LiteInjector: A LiteX Extension for Fault Injection

Adam Henault, Tanguy Philippe, Vianney Lapôtre

LiteInjector: A LiteX Extension for Fault Injection

Adam Henault
Université Bretagne Sud
UMR 6285, Lab-STICC,
Lorient, France
adam.henault@univ-ubs.fr

Philippe Tanguy
Université Bretagne Sud
UMR 6285, Lab-STICC,
Lorient, France
philippe.tanguy@univ-ubs.fr

Vianney Lapôtre
Université Bretagne Sud
UMR 6285, Lab-STICC,
Lorient, France
vianney.lapotre@univ-ubs.fr

***Abstract*—In this article, we focus on the emulation of fault injections on FPGA boards using saboteurs by presenting LiteInjector. LiteInjector is an open source bit- and cycle-accurate logic fault emulator written in Python, developed with the aim of accelerating security evaluation campaigns for systems-on-chip (SoC). LiteInjector has been tested on several use cases. These use cases rely on VerifyPin codes from the FISSC security benchmark running on a Linux-capable system-on-chip with a RISC-V core. We demonstrate that the tool provides a flexible solution that supports a large set of fault models. Furthermore, results show that, compared to an HDL simulation-based tool, LiteInjector allows reducing the fault injection campaign time by a factor of 210 considering a Linux-based system.**

***Index Terms*—Hardware security, fault injection attacks, emulation, FPGA**

I. Introduction

The widespread use of embedded systems has increased the demand for security guarantees in both software and hardware. In a security context, faults can be intentionally injected with the goal of altering hardware or software behaviour to bypass protection mechanisms [1], [2]. To defend against fault injection attacks (FIA), the development of tools to evaluate a circuit's resilience to fault injections is crucial. Various approaches exist to assess the robustness of hardware designs. For post-silicon evaluation, the most common approach relies on actual fault injection, which uses various techniques such as voltage manipulation [3], clock signal modification [4], electromagnetic pulses [3] or lasers [3], [5] to introduce faults into a chip. This post-silicon approach allows the security of a chip to be evaluated under real-world conditions, enabling the assessment of a target's resistance to physical FIA. It also allows attacks aimed at extracting sensitive information or altering the target's behaviour for malicious purposes. However, this approach requires expensive equipment and a deep understanding of the target. FIA campaigns can also be time-consuming, depending on the technique used and the complexity of the target.

Recently, approaches relying on simulators or emulators have taken precedence since they are cost-effective and require less expertise. Furthermore, such approaches can be used during design time, allowing the designer to perform early security evaluations and correct potential vulnerabilities in both software and hardware descriptions. Nevertheless, pre-silicon approaches are often time-consuming or impractical to conduct on complex architectures using conventional simulation methods.

In this article, we present LiteInjector, an open-source tool allowing to build fault emulation platform on FPGA. It is written in Python with the goal of accelerating security evaluation campaigns for systems-on-chip (SoCs). The tool was specifically designed for seamless integration into SoCs generated using the LiteX framework [6], enabling users to quickly and easily build complete SoCs. By combining LiteInjector with LiteX, users can efficiently create a SoC platform upon which comprehensive security evaluation campaigns can be executed. Similarly to LiteX, LiteInjector is built on the Migen [7] domain-specific language (DSL), which translates hardware descriptions written in Python into hardware description languages (HDLs) such as VHDL or Verilog. Unlike previous approaches, LiteInjector is specifically designed to inject logical faults and does not aim to replicate physical fault injections, as seen in electromagnetic fault injection emulation [8].

The remainder of the paper is structured as follows. Section II introduces related work. Section III details the proposed approach. Section IV presents four case studies that illustrate the features of LiteInjector. Section V discusses the benefits and limits of the proposed approach. Section VI concludes the work and draws some perspectives.

II. Related work

Fault injection evaluation tools are designed to evaluate the impact of fault injection on hardware/software applications. Numerous existing tools are dedicated to cryptographic applications. Although cryptographic primitives are critical, other security mechanisms have to be evaluated against fault injection attacks. In this section, we focus on existing tools that allow the evaluation of various software and hardware applications. We classify existing work into three main groups: Formal-methods-based, simulation-based and circuit emulation-based approaches.

Formal methods provide mathematical proofs to ensure a rigorous verification of the system's behavior during fault injection experiments. For instance, [9] allows the detection of sensitive logic or sequential hardware elements while [10], [11] and [12] present formal verification methods to analyze HDL implementations. However, these approaches usually suffer from restrictions that limit their actual usage on a

complete processor. In particular, the circuit structure it can analyze is usually limited.

A simulation-based approach can be used at both the hardware and software levels using different methods. Fault injection attacks can be simulated at the instruction level. [13] evaluates four open-source fault simulators, comparing their techniques and suggest enhancing them with AI methods inspired by advances in cryptographic fault simulation. One of the most popular tools for fault injection at the instruction level is FiSim [14]. This approach is cost-effective, but does not consider the microarchitectural details of the underlying hardware. An other method relies on QEMU (Quick EMUlator) for fault simulation as presented in [15] [16] and [17]. Although QEMU provides an efficient and complete platform emulation, the microarchitectural details of the hardware components are not fully modeled, limiting the scope of the security evaluation.

To leverage this limitation, fault injections can be simulated at the Register Transfer Level (RTL) using commercial simulators such as Vivado or QuestaSim [18] or a post-layout netlist level [19]. These methods allow to finely study the impact of fault injection at the hardware level. However, it is very time-consuming, especially when dealing with complex hardware designs or large software systems such as operating systems.

