D4.2

Prototype of automatic security pre-silicon SW 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	Demonstrator
Deliverable reference number	CL3-2021-CS-01 / D4.2 / 1.0
Work package contributing to the deliverable	WP4
Due date	Jun 2025 – M33
Actual submission date	4th July 2025

Responsible organisation	ECM
Editor	Volodymyr Bezsmertnyi, Aurélien Hernandez, Aurélien Francillon
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. This deliverable presents the prototypes results.
Keywords	Security Testing, Fault injection, fuzzing, RISC-V, Debugger, Logical vulnerabilities, Firmware rehosting, Hardware-in-the-loop

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 (NXP)
EURECOM (ECM)

Contributors (ordered according to beneficiary numbers)
Volodymyr Bezsmertnyi (NXP)
Aurélien Hernandez (ECM)
Aurélien Francillon (ECM)

Reviewers
Olivier Thomas (TXP)
Clarisse Ginet (TXP)

Executive Summary

Deliverable D4.2 of the ORSHIN project presents three innovative prototypes for automatic pre-silicon software security testing. These prototypes aim to enhance the resilience of embedded and IoT systems through open-source hardware and software solutions.

- **Fault Injection Demonstrator**: Utilizes FPGA-based emulation to accelerate fault injection analysis on RISC-V systems. A custom Fault Injection Module (FIM) and high-speed debugger enable automated instruction skipping and vulnerability detection, significantly improving testing throughput and precision.
- **Hybrid Fuzzing Demonstrator**: Combines fuzzing with concolic execution to overcome the limitations of traditional fuzzing. A Software Testing Acceleration Module (STAM) enables high-speed branch tracing and symbolic input generation, achieving superior coverage and vulnerability discovery rates.
- **Firmware Rehosting Platform (DMON)**: Enables secure and efficient testing of embedded firmware by rehosting it on a more capable system while maintaining hardware interaction fidelity. Built on the Zynq-7000 MPSoC, DMON supports real-time tracing and dynamic patching, facilitating advanced analysis of low-level firmware behavior.

These prototypes demonstrate significant advancements in pre-silicon security testing, leveraging open-source architectures like RISC-V. The work contributes valuable tools and methodologies for future secure system development and auditing.

Table of Content

List of Figures

List of Tables

Chapter 1 Introduction

The ORSHIN project, funded by the European Union under grant agreement no. 101070008 focuses on enhancing the security evaluation and resilience of IoT devices and embedded systems. In recent years, the advent of open and license-free processor architectures, namely RISC-V, presented both new opportunities and challenges to build more scalable security testing techniques. Leveraging from such open-source hardware platforms, the Work Package 4 (WP4) of the ORSHIN project focused on pre-silicon security testing to address, before market release, vulnerabilities occurring at both physical and logical levels. This focus is crucial due to the broad adoption of IoT devices in the various commercial and industrial sectors, often used for critical applications. Due to the tight relationship between hardware and software on such devices, post-silicon testing techniques cannot fully detect and address all potential bugs and vulnerabilities once the silicon tape-out is achieved, shifting priority towards pre-silicon counterparts.

Pre-silicon security testing is usually backed by virtual environments, where the internal logic design and behavior of a microcontroller to be manufactured (processor, hardware accelerators) is modelled and tested. Such virtual environments typically run a mix of CPU instruction set emulation and RTL logic simulation, enabling extensive testing without effectively manufacturing the device. Although offering the convenience of a virtual platform with high behavioral inspection capability, such approaches are however computationally complex (time-consuming) and potentially prone to inaccuracy and human errors, diverging from the actual physical device behavior. As such, WP4 has responded to these challenges by developing hardware-backed, open-source frameworks for both firmware fault injection and logic bug testing, enabling comprehensive pre-silicon evaluation for a broad range of well-exploited vulnerabilities.

This work resulted in three publicly available prototypes leveraging FPGA designs based on open-source hardware and software components. The present deliverable describes the technical details and requirements to reproduce each prototype. As such, **Chapter 2** introduces two demonstrators for hardware accelerated Fault Injection Emulation platform via a custom Fault Injection Module (FIM) and a logical hybrid Fuzzing platform backed by a high-speed firmware tracing hardware module. **Chapter 3** introduces a third prototype of an open-source firmware rehosting platform, facilitating logical testing and security analysis in mixed scenarios.

Chapter 2 Fault Injection and Hybrid testing

This chapter introduces two key demonstrators developed under Work Package 4, task 4.1 and 4.2 of the ORSHIN project, aimed at advancing pre-silicon software security testing. The first demonstrator focuses on fault injection, leveraging a custom hardware-accelerated platform to emulate and analyze fault scenarios in embedded firmware. The second demonstrator explores hybrid fuzzing, combining traditional fuzzing techniques with symbolic execution to enhance vulnerability discovery in complex software logic.

Both approaches are implemented on an open-source RISC-V-based FPGA platform and are designed to improve testing speed, coverage, and automation. The chapter begins by detailing the common hardware and software setup required for both demonstrators, followed by in-depth descriptions of each testing methodology and its implementation.

2.1 Common platform setup

This section provides instructions for setting up the demonstrator for pre-silicon testing. It is based on CORE-V-MCU project and features Fault Injection Module (FIM) integrated in the system-on-chip (SoC). It can be used to inspect and modify system state while the core is halted and in debug mode. This component is especially useful to perform fault injection analysis at high-speed on FPGA to develop pre-silicon countermeasures or to perform software testing and dynamic binary analysis. The references to all necessary Git repositories and binary components can be found in the following GitHub repository.

Note: the final code and binaries are in a pending internal release process, so the content of the following GitHub repository can be updated.

2.1.1 Hardware overview

The basis for the demonstrator is a CORE-V-MCU. This is possible to follow their Quick Start Guide to setup the hardware and provide help with software components. We provide the bitstream files which can be programmed onto Xilinx Genesys2 via Vivado. Please refer to following GitHub repository for further instructions.

