Formal verification of processor microarchitecture to analyze system security against fault attacks

Simon Tollec

université
PARIS-SACLAY

Formal Verification of Processor Microarchitecture to Analyze System Security against Fault Attacks

Vérification formelle de la micro-architecture de processeurs pour l'analyse de sécurité des systèmes contre les attaques par injection de fautes

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580 : Science et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique
Référent : Faculté des sciences d'Orsay

Thèse préparée à **l'institut LIST (Université Paris-Saclay, CEA)**, sous la direction de **Mathieu JAN**, directeur de recherche, le co-encadrement de **Karine HEYDEMANN**, maîtresse de conférences, de **Mihail ASAVOAE**, ingénieur-chercheur, et de **Damien COUROUSSÉ**, ingénieur-chercheur.

Thèse soutenue à Paris-Saclay, le 15 novembre 2024, par

Simon TOLLEC

Composition du jury

Membres du jury avec voix délibérative

Emmanuelle ENCRENAZ-TIPHENE Professeure des universités, Sorbonne Université	Présidente
Vincent BEROULLE Professeur des universités, Université Grenoble Alpes	Rapporteur & Examinateur
Jean-Max DUTERTRE Professeur des universités, École des Mines de Saint-Étienne	Rapporteur & Examinateur
Guillaume BOUFFARD Ingénieur de recherche, Agence nationale de la sécurité des systèmes d'information (ANSSI)	Examinateur
Guillaume HIET Professeur des universités, Centrale Supélec	Examinateur
Ulrich KÜHNE Maître de conférences, Télécom Paris	Examinateur
David MONNIAUX Directeur de recherche, Université Grenoble Alpes	Examinateur

THÈSE DE DOCTORAT
NNT : 2024UPASG077

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Vérification formelle de la micro-architecture de processeurs pour l'analyse de sécurité des systèmes contre les attaques par injection de fautes.

Mots clés: Attaques par injection de fautes · Vérification formelle · Processeur · Contremesures · Model Checking · SAT.

Résumé: Les attaques par injection de fautes représentent une menace majeure pour la sécurité des systèmes, car elles permettent aux attaquants de déjouer des mécanismes de protection ou d'accéder à des informations sensibles. Alors que la sécurité de ces systèmes est traditionnellement évaluée au niveau du logiciel ou du matériel, des recherches récentes soulignent la nécessité de prendre en compte les deux niveaux et d'analyser la microarchitecture du processeur pour comprendre pleinement les conséquences des attaques par injection fautes.
Dans ce contexte, cette thèse vise à élaborer une méthodologie d'analyse exhaustive et automatisée, prenant en compte à la fois les descriptions logicielles et matérielles du système, afin d'avoir une évaluation fine des conséquences des fautes sur le logiciel ou de fournir des garanties formelles sur la sécurité du système. À cette fin, nous proposons µArchiFI, une méthodologie formelle de modélisation et de vérification permettant d'évaluer les effets des fautes sur les systèmes combinés matériel/logiciel. Contrairement aux méthodologies existantes, µArchiFI est exhaustif et permet l'identification automatique des vulnérabilités difficiles à détecter, ainsi que la preuve de robustesse des systèmes contre les attaques par injection de fautes. Implémenté à partir de l'infrastructure de compilation Yosys, notre approche génère une modélisation du système adaptée aux techniques de vérification formelle telles que le model checking borné. Nous validons notre méthodologie sur les processeurs RISC-V en identifiant automatiquement les attaques connues exploitant les mécanismes microarchitecturaux et en découvrant des effets de fautes jusqu'alors inconnus que les techniques de simulation classique pourraient manquer. De plus, nous évaluons formellement la sécurité d'une contre-mesure conjointe logiciel et matériel MAFIA, tâche qui ne n'aurait pas été possible en ne travaillant uniquement qu'à l'un de ces niveaux.
Pour améliorer les performances de notre approche et résoudre le problème de l'explosion de l'espace d'état, l'un des défis majeurs des techniques exhaustives, nous décomposons la co-vérification matériel/logiciel en étapes plus faciles à résoudre. Cette décomposition s'appuie sur une évaluation préliminaire des contre-mesures matérielles potentielles existantes. Par conséquent, nous démontrons que des problèmes auparavant insolubles, tels que l'analyse de la robustesse de l'élément sécurisé OpenTitan exécutant un processus de démarrage sécurisé, peuvent désormais être résolus grâce à notre méthodologie. Notre approche a également identifié des vulnérabilités dans le banc de registres, pour lesquelles nous avons fourni et prouvé un correctif de sécurité qui a ensuite été intégré dans le projet OpenTitan.

Title: Formal Verification of Processor Microarchitecture to Analyze System Security against Fault Attacks.

Keywords: Fault injection attack · Formal verification · Processor · Countermeasures · Model Checking · SAT.

Abstract:
Fault injection attacks are a serious threat to system security, enabling attackers to bypass protection mechanisms or access sensitive information. While the security of these systems is traditionally assessed at the software or hardware level, recent research highlights the need to consider both and analyze the processor microarchitecture to fully understand the consequences of fault attacks.
In this context, this thesis introduces an exhaustive and automated analysis technique, comprising both software and hardware system descriptions, to better understand the final consequences of hardware-level faults on software and provide formal guarantees of software security. For this purpose, we propose μArchiFI, a formal modeling and verification methodology to evaluate fault effects on combined hardware/software systems. Built on top of the Yosys compiler, this tool generates a system model suitable for formal verification techniques such as bounded model checking. Unlike previous methodologies, μArchiFI is exhaustive and allows for automatic identification of corner-case vulnerabilities, as well as proving system robustness against fault attacks. We validate our methodology on RISC-V processors by automatically identifying known fault attacks exploiting microarchitectural mechanisms and by discovering previously unreported fault effects that existing simulation-based techniques might miss. Additionally, we formally evaluate the security of the combined countermeasure MAFIA, something that would not be possible through hardware or software verification alone.
To improve performance and address the state space explosion problem—one of the most significant challenges of exhaustive techniques—we decompose hardware/software co-verification into more manageable steps. This decomposition leverages a preliminary evaluation of potential hardware-level countermeasures. Consequently, we demonstrate that previously intractable problems, such as analyzing the robustness of the OpenTitan secure element running a secure boot process, can now be solved using our methodology. Our approach also identified vulnerabilities in the register file, for which we provided and proved a security fix before integrating it into the OpenTitan project.

Remerciements

Après avoir rédigé ce manuscrit de thèse de 120 pages, il n'est pas facile de sortir de son rôle de scientifique pour prendre la plume et partager l'expérience que cela a représenté. Cependant, il me tient à cœur de remercier toutes les personnes qui ont contribué à l'accomplissement de cette thèse, enrichi mon quotidien et rendu cette période, entre le monde académique et le monde professionnel, aussi formatrice qu'enrichissante.

Je tiens tout d'abord à exprimer ma profonde gratitude à l'équipe encadrante de cette thèse, qui m'a chaleureusement accueilli lors de mon stage au CEA, et avec qui j'ai choisi de prolonger l'aventure sur un sujet à la fois passionnant et complexe en raison des nombreuses thématiques qu'il recoupe. Mihail Asavoae, Damien Couroussé, Karine Heydemann, Mathieu Jan, je vous remercie pour votre accompagnement et votre expertise, qui ont largement contribué à la réussite de cette thèse et ont grandement contribué à mon développement tout au long de ces trois années.

Je souhaite également remercier tous mes collègues du CEA, chercheurs et doctorants, dont la richesse des échanges, parfois informels autour de la machine à café, a marqué ces trois années. Sans vous, cette aventure aurait été bien plus solitaire. Un merci particulier à notre groupe de doctorants : Benjamin Binder, qui m'a ouvert la voie dans l'exercice de la thèse, mais aussi Pierre-Emmanuel Clet, Jonathan Fontaine, Valentin Gilbert, Robin Ollive, Marc Renard, Julien Rodriguez, Guillaume Roumage, et tant d'autres, avec qui j'ai partagé aussi bien les bons moments que les défis et difficultés de la thèse. Au-delà du travail, nous avons su trouver des moments de respiration, que ce soit lors de nos sorties d'escalade, au bar, au restaurant, lors de notre excursion au CERN, d'un périple au Mont-St-Michel, ou encore sur les pistes de ski. Je souhaite par ailleurs bonne chance à la prochaine génération de doctorants.

Je tiens à remercier le CEA d'avoir accueilli ma thèse, ainsi que ma hiérarchie pour son soutien et pour m'avoir permis de participer à de nombreux événements : conférences, écoles d'été, séminaires et mobilités internationales. Je remercie tout particulièrement Yann Gallais, mon chef de laboratoire, pour avoir facilité mon projet de visite scientifique à l'Université technologique de Graz au cours de ma deuxième année de thèse.

I would like to extend my thanks to the entire IAIK team at TU Graz for their warm welcome and for the collaboration that led to the publication of an excellent paper at the CHES conference. I am especially grateful to Prof. Stefan Mangard for offering me the opportunity to join the SESYS team for three months. This experience was deeply enriching, both scientifically and in terms of international collaboration. I highly recommend such an experience to any PhD student. A special thank you goes to thank Pascal Nasahl, who generously found time to collaborate with me despite his

busy schedule at the end of his thesis. I wish you all the best in your new chapter in Switzerland, and I'm confident our paths will cross again soon. I would also like to express my gratitude to Vedad Hadžić and Prof. Roderick Bloem for joining the project and contributing to the rigor and quality of our results. Your involvement, even after my departure from Graz, was decisive in the successful publication of our paper. Lastly, my thanks to all the PhD students and post-docs I had the pleasure of meeting during my stay, who made those three months so enjoyable. While I cannot mention everyone, I would like to give special thanks to Gaëtan Cassiers, Barbara Gigerl, Lukas Maar, Rishub Nagpal, Robert Primas, and Moritz Waser for the insightful discussions and memorable mountain excursions.

Je souhaite enfin remercier mes proches et ma famille, qui m'ont soutenu tout au long de cette période. Je sais qu'il n'a pas toujours été facile de comprendre la nature de mon travail, mais cela m'a permis d'affiner mes compétences en communication et en vulgarisation, même si, pour certains, je resterai un *dresseur de puce*. Il était également souvent difficile de justifier que, malgré mes 10 semaines de congés et de nombreux déplacements en conférences ou formations, il m'arrivait parfois de travailler ! Un merci tout particulier à Emma, qui a partagé avec moi ces trois années de labeur, relu mes articles avec une rigueur parfois supérieure à celle de certains reviewers, et avec qui nous allons bientôt pouvoir tourner la page vers un nouvel épisode de notre vie.

Pour conclure, je remercie chaleureusement les membres de mon jury qui ont accepté d'examiner mes travaux, et tout particulièrement Vincent Beroulle et Jean-Max Dutertre, qui ont relevé le défi de rapporter mon manuscrit, parfois dans l'urgence face à des changements de dernière minute.

Synthèse en français

Les attaques par injection de fautes représentent une menace majeure pour la sécurité des systèmes embarqués, car elles peuvent permettre à un attaquant d'extraire des secrets ou d'obtenir un accès non autorisé à des fonctionnalités sensibles du système. Si ces menaces sont traditionnellement évaluées séparément au niveau logiciel (programme) ou matériel (circuit du microcontrôleur), des recherches récentes soulignent la nécessité de prendre en compte les deux niveaux et d'analyser la microarchitecture du processeur pour comprendre pleinement les conséquences des attaques par injection fautes. Dans ce contexte, cette thèse vise à élaborer une méthodologie d'analyse exhaustive et automatisée, prenant en compte à la fois les descriptions logicielles et matérielles du système, afin d'avoir une évaluation fine des conséquences des fautes ou de fournir des garanties formelles sur la sécurité. La thèse est structurée en quatre chapitres auxquels s'ajoutent une introduction et une conclusion.

Le chapitre 1 dresse un état de l'art des attaques par injection de fautes, des contre-mesures, et des outils d'évaluation, avant de poser la problématique adressée par le manuscrit de thèse. Il met en évidence la nécessité d'une analyse multi-niveaux, englobant les différents niveaux d'abstraction, du matériel au logiciel. Les contre-mesures présentées incluent des solutions matérielles, logicielles et mixtes, avec un accent particulier sur celles visant à protéger l'intégrité du flot de contrôle du code embarqué (CFI) qui nécessitent une analyse multi-niveaux. Cependant, l'analyse effectuée dans cet état de l'art révèle une lacune dans les méthodologies actuelles : aucune ne propose d'analyse logicielle/matérielle basée sur les techniques formelles. Les méthodes existantes reposent sur une approche par simulation non-exhaustive, n'apportant pas de garanties de sécurité.

Le chapitre 2 présente µArchiFI, une méthodologie innovante pour la modélisation et la vérification formelle de systèmes combinés matériel/logiciel. Contrairement aux méthodologies existantes, µArchiFI est exhaustif et permet l'identification automatique des vulnérabilités difficiles à détecter, ainsi que la preuve de robustesse des systèmes contre les attaques par injection de fautes. Cette méthodologie repose sur une machine à états finis de type Mealy, où le matériel est modélisé comme un système de transition d'états, tandis que le code logiciel est encodé dans l'état initial. Les fautes sont intégrées en modifiant les transitions du système et l'objectif de vulnérabilité d'un attaquant est exprimé par une propriété d'accessibilité sur un sous-ensemble d'états du système. La vérification des propriétés de sécurité est effectuée grâce à des techniques formelles du type model checking borné. µArchiFI est mis en oeuvre dans l'infrastructure de compilation Yosys, et disponible publiquement en open source.

Le chapitre 3 illustre l'utilisation de µArchiFI dans deux cas d'étude : la détection

de vulnérabilités dans la microarchitecture d'un cœur CV32E40P et la validation d'une contre-mesure CFI (MAFIA). Dans le premier cas, µArchiFI identifie des vulnérabilités complexes dans le forwarding et le prefetch buffer, démontrant l'efficacité de la méthode. Dans le second cas, la robustesse de MAFIA face à des fautes symboliques est formellement prouvée, tâche qui ne n'aurait pas été possible en travaillant séparément aux niveaux d'abstraction logiciel ou matériel. µArchiFI s'avère également performant pour évaluer la sécurité de codes et de cœurs matériels, avec des résultats obtenus en moins de cinq minutes pour des circuits jusqu'à 50 000 portes.

Pour répondre aux défis posés par des systèmes plus complexes, le chapitre 4 propose une amélioration de la méthodologie en deux étapes : une évaluation préliminaire des contre-mesures matérielles (intitulée *partitionnement résistant aux fautes d'ordre k*) pour réduire l'espace des états, suivie d'une vérification multi-niveaux des fautes restantes, en intégrant le matériel et le logiciel. Cette approche a été validée sur des implémentations sécurisées d'algorithmes cryptographiques symétriques tels que AES, où elle prouve la robustesse des circuits jusqu'à trois fautes, surpassant les outils existants. La méthodologie a ensuite été appliquée au cœur sécurisé Ibex d'Open Titan, où elle identifie des vulnérabilités dans le banc de registre du processeur. L'étape de co-vérification démontre cependant que ces fautes sont capturées par les contre-mesures logicielles, prouvant ainsi la sécurité du premier étage du boot sécurisé d'OpenTitan face aux fautes uniques. En conclusion, cette méthodologie permet d'analyser des systèmes de grande complexité, comprenant jusqu'à 130 000 portes et des logiciels de plusieurs milliers d'instructions, ce qui constitue une avancée majeure par rapport à l'état de l'art.

Contents

Introduction

Contents

Context

Embedded systems are designed to perform specific tasks efficiently and reliably while operating under strict resource constraints, such as limited power, memory, and computing capacity. These systems are omnipresent in our daily lives and are widely used in security-critical applications, making them privileged targets for cyberattacks. For instance, in medical devices that control life-sustaining functions, such as pacemakers and insulin pumps, a security breach could lead to severe health risks or fatalities. In automotive systems, a security compromise could alter braking, acceleration, or navigation, potentially leading to accidents or loss of vehicle control. Similarly, a security incident in industrial control systems, such as those used in nuclear plants or manufacturing lines, could disrupt essential services, causing economic damage and safety hazards. Last but not least, personal devices like smartphones, smartwatches, and smart home systems collect and process personal data, making security vulnerabilities in these systems a serious concern for privacy. Ensuring the security of embedded systems is therefore crucial, as breaches can result in significant consequences, including safety risks, financial losses, and compromised privacy.

At the heart of an embedded system is typically a microcontroller or processor executing specialized software. Each of these components—the processor and the software—can be targeted by various types of attacks. Famous illustrations of these attacks include buffer overflows such as Morris Worm exploit [Con88], buffer over-read such as Heartbleed [Con14], and side-channel attacks that leverage extra information, such as the computation timing, power consumption, and electromagnetic emissions, to gain knowledge on the sytem [Koc96]. Among this wide range of existing security threats, *fault injection attacks* are particularly powerful and con-

cerning. In these attacks, malicious adversaries apply physical stress during system operation, such as manipulating the supply voltage, applying an electromagnetic field, or shooting the circuit with a laser. These attacks create incorrect values within the microelectronics, known as *faults*, which can alter system behavior, allowing attackers to compromise security by bypassing authentication mechanisms or gaining access to sensitive information.

Protecting against these attacks is a challenging but tremendously important task in the development of secure and reliable systems. During the design process, security engineers must first establish a threat model—or *fault model*—to describe the capabilities of a malicious attacker. Accurately understanding and modeling potential attacks is critical because inaccuracies can lead to ineffective protections or excessive cost and performance overhead. In a second step, hardware designers and software developers must carefully select or develop effective countermeasures to protect the system against the considered fault attacks. These countermeasures can be implemented across different system components, including hardware and software, and are designed to detect, prevent, or mitigate fault attacks. Finally, once the countermeasures have been implemented, the overall system security must be rigorously evaluated to ensure that these protections effectively thwart fault attacks. These security evaluations are conducted by the manufacturers themselves as well as by independent certification authorities to ensure compliance with security standards such as Common Criteria [Cri22]. Numerous techniques have been proposed in the literature to assess the effects of fault attacks on embedded systems at both the software and hardware levels, using techniques such as experimental characterization, simulation, and formal verification.

Motivation

Over the past decade, the emergence of open-source initiatives like RISC-V processors has facilitated access to implementation details, enabling the development of new approaches to better characterize the consequences of faults. While traditional security evaluation techniques have typically focused on either software or hardware analysis, a few recent studies have considered both and investigated fault effects on processor microarchitecture.

Notably, the PhD theses of Bilgiday Yuce [Yuc18] and Johan Laurent [Lau20] emphasize the importance of considering microarchitectural processor optimizations, such as pipelining and forwarding. These studies reveal that fault effects can be far more subtle and potent than previously understood, resulting from a complex interplay between software and hardware. Consequently, a single fault attack can compromise countermeasures like instruction duplication, which are designed to protect against the instruction skip fault model. These findings challenge the conventional evaluation techniques that independently analyze system security at the software or

hardware level. Furthermore, they raise concerns about current security evaluation methodologies, which may rely on oversimplified fault models and, consequently, fail to accurately assess system security.

Establishing more accurate fault models and understanding the final consequences of faults requires to adopt cross-layer evaluation methodologies. Therefore, developing systematic analysis methods that integrate both software and hardware descriptions is essential for a better understanding of fault effects and for assessing system security against fault attacks with stronger guarantees.

Contributions

The research conducted during this PhD and presented throughout this manuscript provides the following key contributions:

- We introduce µArchiFI, a novel methodology for modeling combined hardware/software systems and evaluating system security against fault attacks using bounded model checking techniques. Unlike existing approaches, our methodology is exhaustive and allows the automatic identification of corner-case vulnerabilities as well as proving system robustness against fault attacks. The details of µArchiFI are discussed in Chapter 2.

- We evaluate the security of RISC-V processors, including the CV32E40P and Ibex cores, running various lightweight software applications such as the authentication program VerifyPIN. We validate our approach by reproducing known results, such as the vulnerability exploiting the forwarding mechanism highlighted by Johan Laurent. Additionally, our methodology discovers previously unreported fault effects, such as those targetting the prefetch buffer and the multiplication unit, allowing to skip or replay multiple instructions under specific software conditions. These experimental findings are detailed in Chapter 3.

- We formally assess the security of a combined hardware/software countermeasure called MAFIA, implemented in the CV32E40P processor. We prove that this countermeasure effectively protects known vulnerabilities in VerifyPIN from fault attacks. Such an analysis is not feasible using hardware or software verification alone, as it requires modeling software integrity signatures that are checked during hardware execution. The results of this analysis are also presented in Chapter 3.

- We enhance the fault evaluation methodology by performing a preliminary evaluation of hardware-level countermeasures. For this purpose, we introduce and formalize the novel concept of *fault-resistant partitioning*, which ensures

that faults are effectively captured by hardware countermeasures. This preliminary evaluation allows the co-verification step to focus only on potentially harmful faults not captured by hardware protections, thereby improving the performance of the approach. We also propose an algorithm to identify and prove such fault-resistant partitions. The fault-resistant partitioning methodology is described in Chapter 4.

