Physical Model-Assisted Secure Software Execution Against Fault-Injection Attacks

Henian Li, Md. Habibur Rahman, Arunabho Basu, Rakibul Hassan, Farimah Farahmandi, Mark M. Tehranipoor
Department of Electrical and Computer Engineering
University of Florida, Gainesville, Florida, 32611
Email: {henian.li, rahman.mdhabibur, arunabhobasu, rhassan1}@ufl.edu, {farimah, tehranipoor}@ece.ufl.edu

***Abstract*—Modern software running on vulnerable circuits such as SoCs and processors is increasingly susceptible to cyber and hardware attacks. While protections against cyberattacks are widely integrated across the software development and compilation cycle, physical attacks such as fault-injection attacks directly target hardware implementations. Such attacks bypass upper-level protections and compromise the confidentiality, integrity, and availability of modern chips. Since the hardware has already been fabricated, the cost of redesign is extremely high, hence hardening software/firmware against the identified physical vulnerabilities would be a recommended and less costly approach. However, existing compilers lack considerations and hardening techniques for physical attacks. In this paper, we address physical threats from fault injections at the software compilation stage. Our approach models fault-injection attacks based on physical design characteristics, facilitating secure software compilations with low-cost and effective physically-aware hardening techniques.**

***Index Terms*—fault injection; physical attacks; fault model; compiler; software countermeasures**

I. Introduction

With the emergence of the Internet of Things (IoTs) regime that promises exciting new applications from smart cities to connected autonomous vehicles, security and privacy have emerged as major design challenges. These advancements mean that modern software applications running on vulnerable circuits such as SoCs and processors are facing heightened risks from both cyber and hardware attacks. Traditionally, the software has been protected against cyberattacks integrated across the software development and compilation cycle [1]. However, physical attacks (e.g., fault injection attacks) could effectively bypass these upper-level software protections, threatening the system's confidentiality, integrity, and availability.

Physical attacks, such as fault-injection attacks, directly exploit the hardware's physical properties rather than the software's logical vulnerabilities. For example, fault-injection can intentionally alter the state or physical behavior of a circuit using a clock glitch or laser-induced photocurrent shown in Fig. 1, cause a variety of disruptions in a processor (such as memory value modifications, instruction skipping, or calculation errors) [2], and facilitate leaking secrets of a system. Fault-injection can be non-invasive (e.g., clock glitching, voltage glitching, optical), semi-invasive (e.g., local heating and laser), or invasive (e.g., focused ion beam and probing), which can be carried out by a variety of techniques and instruments with different cost and precision. Different forms of fault-injection attacks have been successfully demonstrated by researchers as well as practitioners in the industry on many applications, and almost all platforms, such as microcontrollers [2], SoCs, FPGA-based embedded systems, and IoT devices are vulnerable to such attacks. In today's landscape, properly identifying and mitigating hardware vulnerabilities could reduce overall system vulnerabilities and software attacks by up to 43% [3].

Unlike the awareness and sophisticated protections against cyberattacks, current software developers lack information on physical attacks. Besides, existing compilers such as GCC and Clang lack considerations and hardening techniques for physical attacks. Although there are a few explorations in the literature on software code simulations and hardenings against hardware attacks [4], these efforts do not consider realistic physical attack parameters and models. As a result, only a limited number of physically vulnerable scenarios can be captured by such software-side predictions. Because the vulnerable hardware has already been fabricated, it is inaccessible to software developers and it costs highly to do a redesign, exposing the software to risks from physical attacks.

Our Contributions: In this paper, for the first time to the best of our knowledge, we propose a cost-effective and scalable approach for addressing physical threats from fault injections at the software compilation stage. Our method leverages fault-injection attack modeling based on physical design characteristics, enabling secure software compilation through precise and effective physically-aware hardening techniques.