It features the open-source 4-stage, in-order 32-bit RISC-V core CV32E40s (which we integrate instead of original CV32E40p), operating at a core frequency of 10 MHz. We customized the debug module of the emulated design according to our needs for a faster instruction skip, owing to our access to the source code of the emulated SoC. Highlighting the advantages of the RISC-V architecture's open-source nature, we developed a custom debugger.

This debugger communicates Debug Module Interface (DMI) commands over a custom QSPI protocol. Two debugging processes operate on the host: the conventional OpenOCD and our custom debugger. OpenOCD communicates with the HS-2 Debugger via USB, and HS-2 utilizes JTAG transport layer to interact with the SoC's debug module. Simultaneously, our custom debugger communicates with a Teensy 4.1 Arduino microcontroller. The Teensy is programmed to function as a QSPI master, receiving DMI commands from the host over USB and transmitting them to the SoC using a software-based QSPI implementation. Essentially, we supplemented the SoC with a hardware multiplexer responsible for managing JTAG and QSPI slave communication. The FIM (aka STAM according to D4.1) translates debugging commands into general Debug Module Interface commands, adhering to the RISC-V Debug Specification.

In this particular design, DM implementation operates on an execution-based principle. This means that when the core enters debug mode, the code in the DM's ROM, referred to as the "park loop", is executed. We extended the DM registers and ROM code in line with our custom extension's specifications to allow automated program counter modification upon breakpoint. This fully automates the process of emulating an instruction skip on the chip without instrumenting the code being executed. To enable high-speed debugging and accommodate the use of two different debugger implementations, we adjusted the SoC's pinout, introducing 10 additional pins to integrate the QSPI interface.

2.1.2 FPGAs and Debugger

The demonstrator runs on Xilinx Genesys2 FPGA board and can be programmed via Vivado using USB-JTAG interface with our bitstream file. For emulating fault injections using the regular OpenOCD+GDB method, the setup also requires Digilent HS2 Debugger connected to the boards.

2.1.3 FIM and CORE-V-MCU modifications

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.

QSPI Interface

The communication between the host and the DUT is based on a QSPI protocol; only mode 0 is supported. Teensy serves as a QSPI master, while the FIM is the slave. The FIM receives DMI commands (read/write DM register) through 10 QSPI pins and routes them to the DM. The DM responds with a 32-bit value in case of a read operation and FIM outputs via MISO pins.

The FIM receives commands from the user via a QSPI interface. The commands are routed either towards the SoC's debug module (DM), through an adapter which can generate either APB or DMI transactions, or towards the internal extending component. Two asynchronous FIFOs take care of the clock-domain crossing between SPI clock and system clock.

PIN	SIGNAL
JC1	sclk
JC2	scl
JC7	FIM sel
JD1	MISO0
JD2	MISO1
JD3	MISO2
JD4	MISO3
JD7	MOSI0
JD8	MOSI1
JD9	MOSI2

Table 1: FIM QSPI Pinout

Clock cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	Addr[6:3]	Addr[2:0] & 1' b0	dummy	dummy	dummy	dummy	dummy	dummy	dummy	dummy	dummy	dummy	dummy	dummy	Data[31:28]	Data[27:24]	Data[23:20]	Data[19:16]	Data[15:12]	Data[11:8]	Data[7:4]	Data[3:0]

Figure 1: Read transaction

Clock cycle	0	1	2	3	4	5	6	7	8	9
	Addr[6:3]	Addr[2:0] & 1' b1	Data[31:28]	Data[27:24]	Data[23:20]	Data[19:16]	Data[15:12]	Data[11:8]	Data[7:4]	Data[3:0]

Figure 2: Write transaction

2.1.4 *Teensy Microcontroller*

We provide a library for Teensy which is able to communicate with the FIM through QSPI interface. Use flying wires or custom PCB to connect Teensy pins to FPGA. To compile and flash firmware via Arduino Studio follow these steps:

- Connect Teensy via USB cable to computer.
- Create a new Sketch in Arduino Studio.
- Install Arduino Add-on for Teensy.
- Select Teensy board by going to Boards Manager.
- Configure it to run at 528 MHz by clicking Tools → CPU Speed → 528MHz.
- Add horizon_orshin_teensy_fw library by going Sketch → Include Library → Add ZIP Library. Please refer to the following GitHub repository for further instructions for library download.
- The Teensy Platform does not allow building with precompiled libs for some reason. Hence, we need to add this line *compiler.libraries.ldflags=* to the platform description file *C: \Users\<username>\AppData\Local\Arduino15\packages\teensy\hardware\avr \1.59.0\platform.txt*
- Use the following snippet to use the library:

```
#include "Arduino.h"
#include "horizon_orshin_teensy_fw_lib.h"

// Define serial number for Teensy remote connection
#define STR_SERIAL_NUMBER   L"MY_PROGRAM_V1"

// Set this to the hardware serial port you wish to use
#define HWSERIAL Serial1

unsigned int proxy_readBytes(char* buffer, unsigned int size) {
  return Serial.readBytes(buffer, (size_t) size);
}

unsigned int proxy_write(const uint8_t* buffer, unsigned int size) {
  return Serial.write(buffer, (size_t) size);
}

unsigned int proxy_available() {
  return Serial.available();
}

void proxy_interrupts() {
  interrupts();
}

void proxy_noInterrupts() {
  noInterrupts();
}

ext_funcs_t funcs = {
  proxy_readBytes,
  proxy_write,
  proxy_available,
  proxy_noInterrupts,
  proxy_interrupts
};

void setup() {
  Serial.begin(20000000);
  Serial.print("Start QSPI:\n");
  horizon_orshin_setup(&funcs, 4);
}

void loop(){
  horizon_orshin_loop();
}
```

- Now you can flash the firmware by clicking “Upload”.