- We validate our *fault-resistant partitioning* methodology by replicating known results on fault-hardened Skinny and AES circuits [AMR+20], as no similar work has been done on processors. We show that our approach is competitive with related work by proving the 2-fault security of AES in less than 4 hours, compared to 130 hours for FIVER [RRSS+21] or 30 minutes for FIRMER [TGC+23]. We also demonstrate the scalability of our approach by analyzing the 3-fault security of AES, which has never been done before. Evaluations of Skinny and AES are detailed in Chapter 4.

- Finally, we analyze the fault-hardened Ibex processor [lowa] used in the OpenTitan secure element [JRR+18] which was previously intractable using existing methodologies. Our analysis reveals that the Ibex dual-core lockstep protection is secure against one fault but its register file does to capture certain bit-flips, potentially leading to software vulnerabilities. The co-verification process shows that these vulnerabilities could be exploited in VerifyPIN or during differential fault analysis of AES software [kok19]. However, we prove the robustness of OpenTitan running the first step of a secure boot, as its software countermeasures prevent the register file vulnerability from being exploited. We disclosed and fixed the register file vulnerability, which has since been integrated into the OpenTitan project. The evaluation of OpenTitan is detailed in Chapter 4.

The next chapter introduces fault injection attacks and reviews existing fault characterization and security evaluation techniques to establish the problem that this manuscript addresses.

Chapter 1

State of the Art and Problem Statement

Contents

This thesis explores the consequences of hardware-level fault injection on the security of CPU-based systems. To effectively address this topic and propose meaningful advancements, it is crucial to thoroughly understand the current state of knowledge in this field. This chapter surveys the literature related to fault injection, providing a comprehensive overview while also highlighting key observations and identifying an open question that this thesis aims to address. This chapter is organized as follows.

First, Section 1.1 examines the root causes of fault injections in microelectronics and details their final consequences on program execution. Next, Section 1.2 describes how software and hardware security designers develop countermeasures to enhance system security against these attacks. Finally, Section 1.3 reviews automated analysis methodologies and tools used to understand fault effects and assess the security of embedded systems. This chapter concludes with observations and questions on the necessary steps to better evaluate the effects of faults, thereby establishing the problem addressed in this manuscript.

1.1 Fault Injection Attacks

Fault injection research originally emerged from safety-critical domains such as aerospace and nuclear sectors, where integrated circuits are exposed to hazardous perturbations like radioactive particles and cosmic radiation [MW78, ZL79]. The reliability of these systems is crucial, as malfunctions can lead to dangerous hazards. In the security domain, adversaries realized they could intentionally induce faults in systems to reproduce these effects and create security breaches. In 1997, Boneh, DeMillo, and Lipton showcased that hardware faults can break the security of the RSA cryptosystem [BDL97]. Later, in 2002, Skorobogatov and Anderson used a laser source to inject faults into the SRAM memory of microcontrollers [SA02]. These pioneering works paved the way for fault injections in the field of computer security, highlighting the critical need to protect embedded systems against such vulnerabilities.

In this section, we delve deeper into fault injection attacks. We describe how they manifest in the physical platform, their impact on software, and how research has explored and characterized their effects over the past few decades.

1.1.1 Physical Means

Fault injection attacks aim to apply abnormal physical stress to integrated circuits to produce incorrect values in the microelectronics. These erroneous values are denoted as *hardware faults* and alter the system behavior. In the following of this section, we survey the multiple fault injection means existing in the literature [BCN+06, BBKN12, YSW18, RSG22], starting from the most affordable before describing the most advanced ones. We report their main characteristics and detail their consequences.

Clock Glitches. Sequential circuits rely on a clock signal to sample the circuit's internal state at a given operating frequency and store digital values in state-holding units like flip-flops. Tampering with the clock signal disrupts this sampling process, leading to the storage of incorrect values in flip-flops. *Clock glitches* are a relatively inexpensive technique that exploits this erroneous sampling process by instrumenting the external clock signal. Clock glitches have a high temporal accuracy, but the exact location of the resulting hardware faults is uncertain. In practice, this fault injection technique typically affects the circuit's critical path, i.e., the longest combinational path between two flip-flops. In the literature, clock glitches were first demonstrated efficient to break cryptographic algorithms on dedicated integrated circuits [AD10, ESH+11] before being later applied to processors [BGV11, KHEB14]. From a security standpoint, clock glitches were used to defeat software countermeasures [YGS+16], and bypass authentication mechanisms [CPHR21] skipping multiple CPU instructions.

Table 1.1: Fault injection techniques overview.

Technique	Spatial Accuracy	Temporal Accuracy	Cost	Remote Access	Injection Parameters
Overheating	Low	Low	Low	✗	Temperature
Clock Glitches	Low	High	Low	✗	Timing, glitch width, glitch amplitude
Underpowering	Low	Low	Low	✗	Voltage amplitude
Voltage Glitches	Low	High	Low	✗	Timing, glitch width, glitch amplitude
Electromagnetic Pulses	Moderate	High	Moderate	✗	Timing, intensity, duration, location
Laser Beams	High	Very High	High	✗	Timing, intensity, duration, location
Body Biasing	Moderate	High	Moderate	✗	Timing, intensity, duration, location
X-ray	Very High	Low	Extreme	✗	Location, intensity
Software-based FI	Varies	Varies	Low	✓	Platform-dependent

Underpowering and Voltage Glitches. Similar to manipulating the clock signal with clock glitches, underpowering and voltage glitches are low-cost fault injection techniques with limited spatial accuracy. *Voltage glitches* are temporally precise, as they involve lowering the supply voltage for a very short duration. *Underpowering*, on the other hand, reduces the power supply throughout the entire execution of the circuit. By decreasing the power supply, the propagation delay of the circuit gates increases, leading to an erroneous sampling process akin to clock glitches [ZDCT13]. Initially, the first practical experiments were achieved on cryptographic algorithms [SGD08, BBPP09, ZDCT13]. More recently, advanced attacks have targeted CPUs to escalate privileges [TM17] and compromised bootloaders [TS16, BFP19].

Electromagnetic Pulses. Electromagnetic fault injection (EMFI) is one of the most popular attack methods due to its practicality, affordability, and accuracy. By creating a local electromagnetic field with a probe, an attacker can induce a current in the circuit that disrupts the power supply [DLM19]. Experiments have shown that EMFIs can cause timing violations on the critical path [DDRT12, ZDT+14]. Further studies refined these observations and proposed the *sampling fault model* [OGT+15, OGM15, OGM17, DLM19, DLM21]. From a security perspective, a seminal paper in 2002 demonstrated the injection of EM faults in microprocessor memory [QS02]. Subsequent research targeted more complex architectures, breaking cryptographic implementations [SH07, DDRT12, DMM+13] and creating software vulnerabilities, including buffer overflows [BLLL18].

Laser Beams. Laser fault injections (laser FI), popularized by Skorobogatov and Anderson in 2002 [SA02], focus a light beam on circuit transistors to create a photoelectric effect, inducing a transient current [RSDT13, DDCS+14]. In recent years, laser techniques have become more accurate, demonstrating effectiveness on 45 nm [SBHS16] and 28 nm technologies [DBC+18], with timing precision down to a

few picoseconds [LBC+15]. Consequently, lasers are also effective tools for injecting multiple faults in a single attack [WMP20, CGV+22]. Initially, the practical use of laser FI in security was demonstrated in cryptography [TK10]. Recently, laser FIs have also enabled attackers to bypass authentication mechanisms [DRPR19] and secure boot processes [VTM+17].

Software-controlled. In all the previously mentioned scenarios, hardware faults are injected by an attacker with physical access to the system. However, the situation has changed and recent research has shown that hardware faults can also be triggered remotely by a virtual attacker using malicious software. In these cases, the hardware fault no longer arises from abnormal environmental conditions but from internal software-controlled mechanisms. A well-known example of these *software-based fault attacks* is the Rowhammer vulnerability [KDK+14], which induces bit flips in DRAM memory by performing intensive data accesses in adjacent memory rows. Other notable attacks exploit the power management unit, such as CLKSCREW [TSS17], Plundervolt [MOG+20], and PMFault [CO23].

Miscellaneous Techniques. Other fault injection techniques have also proven to be effective in the literature. These include overheating [Sko09], body biasing [MTOL12, BLE+16], and X-ray [ABC+17, BAM+23]. A summary of all the aforementioned techniques is provided in Table 1.1 and details the characteristics and the parameters of each fault injection means.

1.1.2 Abstraction Layers

Attackers have a wide diversity of fault injection techniques to disturb the operation of embedded systems, ranging from clock glitches to laser beams. However, understanding the final consequences of these attacks on the running software is a challenging task. Since the software is the primary target of the attack and the vulnerability originates from the hardware, all system components must be considered in the analysis. To better understand how fault attacks lead to security vulnerabilities, we propose to open the box and delve into the typical anatomy of a secure embedded system from its hardware layer—including the processor—to its software component, the program [YSW18]. The boundary between the software and hardware worlds is known as the instruction set architecture (ISA). A detailed description of the abstraction levels is depicted in Figure 1.1. In the following, we examine each of these levels, starting from the physical layer. Additionally, for each level, we discuss the various *fault effects* induced by fault injection attacks.

Physical Level

The physical level of an embedded system refers to the actual implementation of the integrated circuit (IC). This includes the *transistors*—the fundamental IC's building blocks acting as switches and performing the operations—and the *metallic connections* between transistors. As the lowest level of abstraction, the physical level's com-

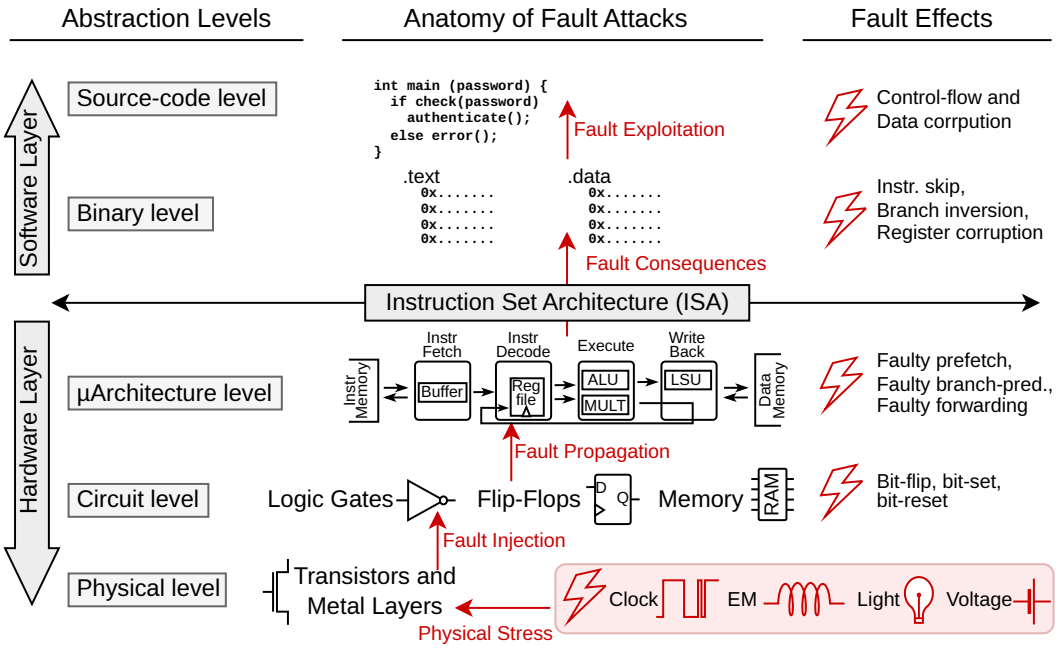


Figure 1.1: Fault abstraction levels and fault effects, adapted from [YSW18].

ponents are highly sensitive to external perturbations. As discussed in the previous section, dysfunctions can occur when tampering with the clock signal or applying an electromagnetic field to the IC.

At this level, the effects of faults are described by the physical stress applied to the system. This includes specifying the fault injection technique used, such as laser or underpowering, and detailing the experimental parameters, like timing, intensity, and duration, as summarized in Table 1.1. In the remainder of this manuscript, we set aside physical considerations to focus on the digital level of systems, where data is represented as digital values rather than voltages with propagation delays. Therefore, the lowest abstraction level we consider is the circuit level, which we introduce next.

Circuit Level

The circuit level represents the lowest logical view of a system and is usually one of the key elements provided to manufacturers for chip production. The circuit's building blocks include logic gates such as AND, OR, NOT, and XOR gates, as well as flip-flops and memory elements, all of them primarily constructed from transistors. These components consequently assume that the underlying physical level behaves as expected, enabling circuit elements to implement logic, arithmetic or memorizing functionalities.

However, when physical stress is applied to the system, this assumption fails, and we say that faults are *injected* into the circuit. The fault injection process is illustrated in Figure 1.1. Representing fault effects at the circuit level involves

describing how the basic circuit's building blocks malfunction. This description usually includes parameters such as the *location* (e.g., flip-flops, AND gates), the *logical effect* (e.g., bit-flip, bit-reset), the *duration* (e.g., transient, permanent), and the *multiplicity* (e.g., single or multiple faults). A wide range of circuit-level fault effects exist depending on the type of physical stress applied.

Microarchitectural Level

The microarchitectural level is the highest abstraction level within the hardware layer. While the physical and circuit descriptions apply to both application-specific systems, like cryptographic accelerators and general-purpose CPUs, the microarchitectural view is specific to processors. This level details the implementation choices made by hardware designers to meet the requirements specified in the ISA.

The microarchitecture typically includes mechanisms to optimize cost and performance, which are hidden from software programmers. Key optimizations include pipelining, forwarding, and memory hierarchy [HP12]. We provide a brief description of these mechanisms below.

Pipelining: Processor pipelining enhances CPU performance by dividing the processing of instructions into several elementary steps, each handling a different stage of the instruction execution. This allows multiple instructions to be processed concurrently. Typical pipeline stages include *instruction fetch, instruction decode, execute*, and *write-back*, as depicted in Figure 1.1. However, pipelining can introduce pipeline hazards, where dependencies between consecutive instructions cause performance penalties.

Forwarding: Data forwarding addresses data hazards caused by pipelining. This technique passes the result of an instruction directly to the subsequent instruction that needs it, bypassing the register file to speed up execution. Other techniques to mitigate hazards include stalling, branch prediction, and speculative execution, which are not detailed here.

Memory Hierarchy: Memory hierarchy refers to the way of organizing data storage to optimize speed, cost, and capacity. Since memory operations are time-consuming, memory is structured in layers, from the faster to the slower: register file, cache, and main memory. The closer the data storage is to the processor, the faster the operations, but the lower the capacity. Replacement policies ensure that the most frequently used data remains near the processor. In addition, prefetch buffer also exist to store instructions before the processor executes them.

When a fault is injected into the underlying circuit, we say that the fault *propagates* through the microarchitecture. The faulty values are memorized in registers and moved through pipeline stages. In contrast with circuit-level fault effects, fault locations are not identified by logic gates but by dysfunctional blocks or modules. These erroneous values are described using bit-vector variables, referred to as the *word level*, rather than at the bit level. For example, as shown in Figure 1.1, microar-

chitectural fault effects may include malfunctioning forwarding or branch prediction mechanisms.

Binary Level

The binary language operates at the lowest level of the software layer, specific to a given processor architecture (ISA), and provides the machine code executed by the CPU. Its human-readable form is known as *assembly* language, which uses mnemonics (or opcodes) such as `mov`, `add`, and `sub` to represent machine-level instructions. These instruction mnemonics are followed by operands, which can be registers (e.g., `r1`, `r9`), constants, or memory addresses. A binary program is structured into multiple sections. Typically, the `.text` section contains the instructions, which are stored in the instruction memory, while the `.data` section holds data values.

At the binary level, there is no information about the actual implementation of the processor. Instead, the instruction set architecture (ISA) serves as a contract between software and hardware, specifying the expected behavior of the underlying hardware to ensure the correct functioning of the software. For instance, when executing the instruction `add r1, r14, r15`, it is expected that `r1` will contain the sum of `r14` and `r15` before the next instruction is executed. However, when fault injections compromise the processor's microarchitecture, the ISA specification is no longer valid, and assumptions made at the software layer about the correct operation of the hardware fail.

The ISA *fault effects* describe the impact of faults on binary program execution. Faults are often categorized into those affecting the control flow—the order of instruction execution—and the data flow, which involves data values and computation results. Control-flow faults include instruction skips, instruction replacements, or branch inversions. Data-flow faults include register corruption or operand corruption. These effects are mentionned in Figure 1.1.

Source-Code Level

The source-code level represents the highest abstraction of the software layer. Programmers typically write code using languages like C, C++, or Rust, which are then compiled into lower-level binary code. This manuscript focuses on secure software that implements sensitive functionalities like authentication mechanisms, integrity checks, or secret data manipulation.

Faults originating from the physical level and manifesting at the binary level can lead to misbehavior of the executed program. Representing faults at the source-code level is often very inaccurate as significant discrepancies exist between source code and the binary program. Nevertheless, fault effects are coarsely modeled between *data corruption* and *control-flow corruption*.

Fault Observation and Exploitation

From a security perspective, a malicious attacker selects a fault injection technique to disrupt hardware operation and observe the resulting fault consequences at the software layer. The following paragraphs detail various exploitation methods attackers use to leverage fault observations, thereby compromising the security of embedded software.

Direct Attacks. Direct attacks occur when the consequences of a fault on the running program immediately create a vulnerability without requiring additional effort from the attacker. Examples of such attacks include control flow hijacking [BIL11, TSW16], bypassing secureboot [TS16, VTM+17], privilege escalation [GMM16, TM17], and subverting memory isolation [BTG10, TSS17].

Cryptanalysis. Fault injection attacks have been extensively used to compromise the security of cryptographic algorithms that are mathematically robust but whose implementation has weaknesses. For instance, *differential fault analysis* exploits observable behavioral differences between two encryptions to gain knowledge of secret variables, such as plaintext or cryptographic keys [BS97, PQ03, Gir05, TFY07, AM11, TMA11]. Additional techniques, such as *differential fault intensity analysis* [GYTS14] and *safe error attacks* [SJ00], have also been studied in the literature.

Combined Attacks with Side Channel. Similar to fault injection attacks, side-channel attacks (SCAs) exploit the hardware implementation of a microprocessor to compromise the security of a running program. SCAs involve observing the physical leakage of a device during sensitive operations, such as cryptographic encryption. Common sources of leakage include power consumption, timing information, and electromagnetic emissions. Combined attacks enhance the effectiveness of side-channel attacks by leveraging additional leakage induced by fault injections [Sko06, AVFM07, RLK11]. Additionally, combined attacks can use side-channel measurements to set up fault injections and choose the best timing to perform the attack [FGA+23].

This section has established that fault attacks on embedded systems target software security, even though the root vulnerability lies in the hardware. Given the various representation levels involved from fault injection to its exploitation at the software level, it is essential to consider these multiple system layers in the analysis to understand and evaluate the impact of hardware-induced faults on software. The necessity for cross-layer evaluation raises the following questions:

Questions for Problem Statement

Q1. How can system security against fault attacks be evaluated across multiple abstraction levels?

Q2. How can the causal relationship between hardware-level fault effects and their consequences on software security be better understood?

1.1.3 Fault Models

At this stage of this Section 1.1, we have reviewed existing fault injection techniques at the physical level and described their corresponding effects for each abstraction level. However, understanding the actual fault effects resulting from a specific fault injection technique on a given hardware/software system is challenging. This task becomes even more complex when evaluators cannot observe fault effects at intermediate abstraction levels, as is often the case with already-manufactured chips or closed-source processors.

To address this issue, *fault models* are proposed to describe fault effects observed in practice. Establishing accurate fault models at various abstraction levels is crucial for developing effective countermeasures and methodologies to analyze system security. Inaccurate fault models can lead to ineffective protections or can increase costs due to oversized countermeasures.

In the following, we review works that characterized the consequences of faults on secure embedded software in order to derive the so-called *fault models*.

Blackbox Experimental Characterization

To understand the effects of fault injection attacks and derive a fault model, experimental studies use simple test cases and observe the system's state before and after applying physical stress. Given that designs are typically proprietary (e.g., from vendors like ARM or Intel), evaluators lack information about implementation details. Consequently, they limit their observations to values specified in the ISA and visible during software execution, such as data in memory or in the register file. Based on these black-box observations, they propose explanations for the software-level fault consequences, also known as the *ISA fault model*.

Over the past decades, numerous fault models have been developed. Initially, the effects of fault attacks on software were categorized into those affecting the control flow and those affecting the manipulated data [BCN+06]. Faults impacting the control flow were initially associated with *instruction skips* and used in practical attacks to bypass critical instructions, such as counter increments or function calls, during AES or RSA execution [CT05, KQ07, SH08]. Further experimental studies on the instruction skip fault model revealed that instructions are more likely to be replaced by other instructions rather than skipped. When the faulty instruction opcode is invalid, some microprocessors treat them as a `nop` (no operation) [BGV11]. More recently, more complex effects have been reported, such as multiple instruction skips [MDP+20] and instruction replay targeting the instruction cache [RNR+15].

