Lightweight Countermeasures Against Original Linear Code Extraction Attacks on a RISC-V Core

Théophile Gousselot, Olivier Thomas, Jean-Max Dutertre, Olivier Potin, Jean-Baptiste Rigaud

Author's version HOST May 2023 DOI: 10.1109/HOST55118.2023.10133316

Lightweight Countermeasures Against Original Linear Code Extraction Attacks on a RISC-V Core

Théophile Gousselot*, Olivier Thomas†, Jean-Max Dutertre*, Olivier Potin*, Jean-Baptiste Rigaud*
*Mines Saint-Etienne, CEA, Leti, Centre CMP, F - 13541 Gardanne France
theophile.gousselot@emse.fr dutertre@emse.fr olivier.potin@emse.fr rigaud@emse.fr
†Texplained olivier@texplained.com

***Abstract*—Linear Code Extraction (LCE) is an invasive attack aiming at fully extracting a code from a device's memory for reverse engineering purposes. The core instruction bus is identified and microprobed using Failure Analysis tools. Meanwhile, other microprobes force internal nodes of the core to logic states which allow a full memory linear extraction.**

This paper demonstrates the first assessment of a RISC-V core vulnerability to LCE. It evaluates the complexity to extract the code in the right order by freezing the instruction register or by editing the incoming instructions. This paper introduces three original countermeasures to detect an ongoing LCE by monitoring symptoms such as the lack of branch instruction execution. These hardware countermeasures are lightweight and adaptable to other core architectures. We develop an experimental setup based on a functional simulation framework and an FPGA-based demonstration. This setup made it possible to study and assess the LCE vulnerabilities of our RISC-V target and to validate the effectiveness of our proposed countermeasures. The area overhead was measured between 0.52% and 1.47% of the cv32e40p RISC-V core. Depending on the detection latency target, the clock cycle overhead using the Embench™ benchmarks can be null or kept below 1%.

***Index Terms*—linear code extraction, reverse engineering, hardware security, countermeasure, RISC-V**

I. Introduction

Embedded systems execute applications stored in their memories. They may be the target of hardware or software attacks which aim to extract the application code. As introduced in [1, §7.1], reverse engineering the code running on an Integrated Circuit (IC) is popular as it can lead to a profitable business. Manufacturers designing an IC based on a competitor's design reduce their development costs. Manufacturers may also sell low-cost devices which are compatible with a competitor's product. [2] discusses the fact that in the 90s, compatible pirate smart cards to access paid TV services were selling well. Now, peripheral devices as gamepad controllers or ink cartridges are the main targets such as they are increasingly widespread. Some well-funded and organized manufacturers such as *Apex Microelectronics* sell compatible ink cartridges [3].

Despite elementary countermeasures against code extraction, such as destroying the embedded memory readback logic after programming suggested in [4, 8.3], the attackers may employ Failure Analysis (FA) tools to extract a code anyway. The instruction bus between the core and the memory may be microprobed as introduced in [5]. The first step is to locate the bus, for example by imaging the IC layers with a Scanning Electron Microscope (SEM). Then, signals from the instruction bus are routed to the top layer with Focus Ion Beam (FIB) edits, in order to be put in contact with microprobes as carried out in [6]. All the executed instructions are eavesdropped. To extract all application functionalities, the code must be extracted entirely as discussed in [7, §16.6]. Other microprobes must be set up to force specific internal nodes of the core (e.g., the reset of the instruction register) to logic states which induce an execution of the instructions one after the other, as ordered in memory (i.e., a linear execution). This type of attack is called a Linear Code Extraction (LCE). The targeted internal nodes constitute the attack path of the LCE. The attack paths involve different numbers of microprobes, depending on the targeted core and its Instruction Set Architecture (ISA). To date, there have been no studies on the RISC-V vulnerabilities to LCE, despite its growing use in Embedded Systems. There is no existing infallible countermeasure against LCE. Most efficient ones such as the Probe Attempt Detector [8] created a significant area overhead to protect all the core nodes and instruction bus sections. Memory encryption [6, §5] may be circumvented by microprobing the instruction bus after it is deciphered. Using design obfuscation [9] does not prevent the instruction bus from being identified.

The main contributions of our research are threefold:

- To assess the vulnerabilities of a RISC-V core to LCE. Theoretical analysis and practical experiments determine three types of attack paths to induce a linear execution: by tampering with the incoming instructions, the instruction register, or the Program Counter (PC) circuitry. Some of them are briefly mentioned in the state-of-the-art in [5], [6], [10]. Our contribution is to specially study how to take advantage of the RISC-V core considered: the cv32e40p. In particular, we evaluated the complexity in terms of microprobe's number and their locations.
- To introduce three original countermeasures as a way to protect against LCE. Focused on detecting LCE, they are lightweight enough to be integrated into small embedded devices without significant performance degradation. They

The ARSENE project was funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference "ANR-22-PECY-0004".
This research was also funded by the European Union under grant agreement no. 101070008.

monitor symptoms of LCE, such as the lack of branch or custom security instruction execution. They are implemented and integrated into the cv32e40p core. Simulations and FPGA-based demonstration allow us to assess and characterize the countermeasures in practice (effectiveness, detection latency, clock cycle overhead, maximum frequency, area overhead).

• To develop and publish an Experimental Setup including one functional simulation framework and one FPGA-based demonstration. It ascertains the complexity of custom LCE attack paths when taking advantage of the vulnerabilities of a RISC-V core. It demonstrates and characterizes the introduced countermeasures.

This paper is structured as follows: Section II introduces additional background on code extraction threats and motivations, the LCE methodology, and the existing countermeasures. Section III describes the Experimental Setup. Section IV reports on the vulnerabilities of a RISC-V core and the experimented LCE attack paths. Section V presents the three conceived countermeasures and their practical implementation. Section VII provides access links to the countermeasure descriptions and the Experimental Setup in order to replicate the paper results.

II. Background

This background section reviews the motivations and methods of LCE attacks, as well as the existing countermeasures.

A. Code extraction: threat analysis

Embedded systems are built around a core which executes a program stored in a memory. This program, also called firmware or code, contains instructions and data. The instructions are executed by the core to carry out an application. The code in memory could be targeted by software or hardware attacks. Skorobogatov introduced five reasons to attack a device [1, §7.1]: theft of service, cloning, overbuilding, IP piracy, and denial of service. Extracting a code stored in a memory may lead to achieving these goals. For example, to design a gamepad controller compatible with a video game console, the communication and authentication protocols must be reproduced. For that purpose, code from an original gamepad controller memory has to be reverse-engineered. Therefore, it has to be extracted beforehand.

1) Extraction from external memory interfaces: The extraction can take place on external memory interfaces as described in [11]. In order to extract code from memory, some debug interface abuses, JTAG and UART exploits, and raw flash dump can be carried out. Specific voltage glitches may also help to get memory read access, as presented in [12]. Countermeasures against these types of attacks exist, such as memory encryption to prevent any use of the extracted code. Removing the external memory interfaces as suggested in [4, §8.3] addresses these extraction approaches.