Refer to Teensy Firmware Description section regarding implementation details of the firmware.

PIN	SIGNAL
14	MOSI2
15	MOSI3
16	scl
17	sclk
18	MOSI1
19	MOSI0
20	MISO2
21	MISO3
22	MISO0
23	MISO1

Table 2: QSPI Pinout of Teensy

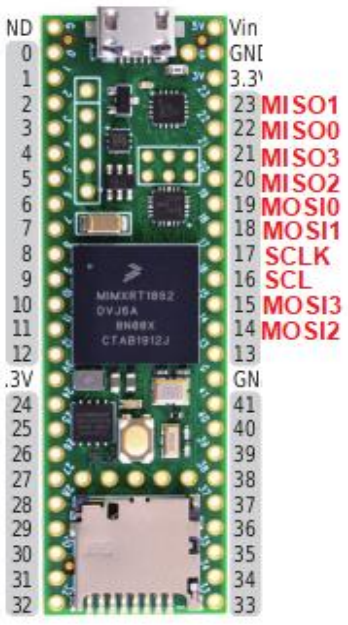

Figure 3: Pins locations on the Teensy Board

2.1.5 Teensy Firmware Description

Teensy firmware implements Length Value protocol for Teensy side.

A packet consists of 2 parts:

- `len` (2 bytes) → Number of commands to execute
- `5*len` → the array of 5-byte commands.

For each packet, it returns an array of 32-bit QSPI responses for each command.

Teensy expects 5 bytes per command. 1st byte specifies the register to be addressed. LSB of 1st byte should be set if write to register is intended, hence the register value should be always shifted by 1 to left as LSB is reserved for R/W indication. Next 4 bytes contain the value to be written. Value bytes will be ignored in case of a read operation. Teensy returns the requested register value (4 bytes) in case of a read operation or 0 in case of a write operation.

As an example, we take commands to activate DM (write `1` to register “DM Control” `0x10`) and read DM status (read from register "DM Status" `0x11`). Teensy should receive a sequence of 2 commands:

`00 02 21 00 00 00 01 22 00 00 00 00` where:

- `00 02` indicates that 2 commands are sent
- `21` is `0x10` shifted by 1 to left and LSB is set to indicate write operation
- `00 00 00 01` is value that we write to "DM Control"
- `22 00 00 00 00` instructs DM to read "DM Status" register `0x11` (which is `0x22` after shift by 1 to left) and `00 00 00 00` are dummy bytes that will be ignored by FIM since it is a read operation

And returned response is something along: `00 00 00 00 00 03 0C 82` where:

- `00 00 00 00` is the result of the write operation, so no data is returned.
- `00 03 0C 82` corresponds to the result of the read operation, so "DM Status" register value is returned.

Internally, Teensy is supposed to implement QSPI in software and send 5 bytes (1 byte for register and 4 data bytes) including some dummy bytes (2 in current implementation) over its MOSI pins. Then it reads 4 bytes of response data and returns to the host via serial interface.

2.1.6 Python Teensy Driver

Clone `rv_qspi_dbg` repository into `rv_qspi_dbg` folder. Go to `rv_qspi_dbg` folder and clone `rv_qspi_dbg_templates` repository into templates folder. Please refer to the following GitHub repository to locate necessary Git repositories.

Main script (`teensy_dbg.py`) connects to Teensy over serial connection. It provides python interface and ability to apply proper signals for debugging operation (e.g., halt, resume, read/write memory, set breakpoints) according to the RISC-V debug specification 0.13.

To initialize the driver in python, run:

```
dbg = RvTeensyDebugger('com15',memChunkSize=5000, timeout=5)
```

Where `com15` is the serial port under which Teensy is reachable and `memChunkSize=5000` is a variable parameter which specifies the command throughput (number of commands that can be supplied at once to FPGA). This parameter varies depending on the host CPU and/or I/O handling capabilities as well as on type of connection (flying wires, interposer, PCB). The greater `memChunkSize`, the more prone the setup is to faulty transmission. To test if current `memChunkSize` is sufficient, run `writeSpeed, readSpeed = dbg.memTest(0x1c000800,100000)` and observe the fault ratio in the terminal output. In addition, the Teensy CPU speed can be reduced by reloading firmware via Arduino if faults are persistent.

2.2 Fault Injection Demonstrator

The demonstrator is based on the work that was conducted for a scientific paper.

For more details regarding FI testing methodology and implementation refer to article Duplication-Based Fault Tolerance for RISC-V Embedded Software

Fault injection is a method to evaluate the effects of faults in a system, in order to validate the effectiveness of countermeasures and mitigation techniques and to detect new weaknesses.

Fault injection analysis can be performed using software-based simulation techniques, which can offer a very detailed view of the insight of the system but are extremely slow. FPGA-based emulation techniques, like the one proposed here, offer a less detailed view of the behavior of the system (which is anyway not always necessary) but can achieve much higher execution speed and can therefore be used to stress many more scenarios.

2.2.1 Hardware

Setup the platform as described in section Common platform setup.

2.2.2 Software

In other repositories you will find all necessary software components to run an FI campaign on a provided test code.

2.2.3 Python FI Scripts

Please refer to the following GitHub repository to locate necessary Git repository with code for fault injection demo. In `fault_injection_testing` there are 2 python scripts that conduct FI testing on the test binary: `fi_runner.py` uses standard OpenOCD+GDB debugging toolchain and `teensy_fi.py` uses custom high-speed debugger and DMI extension.

To run `fi_runner` you need to:

- Download and install in PATH `riscv32-corev-elf-*` the gcc toolchain
- Setup and start OpenOCD version 0.11.0-rc2 with config file for HS2

To run `teensy_fi` you need to:

- Run `pip install -r requirements.txt`
- Have Teensy connected and firmware uploaded

Both scripts perform FI campaign of the test binary while using different methods of injecting faults. Scripts parse output of objdump and compile a list of instruction addresses that need to be skipped. Every instruction in that list is skipped and success of an attacker is determined. Finally, it outputs the list of vulnerable instructions.