Another approach to developing ISA fault models involves conducting both experimental fault injections and faulty simulations to find correlations between simulations and experimental observations [MDH+13, GJL20]. To better understand fault effects, a top-down approach was used in [TBC20], where the assembly test vector is crafted and adapted according to the observed effects to finely characterize

the consequences of faults. The authors also reported that new effects can appear depending on the processor state. The work conducted by Proy et al. on a superscalar processor successfully reproduced known effects, including instruction skips, operand corruption, register corruptions, and instruction replay [PHB+19]. Additionally, they observed combinations of these effects, as well as inexplicable ones dubbed *magic edges*, where the program branches to an illegal destination block in the control-flow graph.

These recent findings highlight the limitations of black-box experimental characterization in interpreting the consequences of faults at the software level. The effects of fault attacks result from a complex interplay between the hardware undergoing physical stress and the software running on the platform.

Whitebox Manual Characterization

With the emergence of open-source initiatives like RISC-V processors, new methods have appeared for characterizing the consequences of faults on embedded systems. The consequences of faults on software security can now be evaluated through a white-box approach that also considers intermediate levels of abstraction in the analysis, such as the circuit or the microarchitectural levels. Including these additional descriptions in the analysis helps evaluators to understand the impact of implementation details on the consequences of faults at the ISA level. The white-box approach also helps to refine and improve fault models at intermediate levels by leveraging insights into the processor microarchitecture. It is also a step forward in explaining the misunderstood or inexplicable fault effects observed through experimental characterization.

Recently, a few works proposed to include observations from the microarchitecture to better understand the effects of fault attacks. These works highlight the impact of microarchitectural optimizations such as pipelining, forwarding, and memory hierarchy. Some detailed results from these studies are provided below.

Pipeline. As previously detailed, processor pipelining allows multiple instructions to be processed concurrently into distinct stages. In their works [BGV11, YGS15], Balash et al. and Yuce et al. highlighted that a fault injected into a pipelined processor can affect multiple stages, thus corrupting multiple instructions at the same time. The resulting consequences are complex and powerful enough to defeat fault protections designed based on the instruction skip fault model, such as instruction duplication [YGS+16].

Forwarding. Later, Laurent et al. reported an attack that exploits the forwarding mechanism of pipelined processors [LBD+18]. As a recall, forwarding mitigates data hazards caused by pipelining by transmitting the result of an instruction directly to the subsequent instruction that needs it. This mechanism is consequently exploited in a fault attack to reintroduce a previously computed value into the pipeline. This attack can bypass a comparison by forwarding a wrong value or skip instructions protected with instruction duplication countermeasure [LBD+19].

Hidden Registers. Another notable characteristic of processor implementations is the existence of microarchitectural registers that temporarily hold data or control values during code execution. In [LBDP19], the authors reported several examples of these so-called *hidden registers* such as the last computed result from the multiplication unit or the most recent data read from memory. They showcased how values stored in hidden registers can be leveraged in an attack to bypass a PIN comparison or leak sensitive data, thereby compromising system security.

Memory Interface. In [TBE+21], the authors conducted a comprehensive study of fault effects on the memory hierarchy of a modern CPU. They first targeted the instruction cache and confirmed the instruction corruption fault model. They also targeted the memory management unit (MMU) and observed more complex effects where faults in the MMU shift the page tables in memory.

Variable-length Instructions. Fault analysis was also conducted on the instruction buffer and decoder unit. Alshaer et al. reported that fault injections on variable-length instructions can lead to effects akin to *skip* or *repeat and skip* fault models based on instruction misalignment [ACD+22].

All the aforementioned fault models resulting from microarchitectural mechanisms pave the way for a deeper understanding of fault consequences on secure embedded systems. The actual consequences of faults on software execution are far more complex than they may initially appear in Figure 1.1. Additional parameters must be considered to make the correspondence between hardware-level and software-level fault models, such as the implementation details, the processor's internal state, e.g., hidden registers, and the program being executed. As a result, a single fault in the underlying hardware can result in a wide range of ISA-level fault effects. As illustrated in Figure 1.2, a single *bit-set* fault can lead to software data corruption, an instruction replay, or a yet-unreported fault effects, for instance. Conversely, a fault observed at the software level may have its root cause in a wide variety of hardware faults, such as a *bit-flip* or a *bit-reset* in various processor areas. There is no one-to-one mapping between hardware and software fault models. While the well-known *instruction skip* fault model is a good approximation, it may have additional side effects in practice. Moreover, these subtle and complex fault effects may have significant implications for software security. They can be exploited in attacks to defeat protections such as software-level redundancy [PCNM15, YGS+16, LBD+18] or to bypass secure authentication processes [VWWM11, LBDP19].

Several works have proposed categorizing all possible fault effects into tables enumerating potential hardware faults, microarchitectural implementation details, and the type of instruction currently executed in the pipeline [YSW18, LBD+19]. However, enumerating all possible fault consequences is an endless task, given the vast diversity of instruction sets and microarchitectural implementations.

Over the past decades, researchers and analysts have accumulated extensive knowledge about fault models. However, these analyses have primarily relied on

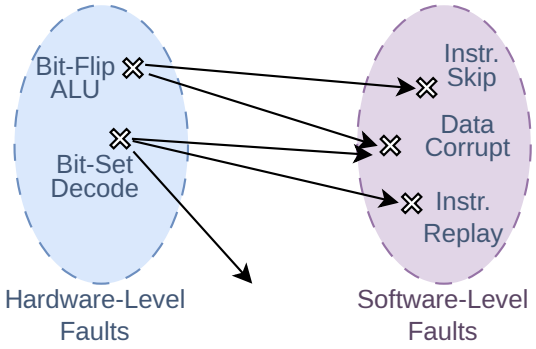


Figure 1.2: Mismatch between hardware-level and software-level fault models.

manual processes, such as experimental characterization and microarchitecture inspection. To deepen our understanding about fault consequences on software security and to determine the conditions under which faults are exploitable, manual characterization alone is insufficient—it is time-consuming, incomplete, and prone to errors. We have now reached a stage where it is crucial to integrate this vast amount of knowledge into cross-layer automated methods that consider both the software and hardware characteristics of embedded systems. This necessity raises the following questions:

Questions for Problem Statement

Q3. Which analysis techniques are best suited to systematically explore the entire system state space, including system behaviors, and fault effects?

Q4. Can these techniques be adapted to determine which hardware-level faults lead to security exploits at the software level?

This section has introduced fault injection attacks and illustrated that attackers have a wide diversity of techniques, leading to a wide variety of fault effects, to compromise the security of embedded systems. The next section demonstrates that hardware designers and software programmers also have a broad panoply of countermeasures to protect against fault attacks.

1.2 Countermeasures against Faults

Embedded systems ensuring security functionalities, such as data confidentiality or the authenticity of executed programs, must be protected against fault attacks to prevent undesired system behaviors. Countermeasures can be deployed at various abstraction levels, from hardware to software, and are designed based on specific fault models they aim to protect against. As mentioned in the previous section, establishing accurate fault models is crucial for developing or selecting effective countermeasures, as inaccurate fault models can lead to ineffective protections.

Mitigating fault attacks often relies on spatial or temporal redundancy [BCN+06] to detect erroneous values and involves a trade-off between the effectiveness of the countermeasure and its overhead in terms of cost and performance.

In the rest of this section, we describe state-of-the-art fault injection countermeasures, covering the hardware layer, the software layer, and combinations of both.

1.2.1 Hardware

Hardware countermeasures against fault attacks are designed to detect, prevent, or mitigate fault attacks at various levels of the hardware layer. At the physical level, circuits implement detectors or shielding mechanisms to detect when an attacker attempts to manipulate environmental conditions to induce a fault. Detectors monitor light, power supply, or clock frequency to ensure that physical environmental parameters remain within the system's nominal operation range [ZDT+14, MNH+16]. Additionally, protections such as active or passive shielding have been proposed, mainly to guard against laser fault injections (FIs) that require chip decapsulation before performing the optical attack [LM06].

At the circuit level, protections are based on either temporal redundancy, i.e., performing the same computation multiple times, or spatial redundancy, i.e., performing the same computation on several hardware blocks in parallel. Spatial redundancy, also referred to as Concurrent Error Detection (CED) schemes [MM00], often involves duplicating parts of the circuit and comparing each result for equality, though this results in considerable overhead. Alternatively, CED can rely on informational redundancy using error detection codes [BBK+03, AMR+20].

Another category of countermeasures aims not at detecting or correcting injected faults but at making the attacker's task more difficult or the results of an attack unusable. For example, inserting random dummy clock cycles or dummy instructions during processor execution complicates the experimental setup of an attack [GST12, low18]. Similarly, since many attacks exploit data on the bus or in memory, scrambling is an effective countermeasure to randomize the visible data from an attacker's point of view [lowc].

Hardware countermeasures are undoubtedly the most effective means of protecting against attackers, as faults are detected at their root cause. However, they significantly increase the area and the cost of the design. Additionally, manufactured chips cannot be updated when a new vulnerability is exposed. Consequently, designers also develop countermeasures on the software side.

1.2.2 Software

The main advantages of software-based countermeasures are their low cost compared to hardware-based protections and their portability across various systems without

requiring any hardware modifications. Additionally, software countermeasures can be updated to address new vulnerabilities as they are discovered. However, performance penalties offset these benefits, as countermeasures increase program size and execution time.

Software countermeasures assume a given ISA fault model to propose protections that thwart its effects. Since fault attacks can affect both control and data, software countermeasures are categorized into control and data protections.

On the one hand, control-flow protection ensures that instructions are executed correctly and in proper order. A primary protection against the ubiquitous instruction skip fault model is instruction duplication [OSM02b, RCV+05, BBK+10]. Here, the compiler duplicates program instructions and intertwines them with the original instructions before comparing the results. Instruction duplication using idempotent instructions, i.e., instructions that can be executed multiple times in a row, has also been proposed [Mor14]. Control-flow integrity (CFI) protection, first introduced in 2002 [OSM02a], implements control-flow checks at runtime, also known as checkpoints. Counters are incremented, or signatures computed, during program execution, and their values are checked at strategic locations such as function calls and returns [RCV+05].

On the other hand, data integrity protections ensure that data manipulated in memory or in the bus connecting memory with the processor are not modified by faults. These protections often rely on detection codes to guarantee data integrity or cryptographic signatures to ensure both the authenticity and integrity of manipulated data and instructions [BBK+10].

However, while software countermeasures aim to thwart fault exploitability at the software level, the root cause of fault attacks originates from hardware. Sometimes, the fault model considered for designing the countermeasure is too simplistic, and single fault injection attacks can defeat these countermeasures due to complex fault effects related to the microarchitectural implementation details [YGS+16]. Nonetheless, software countermeasures are a valuable complement to hardware protections, working together to ensure better fault coverage.

1.2.3 Combined Countermeasures

Software-only or hardware-only countermeasures have limitations in their ability to protect against fault attacks [YGS+16] and often come with significant overhead [AMR+20]. Consequently, recent works have combined hardware and software aspects in their proposed countermeasures.

To achieve the highest level of security, software-based CFI protections have been ported to hardware [DCGÜ+17, WUSM18]. In these approaches, signatures are inserted into the code at compile time and checked on the fly during execution by a hardware module extension.

These protections have been further enhanced to provide stronger security guarantees. For example, in [CCH22, CCH23], the authors propose mechanisms for code integrity to protect the program during transfers between the core and memory, as well as control-signal integrity to protect against faults in the microarchitecture. Hardware-based control-flow integrity has also been implemented on RISC-V cores [ZPRD23], secure elements [NM23], and Intel platforms [NSL+23].

Conclusion

After selecting and implementing countermeasures, the overall system security must be rigorously evaluated to ensure that chosen protections effectively thwart fault attacks. Common verification techniques typically operate at either the hardware or software level and require exhaustive coverage of all possible attack scenarios specified in the fault model. However, combined countermeasures—such as hardware-CFI—as well as subtle microarchitectural fault effects—as highlighted in Section 1.1—necessitate cross-layer security evaluation techniques to accurately capture the complex interplay between software and hardware. This need for exhaustive and combined techniques raises the following question:

Questions for Problem Statement

Q5. How to guarantee that countermeasures effectively prevent all hardware-level faults from compromising system security at the software level?

1.3 Fault Evaluation Tools

First, Section 1.1 presented fault injection attacks, described their effects at various abstraction layers, and introduced characterization techniques to derive the so-called *fault models*, representing the experimentally observed fault effects. Section 1.1 concluded by emphasizing the need for cross-layer automated fault evaluation methods. Second, Section 1.2 described existing fault countermeasures at various abstraction layers and stressed the need for exhaustive security evaluation techniques to ensure that protections effectively thwart fault attacks. The present section, therefore, describes existing tools and methodologies to systematically evaluate the consequences of various fault models.

The tools discussed here are pre-silicon, meaning they perform their evaluations on system models before chip manufacturing. These techniques complement the experimental characterizations described earlier in Section 1.1.3 and are crucial for identifying security vulnerabilities at the early stages of development. The tools presented operate at the software layer, the hardware layer, or a combination of both. They employ various verification techniques such as *simulation*, which executes con-

Table 1.2: Tools for evaluating fault consequences at various levels of abstraction.

	Tool	Target	Input Format	Verif. Technique	Fault Model Description
Hardware	AutoFault [BGE+17]	Cipher	RTL: VHDL	Formal: SAT	Single or multiple bit-/byte-flips.
	VerFI [AWMN20]	Cipher	RTL: Verilog/VHDL	Simu: custom	Multiple Stuck-at/bit-flip
	FIVER [RRSS+21]	Cipher	Netlist: Verilog	Formal: EC (BDD)	Fault model from [RSG22]
	SYNFI [NOV+22]	Cipher/CPU	Netlist: Verilog	Formal: EC (SAT)	Multiple faults, stuck-at or transient.
	FIRMER [TGC+23]	Cipher	Netlist: Verilog	Formal: SAT	Fault model from [RSG22]
Software	SymPLFIED [PNKI08]	CPU	Binary: RISC	Formal: BMC (Maude)	Transient in memory/registers.
	LAZART [PMPD14]	CPU	LLVM IR	Formal: SE (KLEE)	Symbolic data, test inversion.
	Given-Wilson [GJLL17]	CPU	Binary: 32-bit x86	Formal: BMC (LLBMC)	Binary manually edited.
	RobustB [BHE+19]	CPU	Binary: ARMv7-M	Formal: SA + EC (SMT)	Single skip or register corruption.
	FiSim [Ris20]	CPU	Binary: ARM	Simu: custom	Skip, flip, custom.
	ARCHIE [HGA+21]	CPU	Binary: ARM, RISC-V, x86	Simu: QEMU	Transient, permanent, data.
	ARMORY [HSP21]	CPU	Binary: ARM-M	Simu: ARMv7	Skip, replace, data/op/addr corrupt.
	SAMVA [GHHR23]	CPU	Binary: ARMv7-M	Formal: SA	Multiple instruction skips.
	BINSEC [DBP23]	CPU	Binary: 32-bit x86	Formal: SE (BINSEC)	Symbolic data, test invers., instr skip.
Combined	MEFISTO [JAR+94]	CPU	RTL: VHDL	Simu: custom	Single stuck-at or transient bit-flip.
	VeriFY [STB97]	CPU	RTL: VHDL	Simu: custom	Single stuck-at, transient bit-flip.
	LIFTING [BN08]	CPU	RTL: Verilog	Simu: custom	Multiple stuck-at and single bit-flip.
	SimpliFI [GS21]	CPU	Netlist + RISC-V Binary	Simu: Xilinx	Clock glitch (timing violation)

SE: Symbolic Exec. SA: Static Analysis EC: Equiv. Check. BMC: Bounded Model Check. BDD: Binary Decision Diag.

crete instances of the system to observe fault consequences, and *formal methods*, which allow reasoning on the system model to prove properties, thus considering multiple system executions simultaneously. The tools we discuss are summarized in Table 1.2 where the columns indicate 1) the name of the tool, 2) whether the tool supports cryptographic- or CPU-based systems, 3) the input description format, 4) the verification technique and, 5) the supported fault models.

1.3.1 Hardware

Very few works focus on analyzing CPUs at the hardware level, and recent advances in hardware fault evaluation are primarily driven by the cryptographic field. Table 1.2 reports five recent tools that evaluate fault effects on cryptographic circuits. Although these tools do not model software instructions and operate solely at the hardware level, they are noteworthy for their exhaustive evaluation of fault effects, which is crucial for ensuring the security of cryptographic primitives.

One category of work focuses on evaluating cryptographic accelerators against cryptanalysis fault attacks. For example, AutoFault [BGE+17] proposes a framework to automatically evaluate the security of block ciphers against differential fault analysis (DFA). Given a fault model, AutoFault generates a Boolean formula solved by a SAT solver. However, AutoFault's methodology is specific to the cryptographic algorithm evaluated and cannot be adapted for CPU analysis.

A second category, represented by VerFI, FIVER, and FIRMER in Table 1.2, evaluates whether faults can modify circuit behavior by comparing fault-free and faulted versions of the circuit. These techniques are well-suited for evaluating the

security of hardware-level countermeasures designed to detect or correct faulty values. VerFI [AWMN20] introduces an automated process operating at the bit-level granularity on circuit netlists to evaluate countermeasure robustness. The tool simulates all possible faults specified in the fault model and classifies them as *detected*, *ineffective*, or *non-detected*. FIVER [RRSS+21] extends VerFI by exhaustively covering the entire state space using formal methods. Through bounded equivalence checking, the circuit is unrolled over several clock cycles with symbolic input values and faults, and the outputs are compared to reveal potential fault consequences on cryptographic circuits. FIRMER [TGC+23] presents a methodology to classify faults as effective or ineffective, using a new encoding of the problem with SAT solvers instead of Binary Decision Diagrams (BDD) as in FIVER. It is noteworthy that FIRMER can evaluate AES against one fault in a few minutes and two faults in approximately 30 minutes.

SYNFI [NOV+22] is another relevant tool operating at the hardware level. It is a pre-silicon fault analysis framework that allows hardware designers to evaluate the robustness of a circuit and its countermeasures against faults. The authors demonstrated that SYNFI can also evaluate small parts of CPUs, such as finite-state machines.

Despite their capabilities, existing hardware-level tools have limitations that prevent their use in evaluating fault consequences at the software level on CPU-based systems. First, they lack support for specifying the executed program, and the evaluated properties either focus only on algorithm-specific attacks like DFA or on evaluating whether the countermeasure detects the fault. Consequently, the verification techniques employed are not suited to observe fault propagation in the circuit and evaluate its final consequences on software execution. Second, most of these tools evaluate circuits over a single clock cycle, such as a single encryption round, while CPUs need to be evaluated over multiple clock cycles, as faults may have visible consequences only after a certain time period.

1.3.2 Software

At the software level, a wide variety of tools have been developed to evaluate fault effects. These tools typically take a binary program as input and execute it with faults for a given instruction set, such as RISC-V or ARM. The following describes various techniques, including simulation, static analysis, and symbolic execution, to determine if faults can lead to undesired instruction executions.

In the safety and dependability domain, SymPLFIED [PNKI08] was one of the first works to propose a formal framework based on bounded model checking to evaluate fault effects on running programs. In the security domain, relevant works are more recent. One category of work relies on simulation [Ris20, HSP21, HGA+21]. ARMORY [HSP21] is a framework capable of automatically injecting faults during program execution using an ARMv7-M emulator to analyze their effects. Simi-

larly, ARCHIE [HGA+21] injects faults into software executed on an emulator and supports multiple architectures, including ARM, RISC-V, and x86. FiSim [Ris20] injects faults into instructions to determine if specific attack goals, such as skipping a password check, can be achieved.

A second category of tools relies on static analysis to assess program robustness to fault injection without executing it. Tools like RobustB [BHE+19] and SAMVA [GHHR23] evaluate the consequences of the instruction skip fault model based on a structural evaluation of the control-flow graph.

A third category of work uses symbolic execution, a formal technique where program inputs are treated as symbolic variables representing a range of possible values rather than a single concrete value [Kin76, BCD+18]. Faults injected into the program can also be treated as symbolic variables, allowing symbolic execution to explore multiple paths simultaneously. This technique is exhaustive but suffers from the state space explosion problem, limiting the number of instructions that can be analyzed. It has been implemented using the intermediate language of the LLVM compiler [PMPD14] and on binary programs [DBP23], scaling up to ten faults.

While formal techniques such as static analysis or symbolic execution offer exhaustive evaluation of software security, these frameworks perform their analysis using architectural models instead of actual implementations. Consequently, they are unable to identify vulnerabilities induced by subtle microarchitectural effects.

1.3.3 Combined Frameworks

The earliest works proposing combined hardware/software methodologies to evaluate fault effects were in the context of safety. These tools, based on simulation techniques, aimed to assess the dependability of fault-tolerant systems against cosmic radiation. In 1994, MEFISTO [JAR+94] was among the first to introduce automated techniques for emulating hardware faults and observing their consequences on running software. This work involved modifying a 32-bit processor design by inserting a *saboteur* or *mutating* the RTL model to reproduce fault injection effects. MEFISTO analyzed two sorting programs with instructions directly encoded in the VHDL description. This allowed the authors to classify the observed fault consequences during program execution and report the fault latency between the injection instant and its visible manifestation. Similar tools include VERIFY [STB97] and LIFTING [BN08].