2) Extraction from the instruction bus: Another way to extract code is to eavesdrop the instruction bus between the memory and the core. Such an invasive attack is carried out using FA tools as illustrated in [6]. The eavesdropping is accomplished by getting access to the instruction bus in order to probe it electrically with microprobes. Finding the bus and identifying the right probing location is possible by using a SEM and delayering tools as explained in [13]. Once the probing locations are known, a FIB etches and deposes silicon oxide and metal to route the instruction bus bits to microprobes. At the time instructions are executed, they are extracted. The extracted instructions may be reordered to rebuild the code. Unfortunately, only the executed instructions are extracted, which also means that application functionalities may be missing. For example, some authentication features may be enabled by the video game console later after commercialization, when the first compatible gamepad controllers appear as noticed in [7, §16.6].

3) Linear Code Extraction: LCE is still about eavesdropping the instruction bus, but in addition, some core nodes are forced into logical states that allow a linear execution. All the memory's instructions pass through the instruction bus and are extracted, including all application functionalities. The reverse engineering of the code is also straightforward, as there is no need to put the instructions back in order.

4) LCE funding: As an invasive attack, performing an LCE is reserved for well-equipped laboratories. Some manufacturers are so profitable by selling compatible products that they finance LCE attacks on targeted devices like *Datel*, a compatible game console peripheral maker. The market for compatible products is driven by ink cartridges through companies like *Static Control Components* or *Apex Microelectronics*. Despite the lack of market data, a trend is apparent. A Market Research Report [14] indicates that the global revenue for ink cartridges and toners for 2021 was USD 16.96B and is forecasted to reach USD 24.60B by 2027. Furthermore, Hamm revealed in [15] that compatible product manufacturers sell 25% of compatible cartridges. According to its growth, it can be concluded that the cartridge market is attractive and profitable (e.g., *Ninestar Corp*, a major actor in the design of compatible devices and ICs, declared revenue of USD 4.3B in 2019 [16]). Selling compatible products is legal in most countries, as highlighted in [17] by the American legal case concerning ink cartridges developed by *Static Control Components* compatible with the *Lexmark* printer authentication system. Even though compatible device manufacturers are not authorized to copy IPs that belong to the genuine source, proving these infringements is a difficult task that also requires the reverse engineering of compatible products.

Given the potential profitability, compatible product manufacturers can allocate funding for significant resources — equipment and knowledge — to carry out LCE attacks on targeted devices on an industrial scale. According to the IBM attacker classification [18], these manufacturers should be labeled as attackers of class III: Funded organizations like governments.

B. LCE: principles and methodology

An LCE aims at extracting code in a linear fashion from a memory. Two sets of microprobes are used. One set to

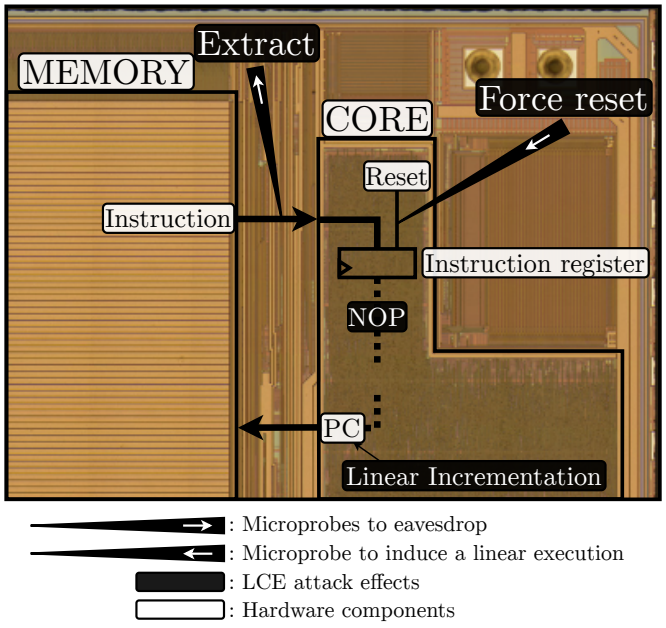

Fig. 1. LCE operation on an Integrated Circuit overview.

eavesdrop the instruction bus and the other set is used to induce a linear execution. This attack principle was introduced by Anderson in [7], [19] and Kommerling in [5]. The principle is illustrated in Fig. 1, there are four steps needed when carrying out an LCE on an IC.

1) Attack path definition: Analyzing the targeted core and its ISA highlights potential vulnerabilities. A classical attack path involves forcing the instruction register to reset. Hence, every instruction is turned into a No Operation (NOP) instruction before reaching the decode stage of the core. A linear read of the memory is thus obtained (see below).

2) Hardware identification: Locations for the microprobes according to the attack path must be found. Depackaging, then delayering the IC with chemicals, allows the layers of interest to be imaged with a SEM. Once a picture (as in Fig. 1) is obtained, the memory and the core can be identified. In order to eavesdrop, microprobes should be located on the instruction bus leaving the memory. Following the instruction, bus leads to the identification of the instruction register and its reset input. The reset signal is the target of the microprobe which induces a linear execution.

3) FIB edits to put microprobes in place: Pads like the cross shape in Fig. 2 must be electrically connected to signals of interest. For that purpose, a FIB must etch and depose silicon oxide and metal. Once cross pads are added to the bits of the instruction bus and the reset signal, microprobes can be put in direct contact with pads.

4) Linear code extraction: During execution, incoming instructions are extracted from the bus. When the reset signal of the instruction register is enabled, only NOPs are decoded and executed by the core. The execution of NOPs linearly increments the PC which will successively point to every memory address. The code from memory is extracted in a linear fashion.

LCE is an invasive attack involving FA Tools like SEM and FIB. Note that, only a limited number of microprobes can be put in place simultaneously (for a lack of access space) while minimizing any possible damage to the IC target. This limits the number of instruction bus bits eavesdropped. During one LCE, several extractions have to be performed in a row (generally on different chips).

C. Existing countermeasures against LCE

Some countermeasures against LCE have already been developed. None of them, though are infallible, and the most efficient ones often cause significant overhead in terms of either area or timing.

1) Clock signal randomization: Randomizing the IC clock as introduced in [5] may avoid the rebuilding of the code from bits extracted at different times. However, microprobing the clock signal is enough to synchronize them.

2) Memory encryption: Encrypting the memory makes the extracted code useless. Nevertheless, as noticed in [6] the core executes only decrypted instructions, thus, there is a deciphering component. Microprobing the instruction bus after deciphering leads to the extraction of the deciphered code. Recently, Skorobogatov challenged an encrypted ROM of a core to highlight its vulnerabilities against microprobing in [20].

3) Design obfuscation: Obfuscating the design alters the efficacy of hardware reverse engineering, as shown in [9]. Nonetheless, the program memory stays identifiable, and following its instruction bus is feasible.

4) Active shield or mesh: Top metal layers may be converted into an active shield (or mesh) as displayed in Fig. 2. An alarm is triggered when opened or shorted with its neighboring lines. However, a line may be cut and rerouted using a FIB, as the U-shape line in Fig. 2 shows, to open a space in the shield in order to access the inner routing. Otherwise, signals may be microprobed from the backside of the IC as clearly illustrated in [6, Fig. 1].