`fi_runner` starts gdb instance and sends commands to it. Within a single run it:

- Resets the core.
- Loads the binary in the memory.
- Sets breakpoints at exception handler, at the end of main function and at FI location.

- Resumes the core.
- Skips an instruction at FI location.
- Evaluates success of FI based on encountered breakpoints.

`teensy_fi` does basically the same, except that it uses Teensy for debug operations and configures special DM registers for on-target FI.

2.2.4 Test binary

To run a FI campaign, you need a test binary, which needs to be compiled from sources provided in the `firmware_core_v` repository in the `demo1` branch. Please refer to following GitHub repository to locate necessary Git repositories. You need the `make` tool and `riscv32-corev-elf-*` toolchain. To compile, go to `Default` folder and run `make sec_boot`. You will find the compiled `sec_boot` ELF binary in the `Default` folder.

`sec_boot` implements a simple routine that verifies integrity of the memory by calculating hash and comparing it with a ground truth value. If the hashes do not match, an exception is raised. Otherwise, it outputs an AES test encryption vector over UART. A fault injection attack is considered successful if the target was faulted and the encryption vector has been extracted.

2.3 Hybrid Fuzzing Demonstrator

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 to 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 is 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.

2.3.1 Hardware

Setup the platform as described in section Common platform setup.

2.3.2 Software

In other repositories you will find all necessary software components to run hybrid fuzzer on a provided test code.

2.3.3 Test Binary

To run a fuzzing campaign, you need a test binary, which needs to be compiled from sources provided in the `firmware_core_v` repository in the `demo2` branch. Please refer to the following GitHub repository to locate necessary Git repository. You need the `make` tool and `riscv32-corev-elf-*` toolchain. To compile, go to `Default` folder and run `make concolic_test`. You will find the compiled `concolic_test` ELF binary in the `Default` folder.

`concolic_test` implements 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.

2.3.4 Hybrid fuzzer

Once you have the hardware setup, you can clone `hybrid_fuzzer` repository. Please refer to the following GitHub repository to locate necessary Git repository. To run the tool, you need to:

- Make sure that our python module `rv_qspi_dbg` is accessible from the environment.
- Run `pip install -r requirements.txt`
- Place compiled test binary in the `firmware` folder
- Adjust configuration file. There is an example configuration with explanations under `./configs/corev_config.yaml`
- Run `python ./src/main.py` to execute tool with `./configs/corev_config.yaml` configuration.

Chapter 3 Firmware rehosting on Open-Source hardware

The overall ORSHIN-funded project has the ambition to create secure design methodologies, implementation, and testing approaches of secure hardware available in the open-source community. In this respect, a notable observation can be made on the significant prevalence of computing systems relying on constrained embedded devices whose software is strictly tailored for their physical capabilities to meet cost and efficiency requirements. Those embedded systems are particularly difficult to test in pre-production scenarios due to their inherent reduced physical interfaces and internal debugging capabilities. This observation raises some level of security and safety concerns for end users, companies, and manufacturers, which remains an open and active research topic. This third demonstrator aims to show the feasibility and provide guidance on building better open-source testing and debugging tooling for embedded systems via firmware rehosting.

Firmware rehosting refers to the process of extracting a program and executing it in an environment different from the original hardware on which it was designed to run. This serves the main purpose of offering a more capable environment to inspect and instrument a given program for reverse engineering and testing purposes. In the context of embedded systems with reduced debugging capabilities, DMON aims to provide an open source rehosting platform that eases firmware testing and security analysis by enabling real-time firmware behavior inspection and memory tracing. Such capabilities are particularly useful in detecting and understanding the origin of various known software issues, such as memory corruption.

Performing rehosting is a challenging task as it requires recreating an environment with enough fidelity for the rehosted firmware to run in a manner analogous to its original environment. The difficulty lies in the fact that recreating the hardware interactions of the original environment is typically hard to achieve due to the inherent complexity and diversity of physical devices. This difficulty generally varies depending on the level of abstraction inherent to the system being studied. Higher levels of abstraction decorrelate the software execution from its interactions with the underlying physical peripherals. As such, embedded systems firmware based on operating systems offers a clear distinction between high-level software and the underlying hardware interactions (e.g., drivers), ensuring portability over various physical devices.

However, firmware found in constrained or critical embedded systems is usually tightly coupled to their physical environment to meet firmware size, costs, and real-time requirements. Due to those engineering constraints, embedded systems running bare-metal firmware hardly offer any layer of abstraction and, thus, are particularly hard to rehost in a viable and reproducible manner. The proposed concept of the present demonstrator asserts this issue by providing an Open-Source rehosting platform where firmware can be conveniently executed and tested on a more capable computing system while ensuring environment consistency through a hardware-in-the-loop approach. Hardware-in-the-loop keeps the original target coupled with the rehosting environment, providing hardware access to the software. However, approaches found in the literature hardly achieve proper timing performance of interactions and are, thus, hardly applicable to real-world, complex devices. The proposed architecture in DMON is based on an open-source logic design that is tightly coupled with a rehosting processor, which significantly improves timing consistency.