To our knowledge, the first work proposing a combined hardware/software evaluation tool to assess security against fault attacks was SimpliFI [GS21] in 2021. SimpliFI offers a framework for hardware simulation of a processor running software. The authors included timing information in their hardware model to reproduce realistic faults similar to clock glitch attacks. The simulator can shorten the clock period during one cycle to create timing violations. The usability of this approach was demonstrated on a RISC-V embedded processor running an AES application.

All the aforementioned tools rely on simulation techniques to explore the consequences of fault injections on software. However, simulation is limited in its ability to explore the entire state space, including all possible inputs and fault attacks, and cannot provide security guarantees when the completeness threshold is not reached. Conversely, exhaustive techniques such as formal methods, employed in hardware- and software-only tools, have proven to be better suited for exploring large state spaces and are valuable for identifying corner-case vulnerabilities that simulation-based approaches might miss. To the best of our knowledge, no such formal method-based tool currently exists, which naturally leads us to the following questions:

Questions for Problem Statement

Q6. Can existing formal verification techniques at the hardware or software layers be adapted to cross-layer methodologies for evaluating fault effects?

Q7. Can the state explosion problem, inherent to the exhaustive exploration of system behaviors be overcome or contained?

1.4 Problem Statement and Manuscript Outline

Overview. This chapter has reviewed the literature on fault injection attacks for processor security evaluations. In Section 1.1, we presented fault injection attacks, described their effects at various abstraction layers, and introduced characterization techniques to derive the so-called *fault models*. Throughout this section, open questions were raised, emphasizing the need for cross-layer fault evaluation to understand the causal relationship between hardware-level fault effects and their consequences on software security (Questions Q1 and Q2). Additionally, the necessity for automated verification techniques to discover corner-case vulnerabilities that manual approaches might miss was highlighted (Questions Q3 and Q4).

In Section 1.2, we presented existing fault countermeasures at the software layer, the hardware layer, and combinations of both. In the conclusion of this section, we stressed the need for exhaustive security evaluation techniques to ensure that protections effectively thwart fault attacks (Question Q5).

Finally, in Section 1.3, we explored existing tools and methodologies to evaluate the consequences of various fault models. The tools presented operate at different abstraction layers and employ various verification techniques such as *simulation* and *formal methods*. This section concluded by emphasizing the benefits of formal methods for exhaustively exploring the state space and noted the absence of such methodologies for combined hardware/software evaluation (Questions Q6 and Q7).

These observations and the open questions raised in this preliminary chapter lead us to formulate the following problem statement, which this manuscript addresses:

Problem Statement. Establishing exhaustive and automated analysis methods comprising both software and hardware system descriptions is crucial to better understand the final consequences of hardware-level faults or provide formal guarantees of software security.

However, developing such an analysis method involves several significant challenges.

1) System modeling. Exhaustive verification techniques such as formal methods require establishing a modeling of the system to reason about and prove invariants or security properties. This modeling process is challenging because it requires describing both the software and hardware layers to accurately represent the security objectives of an attacker and the effects of hardware faults, which are the root cause of the attack. As described in Section 1.3, existing tools have proposed combined hardware/software models for simulation purposes only.

2) State space explosion problem. Analyzing fault effects on systems modeled at the software or hardware level is known to encounter scalability issues, as highlighted in Section 1.3. Establishing an exhaustive cross-layer verification approach will undoubtedly face similar challenges since the analysis must explore the entire system state space, encompassing all system behaviors and fault effects. Addressing the state space explosion problem is therefore essential for the practical applicability of a cross-layer fault evaluation methodology.

Outline. To address the problem statement outlined above and overcome the associated challenges, the contributions of this thesis are organized into three chapters.

First, Chapter 2 introduces µArchiFI, a formal modeling and verification methodology to evaluate fault effects on combined hardware/software systems. This chapter proposes a first solution by integrating the hardware description of the processor with the running program into a single model suitable for formal verification techniques.

Second, Chapter 3 evaluates the security of several software and hardware case studies using µArchiFI. We first validate the methodology by automatically identifying known fault attacks exploiting microarchitectural mechanisms such as forwarding, and by discovering previously unreported fault effects. Then, we formally evaluate the security of a combined countermeasure, which would not have been feasible through hardware or software verification alone. Finally, we report the performance of µArchiFI approach across various use cases.

Third, Chapter 4 enhances the methodology by developing a preliminary analysis of hardware-level countermeasures. This improvement simplifies the co-verification process since only faults not captured by hardware protections need to be evaluated, therefore helping to contain the state space explosion.

Chapter 5 concludes this manuscript and discusses future research directions.

Chapter 2

µArchiFI Workflow: Formal Modeling and Implementation

Contents

The background on fault injection attacks presented in Chapter 1 led us to the following conclusion: a comprehensive, automated verification methodology to analyze fault effects, considering both hardware and software layers, would be a significant step forward in evaluating system security.

This Chapter 2 is the first contribution of this Ph.D. thesis. We introduce an initial solution to the problem statement through the µArchiFI tool. µArchiFI is a workflow that combines the hardware description of a processor with a binary program to formally analyze the effects of faults. To achieve such a tool, we first need to integrate the different components of our system—hardware, software, faults—into a single model. This is the focus of Section 2.1. Section 2.2 presents the background on model-checking techniques to verify whether the system model satisfies a given property. This section also reviews the languages and model-checking tools commonly used in the literature. To automate the security evaluation of our system, we need to automatically build the system model and convert it into a language suitable for formal verification techniques. Several tools are available to facilitate this transformation. This is notably the case with Yosys, which is introduced in Section 2.3. Finally, Section 2.4 describes µArchiFI's workflow, details its usage, and provides two verification strategies to enhance security evaluation.

The content of this chapter is adapted from our publication at *Workshop on Fault Detection and Tolerance in Cryptography (FDTC)* [TAC+22] in 2022, and *Formal Methods in Computer Aided Design (FMCAD)* [TAC+23] in 2023. This work was also presented at *Groupe de Travail sur les Méthodes Formelles pour la Sécurité (GTMFS)* [Tol23] in 2023.

2.1 System Modeling

The methodology we propose in this chapter aims to evaluate the security of a hardware/software system. This section addresses our first challenge: combining the hardware description of the processor with the running software program into a single model. This model must also represent how fault injection attacks alter system operation. The rest of this section is organized as follows.

First, we describe hardware circuits as a directed graph and model their execution through time with a Mealy machine [Mea55]. Second, we characterize the software as constraints on the system to restrict the possible executions to those corresponding to the evaluated program. Third, we describe how fault attacks modify the circuits' operation by introducing new *faulty* behaviors during system execution. Finally, we define the attacker goal as a reachability property. The notations used in this thesis are summarized in Table 2.1.

Table 2.1: Notations used in this manuscript to model hardware/software systems.

	Notation			Description
Structural Circuit View	$\mathcal{C}$	$=$	(G, W)	Circuit modeled as a directed graph.
	G	$=$	$I \cup O \cup R \cup C$	Set of circuit elements a.k.a. gates.
	W	$\subseteq$	$G \times G$	Set of wires connecting two gates.
	I	$=$	$\{x_1, \ldots, x_{\lvert I\rvert}\}$	Set of circuit inputs.
	O	$=$	$\{y_1, \ldots, y_{\lvert O\rvert}\}$	Set of circuit outputs.
	R	$=$	$\{r_1, \ldots, r_{\lvert R\rvert}\}$	Set of sequential gates a.k.a. registers.
	C	$=$	$\{c_1, \ldots, c_{\lvert C\rvert}\}$	Set of combinational gates.
Functional Circuit View	X	$\subseteq$	$\mathbb{B}^{in}$	Set of variable vectors holding input values.
	Y	$\subseteq$	$\mathbb{B}^{out}$	Set of variable vectors holding output values.
	S	$\subseteq$	$\mathbb{B}^{states}$	Set of variable vectors holding circuit states.
	g	$:$	$\mathbb{B}^u \to \mathbb{B}^v$	Boolean function implemented by gate g.
	$\texttt{val}(g)$	$\in$	$\mathbb{B}^v$	Gate output value.
Temporal Circuit View	$\mathcal{M}_\mathcal{C}$	$=$	$(X, Y, S, S_0, \delta, \lambda)$	Mealy machine associated with circuit $\mathcal{C}$.
	S_0	$\subseteq$	$X \times S$	Set of variable vectors holding initial states.
	δ	$:$	$X \times S \to S$	Next-state transition function.
	λ	$:$	$X \times S \to Y$	Circuit output function.
	$(s_i)_{i=1}^n$	$\in$	S^n	Circuit execution trace of length n.
Faults	$\mathcal{F}$	$\subseteq$	$G \times E$	Transient fault model.
	e	$:$	$\mathbb{B}^v \to \mathbb{B}^v \in E$	Fault effect as a Boolean function.
	$\mathbf{F}$	$\subseteq$	$G \times E \times \mathbb{N}$	Fault attack a.k.a. timed fault model.
	k	$=$	$\lvert\mathbf{F}\rvert \in \mathbb{N}$	Fault attack order.
	φ	$:$	$X \times S \to \{0, 1\}$	Attacker goal.

2.1.1 Hardware Modeling

Modeling a system to analyze fault effects requires multiple levels of representation. First, a structural view of the hardware, formalized in Definition 2.1, allows to accurately describe which hardware component is targeted with a fault by an attacker. Then, a functional view of the circuit, formalized in Definition 2.2, defines the logical functions the circuit implements. This functional view is also essential to model the fault effects and to describe how such an attack alters the system operation. Finally, since circuits process values through time, we need to represent its behavior as a transition system.

Definition 2.1 (Circuit Model). A hardware circuit is structurally defined as a directed graph $\mathcal{C} = (G, W)$, where G is a set of circuit elements (gates), and $W \subseteq G \times G$ is the set of wires connecting the gates. Furthermore, each gate $g \in G$ has a type and belongs to one of the disjoint sets representing inputs I, outputs O, register gates R, and combinational gates C such that $G = I \cup O \cup R \cup C$. Additionally, every loop in the circuit must contain at least one register $r \in R$ to prevent combinational loops.

Example 2.1. Figure 2.1 illustrates a simple circuit example where $I = \{x_1, x_2, x_3\} \subseteq$

G are the inputs, $R = \{r_1, r_2, r_3\} \subseteq G$ are the registers, $C = \{c_1, c_2, c_3\} \subseteq G$ are the combinational gates, and $O = \{y_1\} \subseteq G$ is the circuit output.

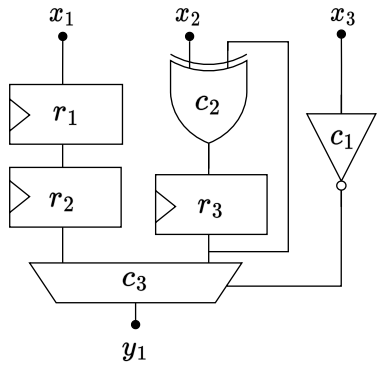


Figure 2.1: Simple circuit example.

Since circuits are designed to process digital values, Definition 2.2 introduces the circuit functionality.

Definition 2.2 (Gate Function). Let $\mathcal{C} = (G, W)$ be a circuit. Each structural gate $g \in G$ of the circuit implements a Boolean function $g\colon \mathbb{B}^u \to \mathbb{B}^v$ where $\mathbb{B}$ is the Boolean field $\{0, 1\}$, and u (resp. v) is the number of input (resp. output) bits of the gate g. We denote $\texttt{val}(g) \in \mathbb{B}^v$ the output value of gate g.

In the rest of this work, we consider sequential circuits, and we assume that all registers $r \in R$ and inputs $x \in I$ are synchronized on the same clock signal. Consequently, we use the clock cycle as the timing unit of the circuit. As a result, since circuits process values through time, we model their execution as a Mealy Machine [Mea55].

Definition 2.3 (Mealy Machine). Let $\mathcal{C} = (G, W)$ be a circuit, $I = \{x_1, \ldots, x_{|I|}\} \subseteq G$ be its inputs, and $R = \{r_1, \ldots, r_{|R|}\} \subseteq G$ be its registers, where $|\cdot|$ is the cardinality operator. We model the circuit execution as a Mealy machine $\mathcal{M} = (X, Y, S, S_0, \delta, \lambda)$ where X, Y, S, and S_0 are the set of inputs, outputs, states, and initial states, respectively.

Furthermore, let $s = \big(\texttt{val}(r_1), \ldots, \texttt{val}(r_{|R|})\big)$, $x = \big(\texttt{val}(x_1), \ldots, \texttt{val}(x_{|I|})\big)$, and $y = \big(\texttt{val}(y_1), \ldots, \texttt{val}(y_{|I|})\big)$ be variable vector holding the values for the states $s \in S$, the inputs $x \in X$, and outputs $y \in Y$, respectively.

$\delta\colon X \times S \to S$ denotes the next-state function, and $\lambda\colon X \times S \to Y$ is the output function. Finally, a transition between two states $s, s' \in S$ is valid iff $\exists x \in X$ such that $s' = \delta(x, s)$.

A graphical representation of a Mealy machine $\mathcal{M}$ is given in Figure 2.2. In comparison with the circuit view presented in Definition 2.1, the Mealy machine does not expose the circuit structure, i.e., the logic gates. In contrast, the important elements highlighted in the Mealy machine are the input vector x, the output vector y, and the registers R holding the current state s. Notably, the atomic Boolean

functions implemented by the circuit combinational gates $c \in C$ are hidden within the next-state function δ and the output function λ.

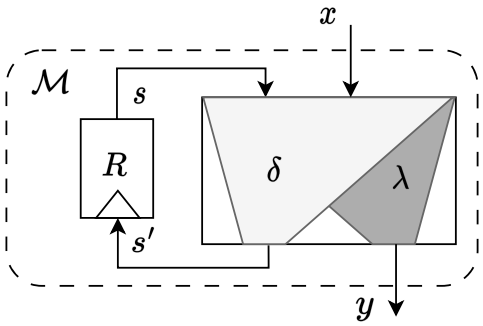
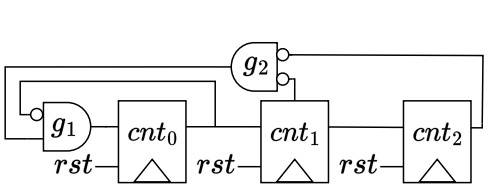
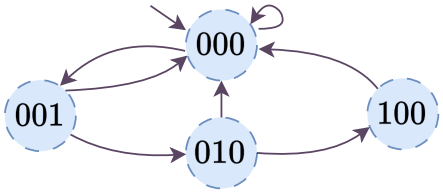


Figure 2.2: Mealy machine $\mathcal{M}$ executing the sequential circuit $\mathcal{C}$.

Example 2.2. Figure 2.3 illustrates a simple 3-bit counter. The left-hand side **(a)** of the figure shows the circuit representation of the counter, while the right-hand side **(b)** presents the associated state-transition diagram. Since the circuit has 3 bits of registers, $2^3 = 8$ states are possible. However, only the reachable states are represented in the figure. Valid transitions between states are indicated with arrows, and the incoming arrow on the state (0, 0, 0) symbolizes the initial state.



(a) Counter circuit model.



(b) State-transition diagram of the counter with $s = \{cnt_2, cnt_1, cnt_0\}$.

Figure 2.3: Counter modeling.

Finally, Definition 2.4 defines a sequence of consecutive circuit states called an *execution trace* where each state depends on its predecessor.

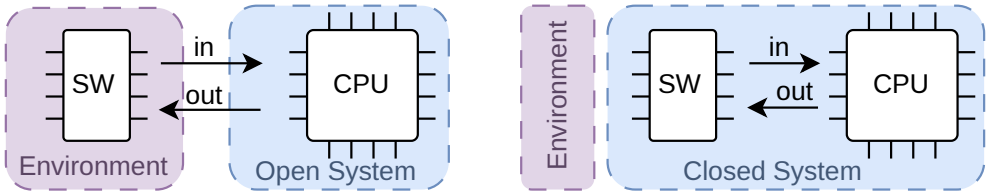
Definition 2.4 (Execution Trace and Reachability). Let $\mathcal{C} = (G, W)$ be a circuit and $\mathcal{M} = (X, Y, S, S_0, \delta, \lambda)$ be the associated Mealy machine. An *execution trace* $(s_i)_{i=1}^{n} \in S^n$ is a sequence of n circuit states $(s_1, \ldots, s_n)$ such that $s_1 \in S_0$, and $\forall i < n, \exists x \in X, s_{i+1} = \delta(x, s_i)$. Furthermore, we say that a state $s_n \in S$ is *reachable* if there exists such an execution trace.

2.1.2 Software Modeling

As this Ph.D. thesis focuses on hardware/software systems, the central component to model is the processor, also known as a central processing unit (CPU). In the

previous paragraphs of this section, we described the structural, functional, and temporal views of a circuit. In the following, we see how to restrict the range of behaviors exhibited by CPU-based circuits to those specified by the program of interest.

Essentially, a program is composed of instructions and data stored in memory. The processor interacts with memory by fetching instructions for computation and by reading/writing data. During its operation, the processor updates its internal state by storing values in the register file. Modeling the execution of a given program on a processor implies representing these memory operations in the circuit transition system defined in Definition 2.3. As illustrated in Figure 2.4, two possible solutions exist depending on whether the memory is included in the modeling or not.



(a) Software as environmental constraints. **(b)** Software as internal state.

Figure 2.4: Software modeling.

The first option, shown in Figure 2.4a, excludes the program memory from the system modeling. The Mealy machine $\mathcal{M}$ that models the processor behavior treats the program as environmental constraints on the input values X. This approach has the advantage of maintaining a smaller size for the resulting hardware/software system since the memory is not modeled alongside the CPU. However, the main drawback is the challenge of generating these environmental constraints. As detailed in the next section, precomputing these constraints is not always possible due to non-deterministic control flow resulting from symbolic data or fault attacks. Additionally, these environmental constraints must be updated according to the store operations performed by the CPU. For example, when the processor performs a read operation after a write at a specific memory location, the value provided by the environment must match the one previously written back into memory.

A second option, shown in Figure 2.4b, includes the instruction and data memories in the system modeling. The hardware encompasses both the processor design and the program memory. Instructions and data are encoded in the initial states S_0 of the Mealy machine $\mathcal{M}$. The resulting system is closed as it does not interact with its environment. This approach is easier to implement since the memory content is updated along with the processor execution. However, the main limitation is the increase in the system's state space. This latter option is the one we use for the remainder of this chapter, while keeping the memory size as small as possible.

2.1.3 Faults Injection Attacks

As mentioned in Chapter 1, attackers can inject faults during circuit computation to induce incorrect system behavior. In Sections 2.1.1 and 2.1.2, we formalized a hardware/software system as a transition system whose initial state corresponds to the program to be executed. In the following, we extend this modeling to describe an attacker's ability to modify system operation using fault injection attacks.

First, Definition 2.5 defines the *fault model* as a functional modification of circuit operation. Second, Definition 2.6 introduces the transient temporal dimension and defines a *transient fault attack*, which modifies an execution trace. Finally, Definition 2.7 introduces the *attacker goal* as a reachability problem over a transition system achieved through fault injection attacks.

Definition 2.5 (Fault Model). Let $\mathcal{C} = (G, W)$ be a circuit. A *fault model* $\mathcal{F} \subseteq G \times E$ for the circuit $\mathcal{C}$ is a set of pairs where each fault $(g, e) \in \mathcal{F}$ has a *fault location* $g \in G$, and a *fault effect* $e \in E$. As a consequence, the fault model results in a faulty circuit $\mathcal{C}^{\mathcal{F}}$ where each fault $(g, e) \in \mathcal{F}$ causes gate g to compute $e \circ g$, with $g \colon \mathbb{B}^u \to \mathbb{B}^v$ the gate function, and $e \colon \mathbb{B}^v \to \mathbb{B}^v$ the fault effect.

Intuitively, a fault model $\mathcal{F}$ describes the range of possible faults an attacker can inject into the circuit. The *fault locations* correspond to the circuit gates affected by the attack and the *fault effects* represent the functional modifications induced in the gate due to the perturbations. As an example, unary fault effects E_1 for bit-level gates include *bit-reset*: $x \mapsto 0$, *bit-set*: $x \mapsto 1$, and *bit-flip*: $x \mapsto \neg x$.

Although in general $\mathcal{F} = G \times E$, Definition 2.5 allows the user to restrict the set of considered faults to a subset. This restriction occurs either because the attacker cannot fault certain gates due to protection or infeasibility, or because only specific fault effects can be introduced due to circuit technology or fault injection methods.

Example 2.3. Electromagnetic fault injections are known to induce an erroneous sampling process in the circuit, which is akin to bit flips in registers (cf. Section 1.1). Figure 2.5 depicts such a situation where fault f_i induces a *bit-flip* in register r_i.