5) Probing Attempt Detector: As microprobes introduce an additional capacity load on the target lines, a delay between symmetric lines may reveal the microprobe. The detector area is gradually decreasing, as proposed in [8], however, in order to protect the line, one detector per signal is still required. Costs may significantly increase to cover every line where instructions are extractable.

6) Restricted PC: The PC may be split in two counters the sum of which is the PC value, as suggested by Kommerling in [5, §3.5]. The smaller one increments linearly. At every jump, the other one is updated with the destination while the smaller one is reset. This limits the size of a one-shot linear extraction. Nonetheless, block-per-block extraction of the entire code remains feasible. Indeed, there is always a jump to execute before being able to extract in a linear fashion a block of code since the smallest counter must not overflow into a nominal mode.

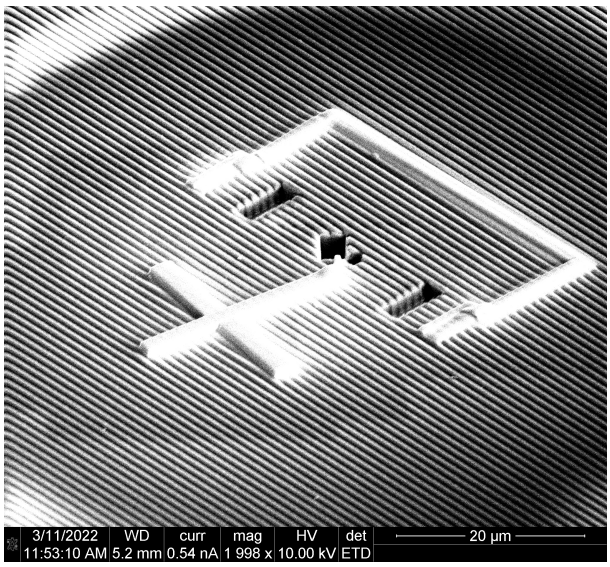

Fig. 2. Example of a cross pad built by Focus Ion Beam edits.

III. Experimental setup

We develop an Experimental Setup in order to ascertain the complexity of LCE attacks when taking advantage of specific vulnerabilities of a RISC-V core detailed in Section IV. This Experimental Setup also demonstrates and characterizes the countermeasures presented in Section V. There is one functional simulation framework and one FPGA-based demonstration.

The goal is not to reproduce every step of an LCE: hardware identification, FIB edits, and signal microprobing as [6] and [20] did in practice, but rather to quickly assess an attack path and to validate the countermeasures that we have developed in our study.

A. RISC-V and cv32e40p target

As RISC-V cores are increasingly used in embedded systems, we chose to analyze the potential vulnerabilities to LCE on a RISC-V core, since there are no existing studies on the topic. The authors studied the cv32e40p [21], a 32-bit, 4-stage, in-order, core provided by the OpenHW Group as it is representative of small ISA RISC-V cores. It was targeted for experiments, as it is also open-source (i.e., intrinsic countermeasures can be added to its Register Transfer Level (RTL) description).

B. Simulation framework

The functional simulation framework was developed for simulation-based validation of either LCE attack paths or countermeasures effectiveness. It runs fast campaigns of code executions to perform LCE by focusing on various vulnerabilities.

The proposed framework is depicted in Fig. 3. It is based on the core-v-verif framework [22] provided by OpenHW group. Simulations are sped up by using the Verilator [23] simulator. They are performed at the RTL level to ensure the accuracy

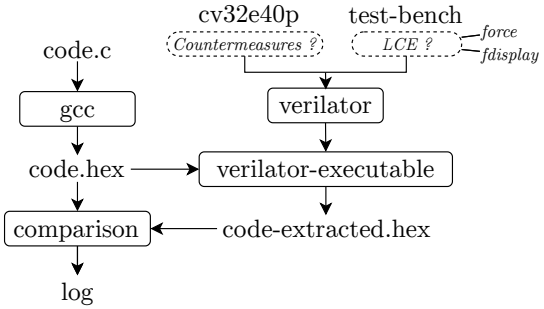

Fig. 3. Functional simulation framework.

of the observed behavior. From the cv32e40p RTL description and an LCE attack path described in a test bench, an executable Verilator model is built. The cv32e40p core can include a countermeasure. The LCE is simulated by using the *fdisplay* task and the *force* statement defined in the IEEE Standard for SystemVerilog [24] instead of microprobes. The *fdisplay* tasks eavesdrop the instruction bus and save the extracted instructions in a separate file. The *force* statement induces stuck-at faults to signals targeted by the defined attack path to ensure a linear execution. A code compiled by a gcc toolchain is given to the Verilator model to simulate its execution on the cv32e40p. The executed codes were taken from the benchmark suite Embench [25] as they are representative of embedded system applications. Simulation outputs like the percentage of correct extracted code, clock cycle overhead, or latency detection are generated by the framework and are compared to theoretical estimations in Section V.

C. FPGA-based demonstration

The FPGA-based demonstration target to quickly assess and observe core behaviors for many LCE attack paths on protected or not core. For that purpose, the Hardware identification and the FIB edits steps are emulated on FPGA. In particular, the software processing to automate the digitalization of extracted bits to rebuild the code is reproduced. The FPGA-based demonstration leads to the same conclusion as the simulator framework to validate the countermeasures and reports the utilization overhead. It is depicted in Fig. 4.

A bitstream of a microcontroller core-v-mcu [26] containing a cv32e40p core and a code in memory is downloaded to an Artix 7 FPGA (xc7a200tsbg484-1) on a Nexys Video board. The LCE is emulated by replacing the microprobes inducing a linear execution by inserting multiplexers on targeted signals (e.g., the instruction register reset in Fig. 1). These multiplexers are driven by switches which select the original input or the forced one. The microprobes eavesdropping the instruction bus are emulated by oscilloscope probes connected to FPGA board outputs. To imitate an attacker who has few microprobes, only four oscilloscope probes are connected to the FPGA board outputs. Bits of the instruction bus are routed to these outputs. Clock and reset signals are also routed for synchronization purposes. Thus, sixteen LCEs are run to extract all the 32 bits of instruction bus. An Integrated Logic Analyzer component

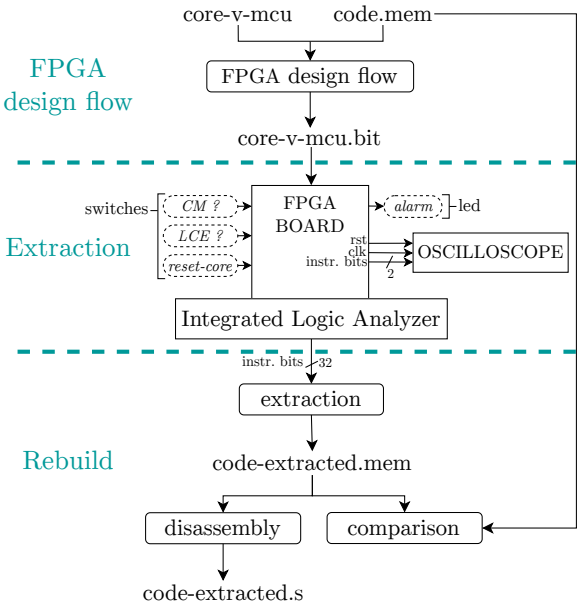

Fig. 4. FPGA-based demonstration framework.

allows emulating these sixteen executions. The extracted waveforms are then digitalized and blended by software processing to rebuild the code. This extracted code can be disassembled to be analyzed or compared to the one in memory. Concurrently, other switches enable and select one of the countermeasures integrated into the cv32e40p core. These switches and the probes connected to the oscilloscope are highlighted in the picture of the FPGA board setup in Fig. 5.