Circuit emulation-based fault injection offers a viable solution to overcome the inherent slowness of simulation and thus accelerate evaluation campaigns. When the circuit is a Field Programmable Gate Array (FPGA) it allows accelerating and reconfiguring evaluation campaigns. One method of emulating faults on an FPGA-based design is through the use of partial reconfiguration (PR), a technique that allows dynamic modification of the state of flip-flops without interrupting the entire system [20], [21]. This allows for run-time targeted fault injection. Another method is to modify the hardware description of the design to incorporate a saboteur strategically placed in the circuit where faults are to be injected [22]–[24]. These methods provide a more flexible and efficient way of assessing a system's resilience to faults than full simulation. Compared to physical fault injection, fault emulation is less costly because it does not require expensive physical equipment, such as lasers or electromagnetic pulse generators. It also offers a significant speed advantage over traditional simulation. However, integrating a fault emulation mechanism into a SoC is a complex task. In this paper, we propose to leverage this limitation by extending the LiteX framework to automate the integration of fault emulation capacity into complex SoC.

To provide a unified view of the landscape, Table I summarises and contrasts the main categories of existing fault injection evaluation approaches discussed above. The comparison draws attention to the different levels of precision, scalability, execution speed and implementation complexity of each approach. Precision refers to the capacity of the approach to consider fault injection effects on a complete systems. Instruction-Level Simulation and Full-System Emulation abstract the hardware layer, reducing the scope of the analysis to high level effects. High precision is reached for approaches relying on a fine description of both hardware and software behavior. It is worth noting that formal methods relying on both software and hardware desciption can reach high precision. In terms of scalability, formal methods-based and RTL/netlist simulation-based approaches are limited to simple systems due to the complexity or the duration of the required computation. Regarding execution speed, RTL/Netlist simulation and Formal methods can lead to prohibitive execution time for complex designs. However, FPGA emulation allows to greatly speed up fault injection campaigns compared to software-based solutions. The expertise required to use the different methods varies considerably: formal methods, RTL/Netlist simulation and FPGA emulation require a high level of expertise to manipulate low level hardware descriptions or models to perform formal verification. Instruction-level simulation and Full system emulation are the most accessible since they abstract the hardware layer. Compared to existing works, the approach presented in this paper provides a high precision, scalable and high speed solution while drastically reducing the complexity to set up of fault injection campaigns on a complex SoC.

III. Proposed Approach

This section details the LiteInjector fault emulator by explaining the implementation of its various logical blocks. Fig. 1 illustrates a simplified representation of the emulation platform designed to conduct a security evaluation campaign on different devices under test (DUT). The LiteInjector fault emulation platform is composed of two main components: a software component and a hardware component.

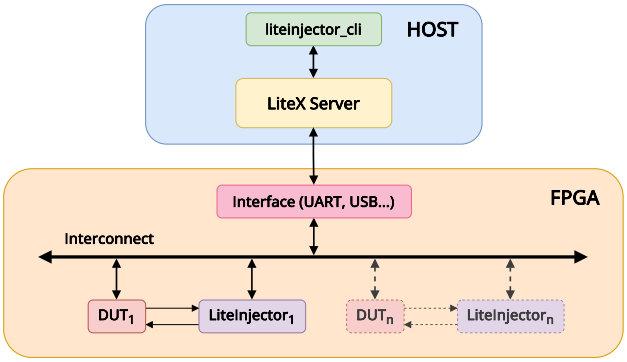

Fig. 1. Overview of the fault emulation platform on FPGA. A command line interface (CLI) allows the hardware design on the FPGA to be configured from the software running on a computer (HOST). The SoC is extended with a set of LiteInjector modules for fault injections on device under test (DUT).

a) Software Component: This includes the software controller, referred to as *litescope_cli*. The primary purpose of this controller is to manage the hardware component of LiteInjector, which resides on the FPGA board. The controller supports multiple communication protocols, enabling seamless interaction between the controller and the interconnect fabric. A detailed explanation of the operation of the software controller is provided in Section III-B.

TABLE I
COMPARISON OF FAULT INJECTION EVALUATION APPROACHES

Approach Type	References	Precision	Scalability	Execution Speed	Expertise
Formal Methods	[9], [11], [12]	Low/High	Low	Low	High
Instruction-Level Simulation	[13], [14]	Low	High	Medium	Low
Full-System Emulation	[15]–[17]	Medium	High	Medium	Medium
RTL / Netlist Simulation	[18], [19]	High	Low	Low	High
FPGA Emulation	[20]–[24]	High	High	High	High
Automated SoC Emulation (LiteX)	Our approach	High	High	High	Low/Medium

b) Hardware Component: This component is designed to be deployed in a SoC to facilitate security evaluation campaigns on various DUTs. The hardware components are described in Python using the Migen language and the LiteX framework. This approach greatly simplifies the modification of the hardware architecture and its integration into LiteX-based SoCs. Additionally, Migen supports exporting the component in Verilog or VHDL, ensuring that LiteInjector can be integrated into a wide variety of hardware designs. As shown in Fig. 1, it is possible to instantiate multiple LiteInjector hardware modules in parallel, all of which can be configured through a single controller. The detailed architecture of the LiteInjector hardware block is described in Section III-A. This modular and scalable approach ensures flexibility and adaptability to the specific requirements of each DUT under evaluation.

A. Hardware Modules

Fig. 2 illustrates the hardware architecture of the LiteInjector module.

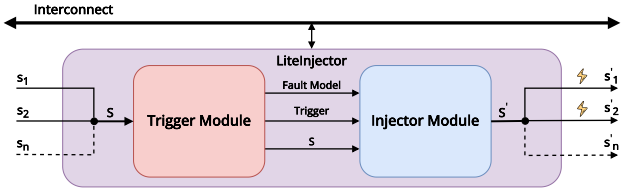

Fig. 2. Architecture of the LiteInjector module. All Signals dedicated to the trigger module configuration are not represented to simplify the schema.