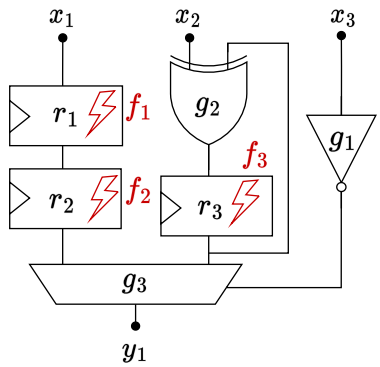
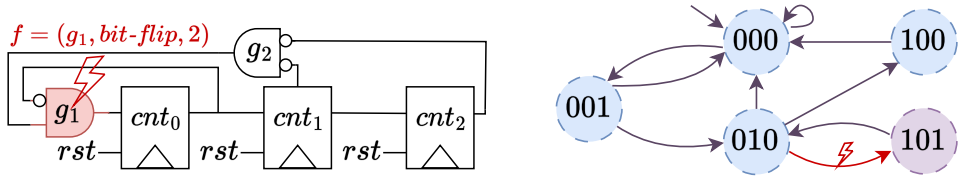


Figure 2.5: Fault model example affecting sequential gates only.

The fault model provides a structural and functional description of faults. However, to evaluate the consequences of faults on hardware/software systems and to enable the attacker to control the timing of the fault injection, we need to include a timing representation in our fault model. Definition 2.6 defines the notion of *transient fault attacks*, which modifies the execution of the Mealy machine.

Definition 2.6 (Transient Fault Attack). Let $\mathcal{C}$ be a circuit, $\mathcal{M} = (X, Y, S, S_0, \delta, \lambda)$ be its associated Mealy machine, and $\mathcal{F} \subseteq G \times E$ be a fault model. A *transient fault attack* $\mathbf{F} \subseteq \mathcal{F} \times \mathbb{N}^*$ is a set of timed faults injected during the execution of $\mathcal{M}$. For each fault $(g, e, i) \in \mathbf{F}$, the temporal dimension of the fault $i \in \mathbb{N}^*$ is denoted as the *fault timing*. As a consequence, the fault-free transition system $\mathcal{M}$ results in a faulty Mealy machine $\mathcal{M}^{\mathbf{F}} = (X, Y, S, S_0, \delta^{\mathbf{F}}, \lambda^{\mathbf{F}})$ where the faulty function $\delta^{\mathbf{F}}$ and $\lambda^{\mathbf{F}}$ relies on a faulty version of the circuit to compute the next states and the outputs. In other words, any valid execution trace $(s_i)_{i=1}^n$ yields a faulty execution trace $\left(s_i^{\mathbf{F}}\right)_{i=1}^n$ where each fault $(g, e, i) \in \mathbf{F}$ causes gate g to compute $e \circ g$ at clock cycle i. Finally, the number of faults $|\mathbf{F}|$ is referred to as the *attack order*.

Example 2.4. Let us consider Figure 2.6, which is an adaptation of the 3-bit counter previously introduced in Example 2.2. Let $\mathbf{F} = \{f\}$ be a transient fault attack with an attack order equal to 1—a single fault in injected—targeting the gate g_1, at cycle 2, with a *bit-flip* effect **(a)**. This results in a faulty state-transition diagram **(b)** where the fault leads to the previously unreachable system state $(1, 0, 1)$.



(a) Counter circuit model.

(b) State-transition diagram of the faulty counter with $s = \{cnt_2, cnt_1, cnt_0\}$.

Figure 2.6: Counter modeling under fault attack $\mathbf{F}$.

At this stage of the section, we have defined a hardware/software system modeling including the possible perturbations an attacker can induce with a fault attack. However, in practice, an attacker utilizes this fault attack on a CPU-based system to create an exploit. Definition 2.7 formalizes this *attacker goal*.

Definition 2.7 (Attacker Goal). Let $\mathcal{C}$ be a circuit, $\mathcal{M} = (X, Y, S, S_0, \delta, \lambda)$ be its associated Mealy machine, and $\mathcal{F}$ be a fault model. An *attacker goal* is a Boolean predicate φ over inputs X and circuit states S determining whether they are desirable, i.e., $\varphi : X \times S \to \{0, 1\}$.

We say that an attacker reaches his goal φ at attack order k if there exists a fault attack $\mathbf{F} \subseteq \mathcal{F} \times [1, n]$, with $|\mathbf{F}| \leq k$, a faulty execution trace $\left(s_i^{\mathbf{F}}\right)_{i=1}^n$, and an input vector $x \in X$ such that $\varphi\left(x, s_n^{\mathbf{F}}\right) = 1$.

Example 2.5. Known attacks on a CPU-based system involve executing a malicious payload. The associated attacker goal $\varphi_{payload}$, which evaluates to *true* when the attack succeeds, can be expressed as an equality between the program counter (PC) value and the payload address: $\varphi_{payload} := (PC = @_{payload})$.

Alternatively, another kind of attack involves reading sensitive data in memory: $\varphi_{secret} := (read\_address = @_{secret})$.

2.1.4 Summary

This section has formalized the modeling of a CPU-based system, including its hardware description, software program, faults, and attacker goal. Figure 2.7 summarizes the proposed modeling and represents system security evaluation as a reachability problem on the faulty state-transition diagram.

First, the hardware circuit is modeled as a Mealy machine, where each green node represents a circuit state and green arrows indicate valid transitions between states. Second, the software program is encoded in the system's initial state, restricting possible execution paths to those corresponding to the program of interest, marked in blue. Third, purple transitions represent faulty transitions allowed by the considered fault model $\mathcal{F}$ over time, including transitions to previously unreachable system states. Finally, undesirable circuit states, where the attacker goal φ evaluates to *true*, are symbolized with red nodes. Given an attack order k, a reachability analysis on this Mealy machine can determine if an attacker can reach his goal using at most k faulty transitions.

Figure 2.7 provides a comprehensive overview of the system modeling to keep in mind for the remainder of this chapter, as it serves as a baseline for building µArchiFI. The next section focuses on reachability analysis techniques to evaluate whether the system is vulnerable to fault attacks.

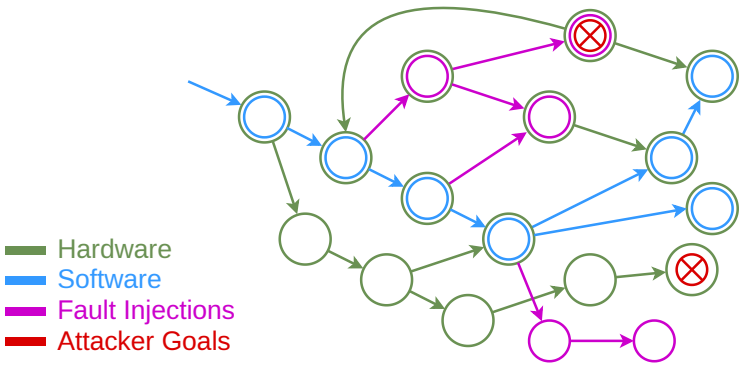


Figure 2.7: Fault injection modeling on a transition system.

2.2 Background on Model Checking

Section 2.1 has laid the foundation of µArchiFI by integrating the hardware and software descriptions, along with faults, into a single model. In this section, we review model checking, an automated formal technique used to verify whether a system satisfies a given property.

Verifying properties on transition systems is a well-known and extensively studied area. Consequently, this section does not contribute to new findings but summarizes existing results on model checking. Nevertheless, it is a crucial step toward achieving µArchiFI, as understanding these methods and selecting the best-suited verification approach is essential.

First, we provide a general overview of model checking before focusing on symbolic methods. Next, we briefly describe decision procedures and discuss some common abstractions. Finally, we present state-of-the-art model-checking languages and tools dedicated to hardware verification.

2.2.1 Overview

Back in the 1980s, computer science was experiencing exponential growth, and computer-aided verification was synonymous with unsolvable problems [Tur37, Ric53]. In 1981, Clarke and Emerson introduced *model checking* [CE81] as an algorithmic method for determining if a system satisfies certain properties expressed in a temporal logic specification. This invention marked a paradigm shift from intractable correctness proofs using deductive reasoning to bug finding and model verification. In today's applications, the major benefit of model checking is its ability to efficiently catch difficult corner-case errors. Additionally, when no bugs are found, it provides a complete proof that the system satisfies the formal specification.

In this work, we focus on a type of temporal properties known as *reachability properties* as defined in Definition 2.7. Roughly speaking, reachability analysis answers the question "*Can the system reach an undesirable state?*".

An initial approach to performing model checking relies on search algorithms, such as depth-first search and breadth-first search, on the transition system. This method is referred to as *explicit model checking*. States are explored successively until a reachability property φ is satisfied. All reachable states are visited as the algorithm starts from the initial states and visits their direct successors. For finite transition systems, as is the case for hardware circuits, reachability analysis theoretically terminates. However, in practice, the number of states to explore may be prohibitively large, consuming excessive run-time or memory space, leading to the classical *state-space explosion problem*. For instance, the 3-bit shown in the previous section (Example 2.2) only has four reachable states to explore. In contrast, a chip with 100 flip-flops has $2^{100} = 1267650600228229401496703205376$ states to explore.

The systems we analyze in the following chapters of this manuscript have thousands of flip-flops. To address this state-space explosion problem, symbolic techniques have been proposed.

2.2.2 Symbolic Model Checking

The key idea of *symbolic* model checking [BCM+92] is to reason about sets of states instead of reasoning on each state individually. For this purpose, both the transition relation and the set of states are modeled by Boolean functions and represented symbolically using propositional logic formulas.

Characteristic function. Let $\mathcal{M} = (X, Y, S, S_0, \delta, \lambda)$ be a Mealy machine, and let $Q \subseteq S$ be a subset of states. The characteristic function of Q, denoted by $[\![Q]\!]$, is a Boolean function over state variables of S, defined as follows:

$$[\![Q]\!] : \begin{array}{ccc} S & \to & \{0,1\} \\ s & \mapsto & \begin{cases} 1 & \text{if } s \in Q \\ 0 & \text{otherwise} \end{cases} \end{array}$$

That is, $[\![Q]\!](s)$ evaluates to *true* if and only if s is an element of Q. Similarly, we define the characteristic function of the transition relation $\delta \colon X \times S \to S$, denoted $[\![\delta]\!]$, by a Boolean function as follows:

$$[\![\delta]\!] : \begin{array}{ccc} S \times S & \to & \{0,1\} \\ (s, s') & \mapsto & \begin{cases} 1 & \text{if } \exists x,\ s' = \delta(x, s) \\ 0 & \text{otherwise} \end{cases} \end{array}$$

In other words, $[\![\delta]\!](s, s')$ evaluates to *true* if and only if (s, s') is a valid transition of the Mealy machine $\mathcal{M}$.

Successor Function. Let $\mathcal{M}$ be a Mealy machine, and let $Q \subseteq S$ be a subset of states. The set of successor states of Q, i.e., the reachable states after a 1-step transition, is defined as follows:

$$\left\{ s' \in S \mid \exists s \in Q, \exists x \in X \colon s' = \delta(x, s) \right\}$$

This set is symbolically represented by the Boolean successor function **succ** as follows:

$$\mathbf{succ}([\![Q]\!](s)) := \left(\exists s : [\![Q]\!](s) \wedge [\![\delta]\!](s, s')\right) [s' \rightsquigarrow s]$$

Intuitively, $\exists s : [\![Q]\!](s) \wedge [\![\delta]\!](s, s')$ is the symbolic representation of successors states of Q, and $[s' \rightsquigarrow s]$ renames variables s' so that the resulting predicate is over current state variables s. That is, $\mathbf{succ}([\![Q]\!](s))$ evaluates to *true* if and only if s is a valid successor state of Q.

Symbolic Model Checking. Let $\mathcal{M} = (X, Y, S, S_0, \delta, \lambda)$ be a Mealy machine. By means of the characteristic and the successor functions previously defined, we can iteratively compute the set of reachable states as illustrated in Figure 2.8.

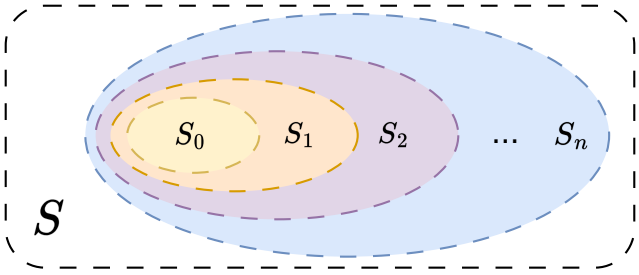


Figure 2.8: Symbolic reachability analysis.

Algorithm 2.1 provides a high-level view of symbolic model checking for a reachability property φ. First, the algorithm initializes the symbolic predicate *reachable* with the predicate associated with the initial states S_0 (line 1). For each iteration, the set of 1-step successors is computed using the **succ** function (line 5). For each new set of reachable states, the algorithm returns *true* is the property φ holds (line 3). Otherwise, the algorithm iterates until it reaches a fixed point (line 6), i.e., it cannot visit new states and $S_n = S_{n+1}$.

Algorithm 2.1: Symbolic reachability analysis using model checking.

Input: a Mealy machine $\mathcal{M} = (X, Y, S, S_0, \delta, \lambda)$, and a state property φ.
Output: a Boolean value indicating whether φ is reachable.

```
1 reachable ← [[S_0]](s);
2 while true do
3     if (SAT(reachable ∧ φ)) then
4         return true;
5     tmp ← reachable ∨ succ(reachable);
6     if (tmp ⟺ reachable) then
7         return false;
8     reachable ← tmp;
```

While symbolic model checking has significantly enhanced verification capabilities by several orders of magnitude [BCM+92], some problems remain intractable. In practice, the number of iterations can be excessively high and Algorithm 2.1 may not converge to a fixed point.

Bounded Model Checking. Bounded Model Checking (BMC) [CBRZ01] limits the search to states reachable within n system transitions. Consequently, BMC is not sound for proving reachability properties, as it does not consider traces $(s_i)_{i=1}^{m>n}$ longer than n transitions. However, BMC is an efficient framework for detecting any bugs that are reachable in less than n transitions. In this context, BMC is said to be n-complete. Algorithm 2.2 introduces a revised version of Algorithm 2.1 using bounded model checking. In contrast with the previous version, Algorithm 2.2 takes a bound n as input and terminates when the *for* loop (line 2) has been completed.

Algorithm 2.2: Symbolic reachability analysis using *bounded* model checking.

Input: a Mealy machine $\mathcal{M}$, a state property φ, and a bound n.
Output: a Boolean value indicating whether φ is reachable within n steps.

```
1 reachable ← [[S_0]](s);
2 for i from 1 to n do
3     if (SAT(reachable ∧ φ)) then
4         return true;
5     reachable ← reachable ∨ succ(reachable);
6 return false;
```

In practice, BMC is often sufficient for catching or proving the absence of bugs, offering a good trade-off between performance and completeness. The *bound* is typically chosen according to practical knowledge on the evaluated system, such as the length of the program under evaluation. Several works in the literature propose methods to extend bounded techniques to provide unbounded guarantees. Examples include induction [SSS00], interpolation [McM03], and property-directed reachability [Bra11, EMB11].

In the remainder of this chapter, we choose to rely on BMC techniques to evaluate the reachability of the attacker goal in the hardware/software modeling introduced in Section 2.1. The bound is chosen according to the length of the program to be analyzed on the hardware.

2.2.3 Decision Procedures

The underlying concept behind symbolic model checking is the reasoning about sets of states using characteristic propositional formulas. While model-checking algorithms iteratively compute the set of reachable states, as illustrated in Algorithm 2.1, they require efficient decision procedures to determine if a formula φ holds for the current set of reachable states (line 3). A *decision procedure* is an algorithm that, given a decision problem, terminates with a correct yes/no answer [KS16]. In the remainder of this subsection, we introduce BDD-, SAT-, and SMT-based decision procedures.

Binary Decision Diagrams (BDDs). Randal E. Bryant has introduced binary decision diagrams in the mid-80s [Bry86, Bry92]. BDDs are efficient data structures to manipulate Boolean expressions based on graph theory and are certainly one of the most famous breakthroughs in the area of formal verification. Each logical operation performed on the propositional formula has an efficient set-theoretic correspondence on BDDs. For example, a conjunction between two formulas is an intersection, a disjunction is a union, and a satisfiability check is a non-emptiness check.

Boolean Satisfiability (SAT). The Boolean satisfiability problem determines whether a variable assignment exists such that a given propositional formula evaluates to true. Besides its theoretical interest due to its NP-completeness, SAT has found many applications, such as model checking. Even if it is very likely that there is no polynomial algorithm to solve these problems, SAT solvers have undergone remarkable improvements since the 1990s. Modern solvers implement search-based algorithms based on the Davis-Logemann-Loveland algorithm (DPLL) [DLL62] or, more recently, the Conflict Driven Clause Learning (CDCL) [MS99].

Satisfiability Modulo Theories (SMT). When encoding systems from programming languages like C into propositional SAT formulas, complex syntax and semantics, such as modular arithmetic or memory reasoning, require non-trivial decision procedures.

Boolean satisfiability problems are formulated using Boolean propositional logic, but other problems are more naturally and compactly expressed in other logic, such as first-order logic, that includes non-Boolean variables, linear arithmetic operators, or quantifiers. In such cases, we speak of *Satisfiability Modulo Theories*.

Choosing some logical *background theories* is always a trade-off between the expressiveness and the ability to automatically check the satisfiability of formulas. For example, encoding a word-level operation between two 32-bit integers requires a translation into bit-level formulas. This task, also known as *bit blasting*, is expensive as it grows exponentially with the size of the bit vector. This exponential explosion can be avoided using the first-order *theory of bit vectors*. Similarly, handling memories to read and store data structures is expensive to model using bit-level operators. The first-order *theory of arrays* addresses this issue.

The work presented in this thesis focuses on hardware/software systems where bit vectors and memories are extensively used to model CPU-based systems. Consequently, we choose to rely on SMT as the backend decision procedure for the model-checking framework. In the rest of this manuscript, SMT refers to the quantifier-free fragment of the first-order *theory of arrays over bit vectors* (QF_ABV). According to the *18th International Satisfiability Modulo Theories Competition* [SMT23], the best state-of-the-art SMT solvers for the QF_ABV theory are Yices [Dut14] and Bitwuzla [NP23]. All these solvers comply with the Smt-lib language [BST10], which is the standard specification to describe SMT problems. μArchiFI, described in detail in Section 2.4, utilizes these two solvers.

2.2.4 Abstractions in Model Checking

While symbolic model checking improves scalability, it can sometimes be insufficient for addressing complex systems. Abstractions provide a more aggressive approach to mitigating state explosion by reducing the state space through the omission of certain system details. Analyzing the effects of faults on systems modeled at the

software or hardware level is known to encounter scalability issues, as highlighted in Section 1.3 when describing related fault evaluation tools. This scalability issue is also inherent to our cross-layer approach and is one of the challenges we must face in this thesis. The following paragraphs introduce existing model-checking abstraction techniques that can be beneficial to µArchiFI.

Localization Abstraction. Localization abstraction [Kur95, CKV10] hides irrelevant system variables for the property φ under verification. Consequently, the resulting abstract system is built by merging all states that agree on the valuation of visible variables.

Intuitively, we may remove irrelevant parts of the hardware design if they are not used by the program under verification. For example, functional units like multiplication or floating point are unnecessary if the considered program does not perform these operations. However, faults break this intuition, and targeting unused functional units may trigger undesired operations that impact the overall system behavior. Such an example is illustrated in Chapter 3, where a fault in the multiplication module leads to a security breach.

Under- and Over-approximations. Under- and over-approximations focus on simplifying or abstracting the system's state space to ease property verification.

Under-approximation restricts the system's state space to a subset likely to contain errors. For instance, bounded model checking under-approximates classical model checking. Under-approximation is used to find bugs, since if a counterexample (bug) is identified, it is a valid error in the real system. However, this technique cannot guarantee that the system satisfies a given specification if no counterexample is found, as it does not explore all possible states. Consequently, under-approximation is a relevant abstraction for bug finding.

Over-approximation enlarges the system's state space to include additional behaviors that may not exist in the actual system, in order to simplify its overall specification, thereby easing the verification process. For example, symbolic model checking over-approximates explicit model checking as symbolic variables may involve spurious behaviors. Over-approximation ensures that if a property is satisfied in the over-approximated model, it is also satisfied in the actual system. However, if a counterexample is found, it might be a false positive (spurious counterexample) due to the inclusion of extra behaviors. Consequently, over-approximation is a relevant abstraction for proving properties and demonstrating the absence of errors.

Over- and under-approximations are not only applied to system's behaviors but can also be applied to a formula φ by checking an easier-to-solve property φ'. If φ' implies φ, proving that a model $\mathcal{M}$ satisfies φ' implies it satisfies φ. Otherwise, nothing can be concluded. Conversely, if φ implies φ', examplifying that $\mathcal{M}$ does not satisfy φ' with a counterexample implies that φ does not hold on $\mathcal{M}$. Otherwise, nothing can be concluded.