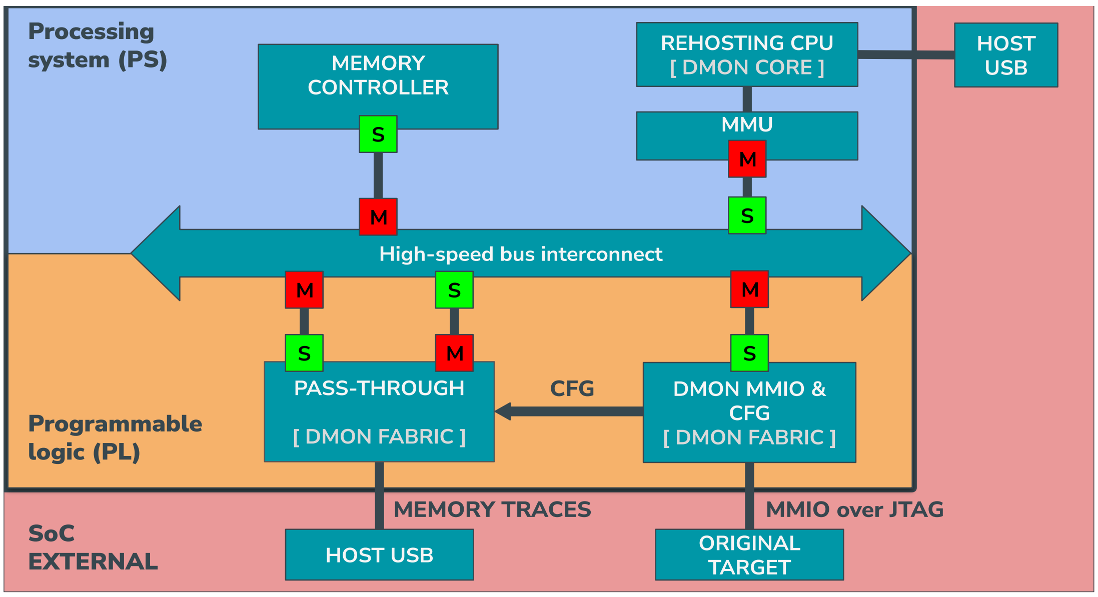

Figure 4: Architecture overview of DMON

3.1 Architecture overview

The core concept of DMON lies on integrated circuits featuring a combination of a Processing System (PS) and a Programmable Logic (PL) (Figure 3). Those components are usually referred to as “Mixed FPGA SoC” or “MPSoC” (Multi-Purpose System-on-Chip). The PS contains all the necessary components of a traditional System on Chip, which are a processor, internal peripherals, and memory controllers. The PS exhibits an execution speed of a higher order of magnitude than the rehosted embedded systems, which offers better performance during firmware bug testing and analysis. On the other hand, the PL embeds a user-programmable FPGA, along with the necessary interconnection with the PS. The PL ensures synchronization with the original target in a hardware-in-the-loop scenario while providing real-time firmware tracing for in-depth analysis.

DMON CORE

The DMON CORE resides in the Processing System of the platform and refers to the high-performance CPU running the rehosted firmware. At first, the DMON CORE runs a minimal code that is responsible for DMON platform initialization and self-testing. It contains a minimal bootloader able to retrieve a target firmware from an internal flash memory or via a host computer (via USB). On loading, the rehosted firmware is natively executed by the DMON CORE. Some specific architectural mechanisms present on the target embedded systems might not be present on the DMON platform. Hence, the DMON CORE contains minimal emulation code for those mechanisms, which generally handles specific CPU context switching and exception mechanisms. The emulation is lightweight and, thus, doesn’t imply a significant performance loss.

MMU (Memory Management Unit)

During ongoing firmware execution, DMON extensively relies on the MMU component of the rehosting CPU to redirect firmware accesses to memory-mapped peripherals. Since the peripherals of the original target are likely not implemented on the rehosting platform, those accesses are either redirected to handlers in the DMON CORE implementing those missing peripherals (emulation), or to the DMON FABRIC which interacts with the original target in a hardware-in-the-loop fashion.

Thanks to the MMU, this redirection is seamless in the sense that the original memory map for which the firmware has been built is kept intact, removing the need of firmware patching and binary rewriting. DMON overshadows relevant memory regions by configuring MMU pages pointing to the DMON FABRIC, instead. Additionally, the MMU can be configured to redirect all access to firmware data and/or code sections to the PASS-THROUGH IP of the DMON FABRIC, providing the tracing and dynamic data/instruction patching capabilities.

Lately, the MMU can be used to manually solve memory-mapping conflicts. A memory mapping conflict occurs when some dereferenced memory regions (addresses) of the target match an address present on the rehosting platform, which would result in inaccurate behavior and, eventually, a crash. Dereferenced memory regions refer to memory locations that are effectively in-use by the re-hosted firmware. Such conflicts are likely to occur on complex targets.

Another case of mapping conflict solved by the use of the MMU is the firmware loading location. To benefit from the execution speed-up during rehosting, the firmware code and data sections must reside in a memory bank internal to the Processing System (PS) of DMON. Since target firmware is not expected to feature position-independent code, section relocation might be required if the mapping doesn't match an available internal memory bank on DMON.

DMON FABRIC

The DMON FABRIC contains two IPs and resides in the Programmable Logic (PL) of the platform. Overall, the DMON FABRIC handles low latency data tracing and interaction with the physical target (hardware-in-the-loop).

DMON FABRIC: Pass-Through

If required for security analysis, the DMON Pass-Through provides high-speed, continuous tracing and patching of all memory operations, including instruction fetching.

In this case, the MMU is further configured to redirect all firmware code fetch and data read/write operations to the DMON Pass-Through instead of the internal memory controller. The Pass-Through IP dumps those memory accesses to the host computer over USB3 for further analysis. The IP ultimately redirects access to the original location in the internal memory of the PS and eventually patches the readout value in the case of a READ operation by the firmware. The patching done by the Pass-Through is what we refer to as a "**dynamic patching**" mechanism, as the patching logic and patch data can be dynamically reconfigured during ongoing firmware execution. This is particularly useful to dynamically instrument certain instructions during firmware analysis.