II. Background

A. Fault-Injection Attack

Fault-injection techniques could maliciously alter the correct functionality of a computing device: In the case of non-invasive attacks, which are inexpensive and more frequent, one can perform clock or voltage glitching attacks and electromagnetic (EM) attacks. For semi-invasive attacks, one can apply optical fault injection techniques. Non-invasive fault-injection attacks include a clock or voltage glitching, while semi-invasive attacks such as laser fault-injection possess

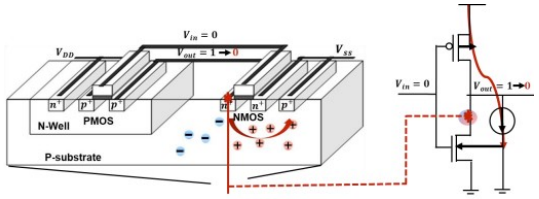


Fig. 1: Laser causing a single-event upset (SEU) [5].

precise temporal and spatial control. Fig. 1 illustrates when the laser illuminations on an inverter excite extra electron-and-hole pairs [5], and these generated electron-hole pairs would create a current pulse through the transistor, and further, create a voltage pulse (a transient fault) to propagate in the circuit. Researchers have demonstrated many practical fault injection attacks using a laser-based technique, such as flipping any individual bit in static random-access memory (SRAM) arrays [6] and extracting key values from cryptographic systems [7].

B. Compiler Hardening

In the existing cybersecurity industry and community, compiler hardening encompasses techniques designed to enhance the security and stability of compiled code, particularly in programming languages like C and C++ that are susceptible to memory safety and control flow integrity violations. Research from Google highlights that nearly 70% of software vulnerabilities stem from memory safety issues [8]. Addressing and mitigating these vulnerabilities during the compilation stage could prevent such issues from becoming executable in the final compiled code.

Modern software compilers detect and address vulnerabilities during compilation by enabling specific flags corresponding to common weaknesses in the code. While considerable progress has been made to identify and mitigate software vulnerabilities during compilation stages, existing compilers like GCC and Clang lack robust techniques to address physical attacks, with many conventional software-level countermeasures proving ineffective or even detrimental to security [1], [9]. Simple fault-injection methods, such as clock glitching attacks, can bypass even well-established software defenses.

C. Related Work

While some software-level physical-attack models exist—such as high-level abstractions describing fault effects (e.g., instruction skipping or calculation errors) and bit-flip simulations in instruction-level or microarchitecture-level [10] environments—they often lack precision in capturing real-world physical attack complexities. These models fail to account for microarchitectural states invisible to programmers or faults occurring during instruction execution. For instance, fault-injection attacks reveal how low-level faults impact multiple architectural states, going beyond what software models can accurately predict. To address such challenges, our work investigates the potential to integrate physically-aware hardening measures into the compilation process, bridging the gap between software-level protections and physical attack models while avoiding costly call-outs for post-fabrication chips.

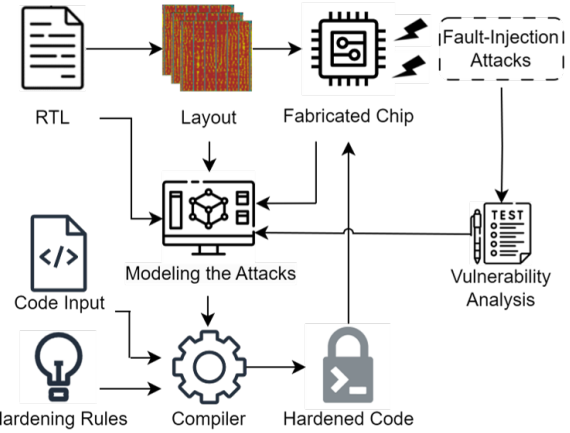


Fig. 2: Physical model-assisted secure software compilation.

III. Physically-aware Software Compilation

We developed a framework, as shown in Fig. 2, to harden the software code execution against fault-injection attacks at the code compilation stage. The overall flow consists of two major steps: 1) Fault-injection modeling and 2) code hardening.

A. Fault-Injection Modeling

Developing a comprehensive modeling methodology for existing physical attack techniques is crucial to ensure fast, reliable, and accurate software execution, even in the presence of sophisticated physical attacks. In this step, we model the logical and physical behavior of laser fault-injection attacks. The models' output, represented as analog variations, will help establish a criticality and attack feasibility ranking of hardware components under various attack scenarios. This information will guide the software compiler in prioritizing and applying appropriate hardening strategies, thereby mitigating such attacks and eliminating the need for call-outs of fabricated chips.