The work presented in this manuscript relies on model-checking techniques, and scalability is a major bottleneck that limits their practical application. Scalability is undoubtedly the biggest challenge we face throughout this work. To address this issue in our contributions, both under- and over-approximations are employed to efficiently analyze hardware/software systems under fault attacks. For under-approximation, *sandboxing*, as described in Chapter 2, is used to improve the identification of security vulnerabilities. For over-approximation, *fault-resistant partitioning* provides a valuable approach to facilitate proving robustness against faults. This optimization is the focus of Chapter 4.

2.2.5 Languages and Tools in Hardware Model Checking

Sections 2.2.1 to 2.2.4 have provided a theoretical introduction to symbolic model checking and underlined how this technique can be employed to perform reachability analysis. This section adopts a more practical view and describes formal specification languages and tools that could be relevant to develop µArchiFI. Since 2007, the *Hardware Model Checking Competition (HWMCC)* [HWM20] showcases the recent advances for hardware symbolic model checkers. The languages and tools overview that follows summarizes these results.

Table 2.2: Languages classically used for hardware model checking.

Language	Year	Level	Description
Aiger 1.9 [BHW11]	2011	Bit-level	HWMCC standard between 2007 and 2017. Describes a bit-level system using an and-inverter graph.
Smv 2.0 [BCC+19]	2019	Word-level	Language of the nuXmv model checker. It supports bit vectors, memories, and infinite types like integers.
Btor2 [NPWB18]	2018	Word-level	Word-level generalization of Aiger to support bit vectors and memories. HWMCC standard since 2019.

Languages. Several languages have been developed over the past few years to describe hardware transition systems for formal verification. Some of these languages, such as Smv [BCC+19], are proprietary and tied to specific verification frameworks. Others have been proposed as standards and have been widely adopted by the community, such as Aiger. These languages vary not only in syntax but also in the features they support for system description. Since 2019, Btor2 has become the new standard for describing hardware systems at the word level. Tools like Yosys and Btor2tools facilitate the conversion of systems described with hardware description languages like Verilog into Btor2. Table 2.2 summarizes the most well-known languages.

Tools. Table 2.3 lists state-of-the-art hardware model checkers. These solvers implement efficient algorithms (see Algorithm 2.1) to explore the system state space and prove the specified formal properties. These model checkers are specialized to evaluate hardware designs as they efficiently handle bit-vector and array data types. Since 2018, all these tools now support the Btor2 format as input.

Table 2.3: Model Checkers for hardware verification.

Tool	Year	Languages	Description
nuXmv [CCD+14]	2014	Smv	Supports finite and infinite transition systems.
Yosys-Bmc [Wolb]	2016	Smt-lib	In-house MC from Yosys, described in Section 2.3.
Ebmc [KP17]	2017	Verilog	Hardware variant of CBMC [KT14].
BtorMC [NPWB18]	2018	Btor2	MC proposed along with the Btor2 language.
CoSA [MMB+18]	2018	Btor2, CoreIR, ...	SMT-based model checker.
Pono [MIL+21]	2020	Btor2	Successor of CoSA. Awarded at HWMCC'19.
Avr [GS20]	2020	Btor2	Winner of HWMCC'20.

The formal modeling of hardware systems and their evaluation using hardware model-checking techniques is a well-established field of research, as attested by the Hardware Model Checking Competition held in conjunction with the CAV and FM-CAD conferences since 2007. The languages and model checkers discussed above are therefore valuable assets for the formal evaluation of systems with faults.

2.2.6 Summary

This section provided a comprehensive overview of model-checking techniques and detailed an algorithmic procedure to perform reachability analysis. We refer the interested reader to the *Principles of Model Checking* [BK08] and the *Handbook of Model Checking* [CHVB18] for more details.

Consequently, symbolic model checking seems perfectly suited to evaluate whether an attacker can reach a vulnerable state in the hardware/software modeling with faults we introduced in Section 2.1. Expressive formal specification and efficient tools also exist as they have been developed for a few decades, as showcased in the Hardware Model Checking Competition [HWM20]. We will rely on them in the remainder of this chapter. However, to automate the complete flow which includes system modeling—Section 2.1—and formal verification—Section 2.2—we must translate the hardware and software description with faults into a formal specification. This is the focus of the next section.

2.3 Background on Yosys

Sections 2.1 and 2.2 represent two major steps in the development of µArchiFI: system modeling and formal verification. To move toward automated security evaluation, we need to automatically build the formal model and provide it to a model checker. Several tools are available to facilitate this transformation. This is notably the case with Yosys, which we describe in this section.

2.3.1 Overview

Yosys [Wolb] is an open-source synthesis tool with a compiler-like infrastructure. Figure 2.9 illustrates the workflow of the tool. Initially, its frontend takes a hardware description language as input and compiles it into Yosys's internal data format, the RTL Intermediate Language (RTLIL). Yosys typically supports languages like Verilog, and VHDL through the third-party GHDL project [Tri]. Once the design is expressed in the RTLIL representation, a wide range of passes can apply transformations to the design, typically performing the various synthesis steps. Finally, Yosys's backends translate the RTLIL design into various outputs, ranging from Verilog netlists to formal languages.

In the following, we provide insights into Yosys to better understand how we utilize and modify it for the purposes of µArchiFI. A comprehensive description of the tool is available online [Wolb].

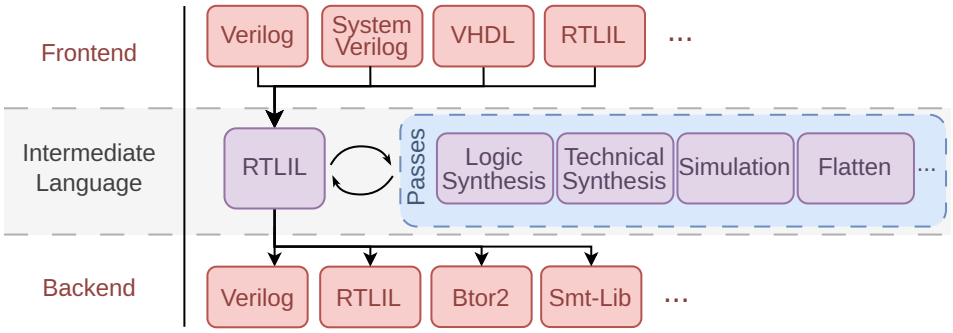


Figure 2.9: Yosys workflow.

2.3.2 Intermediate Representation

RTLIL (Register Transfer Level Intermediate Language) is the intermediate representation used by Yosys for representing hardware designs. It serves as a unified language to apply a wide variety of optimization and transformation passes, as illustrated in Figure 2.9. RTLIL is structured with modules, wires, cells, processes, and attributes.

Modules are the hierarchical blocks of the design that contain wires, cells, processes, and instances of other modules. The design is said to be *flattened* when it has only one module.

Wires represent connections between different parts of the design and carry single-bit or multi-bit digital values.

Cells are the basic building blocks of the design. Different types include combinational cells like `XOR`, `ADD`, `NOT`, sequential cells like `Flip-Flop` and `Latch`,

and formal cells like `Assume`, `Assert`, and `Cover`. Each cell has input and output ports connected to wires.

Processes describe the behavior of sequential logic in the design and include statements for synchronous and asynchronous operations, such as clocked assignments and resets.

Furthermore, each design element has a list of parameters providing additional information, including name, type, data width, etc.

Yosys's RTLIL intermediate representation is reminiscent of the hardware modeling introduced in Section 2.1.1. RTLIL modules composed of cells and wires are similar to circuit $\mathcal{C} = (G, W)$, where G and W are the sets of gates and wires (cf. Definition 2.1). RTLIL appears sufficiently expressive to describe hardware/software systems, where the circuit initial state can be constrained using the `\init` attribute for sequential cells.

2.3.3 Transfomation Passes

Yosys's transformation passes are operations that manipulate and modify the RTLIL representation of a design. These transformation passes are essential for preparing the RTLIL representation for various backends and ensuring that the design meets the necessary requirements for synthesis, simulation, or formal verification. Some of these operations are depicted in the *Passes* box in Figure 2.9 and described in the following paragraphs.

Logic Synthesis. This pass is usually the first one applied to the intermediate representation. It translates the high-level RTLIL produced by the frontend to elementary logic cells such as `ADD`. It also ensures that the sequential and the combinational logics are properly separated and performs basic optimizations as described in the next paragraph.

Optimization. This pass is designed to simplify the RTLIL to reduce the design in size and complexity. It includes various optimizations such as constant propagation, logic simplification, finite-state machine optimization, flip-flop retiming, memory reconfiguration, and dead code elimination.

Flattening. Hardware designs are usually described using a hierarchy of modules, i.e., a top module instantiates several modules, which in turn, instantiate submodules. This pass flattens the design by removing module boundaries, meaning that all the logic and connections defined within the submodules are brought into the parent module.

Selection. This command is useful for selecting a subset of circuit components—modules, cells, wire—and determines the part of the design on which the next command operates. The selection language is a series of commands for a simple stack

machine. Components are selected according to pattern matching, cell type, or cell-width with basic operations like union, intersection, or set difference.

Simulation. This command simulates the circuit using the given top-level module. The result of the simulation can be dumped into a VCD file or used to constrain the design's initial state.

Technology Mapping. This pass implements a very simple technology mapper that replaces the RTLIL representation to specific technology libraries or cells, such as standard cell libraries for ASIC or LUTs for FPGAs.

2.3.4 Backends

Yosys backends are components responsible for converting the internal Rtlil representation of a design into various output formats (cf. Figure 2.9). These outputs can be used for different purposes, such as further synthesis with EDA tools, simulation, or formal verification.

A first category of backends in Yosys that convert designs to netlists for simulation or synthesis purposes include Verilog, Edif, and Blif formats. Alternatively, other backends can dump the Rtlil representation into a text or Json file. These outputs are useful for saving the design or for custom post-processing with other tools. However, this work seeks an automated process to convert a system description into a transition system. For this purpose, Yosys provides specific backends such as Aiger, Smv, Btor2, and Smt-lib. Note that Yosys's Smt-lib output is not a standard model-checking specification and is uniquely designed to work with Yosys-Bmc, as described below.

2.3.5 Yosys-SMTBMC

Yosys-SMTBMC, denoted Yosys-Bmc for short in the following, is a tool that combines Yosys with SMT solvers to perform formal verification of hardware designs.

The toolchain takes as input a Smt-lib description of the hardware design with its properties to be checked. During operation, Yosys-Bmc unrolls the circuit and provides SMT queries to an external solver, e.g., Yices or Boolector, that are checked for satisfiability. This combination between Yosys and SMT solvers facilitates various formal verification tasks such as bounded model checking (BMC) and equivalence checking. When the SMT solver returns a counterexample, Yosys converts it to VCD (Value Change Dump) waveform traces of the successive hardware states.

2.4 µArchiFI Workflow

This section presents the µArchiFI workflow. In particular, we link together the three previous sections of this chapter to provide a tool capable of automatically analyzing the consequences of fault in a hardware/software system. µArchiFI has undergone many improvements throughout the Ph.D., and we present here the latest version of the tool. The main evolutions and differences between tool versions are discussed at the end of this section.

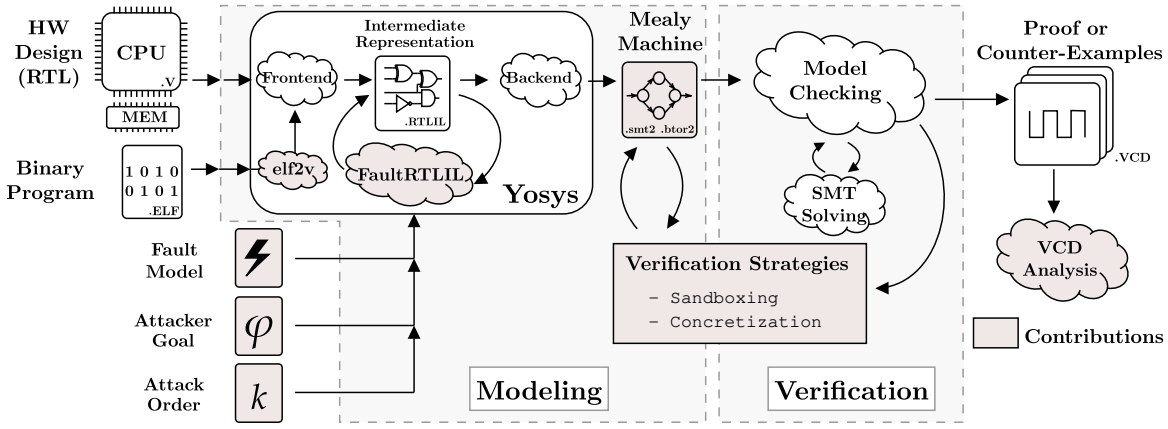


Figure 2.10: µArchiFI architecture and verification toolchain.

2.4.1 Tool Overview

µArchiFI is an automated framework designed to evaluate the effects of faults on a hardware/software system with respect to an attacker goal. The workflow consists of two main parts: modeling and verification, as illustrated in Figure 2.10.

In the *modeling* step, µArchiFI begins by converting the hardware description into a circuit model $\mathcal{C} = (G, W)$ represented in RTLIL. Next, the binary program is utilized to determine the initial state of the memory model. A fault model, along with an attacker goal φ and an attack order k, are provided to the FaultRTLIL pass. FaultRTLIL operates on the RTLIL representation to transform the circuit $\mathcal{C}$ into its faulty version $\mathcal{C}^{\mathcal{F}}$. The faulty circuit is then converted into a transition system $\mathcal{M}^{\mathbf{F}}$ in either the SMT-LIB or BTOR2 format using Yosys' formal backend.

During the *verification* step, µArchiFI uses the previously generated faulty transition system $\mathcal{M}^{\mathbf{F}}$ and relies on third-party model checkers to verify the reachability of the attacker goal. State-of-the-art model checkers compatible with these formats include Yosys-Bmc [Wola], Pono [MIL+21], and BtorMC [NPWB18]. When the (bounded) model checker succeeds in satisfying the attacker goal, a VCD file is generated to precisely report the attack. Analyzing the generated VCD waveform allows for understanding where the fault is injected and how the faulty values

propagate through the microarchitecture, leading to the vulnerability. Otherwise, when no counterexample exists, we have a formal guarantee that the attacker goal is unsatisfiable given the provided fault model.

Practical and technical implementation details on the modeling and verification processes are in Sections 2.4.2 and 2.4.3. Two verification strategies, labeled *sandboxing* and *concretization* in Figure 2.10, are discussed in detail in Section 2.4.4.

2.4.2 Modeling Process

This section provides insights into µArchiFI's modeling process. First, we explain how to use Yosys to model a hardware/software system in Rtlil. Using Yosys saves us from developing a similar tool from scratch, which would be impracticable. However, Yosys does not support specifying a fault model. Therefore, we developed our own transformation pass, FaultRtlil, to model faults in Rtlil, which is also described below.

Hardware Modeling

µArchiFI relies on Yosys to translate a hardware description into a circuit model $\mathcal{C} = (G, W)$. First, we must ensure that the input design falls within the subset of Verilog supported by Yosys. If the design uses SystemVerilog features, we can translate it from SystemVerilog to Verilog using the open-source tool sv2v [Sno]. Additionally, Yosys supports VHDL designs through the Ghdl plug-in [Tri].

The required hardware design includes the processor description to be analyzed and a memory model. In the current version of µArchiFI, the memory model must be provided to Yosys at the RTL level and must comply with the memory protocol interface to communicate with the processor.

Software Modeling

The source code to be evaluated must be compiled for the targeted processor architecture using the riscv32-unknown-elf-gcc tool suite. The resulting object file, which contains the machine code, is then linked with Bsp (board support package) files to produce an Elf (executable and linkable format) file. Essential Bsp files include `Crt0.s` (C runtime zero)—which performs the system initial setup e.g., stack and heap pointers, global variables—the linker script (`link.ld`)— which maps object files in memory, defines memory regions, and sets the boot address—, and `syscall.c`—which contains low-level system functions like `printf` and `memcpy` since the executable does not link to the standard C library to minimize the executable size. Before being loaded into the memory model for execution, the Elf file must be converted to a valid Yosys input using the elf2v script (visible in Figure 2.10) to constrain the initial state of the memory model with Verilog directives.

Initial State Configuration

The previous sections described how to express a hardware/software system in the Yosys infrastructure. However, the current processor's initial state has not yet been constrained and does not match the context of the software program to be verified. To address this, we precompute the system's initial state from the boot point to the software function to be verified.

```
sim -zinit -clock clk_sys -resetn rst_sys_n -n 83 -rstlen 3 -w
```

Listing 2.1: Yosys's simulation pass to set the system's initial states.

As depicted in Listing 2.1, the simulation pass `sim` of Yosys addresses this issue. The parameters are:

`-zinit`: Sets all uninitialized registers and memory to zero,

`-clock`: Specifies the clock signal,

`-resetn`: Specifies a negative reset signal (active low),

`-n`: Provides the number of cycles to simulate the design,

`-resetlen`: Indicates the duration of the reset signal.

`-w`: Uses the simulation result as the design's new initial state.

Choosing the simulation length based on the number of instructions to execute from the entry point to the function of interest ensures the proper initial state for formal verification. By definition, simulation is deterministic and does not allow for symbolic data. However, once the system has been simulated, it is possible to insert symbolic data by removing the `\init` attribute of memory and registers in the RTLIL specification. Consequently, the hardware/software system can have multiple initial states due to unconstrained memory regions.

Fault Modeling

Faults, defined by a fault model $\mathcal{F}$, can be encoded into the RTLIL representation of the system using the FAULTRTLIL pass. This pass operates on the current selection and has multiple options to specify the effect, timing, and maximum number of faults. Listing 2.2 summarizes the FAULTRTLIL syntax.

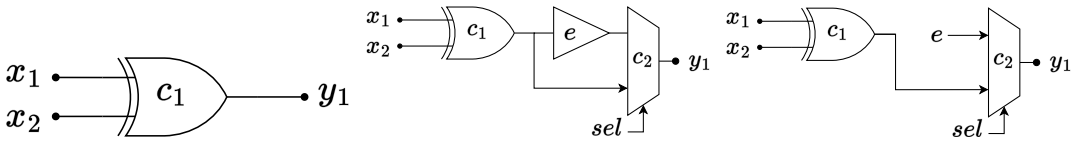
Locations. The FAULTRTLIL pass does not take the fault location as input, as it directly operates on the current selection (see §Selection in Section 2.3.3). Executing Yosys's `select` command before calling FAULTRTLIL allows choosing fault locations based on pattern-matching, cell-type, or cell-width operations.

```
fault_rtlil [options]

    -effect { set | reset | flip | fixed <value> | symbolic | diff }

    -timing <range>

    -cnt [local] <max_fault>
```

Listing 2.2: FaultRtlil command syntax.

Effects. In Section 2.1, we defined fault effects $e \in E$ as functions $\mathbb{B}^v \to \mathbb{B}^v$ that replace the original gate output value with a faulty value. Using µArchiFI, the user can specify multiple fault effects. Options like *set*, *reset*, *flip*, and *fixed* correspond to setting all bits of the target to one, zero, flipping them, or fixing them to a specific value, respectively. µArchiFI also supports symbolic fault effects to cover every possible effect simultaneously. A symbolic fault effect is an uninterpreted function [BD94, BLS02] that maps the fault-free variable to a fresh new symbolic value, encompassing set, reset, or flip effects. The *diff* effect is a symbolic fault with the additional constraint that the faulty value must differ from the fault-free one.

Figure 2.11 shows how faults are injected in Rtlil and summarizes the various fault effects: **(a)** depicts the fault-free circuit, **(b)** shows a fault injection involving a concrete fault effect, e.g., a bit-flip, and **(c)** illustrates a symbolic-effect fault injection. Since we work with transient faults, a multiplexer c_2 selects when the fault effect must be applied.



(a) Fault-free circuit. **(b)** Concrete fault injection. **(c)** Symbolic fault injection.

Figure 2.11: Rtlil transformation using the FaultRtlil pass.

Timing. The timing is specified to FaultRtlil as a *range* of integers, defining when the transient fault effect must be applied to the circuit. To implement this in Rtlil, which has no notion of time (being a structural and functional circuit view), we introduce a *clock_cnt* variable initialized to zero. This variable acts as a simple counter when the circuit model is converted into a transition system using Yosys's backend. As a result, the fault selector signal (*sel* in Figure 2.11) is set to *false* when *clock_cnt* is not within the specified timing range. This constraint is specified in Rtlil using the formal `assume` statement as follows:

$$\texttt{assume} \quad \neg sel \vee (clock\_cnt \in range)$$

Order. The `-cnt` parameter of FaultRtlil specifies the attack order, i.e., the maximum number of faults to be injected into the system. This option generates a *cnt* variable in Rtlil, initialized to 0 and incremented each time a fault is injected.

It serves as a cardinality constraint over fault selector signals (*sel*) to ensure that the number of faults injected into the system cannot exceed the attack order k. The *local* option specifies whether the generated counter should be local to each module in the hierarchy or global for the entire design.