DMON FABRIC: CFG & MMIO

Accesses to memory-mapped peripherals redirected from the DMON CORE to the DMON FABRIC via the MMU configuration are seamlessly actuated on the attached target device via a physical debug interface. Conversely, interrupts triggered internally on the target are forwarded back to the DMON CORE for the firmware to perform the appropriate operation (interrupt handlers). This FPGA IP maintains tight synchronization with the target over the debugging interface. This IP further receives various parameters provided by the DMON CORE, in order, for instance, to configure the patching logic.

High-speed Bus interconnect

The high-speed interconnect enables the DMON CORE and the DMON FABRIC to communicate at low latency. Communications are native to the CPU via memory-mapped read and write operations. A bus interconnect allows multiple Manager interfaces to initiate memory operations on multiple Subordinate ones. Each Subordinate is accessed via a specified range of addresses.

The DMON Pass-Through IP requires two bus interfaces, one Manager and one Subordinate, in order to actuate memory operations from the DMON CPU back into the memory subsystem (internal to the platform).

The DMON CFG & MMIO comprise a single Subordinate interface on the bus to receive MMIO access and configuration.

Physical target

The physical target whose firmware has been rehosted on the DMON CORE processor is flashed a simple firmware stub, synchronizing with the DMON FABRIC. MMIO operations originating from the rehosted firmware are transparently actuated on the physical target on behalf of this firmware stub. However, the purpose of the stub is to catch interrupts happening on the target (e.g., systick timers, peripherals, external peripherals) and to forward them to the DMON FABRIC. The FABRIC subsequently warns the DMON CORE about the outstanding interrupt so that the appropriate interrupt handler can be executed in the rehosted firmware.

Host USB interfaces

DMON provides two distinct external USB interfaces connected to a host computer. Those interfaces are controlled by two individual host clients, serving different purposes.

Main client (platform initialization and logging)

The main client initializes the DMON platform by uploading the configuration of the MMU (relocation and conflict solving) and the firmware blob to rehost. This client accesses the DMON CORE via a low-speed USB 2.0 CDC (COM serial). After initializing the platform, it can further retrieve logging outputs emitted by the DMON CORE during firmware execution (triggered interrupts, faults) for debugging purposes.

On firmware upload, the main client applies a simple binary patching pass by replacing a few specific instructions with dummy ones. This patch permits the DMON CORE to perform the compatibility emulation used to accommodate potential architectural differences, as described in the **DMON CORE** subsection. We refer to this mechanism as “static patching” contrasting with the “dynamic patching” done in real-time by the DMON Pass-Through (see **DMON FABRIC: Pass-Through** subsection).

Optional client (firmware tracing)

The second client accesses the DMON Pass-Through via a high-speed USB 3.0 interface for real-time tracing collection. Additionally, the host can debug the ongoing DMON firmware execution by using the specific physical debug interface available on the rehosting Mixed-SoC platform. Although useful for debugging DMON itself during the development of the platform, such traditional debug interfaces particularly slow down firmware execution on the DMON CORE. Hence, the real-time tracing provided by DMON should be preferred to preserve the platform's performance.

3.2 Rehosting flow overview

The following describes the general flow during firmware rehosting of a microcontroller-based embedded system using the DMON platform:

1. The original firmware is extracted from the microcontroller of the embedded system (original target) and replaced by the firmware stub.
2. The microcontroller is connected to the DMON fabric debug interface.
3. The host client (tracing) initializes the high-speed tracing.
4. The second host client (config) initializes the target, configures the MMU according to the target microcontroller memory map, and finally uploads the previously extracted firmware blob.

5. The DMON CORE loads the firmware on its internal memory subsystem and starts executing it by branching on the firmware's reset handler.
    - During ongoing firmware execution, architectural mechanisms are automatically and efficiently emulated by lightweight code on the DMON CORE.
    - Memory-mapped peripheral accesses (MMIO) are redirected to the original target via the DMON FABRIC.
    - Interrupts triggered on the original target are forwarded by the stub to the DMON CORE via the FABRIC in order to execute the appropriate firmware's interrupt handler in time.
6. The host client collects the firmware trace for real-time or post-analysis.

3.3 Demonstrator requirements

This section describes the requirements for replicating and testing the current DMON demonstrator. It contains references to open-sourced repositories that are enumerated in Chapter 4.

3.3.1 Rehosting hardware requirements

The demonstrator was developed on the ZedBoard development Board, embedding the Zynq-7000 mixed CPU/FPGA system on chip. The latter employs a dual Cortex-A9 as application class processors, interconnected to an individually programmable FPGA fabric via a high-speed AXI bus interconnect.

In summary:

- ZedBoard development Board
- If debugging the DMON platform itself is required: Digilent JTAG-HS3 USB to JTAG adapter or compatible equivalent (e.g., Digilent HS2).
- Breadboard jumper wires (flywires) or the JTAG adapter & multiplexer (from **JTAGMUX** repo).
- Two USB micro to USB-A cables.

For high-speed instruction and data tracing, additional components are required on the host:

- CYUSB3KIT-003 USB FX3 devkit
- CYUSB3ACC-005 FMC interposer

3.3.2 Rehosted target requirements

The current demonstrator can "rehost" all kind of microcontroller-based embedded systems complying with the four requirements. Target requirements are further summarized in Table 3.

R1. TARGET ISA requirement

The demonstrator mainly targets the ARMv7-M variant of the ARMv7 32-bit microprocessor ISA. Although not compatible with other architecture families such as RISC-V or MIPS, the concept and methodologies introduced by DMON are generic and can be applied to various others given the availability of a mixed FPGA SoC for the chosen architecture.

The implementation of the DMON CORE is particularly tailored for the CPU execution modes, special-purpose registers and the Thumb Instruction Set found in the M (microcontroller) variants. For the rehosting to function properly, the DMON CORE particularly relies on emulating architectural specificity found in ARMv7-M on the Cortex A9 (ARMv7-A). Thus, DMON cannot rehost firmware that are built for application class ARMv7-A and ARMv7-R. However, those architectural variants are particularly rare in embedded ecosystems.