It consists of two main components: the trigger module, which detects specific events to initiate fault injection, and the injection module, which modifies the target signal values once the trigger signal is activated.

As shown in Fig. 2, several internal signals s_i can be considered to build the input S. This combined signal S is then utilized by both the trigger and injection modules. In the output stage, the individual signals extracted from $S^{'}$ are reintegrated into the hardware design, thus propagating the injected faults. The trigger module is responsible for managing the configuration of the emulator. It handles receiving event configurations to be detected and fault models to be passed on to the injection module. The combined S signal plays an essential role in our system. It allows us to work on several internal signals at the same time for monitoring and fault injection. The trigger module propagates to the injector module the S signal and the configuration for the faults to be injected (i.e. the fault model), along with the trigger signal that indicates when the faults should be injected. On the other hand, the injection module is responsible for executing the fault injection. If the trigger signal is active, the output of the injection module $S^{'}$ is the faulty version of the S with respect to the configured fault model. If not, $S^{'}$ is equal to the input signal S.

Our hardware component is based on a masking system that is used to select, isolate, or modify manipulated signal while preserving the rest unchanged. This mechanism enables precise logical operations, such as forcing certain bits to fixed values, inverting them, or partially rewriting data while maintaining the integrity of untouched portions. In our approach, the masking system plays a pivotal role in achieving flexible and fine-grained signal management during fault injection campaigns. Additionally, the use of dynamically reconfigurable masks provides enhanced flexibility, enabling real-time adjustments to injection configurations. Furthermore, masking is used for event detection and fault injection triggering. By filtering concatenated signal values, it becomes possible to monitor complex events across multiple signals in parallel with bit-level precision. This capability supports the monitoring of intricate scenarios and the testing of complex systems under various conditions, as shown in section IV.

1) Trigger module: Fig. 3 represents a simplified diagram of the internal logic of the trigger module. The trigger module plays a pivotal role in fault injection campaigns, enabling the detection of specific events in SoC signals to initiate fault injection. This section details its architecture and functionalities, emphasizing its ability to handle complex trigger conditions and dynamic reconfigurations. The primary objective of the trigger module is to detect specific events from SoC signals and initiate fault injection. The module supports advanced trigger conditions, including the comparison of multiple bits in different signals. In addition, it incorporates rising and falling edge detection for individual bits of monitored signals. These capabilities are implemented in parallel using a masking system, enabling the detection of intricate and precise events.

a) Configuration system: The trigger module manages a First-In-First-Out (FIFO) memory that facilitates the configuration of various hardware components. The FIFO is software-controlled and stores multiple configurations that define triggering values, fault injection delays, fault models, and other parameters. Upon completion of a fault injection, the subsequent configuration in the FIFO is dynamically loaded to reconfigure the hardware components. The FIFO's size is adjustable, allowing users to determine the number of sequential fault injections required.

b) Edge detection mechanism: The module includes two submodules dedicated to detecting rising and falling edges on monitored signals. This functionality is achieved by employing two sequential registers that shift values at each clock cycle. By comparing these consecutive values, the system identifies transitions, either rising or falling edges, in real-time. This mechanism is extended to all monitored signals, enabling edge detection across all signal bits simultaneously. This feature enhances the detection system, allowing the identification of complex events based on edge transitions.

c) Configurable delay: Fault injection campaigns often require managing a delay between event detection and fault injection. As illustrated in Fig. 3, the module incorporates additional logic to delay fault emulation by clock cycles N. This delay mechanism is dynamically reconfigurable using the configurations stored in the FIFO. Synchronization with the SoC clock ensures precise timing for fault injection, significantly improving reproducibility and accuracy.

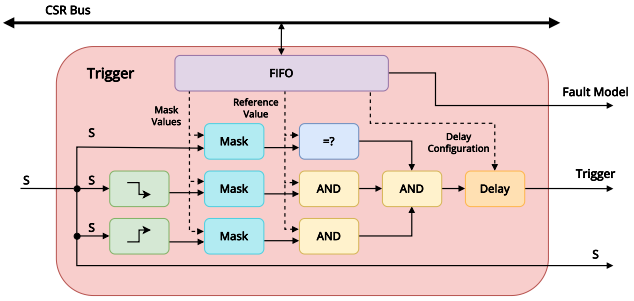

Fig. 3. Architecture of the trigger module. Edge detection and event monitoring are represented with three different logic paths. Those logics paths are configured by the FIFO.

2) Injector module: Fig. 4 represents a simplified diagram of the internal logic of the injector module.

The primary objective of the injector is to modify signal values based on configurations specified by the user. The developed injection system allows manipulation of the entire set of bits in a signal with bit-level precision. Using a masking mechanism and distinct logical paths for each fault model, our system supports five fault models. Each logical path corresponds to a specific fault model. By implementing these five fault models described below, our system enables simulation of various scenarios to evaluate the robustness of hardware components against several FIA.