Attacker Goal

An attacker goal $\varphi\colon X \times S \to \{0,1\}$ is a Boolean predicate over circuit input values X and register values S (cf. Definition 2.7). µArchiFI does not provide a dedicated pass to specify the attacker goal. Instead, the user must encode φ directly in the hardware design using a SystemVerilog Assertions [CDHK15] construction as follows:

```
assert property  ¬φ
```

In this example, we assert φ's negation as the `assert property` keyword is used to specify system invariants. The attacker goal must be expressed using the Verilog syntax. Then, φ is translated into RTLIL using Yosys's Verilog frontend and finally converted into the formal specification of the transition system.

2.4.3 Formal Verification

This section provides insights into µArchiFI's verification process. First, we detail how to use third-party model checkers to evaluate the hardware/software system's security against fault attacks. Then, we explain how to interpret verification results and understand the consequences of faults on software security when the attacker goal is satisfiable.

Model Checking

µArchiFI performs symbolic reachability analysis using any third-party model checker compatible with the BTOR2 language, such as PONO [MIL+21] and BTORMC [NPWB18]. The formal SMT-LIB specification generated by Yosys is uniquely designed to work with YOSYS-BMC.

Each model checker requires additional parameters such as the backend SMT solver, the bound (i.e., the maximum number of transitions to perform in the system), and the output witness file when a counterexample is found.

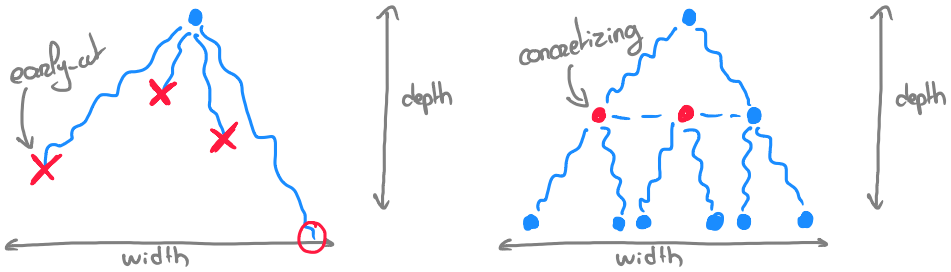
For a reminder of the reachability problem we are checking and the symbolic reachability analysis algorithms, we refer the reader to Section 2.1 (Figure 2.7) and Section 2.2 (Algorithm 2.2).

The witness output by model checkers is a system execution trace $(s_i)_{i=1}^{n} \in S^n$ where $s_1 \in S_0$ is an initial state and s_n fulfills the attacker goal φ. The witness is then converted to a counterexample in VCD (value change dump) format using YOSYS-BMC or BTOR2TOOLS.

In some specific cases, the user may want to enumerate multiple fault attacks that enable an attacker to reach their goal. Currently, µArchiFI does not support model counting or enumeration. A workaround for this limitation is to generate multiple model-checking problems with different fault models.

Counterexample Analysis

The VCD counterexample reports a faulty system execution trace. However, understanding the propagation of faults and their consequences on software security requires human expertise. To ease this task, we developed a VcdDiff tool that computes the difference between the counterexample trace and a reference one. This step is shown in Figure 2.10.



(a) Sandboxing. **(b)** Concretization.

Figure 2.12: Optimization effects on the state space to explore.

2.4.4 Software-Driven Optimizations

As previously introduced in Section 2.2.4, a well-known approach to mitigate the state space explosion consists in reducing the state space through abstractions. In the case of µArchiFI, performance limitations may come from the complexity of the SMT queries sent to the SMT solver. To address this issue, this section proposes two verification strategies to improve the efficiency of the workflow. These optimizations are said *software-driven* as we rely on program counter (PC) values to reduce the state space. First, the *sandboxing* technique restricts the range of possible PC values. Second, *concretization* splits the formula to be solved into several subformulas according to the PC.

Sandboxing

Sandboxing is an example of under-approximation of system behaviors where additional constraints are applied to the model according to the software evaluated. The program counter (PC) is restricted to a range of values that a simple static analysis can extract from binary addresses, e.g., using an *objdump* tool. The benefit of this approach is that it only considers the paths that satisfy the sandboxing condition. This situation often occurs when analyzing a given (set of) function, and we do not

want the program to jump to another memory location as its content is not modeled in the analysis. As illustrated on Figure 2.12a, the verification stops exploring software execution paths that do not satisfy this sandboxing constraint. Consequently, the BMC procedure can also terminate faster when the entire state space has been explored. Using sandboxing, our approach loses the completeness. Even if we assume that the possible solutions we miss are not relevant, this technique must be used to explore possible vulnerabilities rather than prove the system's robustness.

While only one PC exists at the software level, several microarchitectural registers store its value in the processor design. Identifying the proper PC variable to constrain is consequently the main challenge when implementing the sandboxing strategy in practice. The fetch stage PC speculates on the next addresses to be read from memory. Applying the sandboxing technique on this register would thus require relaxing the sandboxing constraint. The execute stage PC misses unconditional branches that are resolved directly in the decode stage. We therefore implement sandboxing by constraining the PC of the decode stage of in-order processors, as presented later in Chapter 3.

Concretization

Concretization is an optimization technique inspired by path-based encodings like symbolic execution [Kin76, BCD+18]. The fundamental idea behind symbolic execution is to split the formula to be checked into sub-formulas according to the different execution paths. However, this technique may encounter an exponential number of paths to check, the so-called *path explosion* problem.

Informally, concretization selects a system state variable and enumerates its possible values to divide the formula into several easier-to-solve sub-formulas. Like sandboxing, we rely on PC values to distinguish between different execution paths.

The concretization procedure is detailed in Algorithm 2.3. First, we define a BMC function, adapted from Algorithm 2.2, to compute the sets of reachable states and check the validity of φ at each iteration (lines 1 to 6). At lines 7 and 8, we initialize the predicate *reachable* with the initial states S_0, and perform BMC up to the concretization depth m. Then, the concretization loop enumerates the possible values for the PC (lines 10 to 16). This loop successively asks an SMT solver to give models of the system with different PC values. It stops when no more system model exists, i.e., ψ becomes unsatisfiable, or after a given number of concretizations nb_{conc}. A new BMC procedure is performed for each enumerated PC value until bound n (lines 17 to 20). When the PC enumeration is incomplete, the remaining paths are encoded within a single formula and checked together (lines 21 to 24). A preliminary program analysis can identify branches' locations and determine the most suitable depth m for which the user should perform the concretization.

Algorithm 2.3: Bounded model checking with concretization.

Input: a Mealy machine $\mathcal{M} = (X, Y, S, S_0, \delta, \lambda)$, a property φ, a bound n, a concretization depth m, and a number of concretizations nb_{conc}.
Output: exit on *success* if φ is reachable within n steps, *failure* otherwise.

Function `BMC_loop`$(\mathcal{M}, \Phi, \varphi, n)$**:** ▷ `BMC (see Algorithm 2.2)`
  for i **from** 1 **to** n **do**
    if $(\mathbf{SAT}(\Phi \wedge \varphi))$ **then**
      exit *success*;
    $\Phi \leftarrow \Phi \vee \mathbf{succ}(\Phi)$;
  return Φ;
$reachable \leftarrow [\![S_0]\!](s)$;
$reachable \leftarrow$ `BMC_loop`$(\mathcal{M}, reachable, \varphi, m)$;
$\psi \leftarrow reachable$; $enum_{\mathrm{PC}} \leftarrow \{\}$; $terminate \leftarrow false$;
for *iter* **from** 1 **to** nb_{conc} **do** ▷ `Concretization loop`
  if $(\mathbf{SAT}(\psi))$ **then**
    $address \leftarrow \mathbf{get\_model}\,(\psi)\,(PC)$;
    $enum_{\mathrm{PC}} \leftarrow enum_{\mathrm{PC}} \cup address$;
    $\psi \leftarrow \psi \wedge (PC \neq address)$;
  else
    $terminate \leftarrow true$; **break**;
for each $address \in enum_{\mathrm{PC}}$ **parallelize** ▷ `BMC runs on concretized paths`
  $reachable \leftarrow reachable \wedge (PC = address)$;
  `BMC_loop`$(\mathcal{M}, reachable, \varphi, n - m)$;
  exit *failure*;
if $\neg terminate$ **then** ▷ `BMC on remaining paths, if needed`
  $reachable \leftarrow reachable \wedge (PC \notin enum_{\mathrm{PC}})$;
  `BMC_loop`$(\mathcal{M}, reachable, \varphi, n - m)$;
  exit *failure*;

2.4.5 Previous µArchiFI Versions

µArchiFI has undergone many improvements throughout the Ph.D. thesis, and this section has presented the latest version of the tool. However, the results presented later in Chapter 3 also rely on a previous version of µArchiFI, which we refer to as µArchiFIv0. The following paragraphs describe the main evolutions and differences between the tool versions.

Incomplete Initial State Configuration. Initially, the system's initial states were not computed using Yosys's `sim` command but rather with an external simulator like QuestaSim. We simulated the design up to the desired clock cycle and dumped the contents of specific design elements, such as memory or the register file, to constrain the system's initial state. However, other flip-flops, such as pipeline-stage registers,

were not retrieved from the simulation and were consequently initialized to their default reset values. This incomplete microarchitectural state initialization could lead to spurious evaluation results.

Fault Expressiveness. µArchiFIv0 modeled faults directly in the generated Smt-lib specification instead of injecting them into Rtlil using the FaultRtlil pass. Consequently, the previous µArchiFI version supported only a limited fault model. Selecting fault locations was a manual task and did not take advantage of Yosys's `select` command to apply automated selection rules. Additionally, some logic gates were not faultable as they were directly optimized when using Yosys's formal backend. Furthermore, only the symbolic fault effect was supported by the tool. Finally, additional constraints encoding the fault timing, attack order, or attacker goal were specified in an external file of constraints (cf. smtc files [Wola]).

Model Checker Support. Since faults were directly encoded in Smt, µArchiFIv0 was bound to use Yosys-Bmc as the Smt-lib output of Yosys is not a standard model-checking specification. The Btor2 format was not supported, and the previous µArchiFI version could not benefit from state-of-the-art model checkers like Pono or BtorMC.

Limitations of the Current Version. Despite its many advantages over the previous version, the current µArchiFI still has some limitations.

First, the hierarchical representation of designs using modules in Rtlil is a bottleneck for specifying global constraints like the fault attack order or properties such as the attacker goal. The visibility of variables from one module to another is restricted, as hardware modules can only exchange values through input/output ports. One solution to this limitation is to flatten the design.

Second, and this is probably the major limitation of µArchiFI, the modeling and verification steps are unable to interact with each other. The only interface between these two steps is the Smt-lib or Btor2 file produced by Yosys. Consequently, everything must be encoded in that formal specification, including the hardware/software system, the fault model, and the attacker goal. This restriction means it is not possible to modify the model during the model-checking phase. Being able to modify the model by pushing and popping assumptions would have been practical for enumerating counterexamples or implementing optimizations. For instance, sandboxing is integrated into µArchiFI as it only requires adding a global constraint on the PC value. However, the concretization technique, which requires interactions between the model checker and the modeling part, cannot be integrated into the tool. Instead, we instrumented Yosys's in-house solver, Yosys-Bmc, as it allows us to stop the model checking at the desired depth to concretize the PC value. This optimization, however, has not been ported to other solvers like Pono, as modifying the model checker itself would have been impractical.

2.5 Conclusion

In this chapter, we first introduced a formal modeling for CPU-based systems, including fault attacks. We then provided the necessary background on model checking to evaluate reachability properties on these models using formal techniques. Additionally, we presented the Yosys synthesis tool with the objective of leveraging its existing synthesis infrastructure to focus on the automated integration of faults into the system. Finally, we described µArchiFI, the first tool designed to formally evaluate the security of hardware/software systems against fault attacks.

µArchiFI is currently available in open access on Zenodo[1], Github[2], and a pull request will be submitted to integrate the FaultRtlil pass into Yosys.

In conclusion, this chapter has proposed an initial solution to bridge the gap between automated fault evaluation tools operating at the circuit level and those working at the binary level. However, this chapter primarily focused on defining the tool, whereas one of the main challenges of formal verification methods lies in their practical usage. Therefore, the next chapter showcases several applications of µArchiFI and evaluates its performance.

[1] https://zenodo.org/records/7958412
[2] https://github.com/CEA-LIST/uArchiFI

Chapter 3

Experimental Evaluation using µArchiFI

Contents

Chapter 1 of this manuscript identified the need for a formal and automated approach to verifying the effect of faults on the security of CPU-based systems. An initial response to this problem was provided by the µArchiFI tool introduced in Chapter 2. However, to determine whether µArchiFI effectively addresses this challenge, it is necessary to evaluate its usage through case studies. This is the focus of the present chapter.

Chapter 3 is divided into two main topics: *security* and *performance* evaluation. Initially, Section 3.1 demonstrates the use of µArchiFI to reveal attacks exploiting microarchitectural mechanisms of processors. Section 3.2 focuses on proving the robustness of a system integrating the control-flow countermeasures. The performance of these evaluations is examined in Section 3.3, which also proposes three additional use cases to evaluate the influence of verification parameters on µArchiFI.

The content of this chapter is adapted from our publications at *Workshop on Fault Detection and Tolerance in Cryptography (FDTC)* [TAC+22] and *Formal Methods in Computer Aided Design (FMCAD)* [TAC+23]. This work was also presented at *Journée sur les attaques par injection de fautes (JAIF)* [Tol22] in 2022.

3.1 Case Study I: Microarchitectural Exploits

As introduced in Section 1.1, recent works have highlighted the importance of considering microarchitectural mechanisms, such as pipelining and forwarding, when evaluating the consequences of fault injections [YGS15, LBD+18, ACD+22]. However, these evaluations were performed manually as the simulation techniques employed were insufficient to exhaustively analyze the effect of faults. In this section, we demonstrate the capability of µArchiFI to perform these analyses automatically. We validate our approach by reproducing known results from the literature and expose previously unknown vulnerabilities. The rest of this section is organized as follows.

First, we describe the evaluated case study—a secure authentication mechanism running on a RISC-V processor—and define the attacker goal as well as the types of faults he can inject into the processor. Second, we detail microarchitectural exploits identified using µArchiFI. Finally, since the security evaluations utilized an earlier version of the tool, µArchiFIv0, a discussion concludes this section.

3.1.1 Experimental Set-Up

The following paragraphs describe the use case analyzed with µArchiFI and the formal model it generates for model-checking verification.

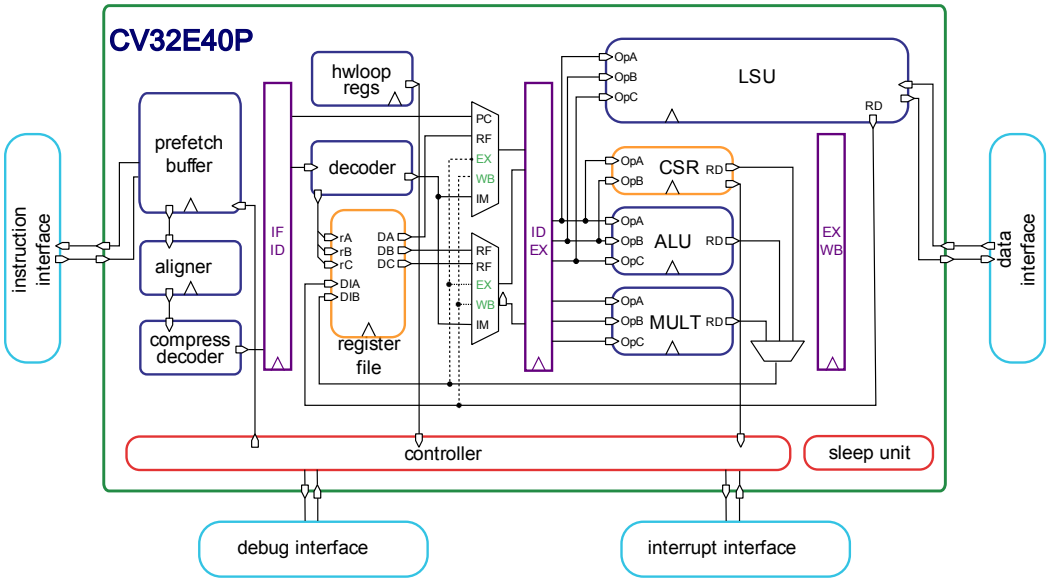


Figure 3.1: CV32E40P block diagram.

3.1.1.1 CV32E40P Processor

The CV32E40P is a 32-bit, in-order, RISC-V core from the OpenHW Group designed for light-embedded use [Opeb]. It implements the RV32IMC ISA with a 4-stage pipeline (IF, ID, EX, WB) as illustrated in Figure 3.1. The CV32E40P is developed in SystemVerilog, and the sources are publically available [Opea]. The CV32E40P does not include protections against faults. In the following, we introduce three microarchitecture features of the CV32E40P processor, which, as detailed later in Section 3.1.2, are vectors of vulnerability during FI attacks.

Forwarding (FWD). Forwarding is a microarchitectural optimization designed to avoid processor stalls due to data hazards (cf. Section 1.1). The forwarding mechanism bypasses the write-back of an operation result to the register file to provide it to the previous pipeline stages as soon as it is available. The implementation is subject to many factors, such as pipeline depth and the location of the bypasses that retrieve the available data. In the CV32E40P, the forwarding mechanism shortcuts data dependencies in the ID stage (dashed lines on Figure 3.1). Any result from functional units, e.g., ALU, MULT, LSU, can then be used in the ID stage without passing through the register file.

Multiplier (MULT). The multiplier, denoted as MULT in Figures 3.1 and 3.2, performs the multiplication between two 32-bit integers and stores the result in the register file. This unit implements two types of instruction that either output the 32 least significant bits (`mul` instruction) or the 32 most significant bits (`mulh` instruction) of the result. The computation time depends on the requested operation. Only one cycle is sufficient for `mul`, whereas five cycles are needed for the `mulh` instruction. In this latter case, a finite state machine drives the operation and two signals indicate to other processor pipeline stages that a multi-cycle multiplication

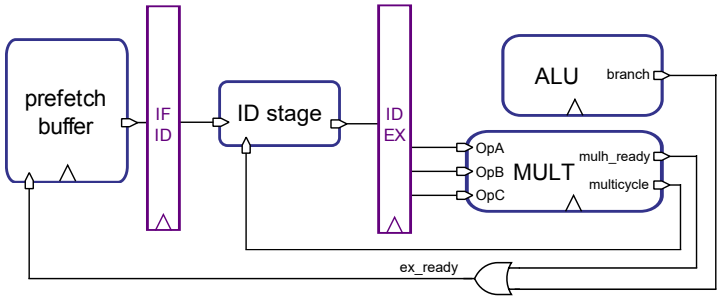


Figure 3.2: Multiplier input/output signals and their use to stall the preceding pipeline stages in case of a multicycle multiplication.

is in progress.

1. `multicycle`: this signal is sent to the ID stage to determine if the intermediate result, computed over iterations, needs to be returned to MULT via the signal `OpC`. Consequently, the ID stage stops decoding new instructions.

2. `mulh_ready`: this signal indicates if a multicycle multiplication is in progress. As a result, the EX stage is tagged as busy, and this information is then propagated to the IF stage to stop fetching new instructions.

Prefetch Buffer (PFB). The CV32E40P processor has a prefetch buffer in the IF stage as shown on Figure 3.1. The PFB performs word-aligned 32-bit memory readings and stores the fetched words in a FIFO with a queue of two instruction words. At the microarchitectural level, two independent program counters (PCs) exist. The first one, PC_{IF}, specifies the last instruction that passed the IF stage. It is used in the following stages, in particular, to compute the target address of direct branches. The second one, PC_{PFB}, indicates the address of the next instruction to fetch in memory. Because of the speculative nature of the PFB, the PC_{PFB} is incremented ahead of time and independently from PC_{IF}. Both PCs are resynchronized when a branch is taken.

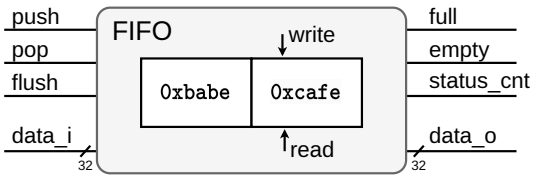


Figure 3.3: Prefetch buffer FIFO in the CV32E40P.