IV. Exploiting RISC-V LCE vulnerabilities

In order to completely extract a code from a core's memory, an LCE entails eavesdropping the instruction bus, while forcing nodes of the core into logic states which allow a linear execution. The microprobing to eavesdrop the instruction bus is quite common to every ISA and core architecture:

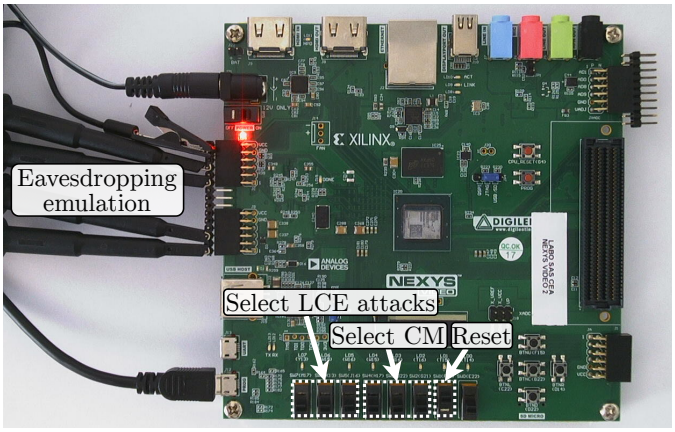

Fig. 5. Setup of the board with probes connected to the oscilloscope.

microprobes have to be set on the instruction bus between the core and its memory. Yet, the microprobing to force the linear execution and its complexity are highly dependent on the ISA and the core architecture vulnerabilities. For example, resetting the instruction register of a RISC-V core as exemplified in Fig. 1 leads to an exception because instruction *0x0000* is illegal in RISC-V.

To the best of our knowledge, our research study is the first to analyze the complexity of LCE attacks when taking advantage of specific vulnerabilities of a RISC-V core. Contrary to ARM or MIPS ISA, the open-source RISC-V ISA and the open-source cores like the cv32e40p make the identification of vulnerabilities completely feasible.

The complexity of ensuring a linear execution may be defined by the number of simultaneous microprobes and their locations. Recently, Skorobogatov performed eight simultaneous microprobes on a chip in [20]. The more microprobes that have to be used, the more time the attackers will spend to identify their locations and perform FIB edits, which amounts to an increased risk of destroying the chip.

A. Theoretical identification of vulnerabilities

Notation: going forward, the term Discontinuity Instruction (DI) refers to every instruction able to break a sequential execution (e.g., branches and jumps in RISC-V). The term Continuity Instruction (CI) refers to every other instruction.

Three types of attack paths (depicted in Fig. 6) exploiting distinct vulnerabilities are identifiable: (I) Freezing a CI in the instruction register, (II) Editing the incoming instructions or (III) Tampering with the PC. Some of them are either concisely or roughly mentioned in the state of the art [5], [6], [10], our contribution is to specially study how to take advantage of RISC-V and cv32e40p vulnerabilities.

1) Freezing a CI (I): Freezing a CI in the instruction register leads to increment linearly the PC. Disabling the *write_enable* (we) or clock inputs of the instruction register will freeze the current instruction, using only one microprobe. Thus the same frozen CI is executed repeatedly, while all the memory's instructions pass through the instruction bus. Caution must be taken not to propagate the forcing value into the core. Finding the instruction register inputs is made possible by following the instruction bus already found in order to set the eavesdropping microprobes. The first instructions of codes

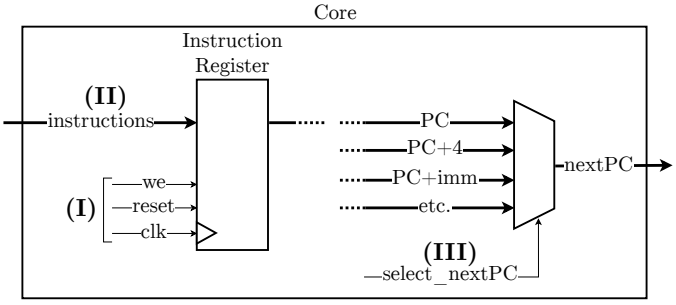

Fig. 6. Three types of identified vulnerabilities.

are often register-initialization (i.e., CI). Thus, a trial-and-error approach to freeze start times may lead to freeze a CI.

2) Editing incoming instructions (II): Editing specific signals of the instruction bus to turn DIs into CIs leads to increment linearly the PC. Every incoming instruction is transformed into a new one to remove all the DIs. The main pitfall is not raising an exception due to invalid RISC-V fields in the created instructions (i.e., opcode, funct3, funct7) or by misaligning load and store. This would fail the LCE attempt. In order to avoid these pitfalls, the simulator framework is useful to detect if an attack path leads to exceptions. A map of the major RISC-V opcodes, taken from documentation [27], is given in Table I. Red cells represent DIs, for all of them: inst[6:5] = 11. Cells with an exponent (e.g., LOAF-FP[2]) show opcodes only used by RISC-V extensions. Firstly, if these extensions are not implemented in the architecture, then forcing bit 5 or 6 of every instruction to 0 will turn DIs into invalid instructions. Secondly, U-type instructions [27, §2.3] (i.e., AUIPC and LUI) are the only CIs to admit all values for their bits from 7 to 31. As a result, at least 4 bits have to be forced to remove all DI without raising an exception. For example, inst[6]=0, inst[4:2]=101, to transform all instructions into AUIPC or LUI, the orange cells in Table I. When using 16-bit compressed instructions, the same forcing is still available once decompressed.

In comparison, for Armv8-A, bits from 26 to 28 of all DIs (named branches in ARM) are equal to 101, according to the documentation [28, p266]. Flipping one of these bits ensures that no DI will be executed. Maier [29, p7], claims that forcing only one bit is enough to ensure linear execution, but consideration should be taken to ensure that no exception will be raised.

Editing instructions by microprobing can be done using the microprobes already set in place to eavesdrop the instruction bus. Identifying the instruction bus is the only identification task for the attackers, but the number of involved microprobes is high.

3) Tampering with the PC (III): Tampering with the PC circuitry may lead to increment linearly the PC. Forcing the PC multiplexer control signal to select the linear increment input (e.g., the input PC+4 in Fig. 6) ensures a linear execution even if a jump instruction is executed (this attack path does not prevent instructions from being executed). On the cv32e40p core (without enabling the Hardware Loop extension), only one microprobe is needed to force the *id_stage_i.pc_set_o* signal of the decoder stage to 0. Finding this signal is feasible by identifying the PC bus and then following it upstream up to the multiplexer. Other design choices for the cv32e40p could increase the number of multiplexer inputs to be tampered with, which would increase the needed number of microprobes.