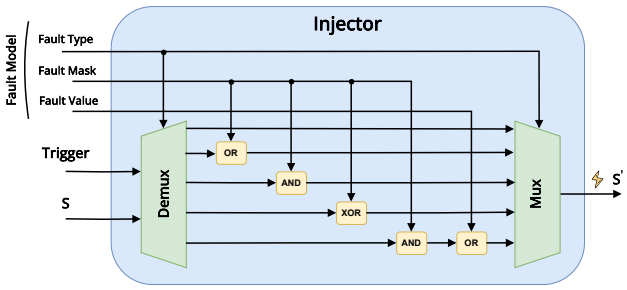

Fig. 4. Representation of the architecture of the injector module. Fault model signals are configuration signals from the FIFO. The trigger module start fault injection using the trigger signal.

a) Bit set: Using a logical OR gate, one or more bits can be forced to 1. The masking system enables the selection of specific bits to be set by applying a mask in which the target bits are 1, and the other bits are masked with 0 to preserve their original value.

b) Bit reset: By employing a logical AND gate, one or more bits can be forced to 0. The masking system precisely selects the bits to reset by applying a mask where target bits are 0, while other bits remain unchanged through a mask of 1.

c) Bit flip and random bit flip: A logical XOR gate allows one or more bits to be flipped. The masking system identifies the target bits to flip with a mask containing $1s$ for bits to modify and $0s$ for others to remain unchanged. For the random bit flip model, the software controller generates a random mask, leading to random modifications of the signal bits.

d) Write value: Combining a logical AND gate and a logical OR gate makes it possible to completely rewrite the value of a signal. First, a 0-mask and the AND gate reset all bits. Then, the OR gate and an appropriate mask set the new signal value. This model is particularly useful for scenarios such as instruction skipping or replaying.

B. Software Controller

The controller is responsible for computing and transmitting the configuration values for both the trigger and injection modules. When the emulator is generated using LiteX, a configuration file is created that lists all emulator input signals and their sizes. As depicted in Fig. 5, the software controller, `liteinjector_cli`, relies on the LiteX Server tool. The `LiteX Server` tool manages the connection between `liteinjector_cli` and the bridge to the SoC communication bus. Through `LiteX Server`, the software controller can support multiple interfaces, including UART, JTAG, USB, Ethernet (TCP/UDP) and PCIe. This versatility allows it to easily adapt to the various interfaces available on different FPGA boards. The controller can be used in two distinct ways: first, via the command line, where users provide trigger and injection parameters, and second, as part of a Python script, enabling automation of security evaluation campaigns. The primary function of the software controller is

to generate the masks required for event detection and fault injection based on user-defined instructions. Each instruction provided by the user is associated with a unique identifier (ID), enabling the management of complex conditions across multiple signals simultaneously. This capability is particularly advantageous for monitoring, where diverse conditions can be applied in parallel to different signals. For each directive, the software controller computes a mask according to the specified conditions. The same principle applies to fault injection, where the ID ensures mapping between the triggering conditions and the associated fault injection. The software controller calculates the masks to implement the targeted faults. However, for fault injection, only one signal can be modified per triggering condition. This restriction ensures precise and targeted modifications, providing reliable and reproducible fault injection.

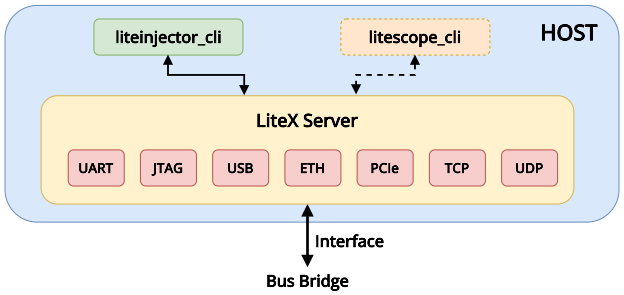

Fig. 5. Overview of the software controller

IV. Case Studies

This section presents the case studies used to evaluate and validate the performance of LiteInjector. Four case studies were executed, each involving fault injection attacks targeting different modules of a system-on-chip incorporating LiteInjector. These case studies highlight the platform's ability to accurately and efficiently emulate attacks, demonstrating its flexibility and applicability in different security assessment scenarios.

A. Experimental Setup

Fig. 6 shows the system-on-chip (SoC) used for our security campaigns targeting the Wishbone communication bus. The target SoC includes a read-only memory (ROM) containing the bootloader, a random access memory (RAM), a RISC-V processor and an Ethernet controller. The SoC also integrates our LiteInjector module, which is positioned between the processor and the communication bus. This module is able to monitor and alter the data transmitted to the processor. Listing 1 details the integration of the LiteInjector into the SoC and the modifications made to the signals connecting the communication bus to the processor to integrate our module. Specifically, the SoC was instrumented by adding the LiteInjector module, which is interconnected to the software controller via a UART bridge, as shown in Fig. 6. The LiteInjector module is connected to two signals on the Wishbone

```
injector_signals = [ self.cpu.pbus.dat_r, self.cpu.pbus.adr]

self.submodules.injector = LiteInjector(injector_signals,
depth        = 16,
csr_csv      = "injector.csv")
self.add_csr("injector")

self.cpu.cpu_params["i_peripheral_DAT_MISO"] =
↪  self.injector.o_pbus_dat_r
```

Listing 1: LiteInjector module integrated into a LiteX SoC for an attack campaign on the communication bus

bus: `DATA_R` and `ADR`, which correspond to `DAT_I` and `ADR_O` in the standard Wishbone bus nomenclature [25]. Fault injections target the `DATA_R` signal to modify the instructions executed by the processor. The `ADR` signal, which represents the address, is used to identify the instructions requested by the processor and is used to trigger fault injections. In addition to the fault emulation module, a LiteScope [26] module has been integrated. As an Integrated Logic Analyzer (ILA), LiteScope validates the behaviour of the LiteInjector module and ensures the correct execution of the fault injection campaign.