We aim to characterize the criticality of faults by examining factors such as fault controllability, timing, the number of concurrent faults (single vs. multiple), fault type (e.g., direct bit-flip, timing fault), and fault duration. In addition to fault criticality, we will also model fault feasibility by simulating attack parameters like current demand, output voltage distortion, IR drop, and propagation delay variations using SPICE simulations.

We model the analog impacts of laser fault-injection attacks and correlate the physical model characteristics with the software. Through SPICE simulations, we model the physical behavior of the attacks as laser-induced current injections within impacted standard cells, therefore, the design's vulnerabilities can be determined and ranked by observing the current demand variations at VDD/VSS pins. The laser-induced current on impacted cells can be modeled as a double exponential current source, where $I(t)$ calculated by Equation 1 [11]:

$$I(t) = \frac{Q}{\tau_\alpha - \tau_\beta}(e^{-\frac{t}{\tau_\alpha}} - e^{-\frac{t}{\tau_\beta}}) \quad (1)$$

where Q is the number of charges deposited by the particle strike, τ_α a process-dependent collection time constant of the junction, τ_β the constant of the ion-track establishing time.

B. Hardening Code Compilations

We perform three major steps to integrate physical attack models into the compilation process for software hardening:

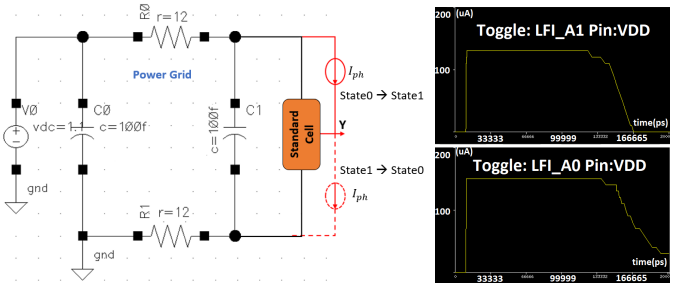
Code feature extraction: Vulnerability varies across code segments based on their context, such as conditional checks in password authentication or loop bounds in cryptographic algorithms like AES. We identify critical code sections, observe active hardware resources during these sections' executions, and detect potentially vulnerable hardware components.

Security property development: Security properties establish clear and precise security margins for the threat scenarios under investigation. These properties define critical aspects such as the protected security asset (e.g., specific instructions or hardware signals), the timing of when constraints on these assets should be verified, and the required behavior the assets must exhibit. We explicitly define security properties using SystemVerilog to protect vulnerable code sections with minimal overhead. For instance, in AES encryption, we will focus on rounds particularly susceptible to Differential Fault Analysis (DFA) or side-channel attacks, securing critical intermediate states and keys during these stages.

Integration of hardening techniques: A database of hardening rules will be created and integrated into the compilation process. We perform two key steps: (1) evaluate and test existing hardening techniques, such as redundant conditional checks, random execution delays, and redundant data storage and memory operations. However, these techniques are proven to be inadequate for many critical physical-attack scenarios [9], hence, we will (2) develop innovative hardening rules informed by insights from our hardware simulations and security property database. This approach aims to provide more targeted and effective countermeasures with reduced overhead. The above-mentioned hardening rules will be combined with the physical modeling data obtained from the previous step to guide the compiler's hardening strategy, ensuring that they are grounded in realistic attack scenarios and can address the specific vulnerabilities identified in the earlier steps. We develop a scalable hardening-rule database including instruction duplication, rearrangements of branch conditions, etc. These rules, combined with physical modeling information, are implemented in the form of GCC plug-ins to be integrated into regular software compilations. The final output is physically hardened assembly codes.