B. Practical validation of attack paths

The authors used the simulation framework described in Section III to assess the previously described attack paths. All three LCEs attack paths succeeded in extracting instructions and data from memory, without raising exceptions due to linear execution.

TABLE I
RISC-V BASE OPCODE MAP, INST[1:0]=11.

[4:2] / [6:5]	000	001	010	011	100	101	110
00	LOAD	LOAD-FP[2]	*custom-0*[4]	MISC-MEM	OP-IMM	**AUIPC**	OP-IMM-32[3]
01	STORE	STORE-FP[2]	*custom-1*[4]	AMO[1]	OP	**LUI**	OP-32[3]
10	MADD[2]	MSUB[2]	NMSUB[2]	NMADD[2]	OP-FP[2]	*reserved*	*custom-2/rv128*[4]
11	**BRANCH**	**JALR**	*reserved*	**JAL**	**SYSTEM**	*reserved*	*custom-3/rv128*[4]
		Discontinuity instructions			**U-type instructions**		

Atomic (A)[1] Floating point (F, D, Q)[2] RV64/128 Base[3] Custom opcodes[4]

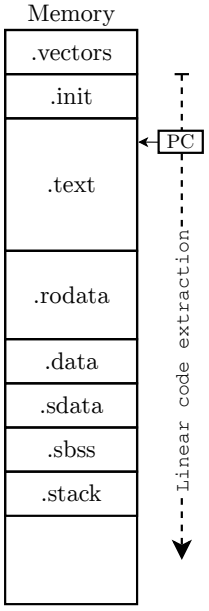

Fig. 7. Memory segmentation.

The memory segmentation of the Von Neumann RISC-V architecture used is depicted in Fig. 7. The memory is segmented into several sections. Instructions are stored in *.init* and *.text* sections, whereas the data is stored in *.rodata*, *.data*, *.sdata* and *.sbss*. For every code of the Embench suite, all instructions and then all data were successively pointed by the PC and retrieved. A Harvard architecture, with separate memories for instructions and data, or a limitation of PC range will obviously avoid this data extraction.

As noticed, enabling the reset signal of the instruction register led to an exception because the generated *0x0000* is not a NOP but rather an invalid instruction. As discussed, Editing the incoming instructions (II) in Section IV-A2, forcing bit 5 or 6 to 0 without using Floating Point extensions led to an exception.

The same LCE attack paths (I) and (II) were also emulated successfully on FPGA (as expected). Fig. 8 exemplifies the oscilloscope waveforms of the three least significant bits of the instruction bus for the Freezing (I) attack path. Bits 0 and 1 are at a high level, revealing the use of full 32-bit instructions. Bit 2 follows the instruction flow of the bus. Though the Tampering with the PC (III) succeeded by simulation, but failed on the microcontroller due to an error in the AXI bus (another attack path to tamper with the PC should be found).

The Experimental Setup exhibits the complete extraction of the instructions and data from memory by the three identified attack paths. The complexity in terms of the number of microprobes for every attack path was confirmed by simulations, and the exceptions suspected to occur were observed.

C. Attack path complexity

Carrying out an LCE is not always done under ideal condition. The starting address pointed by the PC could be at the end of the *.text* section. In that case, an iterative approach

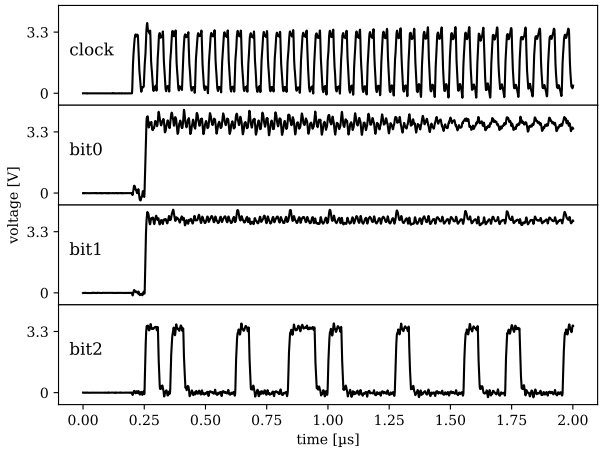

Fig. 8. Waveforms of the clock and the three bits of the instruction bus.

is possible by progressively delaying the start of the LCE (i.e., when specific signals are forced) until it reaches the execution of a DI jumping to one instruction at the beginning of the *.text* section. Delaying the start of the LCE is impossible by LASER or FIB editing of the specific signals instead, microprobes must be used. Note that such an iterative approach could also be used to defeat Kommerling's countermeasure presented in Section II-C6, based on PC partitioning [5] by retrieving the memory content piece by piece.

Therefore, in addition to eavesdropping microprobes, microprobes to induce a linear execution have to be set according to the attack path. The most common attack path in the state of the art: Editing the incoming instructions (II) means installing four microprobes on the instruction bus to induce a linear execution of a RISC-V core. Freezing a CI (I) requires placing only one microprobe on the cv32e40p which is easier to locate compared to Tampering with the PC (III). By enabling the cv32e40p Hardware Loop extension or improving the multiplexer design, additional microprobes are required in order to Tamper with the PC (III).

V. Countermeasures

Countermeasures against LCE are intended to be integrated into relatively simple embedded systems (e.g., inside ink cartridges). They have to be efficient while remaining lightweight.

A. Countermeasure description

Countermeasures must be able to handle the LCE attack paths revealed in Section IV. For the Freezing (I) and Editing (II) attack paths, the instructions coming from the memory are altered. As a result, a full software countermeasure is not possible as the countermeasure instructions could be erased or modified preventing their execution. Thus, the authors chose a hardware-based approach. Nevertheless, from the core point of view, during (I) or (II) LCE attacks, nothing wrong happens. The core receives and executes legal instructions. It cannot detect the occurrence of corrupted ones because it is not able to recognize a corrupted instruction. However, the core can detect some instructions are missing or have been tampered with. This is the operating principle of our first countermeasure.

1) Security Markers Monitoring (SMM): The SMM countermeasure consists of regularly inserting a custom instruction called Security Marker (SM) into the code, and then checking the steady presence of SMs after the instruction register. When an LCE is performed by Freezing (I) or Editing (II), incoming SMs are erased/edited and will be missing in the checking logic.

An SM monitoring component located after the instruction register is added to count the number of instructions executed between two SMs. Beyond a certain threshold, an LCE killer reaction is triggered. This threshold is configurable and defines the LCE detection latency of the countermeasure, it is called Detection Latency Target (DLT). An SM insertion component is added at the output of the program memory before the instruction bus. It inserts SMs on a regular basis, following the DLT. This insertion works by replacing the incoming instruction with an SM. In order to reread the replaced instructions from memory, SM execution must not increase the PC value. Incidentally, freezing an SM in the instruction register maintains the PC value and does not induce a linear execution. The SM insertion and monitoring components are depicted (in red) in Fig. 9.