B. Case study unprotected VeriFyPin

For the first case study, we based our work on the FISSC [27] security benchmark. FISSC provides several software implementations specifically designed for fault injection campaigns. In our case, we used the VerifyPIN program, focusing on its unprotected version. The software is Linux system containing the VerifyPIN binary which is a program designed to verify a PIN code. It does not contain any countermeasures, and our goal was to bypass the verification process without knowing the correct PIN code. The SoC used is the one shown in Fig. 6. After reverse engineering the binary, we identified the instruction responsible for validating the PIN code. We then retrieved the address on the communication bus corresponding to the targeted instruction. These two parameters allowed us to define the trigger condition. During signal monitoring, if both the address and the instruction are detected on the bus, we proceed to inject the fault. The address corresponding to the sensitive instruction (`0x631EF700`, `bne x14, x15, 28`) is `0x100001E3`. To bypass the verification mechanism, we have flipped a bit in the instruction to invert the branch condition. Listing 2 shows how the LiteInjector module was configured to launch this attack. By modifying the 12th bit of the instruction, we successfully inverted the branch condition (`0x630EF700`, `beq x14, x15, 28`), allowing validation of an incorrect PIN code. Table II shows all the configuration values for the LiteInjector module, including the parameters required for event monitoring and fault injection. Finally, Fig. 7 illustrates the target signals during fault emulation. This case study demonstrates the ability of our module to accurately modify a signal value using a specific fault model.

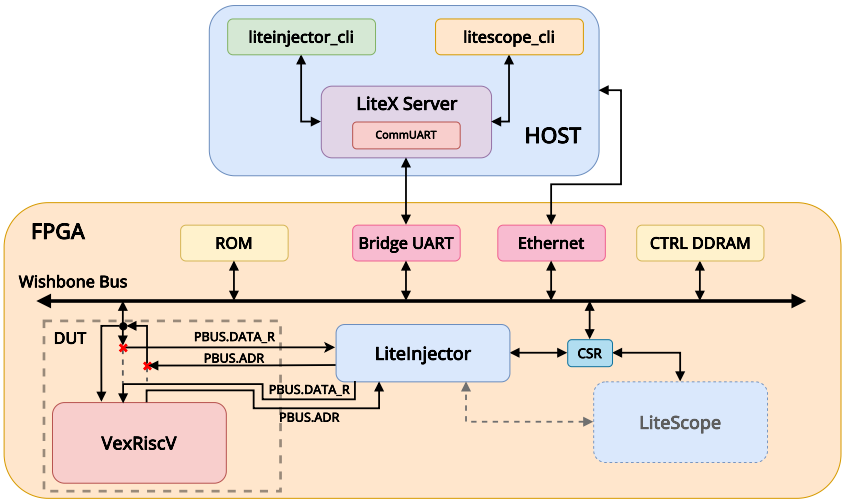

Fig. 6. SoC used for security campaigns targeting the communication bus to alter the instructions executed by the processor

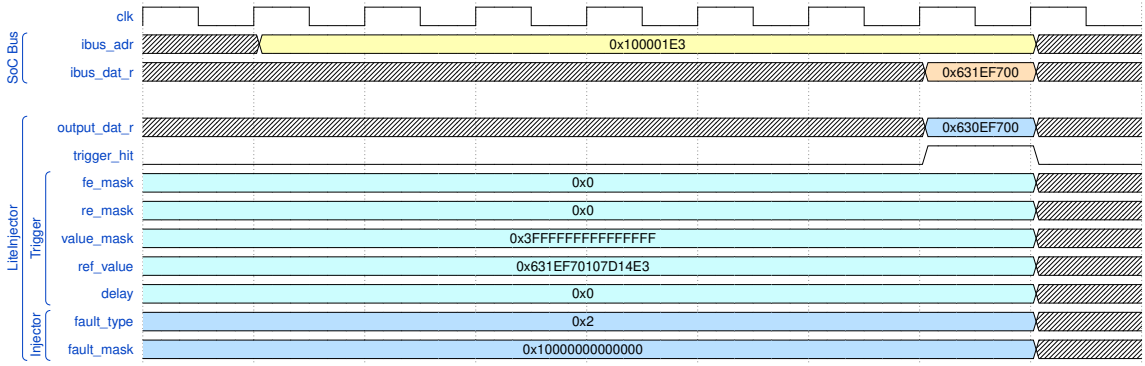

Fig. 7. Waveform representing the internal signals from the SoC and LiteInjector during the attack on the unprotected version of VerifyPin.

```
$ liteinjector_cli -v main_basesoc_pbus_adr 0x100001E3 0 -v
↪ main_basesoc_pbus_dat_r 0x00f71e63 0 -bf
↪ main_basesoc_pbus_dat_r {12} 0
```

Listing 2: liteinjector_cli command line for configuring the LiteInjector module to attack the unprotected version of VerifyPin

TABLE II
LITEINJECTOR MODULE CONFIGURATION VALUES FOR ATTACKING THE UNPROTECTED VERSION OF VERIFYPIN.

Signal	Value
Falling Edge Mask (62 bits)	0x0
Rising Edge Mask (62 bits)	0x0
Value Mask (62 bits)	0x3FFFFFFFFFFFFFF
Reference Value (62 bits)	0x631EF70107D14E3
Fault Type (3 bits)	0x2
Fault Mask (62 bits)	0x10000000000000
Fault Value (62 bits)	0x0
Delay (9 bits)	0x0

C. Case study protected VeriFyPin

For the second case study, we also used the VerifyPIN program, this time focusing on the protected version. This case study was inspired by [28]. The Linux system run the protected version of VerifyPIN which includes several software countermeasures designed to secure the PIN verification process. These countermeasures include: hardened bools, fixed time loop, inlined call, PTC decremented first, double test, step counter and control flow integrity. As in the first case study, the goal is to bypass the PIN verification process. However, in this protected version, it is necessary to perform a multiple skip instruction to bypass 16 branch instructions, each corresponding to the different checks introduced by the countermeasures. To achieve this, the branch instructions must be inverted to allow an incorrect PIN to be recognised as valid. By consulting the RISC-V documentation and using the assembler provided in the Ghidra tool, we found that a single bit flip on the 12th bit on each branch instruction was sufficient to reverse its logic. The parameters required for our fault injection campaign included the addresses and exact values of each targeted instruction. Each injection was

configured similarly to the first case study, with only the address and value of the instructions being adjusted. This case study, although similar to the unprotected version of VerifyPIN, validated the performance of LiteInjector in a scenario requiring multiple consecutive fault injections. It also demonstrated the tool's ability to detect complex events across multiple signals simultaneously, highlighting its effectiveness for more advanced security campaigns.

