

[30] 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

[31] 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

[32] 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

[33] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. 2023. Fuzzing Embedded Systems using Debug Interfaces. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 1031–1042. https://doi.org/10.1145/3597926.3598115

[34] Michal Zalewski. American fuzzy lop (afl). https://github.com/google/AFL, 2014

[35] Kostya Serebryany et al. libfuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2015

[36] F. Wang and Y. Shoshitaishvili, "Angr - The Next Generation of Binary Analysis," 2017 IEEE Cybersecurity Development (SecDev), Cambridge, MA, USA, 2017, pp. 8-9, doi: 10.1109/SecDev.2017.14.

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12). USENIX Association, USA, 28.

[38] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05). USENIX Association, USA, 41.

[39] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. Vancouver, BC, August 2017. USENIX Association.

[40] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: an information theoretic perspective. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020). Association for Computing Machinery, New York, NY, USA, 678–689. https://doi.org/10.1145/3368089.3409748

[41] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In 29th USENIX Security Symposium (USENIX Security 20), pages 1201–1218. USENIX Association, August 2020.

[42] Open On-Chip Debugger: https://openocd.org/