The RTL descriptions of RISC-V open-source cores are editable which allows the insertion of components. The open-source RISC-V ISA is designed to easily add custom instructions like SM. The execution of SM created a clock cycle overhead. This overhead can decrease with the growth of DLT as expressed in 1. The cpi_{SMM} is the average of cycles per instruction in the executed code, and n is the number of needed cycles to execute an SM.

$$Clock\ cycle_{\ overhead} = \frac{1}{\frac{cpi_{SMM} \cdot DLT}{n} - 1} \quad (1)$$

2) Discontinuity Instruction Monitoring (DIM): The DIM is an enhanced version of SMM to reduce the clock cycle overhead by treating DI as SM. When an LCE is performed by Freezing (I) or Editing (II), every incoming DI is erased. Therefore, DIs are missing after the instruction register. The

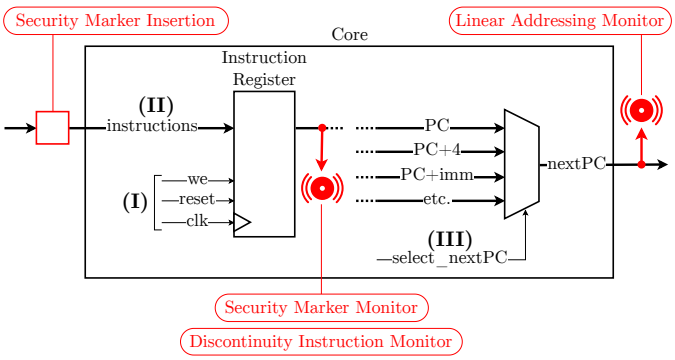

Fig. 9. Potential locations for the monitoring component of every countermeasure. SMs are inserted by a hardware component for SMM and DIM.

monitoring component counts the number of instructions executed between two SMs, two DIs or one SM and one DI. Therefore, inserting an SM is only necessary when there are not enough DIs (i.e., when a Basic Block (BB) is longer than the DLT). A BB is a straight-line code sequence without DIs until the end of the sequence.

Reducing the number of inserted SMs decreases the clock cycle overhead as expressed in (2). The clock cycle overhead depends only on the size $|BB_i|$ and the execution numbers e_i of every BB_i. Knowing the exact path taken on the Control Flow Graph for an execution is the only way to calculate the clock cycle overhead. The simulator framework presented in Section III makes it possible to measure the clock cycle overhead for several codes. The cyc_{ori} is the number of cycles needed to execute the original code.

$$Clock\ cycle\ _{overhead} = \frac{1}{cyc_{ori}} \cdot n \sum_i \left\lfloor \frac{|BB_i|}{DLT} \right\rfloor \cdot e_i \quad (2)$$

Checking that there are not too many consecutive incoming branches is a practical way to address an attack path that aims to freeze a branch with a false condition in the instruction register.

3) Linear Addressing Monitoring (LAM): The LAM monitors the length of linear execution sequences by analyzing PC updates. The LAM addresses the three identified attack paths to perform an LCE, including PC tampering (III). Only a monitoring component, depicted in Fig. 9, counts the number of cycles between two non-linear PC addressing (i.e., when the next PC is different from PC, PC+2 or PC+4 in RISC-V). Beyond a threshold, the LCE-killer reaction is triggered. This triggering threshold must be defined, considering the executed codes to never be exceeded in nominal execution. This threshold defines the detection latency, which is usually higher than the DLT of the previous countermeasures. As there is no need to insert any instruction, there is no clock cycle overhead. Many varieties of LAM can be designed, such as monitoring the driving signal of the PC multiplexer instead of the PC.

Theoretical characterizations of the three countermeasures are summarized in Table II, and are compared to the experimental ones in Section V-C. The area overhead is limited to a bit of combinational logic (measured at approximately one percent of cv32e40p in Section V-C) in addition to one counter for the LAM monitoring component. For SMM and DIM, two smaller counters are required: one for SM insertion and one for SM/DI monitoring components. These counters are smaller, as the DLT may be smaller than the LAM threshold.

A low DLT reduces the amount of extractable code before the triggering of the LCE-killer reaction. Clever access mechanisms which successively extract a small amount of code to rebuild the code are conceivable for high detection latency. However, with a small DLT, these clever access mechanisms are difficult to implement. The final instructions of the longest BBs will never be reached before the LCE-killer is triggered.

B. Further countermeasure issues

There are secondary issues to consider in order to better the full operability of countermeasures.

1) LCE-killer reaction: The LCE-killer reaction following detection is not in the scope of this paper. It has to stop the extraction. Basically, the reaction can be the same as for mesh or Probe Attempt Detector: causing a memory or circuit reset to avoid the access of memory-sensitive contents. A generation of null or random instructions from the memory is quite interesting to make attackers believe they succeed.

In order to avoid tampering with the reaction trigger signal, the pattern of "there is no LCE detected" has to be complex. Ideally, it should alternate between 0 and 1 in the nominal mode and at best it should be based on a lightweight ciphering as applied to active shield in [30]. Thus, circumventing our countermeasures will require more microprobes, a lot of LCE trial and error testing, and therefore more time.

2) Better to insert SMs on-the-fly than at the compilation: Instead of inserting SMs on-the-fly using a hardware component, the SMs could be inserted in the compilation flow. However, the code to be stored in memory is longer due to SMs added. Moreover, SMs must be inserted more often, as all edges of the Control-Flow Graph have to be considered, which ultimately increases the clock cycle overhead. On the other hand, the regularity of the on-the-fly insertion is the lowest one which doesn't exceed the monitoring counter threshold.

These three lightweight and hardware countermeasures are adaptable to every core architecture with an instruction bus and a PC (i.e., all existing cores). Depending on the threat, a specific use of these countermeasures is definable. For example, by using several instances of the DIM and LAM countermeasures to make the LCE very difficult to achieve.

C. Countermeasure practical evaluation

The Experimental Setup made it possible to assess the effectiveness of the countermeasures we created.

1) Countermeasure implementations: The three conceived countermeasures have been implemented in the cv32e40p core. Access links to their RTL description is provided in Section VII. Each CM makes use of a dedicated counter to monitor the core activity and to detect LCE attacks. Table III summarized when it is initialized, its initialization value, and when it is decreased. The role of the counter is to measure the time elapsed between two initialization events (expressed in executed instructions or clock-rising edges. For the LAM the initialization value was set to 105, as this is the minimal threshold value that did not trigger the alarm in nominal mode while testing the Embench codes. For the SMM and DIM countermeasures, ranging DLTs were tested by initializing the counter from 5 to 150, in order to measure the corresponding clock cycle overhead.

Decreasing the SMM and DIM counters at every executed instruction instead of every rising clock edge, allows disregarding instructions executed in several cycles. This leads to less frequent insertion of SMs without extending the detection latency. As the LAM is located on the PC bus, it should operate

TABLE II
THEORETICAL CHARACTERIZATION OF COUNTERMEASURES.