D. Case study VexRiscV PMP Software

This attack aimed to replicate behaviors observed during real-world fault injection campaigns targeting the Physical Memory Protection (PMP) mechanism in RISC-V processors. Authors in [4], [29] demonstrated that clock signal fault injections could alter PMP register configurations, allowing attackers to access memory regions that are normally read- and/or write-restricted. In this campaign, we targeted a RISC-V VexRiscV processor running a baremetal program that configures the PMP module to restrict read access to a protected memory zone. After reverse engineering the bare metal program, we identified the address of the instruction responsible for writing to the PMP (Physical Memory Protection) register. This register contains the security policy parameters to be enforced on the memory area to be protected. By completely modifying the instruction using the write value fault model, we were able to change the `LOAD` instruction to an `NOP` (No Operation). As a result, the PMP register was not reconfigured, leaving the protected memory region accessible for both reading and writing. These results are consistent with the effects observed in previous research on PMP vulnerabilities. Table III provides all the configuration values required for the skip instruction fault model. The address of the target instruction is `0x10000130` and the `NOP` instruction corresponds to code `0x00000013`. With this information, we have successfully configured LiteInjector to bypass the PMP register configuration.

TABLE III
LITEINJECTOR MODULE CONFIGURATION VALUES FOR ATTACKING THE PMP CONFIGURATION.

Signal	Value
Falling Edge Mask (62 bits)	0x0
Rising Edge Mask (62 bits)	0x0
Value Mask (62 bits)	0x3FFFFFFF00000000
Reference Value (62 bits)	0x1000013000000000
Fault Type (3 bits)	0x4
Fault Mask (62 bits)	0x00000003FFFFFF3F
Fault Value (62 bits)	0x0000000000000013
Delay (9 bits)	0x2

E. Case study VexRiscV PMP Hardware

For the final case study, we targeted the PMP module within the VexRiscV processor core. To execute this attack, we modified the hardware description of the PMP module as well as the processor's input/output interfaces to gain access to the PMP signals. Our attack focuses on the 1-bit signal used to interrupt the processor whenever a security policy violation occurs. Unlike the other case studies, this attack does not require advanced reverse engineering. When a security alert is triggered, a rising edge can be detected on this signal. Using LiteInjector's triggering system, we monitored this signal and injected a fault at the exact moment the alert was triggered.By using a bit reset fault model, the processor was not interrupted by the PMP module. As a result, access to the protected memory region was granted without restriction. This case study demonstrates that LiteInjector can also emulate faults on components not described in Migen, assuming that modifications are made to the hardware description. Table IV shows the configuration of LiteInjector used to apply the bit reset fault model to the alarm signal of the PMP module.

TABLE IV
LITEINJECTOR MODULE CONFIGURATION VALUES FOR ATTACKING THE PMP ALERT SIGNAL.

Signal	Value
Falling Edge Mask (1 bit)	0x0
Rising Edge Mask (1 bit)	0x1
Value Mask (1 bit)	0x0
Reference Value (1 bit)	0x1
Fault Type (3 bits)	0x1
Fault Mask (62 bits)	0x1
Fault Value (62 bits)	0x0
Delay (9 bits)	0x0

Table V recaps the main features of our case studies. The diversity of the different case studies we have presented validates the flexibility and the capabilities of LiteInjector's for several scenarios. We were able to validate the operation of the trigger module, which was used to detect events on several signals in parallel. The injection module was able to inject several faults in succession, with a different configuration for each injection.

TABLE V
SUMMARY TABLE OF CASE STUDIES WITH TARGETED SOC LOCATION AND EFFECT OF INJECTED FAULTS.

Name	Target	Fault effect
VerifyPin unprotected	Wishbone bus	Single bit-flip
VerifyPin protected	Wishbone bus	Multiple bit-flips
PMP software	Wishbone bus	Instruction skip
PMP hardware	PMP implementation	Bit reset

Table VI presents post-implementation results targeting a Arty-A7100T FPGA, using Vivado 2019.1. The integration of a LiteInjector module with a 62-bit S input leads to an

TABLE VI
POST-IMPLEMENTATION RESULTS - ARTY-A7100T FPGA.

Resource	SoC	SoC + LiteInjector	Overhead
LUT	9495	10587	+1092
FF	7209	7935	+726
BRAM	21.5	27	+5,5
DSP	4	4	0

overhead of 1092 LUTS, 726 FFs and 5.5 BRAMs. The low resource cost of using LiteInjectors means they can be placed in parallel, or more signals can be added to the module's input. However, the additional cost is not permanent, since the module is only used during the security evaluation campaign. The use of LiteInjector has no impact on the resource cost of a final system. Table VI shows the additional cost of the system used for most of our case studies, which is relatively low.