DMON can rehost embedded systems with an ARMv8-M based microcontroller if the complete firmware image has been built for an ARMv7-M. Conversely, due to the inherent backward compatibility of the ARM ISA family, DMON is functional with ARMv6-M.

R2. TARGET CPU IMPLEMENTATION

DMON doesn't rely on any ARM Cortex-specific feature and is thus compatible with any implementation of the ARMv6-M and ARMv7-M ISA.

Our current implementation targets **single core** applications (most typical). DMON is compatible with dual-core scenarios by leveraging from the dual Cortex A9 embedded in the Zynq-7000 SoC, by further implementing synchronization between the two. Each core owns an individual FPGA pass-through block. Multicore scenarios can also be achieved by the use of multiple physical Zedboard platforms with the addition of proper external synchronization logic between the platforms.

R3. TARGET DEBUG INTERFACE requirement

In order to perform external interactions with peripherals (MMIO), DMON must perform memory READ and WRITE operations on the connected physical target in the place of the original firmware. Microprocessors generally offer such capabilities via dedicated physical "debugging" interfaces.

The demonstrator is currently compatible with the **ARM Debug Interface v5** via the standardized **JTAG-DP physical debug port** (JTAG transport). The ARM Debug Interface further defines the MEM-AP (Memory Access Port), which provides access to the internal memory subsystem of the debugged target and is accessible via the physical debug port. As many microcontrollers offer the capability of disabling the debug port in post-production, one must ensure that the debug port and MEM-AP are fully accessible and functional prior rehosting a given target.

Most ARM-based microcontrollers implement the JTAG-DP ARM debug port. Less common variants include the arm SW-AP (SWD – Single Wire Debug) to access the MEM-AP. SWD is another well-known interface targeting more constrained physical devices (reduced external pinout). The demonstrator in its current state doesn't implement it although being fully compatible with it by adapting the DMON FABRIC physical interface to the SWD protocol (`vhd/dmon.vhd` of the **DMON_ZYNQ7000** repo).

R4. TARGET FIRMWARE requirement

The memory layout of rehosted target (via its firmware memory accesses) must not conflict with the layout of the rehosting platform since all firmware instructions are executed without emulation on the DMON CORE (native execution). Conflict may happen if the original firmware tries to access memory regions belonging to the DMON FABRIC (FPGA IPs) or to the UART (universal asynchronous receiver transmitter) controller of the Zynq-7000 SoC as the latter is being used by the DMON configuration client.

	IMPLEMENTED	COMPATIBLE	INCOMPATIBLE
R1. TARGET ISA	ARMv7-M, ARMv6-M	ARMv7E-M	Other microprocessor architecture families
R2. TARGET CPU IMPLEMENTATION	All ARM CORTEX M in single core setups	Dual Core Cortex M	
R3. TARGET DEBUG INTERFACE	Arm Debug Interface v5 via JTAG transport (JTAG-DP)	Arm Debug Interface v5 via SWD transport (SW-DP) Other physical debug interface with target memory access	Undocumented or locked-out debug port
R4. TARGET FIRMWARE	Single Core bare-metal and RTOS firmwares (without secure extension)	Secure extension mode of ARM-v7 (with TrustZone)	

Table 3: Summary of target requirements for the current DMON demonstrator

3.3.3 Host hardware and software requirements

Preparing a new DMON platform (platform bring-up) by generating the DMON image and programming the Zynq-7000 requires the Vivado 2019.1 design suit. The platform has not been tested on other versions. This version of the Vivado design suit requires an X86 or AMD64 computer running the Ubuntu 18.04 GNU/Linux distribution. This older version of Ubuntu can be installed on a virtual machine or a Docker container. We recommend Distrobox or Toolbox tools to conveniently set up a Docker container for a specified GNU/Linux distribution.

The DMON project automates various building and flashing steps of the Vivado design suit via the CMake build system. Both CMake and GNU Make are required.

Flashing the target embedded systems with firmware stubs further requires various specific toolchains and buildsystems. Those requirements are detailed in the subdirectory for the chosen target in the **DMON_TARGETS** repository.

Once the platform bring-up is done on a Zedboard, rehosting various target firmware (rehosting setup) solely requires the Python interpreter for running the DMON configuration client. The DMON client has been tested on versions 3.6.9 (Ubuntu 18.04) and 3.13.2 (latest maintenance release to date).

3.4 Demonstrator evaluation

The “platform bring-up” refers to the process of preparing a new Zedboard dev board for rehosting by flashing the DMON CORE and FABRIC. The platform bring-up procedure is done only once on a new Zedboard devboard. The “rehosting setup” refers to the process of rehosting a target firmware on bring-up completion. The host requirements for both bring-up and rehosting setup are described in the section Host hardware and software requirements.

This contains references to open-sourced repositories that are enumerated in the chapter Source repositories.

3.4.1 Platform bring-up

The bring-up process should be done only once on a given Zedboard, by taking the following steps:

1. Power-on the Zedboard using the supplied jack power supply.

- (If real-time firmware tracing is required)
  1. Connect the CYUSB3KIT-003 FX3 devkit to its FMC interposer and further connect the FX3 to the host computer using the supplied USB Type-B cable.
  2. Clone the DMON_TRACE repository and follow the instructions to build and flash the FX3 firmware and further build the host client.