	Countermeasures	Detection latency	Attack path covered	Clock cycle overhead	Area overhead
SMM	Security Markers Monitoring	configurable (DLT)	(I) (II)	decreasing with DLT	negligible
DIM	Discontinuity Instruction Monitoring	configurable (DLT)	(I) (II)	decreasing sharply with DLT	negligible
LAM	Linear Addressing Monitoring	higher (code depending)	(I) (II) (III)	null	negligible

without the information that an instruction has been executed, thus it decreases at every rising clock edge. As explained in Section V-A1, an SM must initialize the counter and not increment the PC. To simplify the implementation, SMs are encoded as $JAL\ x0, 0$, since there is none in Embench codes.

TABLE III
COUNTER BEHAVIOR WITH REGARD TO THE CONSIDERED CM

	Initialized at every	Initialization value	Decreased at every
SMM	SM	DLT	instr. executed
DIM	SM or DI	DLT	instr. executed
LAM	nextPC $\notin$ {PC, PC+4}	105	rising clock edge

2) Practical characterization: Campaigns of simulated executions were following the effectiveness and the characterization claims of the three countermeasures. The Freezing (I) and Editing (II) attack paths were detected by the three countermeasures, while only LAM detected the PC tampering (III).

a) Detection Latency: The measured detection latencies met the DLT for SMM and DIM, and 105 for the LAM. In a nominal mode for LAM, the use of a threshold lower than 105 raised an alarm on at least one of the Embench codes. From the FPGA-based demonstration, the oscilloscope waveform of the reaction trigger signal raised during an LCE is plotted in Fig. 10. The LCE begins from the first rising clock edge and the DLT was set to 15 executed instructions. The detection between the beginning of the LCE and the alarm trigger is about 15 cycles, which is consistent, as instructions executed while an LCE is performed need only one cycle.

b) Clock cycle overhead: There is no clock cycle overhead for the LAM as it does not involve the insertion of SMs. Fig. 11 displays the clock cycle overhead of the SMM and DIM solutions for the Embench codes. It depends on the specificity of each code, as expressed in (1) and (2). Only some Embench codes are displayed for readability issues. As SMs — encoded as $JAL\ x0, 0$ — need two cycles to be executed ($n = 2$), both clock cycles overhead could be reduced by a factor of two if the core was edited to speed up the execution of SMs. The reduction of the clock cycle overhead using the DIM is noteworthy. The DIM graph reveals that codes like cubic or nbody contain BBs longer than 15 instructions which are executed plenty of times. The DIM countermeasure with a reasonable DLT makes the clock cycle overhead negligible (e.g., less than 1% for a DLT of 25 instructions executed).

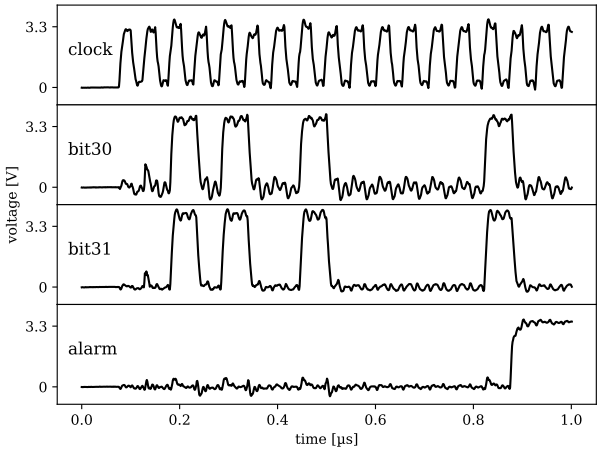

Fig. 10. Alarm signal raised during an LCE.

The effect of DLT on the average clock cycle overhead is drawn in Fig. 12. The cpi_{SMM} is approximated to 1. The clock cycle overhead is basically decreasing for a growing DLT. The SMM average overhead matches the theoretical estimation from (1).

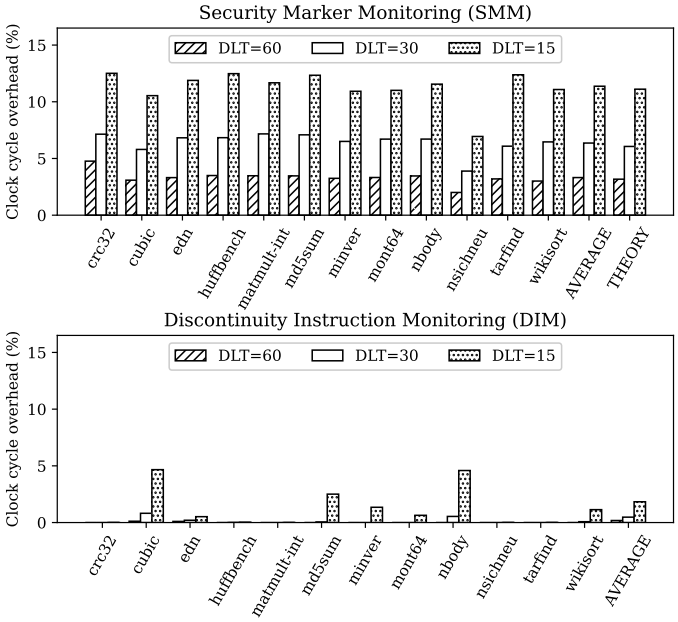

Fig. 11. Clock cycles overhead depending on the DLT for Embench codes.

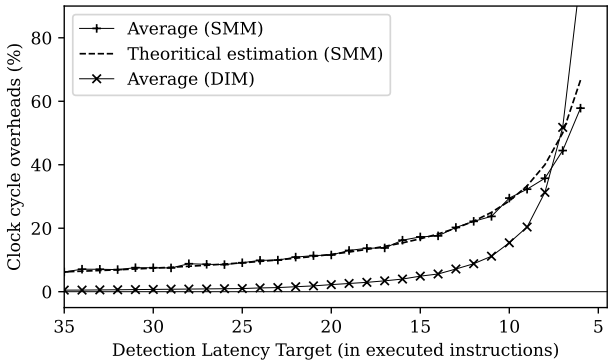

Fig. 12. Clock cycles overhead depending on the DLT.

Additional component logic increases the core logic propagation delays, which may decrease the maximum clock frequency. In the FPGA design flow, after implementation, the Worst Negative Slack was reduced by 2.07% compared to the unprotected microcontroller (from 30.812 ns to 30.173 ns). Countermeasures may therefore slightly limit the maximum frequency of the clock.

c) Area overhead: The area overhead is lightweight but depends on the DLT. The slice LUTs and Registers utilization measured for two DLT values in the FPGA design flow, after implementation, are given in Table IV. The LAM includes only a monitoring component with a DLT of 105 (equivalent to 127), which creates a 0.52% LUT overhead. It is lighter than SMM and DIM. Indeed, those include monitoring and hardware insertion components. Their DLT may be as small as 15, which created a 1.47% LUT overhead. Nonetheless, the increased area overhead for any of the three countermeasures is very low. As a guide, the core-v-mcu microcontroller contains around 48,900 LUTs.

3) Discussions: The authors did not report on the effect of RISC-V extension on the countermeasures design. They did not identify any predictable incompatibility (though it could be ascertained with the proposed simulation framework).