To evaluate the speedup of our platform, we measured the execution time from the Linux system from our two first case study to the end of the target application execution. Compared to verilator simulator running on a i9-9880H core and a 16GB RAM, our approach accelerates the security evaluation by a factor of 210. Indeed, to simulate the fault injection of the first case study it takes 1 hour and 10 minutes while only 20 seconds are required using LiteInjector. This example highlights the usefulness of using LiteInjector when safety assessment campaigns are too time-consuming, or even impossible in simulation. LiteInjector drastically speeds up fault emulation campaigns while offering the same fault injection capabilities as simulation.

V. Discussion

The LiteInjector open source framework[1] provides a practical tool for evaluating both software and hardware against FIA. By integrating it into the LiteX generation flow, it simplifies the evaluation of SoC security. In addition, it significantly accelerates fault injection campaigns compared to traditional simulation methods. LiteInjector is a flexible tool that takes advantage of the LiteX SoC generator. As shown in Listing 1, LiteInjector integrates seamlessly into LiteX-generated systems-on-chip. Furthermore, its development in Migen has enabled us to make LiteInjector generic. For instance, the size of its inputs are automatically computed during the SoC generation and its internal logic is adapted accordingly. The user doesn't need to modify LiteInjector's internal logic for each use case. The module automatically adjusts the logic to the size of the input signals, and automatically creates new output signals for each input signal. The same principle applies to the software: the configuration file needed to drive LiteInjector is generated automatically when the SoC is created. The controller retrieves this configuration and interprets the user's instructions.

LiteInjector is not dependent on any particular FPGA board. Once the Migen description has been converted to HDL, the module can be integrated into any hardware design flow. When LiteInjector is introduced directly into a LiteX SoC, we take advantage of the framework's broad compatibility with most FPGAs on the market. LiteInjector is not dependent on any particular technology that would limit it to a specific hardware. Once the module has been generated, it can be integrated into any FPGA.

[1]https://github.com/labsticc-arcad/LiteInjector

In terms of functionality, LiteInjector incorporates a trigger module capable of handling complex conditions with value and edge detection on several signals in parallel. This trigger module is an asset when it comes to injecting faults with the necessary precision. This module can support a strong threat model where the attacker is assumed to have a high level of information about the target. The injection module enables bit-accurate fault injection. It also supports five fault models, enabling us to reproduce the full range of fault effects which are required for intensive security evaluation campaigns. Such a capacity allows, for example, divers instruction modification leading to instruction skip, as we have shown in one of our case studies. LiteInjector has the ability to detect complex events at cycle and bit level accuracy to inject one or multiple faults over the entire target signal at the bit level. These features enable us to maintain the same capabilities as in simulation, while taking advantage of the FPGA's acceleration.

LiteInjector already has several features. However, we have identified areas for improvement in both its software and hardware components. Currently, LiteInjector associates a single event with a fault injection, requiring event detection for each subsequent fault. It could be beneficial to allow multiple fault injections in sequence, triggered by a single event. By modifying the internal logic of the trigger module, it would be possible to use the trigger signal required for the fault injection as a newly detected event. This extension would allow the creation of a sequence of fault injections based on a single event detected from the DUT signals. The aim here is to extend the capabilities of our tool to cover all possible scenarios.

Regarding the injection module, it does not currently support spatial multi-fault injections on different signals in parallel. This limitation can pose a challenge when conducting security campaigns targeting components with countermeasures operating in parallel with the system. By modifying the logic of the injection module and replicating the corresponding logical paths for each fault model, we could enable spatial multi-fault injections on multiple signals in parallel. Furthermore, by duplicating the logical paths, it would be possible to apply a different fault model to each target signal. This enhancement would allow LiteInjector to handle more complex scenarios involving spatial multi-fault injections across multiple signals simultaneously.

The last improvement we identified was the management of signals within the LiteInjector logic. Grouping all signals together creates unnecessary logic overhead. By creating two distinct signal groups, we could reduce the resource cost of LiteInjector, making it more efficient and optimized.

A significant limitation of LiteInjector and similar tools is the automatic integration of saboteurs. LiteInjector relies on Migen and LiteX to provide integration flexibility and ease of implementation on target modules. However, when working with components that are not described in Migen, it becomes necessary to manually modify the hardware description of the components to access the desired logic. Approaches that attempt to modify the netlist during the design flow to integrate saboteurs represent an interesting solution. However,

this solution introduces a dependency on vendor-specific tools or hardware, as it seems unlikely that a generic approach compatible with all FPGA vendors will be developed. The automatic integration of saboteurs is therefore an important area of improvement for the wider adoption and usability of LiteInjector.

VI. Conclusion

This paper introduces LiteInjector, an open-source tool that can be used with LiteX SoC to assess security against FIA. The interest of the proposed framework is demonstrated through a password check program running on a Linux embedded system. Results show that performing a fault injection is 210 times faster than a fault injection simulation relying on an HDL simulator. In future work, we plan to optimize and extend LiteInjector with extra fault models and the capacity to inject global or local clock glitches in order to emulate electromagnetic fault injections. Furthermore, we plan to extend the software controller to automate vulnerability assessment campaigns. Finally, we plan to automate the integration of Liteinjector modules into a SoC based on a list of target hardware elements to be faulted.

References

[1] X. M. Saß, R. Mitev, and A.-R. Sadeghi, "Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6239–6256. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/sass

[2] S. Nashimoto, D. Suzuki, R. Ueno, and N. Homma, "Bypassing Isolated Execution on RISC-V using Side-Channel-Assisted Fault-Injection and Its Countermeasure," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 1, p. 28–68, Nov. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9289

[3] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, 2006.

[4] K. Quénéhervé, W. Pensec, P. Tanguy, R. Dafali, and V. Lapotre, "Exploring Fault Injection Attacks on CVA6 PMP Configuration Flow," in *27th Euromicro Conference Series on Digital System Design (DSD)*, 2024.