2. Set the Zynq-7000 boot mode to JTAG by setting the following jumpers on the Zedboard. The central pin must be either connected to `GND (OFF)` or `3V3 (ON)`.

```
JMP11 OFF (GND)
JMP10 OFF (GND)
JMP9  OFF (GND)
JMP8  OFF (GND)
```

3. Clone the **DMON_ZYNQ7000** repository

4. Build the DMON FABRIC FPGA bitstream by executing the following commands from the root directory of the repository:

```
mkdir build
cd build
cmake ..
make vv
```

5. Build the full FSBL (AMD bootloader) with DMON CORE firmware and FPGA bitstream.

```
make fsbl
```

6. Flash the Zynq-7000 SoC.

```
make flash
```

7. Power-off the Zedboard.

8. Set the Zynq-7000 boot mode to QSPI flash (previously flashed).

```
JMP11 OFF (GND)
```

```
JMP10 ON  (GND)
JMP9  OFF (GND)
JMP8  OFF (GND)
```

9. Power-on the board.

3.4.2 Rehosting setup

The rehosting setup process is done on every rehosting session for a given target device and firmware, and is performed as follows:

1. Ensure the whole Zedboard is powered-off at this stage.

2. Connect the `UART` labelled micro-USB connector to the host computer. This would show-up as a CDC device (`/dev/ttyACMx` under GNU/Linux)

3. (If debugging the DMON platform itself is required) Connect the Digilent HS3 or equivalent on the `JTAG` labelled 2.0 mm pitch header

4. The target should be flashed with the proper firmware stub. To target the same microcontroller as in the **DMON_TARGETS** repository, follow the subsequent instruction for firmware building and flashing. If targeting a different microcontroller, use the **DMON_TARGETS** repository as a reference to implement a firmware stub.

5. Power-up the target board and connect its JTAG interface to the JTAG adapter & multiplexer from the JTAGMUX repo to the `JA1` and `JB1` PMOD connectors of Zedboard. The JTAG interface must be unlocked and functional.

   Without the JTAGMUX adapter, one can connect the target to the PMOD using flywires following the pinout.

6. (If the real-time firmware tracing is required) Connect the FX3 to the host computer and connect FMC interposer of the FX3 to the FMC connector of the Zedboard prior the power-up of the later. Proceed to reset the FX3 by pressing the `SW1` labelled button.

7. Power-up the Zedboard via the power switch

8. Configure the reference client `cli.py` in the root directory of **DMON_ZYNQ7000** by setting the appropriate MMU settings (`CLI.mmap()`) and dynamic patching (`CLI.add_relacation()`). Refer to the documentation available in `cli.py` regarding the usage of the `CLI` class.

9. Execute the `cli.py` by providing the path to the target firmware as a program option, such as `./python cli.py ../firmwares/stm32_blink`. The firmware image must be in the ELF standard format in order for the client to detect the segments to be loaded along with the

firmware start address. These details can be set manually in the case of a raw firmware image by modifying the Python client.

10. At this point, the firmware is being executed on the DMON CORE, synchronizing with the original target via the DMON FABRIC.

11. The main client continues to trace various logging outputs from the DMON CORE on the terminal.

12. (If real-time firmware tracing is required) Execute the host client as described in the **DMON_TRACE** repository. Memory addresses and values are traced to `stdout`.

13. (If the DMON CORE debugging is required) Debug the DMON CORE using traditional tools such as GDB via the OpenOCD tool or the Segger JLink client. Ensure that GDB interprets the DMON CORE firmware and the rehosted target firmware with the proper memory map (instruction & data addresses). For more details, refer to the reference GDB script `scripts/gdb.init` in the **DMON_ZYNQ7000** repo.

3.4.3 Running the example firmware

We provide examples for two target microcontrollers:

- The LPC1850 from NXP via the LPC1850-db1 devboard from Diolan
- The STM32F4 from Micro via the NUCLEO-F401RE devboard (same company)

Open-source example firmware and instructions on how to build the replacement stub firmware for each target are available in the **DMON_TARGETS** repo under individual subdirectories. A reference Python host client is available on the main **DMON_ZYNQ7000** repo for each target.

The provided examples include various target firmware implementations such as simple bare-metal peripheral test routines (e.g., led blink) or more complex RTOS-based ones. Those target firmware can be executed natively on the original target or via the DMON platform to ensure the consistency of the rehosting process.

Chapter 4 Source repositories

Repository reference	Repository URI
NXP Deliverable Overview	https://github.com/orshinAtNXP
DMON_ZYNQ7000	https://gitlab.eurecom.fr/dmon/dmon_zynq7000
DMON_TARGETS	https://gitlab.eurecom.fr/dmon/dmon_targets
DMON_TRACE	https://gitlab.eurecom.fr/dmon/dmon_usb3_trace
JTAGMUX	https://github.com/eurecom-s3/JTAGMux

Each repository contains the `orshin` git tag, pointing to the repository state at the time of the deliverable release.

Chapter 5 Summary and Conclusion

The present deliverable D4.2, details three prototypes for automatic security testing in pre-silicon, outlining the hardware and software components, testing methodologies, and setup requirements for reproducibility. It focuses on two primary prototypes resulting from task T4.1 “Report on security audit and testing”. The first one describes a hardware-accelerated Fault Injection Emulation platform via a custom Fault Injection Module (FIM) tightly coupled to a RISC-V processor. The other demonstrator focuses on logical hybrid fuzzing fed by a high-speed tracing module implemented along a RISC-V. Both prototypes are implemented on an FPGA platform. Additionally, the present deliverable introduces DMON, an open source rehosting platform that facilitates firmware testing and security analysis by executing firmware in an environment separate from its original hardware, improving inspection and instrumentation capabilities. The platforms are mostly based on open-source hardware and software components, and the materials resulting from this work are open-sourced.

Overall, D4.2 demonstrates practical advancements for automatic security testing in pre-silicon, thanks to the advent of license-free and open processor architectures, addressing critical needs in the development of resilient hardware and software for IoT and embedded systems. Ecosystems based on open architecture like RISC-V made the implementation and experimentation with such architecture possible for firmware vulnerability testing and fault injection emulation. The resulting prototypes exhibit significant enhancements in the performance and depth of security analysis compared to the state-of-the-art approaches applied to IoT and embedded devices, offering valuable open-source tools for future system development and auditing.