IV. Experimental Results

This section provides the results of laser fault-injection attack modeling and the hardened code. We evaluate the hardened codes' resiliency on RISC-V processors [12]. We perform iterative tests of hardened software compilations, broaden the attack scenarios by including more security properties, and fine-tune the physical attack models. This evaluation is conducted in two primary stages. First, we prepare three groups of designs: (1) a general-purpose core/SoC executing unprotected software test cases, (2) the same hardware executing software hardened by existing cybersecurity features in compilers like GCC, and (3) the same hardware executing the physically hardened software developed from our proposed flow. We then check if the software execution could pass all security checks as defined in security properties. This evaluation is based on pre-silicon simulated fault-injection attacks, which is performed by inserting attack points and applying a fault list for fault simulation tools, such as Synopsys Z01X [13]. We developed hardening techniques on top of the RISCV-GCC compiler toolchain in Ubuntu 22.04 LTS, and used Cadence Spectre for SPICE simulations and laser modeling on Cadence 45 nm library.



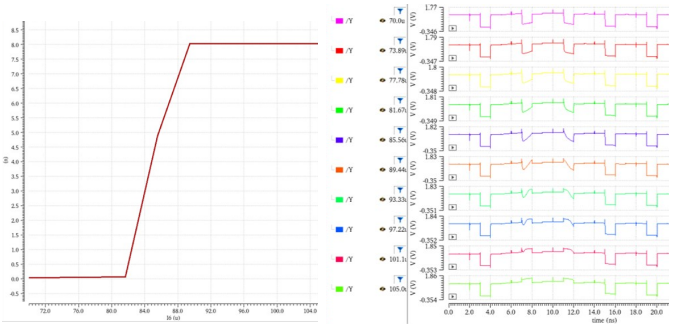
(a) (b)

Fig. 3: (a) SPICE circuit for simulating cell under laser illumination and capturing current demands, (b) inverter cell-level power model capturing currents on power rails during laser illumination.

To simulate the behavior of standard cells under laser illumination, we constructed a SPICE circuit, as shown in Fig. 3a. This setup provided a simplified yet effective means of analyzing the impact of laser fault injection on complex standard cells. Every standard cell is simulated for laser illumination when in State0 and State1, and the currents on VDD and VSS rails are captured for both states. These currents on power rails during laser illumination are recorded for all the standard cells in the library. Fig. 3b shows the current profiles at the VDD rail for the inverter's two states (i.e., LFI State0 and LFI State1) when at the center of the 2W laser spot.

Table I summarized the standard cells with the highest activities from the password-checking software. The second and third columns are the high-to-low delays without and with laser input, respectively. The delay increases while scaling up the laser intensity because the laser-induced current slows the voltage transitions, as shown in Fig. 4a. The third and fourth columns facilitate determining the threshold current for output distortion, which serves as a quantitative indicator to measure each cell's vulnerability level. Fig. 4b illustrates this trend of output distortions on AND2X1 cell when the laser intensity swept from 70 μA to 120 μA.

Building on the observations from the laser fault model,



(a) (b)

Fig. 4: Delay (a) and voltage (b) responses on AND2X1 standard cell when scaling the laser-induced current.

we identified critical vulnerabilities in the password-checking code. Fig. 5a illustrates a highly sensitive if-condition branch that depends on hardware resources previously analyzed for vulnerability. The assembly-level code in Fig. 5b highlights the specific instructions (in red) that are prone to laser-induced faults, such as skipping the line *bne a0, zero, .L3*, which bypasses password verification altogether. This would result in the execution of the *secret_function* regardless of the entered password, a severe security risk. Notably, this vulnerability persists even with all built-in GCC security flags enabled (Fig. 5c, yellow highlights), underscoring the limitations of existing compiler protections.

However, our proposed hardening approach effectively mitigates such vulnerabilities. As shown in Fig. 5d, the hardened code reorganizes the vulnerable section (highlighted in green) to ensure security even under fault-injection attacks. For instance, skipping the critical line *beq a0, zero, .L3* merely branches the program to a scenario where an incorrect password is detected, preserving the system's integrity. This transformation, achieved through our compiler plugin, demonstrates the potential of leveraging physical attack models to produce robust, fault-resistant code that addresses previously overlooked vulnerabilities.

V. Conclusion

In this paper, we address physical threats from fault injections at the software compilation stage. Our approach models fault-injection attacks based on physical design characteristics, facilitating secure software compilations with low-cost and effective physically-aware hardening techniques. Our experimental results demonstrate that a critical conditional branch in a password-checking application could be protected against

TABLE I: Output delay variations and laser current values when signal distortions start and end.