[5] B. Colombier, P. Grandamme, J. Vernay, É. Chanavat, L. Bossuet, L. de Laulanié, and B. Chassagne, "Multi-Spot Laser Fault Injection Setup: New Possibilities for Fault Injection Attacks," in *Smart Card Research and Advanced Applications*, 2022.

[6] F. Kermarrec, S. Bourdeauducq, J. L. Lann, and H. Badier, "LiteX: an open-source SoC builder and library based on Migen Python DSL," *CoRR*, 2020. [Online]. Available: https://arxiv.org/abs/2005.02506

[7] M-Labs, "Migen: A Python toolbox for building complex digital hardware." [Online]. Available: https://github.com/m-labs/migen

[8] P. Maistri and J. Y. Po, "A Low-Cost Methodology for EM Fault Emulation on FPGA," in *Design, Automation and Test in Europe Conference (DATE 2022)*, 2022.

[9] J. Richter-Brockmann, A. Rezaei Shahmirzadi, P. Sasdrich, A. Moradi, and T. Güneysu, "FIVER – Robust Verification of Countermeasures against Fault Injections," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021.

[10] V. Arribas, S. Nikova, and V. Rijmen, "VerMI: Verification Tool for Masked Implementations," in *25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2018.

[11] G. Barthe, S. Belaïd, G. Cassiers, P.-A. Fouque, B. Grégoire, and F.-X. Standaert, "maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults," in *Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Proceedings, Part I*, 2019.

[12] S. Tollec, V. Hadžić, P. Nasahl, M. Asavoae, R. Bloem, D. Couroussé, K. Heydemann, M. Jan, and S. Mangard, "Fault-Resistant Partitioning of Secure CPUs for System Co-Verification against Faults," 2024. [Online]. Available: https://eprint.iacr.org/2024/247

[13] A. Adhikary and I. Buhan, ""sok: Assisted fault simulation"," in *Applied Cryptography and Network Security Workshops*. Cham: Springer Nature Switzerland, 2023, pp. 178–195.

[14] Riscure, "FiSim: An open-source deterministic Fault Attack Simulator Prototype." [Online]. Available: https://github.com/Keysight/FiSim

[15] Y. B. Bekele, D. B. Limbrick, and J. C. Kelly, "A Survey of QEMU-Based Fault Injection Tools & Techniques for Emulating Physical Faults," *IEEE Access*, 2023.

[16] F. Hauschild, K. Garb, L. Auer, B. Selmke, and J. Obermaier, "ARCHIE: A QEMU-Based Framework for Architecture-Independent Evaluation of Faults," in *Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 2021.

[17] K. Murdock, M. Thompson, and D. Oswald, "FaultFinder: lightning-fast, multi-architectural fault injection simulation," in *ASHES '24: Proceedings of the 2024 Workshop on Attacks and Solutions in Hardware Security*. Association for Computing Machinery (ACM), Oct. 2024, not yet published as of 16/10/2024.

[18] W. Pensec, V. Lapotre, and G. Guy, "Scripting the Unpredictable: Automate Fault Injection in RTL Simulation for Vulnerability Assessment," in *27th Euromicro Conference Series on Digital System Design (DSD)*, 2024.

[19] J. Grycel and P. Schaumont, "SimpliFI: Hardware Simulation of Embedded Software Fault Attacks," *Cryptography*, vol. 5, no. 2, 2021.

[20] A. Ullah, P. Reviriego, and J. A. Maestro, "An Efficient Methodology for On-Chip SEU Injection in Flip-Flops for Xilinx FPGAs," *IEEE Transactions on Nuclear Science*, 2018.

[21] F. Ghaffari, F. Sahraoui, M. El Amine Benkhelifa, B. Granado, M. A. Kacou, and O. Romain, "Fast SRAM-FPGA fault injection platform based on dynamic partial reconfiguration," in *2014 26th International Conference on Microelectronics (ICM)*, 2014.

[22] Z. U. Abideen and M. Rashid, "EFIC-ME: A Fast Emulation Based Fault Injection Control and Monitoring Enhancement," *IEEE Access*, 2020.

[23] W. Mansour and R. Velazco, "An Automated SEU Fault-Injection Method and Tool for HDL-Based Designs," *IEEE Transactions on Nuclear Science*, 2013.

[24] S. Eldridge, A. Buyuktosunoglu, and P. Bose, "Chiffre: A Configurable Hardware Fault Injection Framework for RISC-V Systems," in *2nd Workshop on Computer Architecture Research with RISC-V (CARRV '18)*, 2018.

[25] OpenCores, "Wishbone specification." [Online]. Available: https://cdn.opencores.org/downloads/wbspec_b3.pdf

[26] EnjoyDigital, "Small footprint and configurable embedded FPGA logic analyzer." [Online]. Available: https://github.com/enjoy-digital/litescope

[27] L. Dureuil, G. Petiot, M. Potet, T. Le, A. Crohen, and P. de Choudens, "FISSC: A Fault Injection and Simulation Secure Collection," in *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, 2016, pp. 3–11.

[28] A. Gicquel, D. Hardy, K. Heydemann, and E. Rohou, ""samva: Static analysis for multi-fault attack paths determination"," in *Constructive Side-Channel Analysis and Secure Design*, E. B. Kavun and M. Pehl, Eds. Cham: Springer Nature Switzerland, 2023, pp. 3–22.

[29] S. Nashimoto, D. Suzuki, R. Ueno, and N. Homma, "Bypassing isolated execution on risc-v using side-channel-assisted fault-injection and its countermeasure," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 28–68, 2022.