Only the compressed extension challenges the hardware insertion of SMs. The insertion requires replacing the 32 incoming bits with an SM. The risk is to partially replace two incoming unaligned instructions, by an aligned SM, which leads to the creation of two undesired instructions. Information about PC alignment from the core or the memory is needed to address this issue.

The SMM and DIM countermeasures could be enhanced to address the Tampering with the PC (III) attack path, by implementing an original jump or kill mechanism. It would require adding in the code successively: an instruction that jumps over the following instruction and a custom kill instruction which initiates the LCE-killer process if executed. If the jump instruction is not properly executed due to Tampering with the PC (III), then the kill instruction is executed and it triggers the LCE-killer reaction. Implementation and tests could be viable and interesting avenues for future studies.

TABLE IV
COUNTERMEASURE COMPONENTS UTILIZATION OVERHEAD COMPARED TO THE UNPROTECTED CV32E40P.

Slice	Unprotected cv32e40p	DLT=15				DLT=127			
		Hardware insertion		Monitor component		Hardware insertion		Monitor component	
LUTs	4439	46	1.04%	19	0.43%	55	1.24%	23	0.52%
Registers	2132	5	0.23%	10	0.47%	8	0.38%	13	0.61%

VI. CONCLUSION

LCE is an invasive attack aiming at fully extracting a code from a device's memory for reverse engineering purposes. The threat is real and routinely exploited by manufacturers of compatible products (e.g., ink cartridges or gamepad controllers) who can fund the required expensive FA tools. The first contribution of this work is to reveal types of LCE attack paths to induce a linear read of the memory of a targeted RISC-V core. Three attack paths were introduced, notably two of which are linked to vulnerabilities of the instruction register and the PC circuitry. The vulnerability based on instruction editing appears more as a theoretical threat than a practical one because it requires tampering with and forcing at least four bits of the instruction bus. The increase in the number of microprobe-required renders this attack path impractical. These vulnerabilities were assessed through simulations or FPGA-based demonstration thanks to our proposed Experimental Setup (both frameworks we developed are made available on an open-access basis).

To address LCE threats on embedded devices, lightweight countermeasures were introduced. The original approach of monitoring the LCE symptoms leads to an area overhead around 1% of the unprotected cv32e40p core. A widespread deployment on many embedded devices is therefore conceivable. Thanks to the configurable DLT, countermeasures can be adapted to every use case. For example, by decreasing the clock cycle overhead to less than 1% with a DLT lower than 25 instructions executed for DIM. The clock cycle overhead is null for LAM. All these countermeasures were implemented, successfully validated, and characterized by the Experimental Setup, in simulations and FPGA-based demonstration.

VII. AVAILABILITY

The RTL description of the three countermeasures integrated in the cv32e40p core and the Experimental Setup are available (https://github.com/theophile-gousselot/linear_code_extraction). Additional documentation and step-by-step tutorial help to better understand the Experimental Setup and to reproduce the results of this paper. Moreover, original attack paths and countermeasure variants can be tested.

REFERENCES

[1] S. Skorobogatov, "Physical attacks and tamper resistance," in *Introduction to Hardware Security and Trust*. Springer, 2012, pp. 143–173.
[2] C. Loebbecke and M. Fischer, "Pay TV piracy and its effects on Pay TV provision," *Journal of Media Business Studies*, vol. 2, no. 2, pp. 17–34, 2005.
[3] Apex Microelectronics, "Compatible ink cartridges selling by Apex Microelectronics ," 2022.

[4] S. P. Skorobogatov, "Semi-invasive attacks: a new approach to hardware security analysis." *University of Cambridge*, 2005.

[5] O. Kömmerling and M. G. Kuhn, "Design Principles for Tamper-Resistant Smartcard Processors." *Smartcard*, vol. 99, pp. 9–20, 1999.

[6] C. Helfmeier, D. Nedospasov, C. Tarnovsky, J. S. Krissler, C. Boit, and J.-P. Seifert, "Breaking and entering through the silicon," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 733–744.

[7] R. Anderson, "Security engineering: a guide to building dependable distributed systems," *John Wiley & Sons*, 2020.

[8] M. Weiner, S. Manich, R. Rodríguez-Montañés, and G. Sigl, "The low area probing detector as a countermeasure against invasive attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 2, pp. 392–403, 2017.

[9] A. Vijayakumar, V. C. Patil, D. E. Holcomb, C. Paar, and S. Kundu, "Physical design obfuscation of hardware: A comprehensive investigation of device and logic-level techniques," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 1, pp. 64–77, 2016.

[10] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov, "Cryptographic processors-a survey," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 357–369, 2006.

[11] S. Vasile, D. Oswald, and T. Chothia, "Breaking all the things - A systematic survey of firmware extraction techniques for IoT devices," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2018, pp. 171–185.

[12] C. Bozzato, R. Focardi, and F. Palmarini, "Shaping the Glitch: Optimizing Voltage Fault Injection Attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 2, pp. 199–224, Feb. 2019.

[13] H. H. Yap and Z. J. Lau, "Delayering techniques: dry/wet etch deprocessing and mechanical top-down polishing." *Microelectronics Fialure Analysis Desk Reference*, p. 379, 2019.

[14] "Ink-cartridge Market Research Report by Type, Application, Region - Global Forecast to 2027," 2022.

[15] S. Hamm, "Rivals say HP is using hardball tactics," *Business Week*, pp. 48–49, 2007.

[16] I. Distribution, "G&G & Ninestar Corporation," 2019.

[17] B. C. Mank, "Prudential Standing Doctrine Abolished or Waiting for a Comeback: Lexmark International, Inc. v. Static Control Components, Inc." *U. Pa. J. Const. L.*, vol. 18, p. 213, 2015.

[18] D. G. Abraham, G. M. Dolan, G. P. Double, and J. V. Stevens, "Transaction security system," *IBM systems journal*, vol. 30, no. 2, pp. 206–229, 1991.

[19] R. Anderson and M. Kuhn, "Tamper resistance-a cautionary note," in *Proceedings of the second Usenix workshop on electronic commerce*, vol. 2, 1996, pp. 1–11.

[20] S. Skorobogatov, "How microprobing can attack encrypted memory," in *2017 Euromicro Conference on Digital System Design (DSD)*. IEEE, 2017, pp. 244–251.

[21] OpenHW Group, "CV32E40P RISC-V IP," 2022.

[22] OpenHW Group, "CORE-V-VERIF: Functional verification project," 2022.

[23] Veripool, "Fast and open-source Verilog/SystemVerilog simulator," 2022.

[24] IEEE, "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[25] Embench, "Embench: Open Benchmarks for Embedded Platforms," 2022.

[26] OpenHW Group, "CORE-V-MCU: microcontroller," 2022.

[27] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213," *RISC-V Foundation*, December 2019.

[28] ARM, "Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile," 2019.

[29] K. N. P. Maier, "Low-cost chip microprobing," 2012, 29th Chaos Communica- tion Congress (29C3).

[30] X. T. Ngo, J.-L. Danger, S. Guilley, T. Graba, Y. Mathieu, Z. Najm, and S. Bhasin, "Cryptographically secure shield for security IPs protection," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 354–360, 2016.