Gates	t_d (ps)	t_{d_laser} (ps)	I_{start} (μA)	I_{end} (μA)
AND2X1	11.94	22.52	130	146
OR2X1	27.35	50.12	119	131
AO21X1	22.80	40.95	122	134
NAND2X1	13.09	27.25	85	95
NOR2X1	8.71	38.22	116	147
INVX1	6.05	16.67	116	168

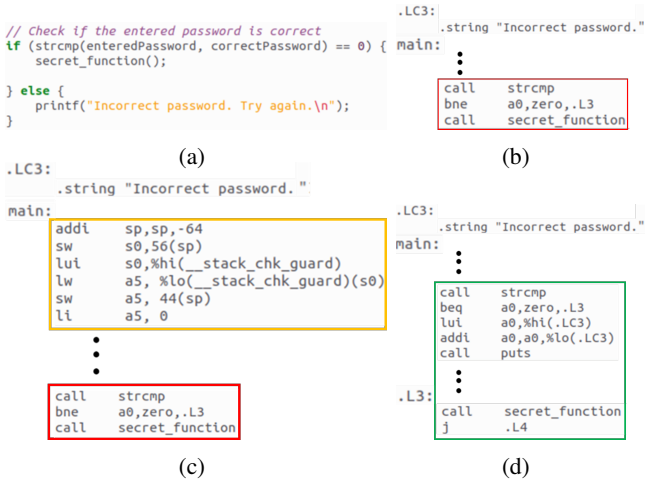


Fig. 5: (a) source code in C, (b) assembly code compiled by GCC-RISCV, (c) code compiled with all GCC built-in security flags enabled, (d) hardened code from our proposed flow.

laser fault-injection attacks by reorganizing the assembly instructions during the compilation stage. In the future, we plan to improve the precision of our current modeling of laser fault injection, model other categories of physical attacks such as clock/voltage glitching and side-channel attacks, and extend the hardening database for this physically-aware compiler.

References

[1] GCC, the GNU Compiler Collection, "Program instrumentation options," https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html.

[2] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2013, pp. 77–88.

[3] T. M. Corporation, "Cve program mission." [Online]. Available: https://www.cve.org

[4] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "FISSC: a fault injection and simulation secure collection," in *SAFECOMP 2016 - The 35th International conference on Computer Safety, Reliability and Security*, vol. 9922, 2016, pp. 3–11.

[5] T. Farheen, S. Tajik, and D. Forte, "Spred: Spatially distributed laser fault injection resilient design," in *2023 24th International Symposium on Quality Electronic Design (ISQED)*, 2023, pp. 1–8.

[6] M. Agoyan, J.-M. Dutertre, A.-P. Mirbaha, D. Naccache, A.-L. Ribotta, and A. Tria, "How to flip a bit?" in *2010 IEEE 16th International On-Line Testing Symposium*, 2010, pp. 235–239.

[7] J.-M. Schmidt and M. Hutter, *Optical and em fault-attacks on crt-based rsa: Concrete results*. Proc. Austrian Workshop Microelectron, 2007.

[8] C. Cimpanu, "Chrome: 70% of all security bugs are memory safety issues." [Online]. Available: https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/

[9] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a risc-v processor," *Microprocessors and Microsystems*, vol. 71, p. 102862, 2019.

[10] A. Gangolli, Q. H. Mahmoud, and A. Azim, "A systematic review of fault injection attacks on iot systems," *Electronics*, vol. 11, no. 13, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/13/2023

[11] F. Lu, G. D. Natale, M.-L. Flottes, B. Rouzeyre, and G. Hubert, "Layout-aware laser fault injection simulation and modeling: From physical level to gate level," in *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era*, 2014, pp. 1–6.

[12] Yosys HQ, "Picorv32 - a size-optimized risc-v cpu." [Online]. Available: https://github.com/YosysHQ/picorv32

[13] Synopsys, "Z01X," https://www.synopsys.com/verification/simulation/z01x-functional-safety.html.