Figure 3.3 shows the FIFO used in the PFB to store fetched instruction words. The signal `status_cnt` corresponds to the number of instructions contained in the FIFO. The signals `full` and `empty` indicate its status, i.e., whether it is full or empty, respectively. The pointers `read` and `write` indicate the buffer location where the next value should be read or written. These pointers are incremented modulo the

```
#define PIN_SIZE 4
SBYTE g_ptc;
BOOL g_authenticated, g_countermeasure;
UBYTE g_userPin[PIN_SIZE], g_cardPin[PIN_SIZE];
void initialize() {
    g_countermeasure = false;
    g_ptc = 3;
    for (int i = 0; i < PIN_SIZE; ++i) {
        g_cardPin[i] = i + 1;
    }
    for (int i = 0; i < PIN_SIZE; ++i) {
        g_userPin[i] = 0;
    }
}
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size) {
    for (int i = 0; i < size; i++) {
        if (a1[i] != a2[i]) {
            return false;
        }
    }
    return true;
}
BOOL verifyPIN() {
    g_authenticated = false;
    if (g_ptc > 0) {
        if (byteArrayCompare(g_userPin, g_cardPin, PIN_SIZE) == 1) {
            g_ptc = 3;
            g_authenticated = true;
            return true;
        }
        g_ptc--;
        return false;
    }
    return false;
}
int main() {
    initialize();
    verifyPIN();
    assert(g_countermeasure == false && g_authenticated == true);  // φ_authen
    assert(g_countermeasure == false && g_ptc >= 3);               // φ_ptc
    return 0;
}
```

Listing 3.1: C code of VerifyPIN_v0 which has no countermeasures.

queue size since the buffer is circular. The FIFO ports for receiving and transmitting instructions are `data_i` and `data_o`, respectively.

3.1.1.2 VerifyPIN

A critical piece of software that needs to be robust against fault injections is the `memcmp`-like mechanism used in authentication or signature verification, e.g., in a secure boot process. As practical examples, the VerifyPIN programs from the FISSC benchmark suite [DPP+16] will be used as a common thread over the next sections. VerifyPIN compares two PIN codes stored in memory: a user and a secret (card) PIN, and allows user authentication when PIN codes are identical.

Listing 3.1 provides the C code of VerifyPIN. First, the `initialize` function initializes the countermeasure flag to *false*, the try counter (`g_ptc`) to 3, and the `cardPIN` and `userPIN` codes. Then, the `byteArrayCompare` function returns *true* if

the two input arrays are equal. Finally, the `VerifyPIN` function performs the comparison and updates the global variables `g_authenticated` and `g_ptc` accordingly. The program is available in eight versions with an increasing number of protections against fault attacks. VerifyPIN_v0 has no protection, while VerifyPIN_v7 is the version with the highest number of countermeasures. The proposed countermeasures are reported in Table 3.1 and their principle is detailed below:

Table 3.1: VerifyPIN suite and its countermeasures, adapted from [DPP+16].

VerifyPIN version	HB	CL	IC	BC	SC	DT
v0						
v1	✓					
v2	✓	✓			✓	
v3	✓	✓	✓		✓	
v4	✓	✓	✓	✓	✓	
v5	✓	✓			✓	✓
v6	✓	✓	✓			✓
v7	✓	✓	✓		✓	✓

Program Features	
CL	Constant time loop
IC	Inline call
Countermeasures	
HB	Harden Boolean
BC	Backup copy
SC	Step counter
DT	Double test

HB Booleans are hardened by using non-trivial *true* and *false* values whose Hamming distance is maximal.

CL The comparison loop has a fixed number of iterations.

IC The `byteArrayCompare` function is inlined to protect against instruction skips.

BC Global variables are duplicated. Their values are affected and tested twice.

SC The loop counter is compared against the expected number of iterations upon loop exit.

DT Critical tests are duplicated.

In the following, we evaluate VerifyPIN_v3 and VerifyPIN_v7 using two different compiler optimization levels, `-Og` and `-Os`. We target the RV32IM instruction set using version 10.2 of riscv32-gcc. The compilation flag `-Og` produces a machine code with a limited number of optimization passes so that it preserves a good traceability between the C code and the generated binary code, while the `-Os` optimization level uses more aggressive optimizations and focuses on code size reduction. We use these two optimization levels for each selected VerifyPIN version to produce binary programs with different structures and instruction ordering. This allows us to analyze more patterns from a single input program and highlight the impact of the binary code on the presence of subtle vulnerabilities. Both selected optimization levels can remove or alter the implemented countermeasures. When necessary, we manually added them to the final assembly file to impose their presence at the binary level. In the remainder, we denote the four resulting binaries as follows: `v3_Og`, `v3_Os`, `v7_Og`, `v7_Os`.

In all experiments, we consider `userPIN` and `cardPIN` to have three 8-bit digits, i.e., `PIN_SIZE` = 3 in Listing 3.1[1]. In addition, each digit is treated as a symbolic variable, meaning it has no concrete value. However, each pair of digits is different, and modifying only one of them is not sufficient for authentication. Equation (3.1) formalizes this assumption.

$$\forall i \in [0, \texttt{PIN\_SIZE} - 1], \quad \texttt{userPIN}[i] \neq \texttt{cardPIN}[i] \tag{3.1}$$

3.1.1.3 Generated Circuit Model

µArchiFI produces a circuit model of the CV32E40P processor. As described in Section 2.1, we considered a closed system, embedding a memory model connected to the CPU and initialized with VerifyPIN. Table 3.2 summarizes the circuit characteristics.

Table 3.2: CV32E40P circuit model characteristics.

Design Name	Size (GE)	Wires		Gates (#)	Registers		Memory Model	
		(#)	(# bits)		(#)	(# bits)	Size	Symbolic Data
CV32E40P	48 000	3 855	39 944	3 180	179	2 786	32×2^{10}	$8 \times 2 \times 3$

First, we report the circuit size in gate equivalent (GE). GE is a standard indicator of circuit complexity often used in the industry. It is independent of the technological library used and the formal model generated.

Then, we provide the circuit characteristics in terms of wires W, gates G, and registers R. Since several levels of circuit representation will be used in this manuscript—word-level in this section and bit-level in Chapter 4—we also report the wire and register sizes in terms of bits. For instance, 179 resisters represent 2 786 bits since most of them are 32-bit registers. The number of registers' bits also indicates the state space size to explore.

Finally, the memory model we use has 2^{10} 32-bit words of storage. Its size is defined according to a bare-metal linker script, which determines the required space to store the VerifyPIN instructions, as well as the necessary space for data storage, e.g., the stack and the heap. The memory is initialized with the content of the VerifyPIN program. Additionally, we precomputed the initial state of the heap, stack, register file, and program counter, initializing these memory locations accordingly so that the formal analysis can start directly from the first instruction of VerifyPIN. We zero out unused memory locations and leave the `userPIN` and `cardPIN` values as symbolic.

3.1.1.4 Attacker Model

The vulnerability analysis in this section is motivated by the desire to highlight the importance of considering microarchitectural details when evaluating fault effects.

[1]VerifyPIN originally uses 4-digit PINs.

Consequently, we exclude faults affecting data and general-purpose registers, as these can be easily modeled at a higher level of abstraction. Instead, we focus on faults occurring in control signals that drive instruction fetching, decoding, and the control path. These faults are particularly interesting because they can lead to effects that cannot be modeled at the ISA level, as discussed in Section 1.1.

Fault model. The fault model $\mathcal{F}$ we consider is defined as follows:

$$\mathcal{F} = \{(g, e) \in G \times E \mid \mathit{size\_of}(g) \leq 6 \wedge \mathit{is\_named}(g) \wedge e = \mathit{symbolic}\} \tag{3.2}$$

We target gates whose output width is less than or equal to 6 bits and that correspond to a named *signal* in the provided RTL description. Informally, a signal is a bit-vector used in hardware description languages like Verilog. We choose to arbitrarily filter out large signals—as they correspond to data—as well as unnamed signals to limit the results obtained to simpler and easier-to-explain effects. Finally, our evaluation considers symbolic fault effects, which encompass every possible effect $e \in E$, as described in Section 2.4.2.

Given the fault model $\mathcal{F}$, we consider that an attacker can inject a single transient fault anytime during the program execution as formalized below:

$$\mathbf{F} \subseteq \mathcal{F} \times [1, n], \quad \text{with } |\mathbf{F}| = 1$$

The integer n corresponds to the program execution time, as shown in Table 3.3. For each version of VerifyPIN, we simulated the program to determine the required number of clock cycles and set the value of n accordingly.

Attacker goal. The attacker wants to authenticate without triggering the countermeasure as formalized in Equation (3.3).

$$\varphi_{authen} := \texttt{g\_authenticated} \wedge \neg \texttt{g\_countermeasure} \tag{3.3}$$

A manual inspection of the VerifyPIN binary files provides the memory addresses of the global variables `g_authenticated` and `g_countermeasure`. Then, the attacker goal φ_{authen} is specified directly in the hardware memory model using SystemVerilog Assertions. This property is translated into Rtlil and finally converted into the formal specification of the transition system.

3.1.1.5 Model Checking Protocol

As mentioned earlier in this section, we use a previous version of the tool to evaluate system security. Specifically, we use µArchiFIv0 to generate an Smt-lib description of the system and rely on Yosys-Bmc for the model checking analysis, with

Table 3.3: Execution time and verification bound for each VerifyPIN version.

VerifyPIN versions	`v3_0g`	`v3_0s`	`v7_0g`	`v7_0s`
Length n (clock cycle)	66	54	68	58
BMC bound $n + \epsilon$	74	62	76	66

Yices as the backend SMT solver. Other formal backends like Pono or BtorMC, which require a Btor2 format input, were not usable.

The BMC bound is set to $n + \epsilon$, where n is the execution time in clock cycles, and ϵ is a small increment relative to n, as shown in Table 3.3.

To enumerate all potential fault attacks that could allow an attacker to reach the goal φ_{authen}, we generated multiple instances of the verification problem. Each instance represents a fault at a specific circuit location and clock cycle according to the fault model $\mathcal{F}$. When φ_{authen} is satisfiable, µArchiFIv0 outputs a counterexample.

3.1.2 Microarchitectural Exploits

The analyses performed on the four configurations of VerifyPIN revealed a total of 250 successful fault attacks. Table 3.4 shows a summarized version of these results since several fault attacks produce exactly the same consequences due to signal renaming in the hardware description, i.e., the same wire can be targeted at multiple locations or timings. Each successful fault injection is given according to its Verilog module (leftmost column, corresponding to the block diagram in Figure 3.1), its targeted signal name (i.e., spatial location), and the processor cycle at runtime (i.e., temporal location). The *Category* column refers to a manual classification of the hardware feature corrupted by the fault. In the remainder of this section, we illustrate the consequences corresponding to each category on selected examples of successful fault injection. The injected faulty value is not mentioned in this table but is given when relevant in the illustrative examples developed below.

3.1.2.1 Forwarding

By faulting the forwarding mechanism, an attacker can retrieve a value previously computed by a functional unit or read from memory, and then forward this value into one of the operands of the EX stage. Laurent et al. have demonstrated similar results on the Rocket Processor [LBDP19]. The vulnerabilities reported in Table 3.4 under the category *FWD* exploit this mechanism. Both `v3_Og` and `v3_Os` are vulnerable to such faults. The fault injection inverts the conditional branch corresponding to the `if` statement at line 26 in Listing 3.1, resulting in a successful malicious authentication. Due to the differences between the two binary programs, the fault is injected at different clock cycles for `v3_Og` and `v3_Os`.

3.1.2.2 Multiplier

The *MULT* category in Table 3.4 corresponds to faults impacting the finite-state machine of the multiplier module, even if no multiplication is performed in the VerifyPIN program. As explained in Section 3.1.1.1, when MULT enters an active state, it stalls the ID stage via the `multicycle` signal for up to 4 cycles. The security concern only arises when the ALU is calculating a branch address at the moment

Table 3.4: Results of the FI analysis on VerifyPIN: each reported *fault model* (designated by the signal name and the FI cycle) satisfies the attacker goal φ_{authen}.

Module	Targeted RTL Signal	Category	Cycle of Fault Injection			
			v3_0g	v3_0s	v7_0g	v7_0s
fifo	empty	*PFB*		46		
	status_cnt_n	*PFB*	18, 21-26	26, 27, 45		
prefetch_ctrl	flush_cnt_q	*PFB*		18, 45, 46		
aligner	instr_valid	*other*	18	18, 19, 46		
	branch_i	*ALGNR*	47			
	update_state	*ALGNR*	47		47, 52	
	state	*other*		18, 19		
id_stage	alu_bmask_b_mux_sel	*other*		39		
	alu_vec_mode_ex	*other*	58	48		
	bmask_b_mux	*other*	46	19, 20, 39		
	branch_taken_ex	*other*	58	48		
	id_valid	*other*	19	19, 20, 47		
	imm_b_mux_sel	*other*		19, 20		
	reg_d_alu_is_reg_a_id	*FWD*	57	20, 47		
	regfile_alu_waddr_mux_sel	*other*	19	19, 20		
controller	ctrl_fsm_cs	*other*		19, 20, 47, 48		
	deassert_we	*other*	19	19, 20, 47		
	halt_id	*other*	19	19, 20, 47		
	is_decoding	*other*	19	19, 20, 47		
	jump_in_dec	*other*	57			
	operand_a_fw_mux_sel	*FWD*	56, 57	19, 20, 46, 47		
	operand_b_fw_mux_sel	*FWD*	57	47		
	pc_set	*other*		48		
decoder	alu_en	*other*		19, 20, 47		
	alu_op_a_mux_sel	*other*	57	19, 20, 47		
	alu_op_b_mux_sel	*FWD*	57	19, 20, 39, 47		
	ctrl_transfer_insn	*other*	57	47		
	regfile_alu_we	*other*	19	19, 20		
	regfile_mem_we	*other*	46, 51	19, 20, 39		
ex_stage	alu_cmp_result	*other*	58	48		
	mult_en	*MULT*		19, 20		
	mult_multicycle	*other*	19	19, 20, 47		
	regfile_alu_we_fw	*FWD*	20	20, 21		
	regfile_we_wb	*other*		21, 22		
alu	cmp_result	*other*		48		
	shift_left	*other*		20, 21		
mult	multicycle	*MULT*	19			
	mulh_CS	*MULT*		46		
lsu	data_be	*other*		17		

of fault injection. In this scenario, the EX stage notifies the IF stage to remain ready (via the `ex_ready` signal shown in Figure 3.2) since a branch might be taken. Consequently, the IF stage continues fetching instructions from memory while the ID and EX stages are stalled by multiplication, leading to the fetched instructions being ignored. If the ALU is calculating no branch, the injected fault does not affect the program's behavior, as the multiplication result is ignored anyway. Depending on the faulty value injected into the multiplication module, up to three instructions following a non-taken branch can be skipped. Similar to the forwarding vulnerability, this fault exploits the comparison at line 26 in Listing 3.1. However, this vulnerability does not exist in `v3_Og` due to the different instruction order.

As mentioned in Section 2.2.4, this microarchitectural fault contradicts the intuition that unused functional units can be safely optimized away, as they can still be targeted by faults to compromise system security.

3.1.2.3 Aligner

The attacks targeting the Aligner module are grouped in Table 3.4 under the category *ALGNR*. Injecting a fault into the Aligner module prevents the PC_{IF} from being updated, desynchronizing it from the PC_{PFB}. Until the next *taken* direct branch, the program keeps running normally since the instructions are correctly read from memory and sent to the next pipeline stage. When a direct branch is taken, the PC_{IF} is used as a reference to calculate the target address. As the PC_{IF} does not correspond to the address of the currently executed instruction because of the attack, the program jumps to an incorrect address. The instruction flow then differs from the expected one.

A fault in the Aligner leads to a successful authentication in `v3_Og` and `v7_Og` by jumping a certain amount of instructions. Here, the exit branch of the comparison loop (line 21 in Listing 3.1) directly jumps at the instruction that allows a successful authentication (line 28).

3.1.2.4 Prefetch Buffer

Faults categorized with the *PFB* label in Table 3.4 consist of modifying signals that control the prefetch buffer FIFO, as depicted in Figure 3.3. For instance, the signal `status_cnt` indicates the number of instructions currently in the queue. The different possible consequences of an attack injecting a non-null value in the signal `status_cnt` when the FIFO is empty are described below and illustrated in Figure 3.4.

1. **Execute the FIFO content:** A fault injection at cycle 22 sets the signal `status_cnt` to 1. As a consequence, the FIFO is no longer considered empty: the instruction $\texttt{IA}^{\dagger}$, which is a witness of a previous execution, is executed (`pop` signal). Moreover, the fetched instruction $\texttt{I1}^{\star}$ is written in the FIFO (`push` signal). The `status_cnt` is then still equal to one as a `push` and `pop` had

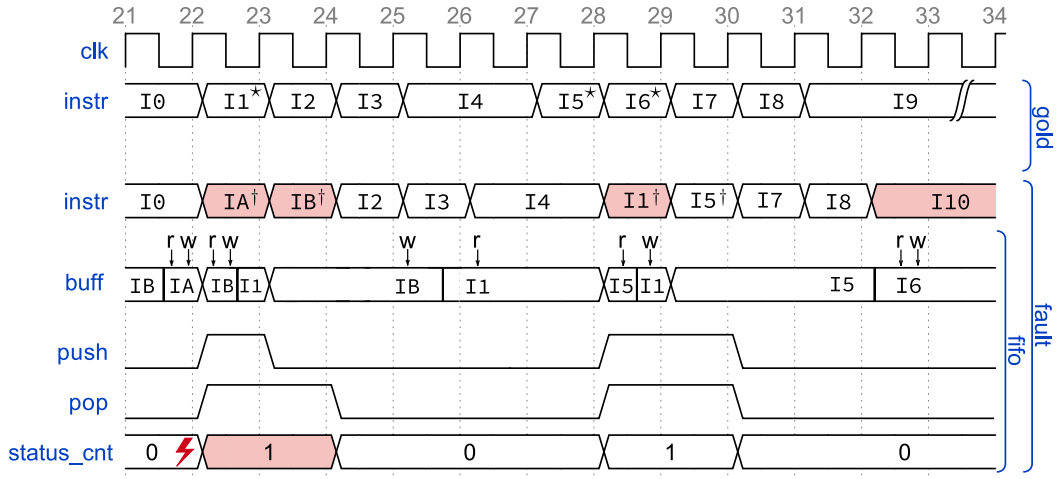


Figure 3.4: Consequences of fault injection on the PFB. The upper part named *gold* represents the non-faulty execution while the lower part named *fault* illustrates the faulty execution where the signal `status_cnt` is subject to fault injection. Symbols $\overset{r}{\downarrow}$ and $\overset{w}{\downarrow}$ indicate the position of the read and write pointers in the prefetch buffer (*buff*). `Ix`* denotes instructions that should have been executed but were not because of the fault. `Ix`† identifies instructions badly fetched from the prefetch buffer due to the fault. Colored signals indicate a divergence wrt. to the golden execution.

occurred at the same time. However, the `read` ($\overset{r}{\downarrow}$) and `write` ($\overset{w}{\downarrow}$) pointers have been incremented. As a result, at cycle 23, instruction `IB`, which is also a witness of a previous execution, is sent to the ID stage (`pop` signal at cycle 23) and executed. The FIFO is now empty (cycle 24), and the `read` pointer points to I1, which has not been executed yet.

2. **Desynchronize the read and write pointers:** When reading instructions `IA` and `IB` in the FIFO due to the fault injection, the `read` pointer is incremented. Consequently, when the FIFO is empty again (cycle 24), the two pointers which should point to the same address are now misaligned. This has no effect during cycles 24-27 because the instructions are directly transmitted to the ID stage without going through the PFB since the FIFO is empty. However, when a stall occurs at cycle 27, the instruction `I5`* is stored and should be executed at the next cycle. Instead, the instruction `I1`† from the location pointed by the `read` pointer is sent to the pipeline at cycle 28. This effect lasts until the next taken branch that resynchronizes the two pointers by flushing the FIFO.

3. **Branching to an incorrect address:** As explained before, the PFB has its own PC (PC_{PFB}) to perform read requests in the instruction memory. However, since the contents of the FIFO have been replayed without notifying the rest of the processor, the PC_{IF} has shifted by the number of replayed instructions, i.e., the value injected in the `status_cnt` signal – one in this example. Consequently, the next branch will be made one instruction too far

(`I10` instead of `I9` at cycle 32).

As a result, a single fault injection in the PFB can lead to various effects with immediate and potentially long-term consequences. In particular, the faulty value set to the 2-bit signal `status_cnt` can further desynchronize the read and write pointers: a branch up to 3 instructions after the correct address may happen.

In both `v3_Og` and `v3_Os`, the instructions contained in the buffer are replayed without side effects for successful faults that fall in this category. However, the desynchronization of the pointers `read` and `write` allows the PIN-comparison loop to be exited prematurely. The countermeasure verifying the loop counter detects this faulty behavior. A call to the countermeasure function is then performed, but it jumps three instructions further. Consequently, the execution continues in the code region where the user is assumed to be authenticated.

3.1.2.5 Remaining Faults

Some faults identified in Table 3.4 are categorized as *other*. Many of these are related to multiplexer selection signals that control operations in the EX stage. These fault effects are mostly computational errors, and we detail one of them below. Injecting a fault into the `bmask_b_mux` signal in both the `Og` and `Os` versions of VerifyPIN allows successful authentication. In the last update the Boolean variable holding the result of the PIN comparison, the Boolean *false* is written instead of *true*. This occurs due to the encoding choice for hardened Boolean values, where *false* = 0x55 and *true* = 0xAA. The fault injection causes a one-bit right shift of the result, i.e., 0xAA $\gg$ 1 = 0x55. This behavior raises concerns about the choice of hardened Boolean encodings. While they maximize the Hamming distance, some elementary operations can convert one value to the other (e.g., *false* + *false* = *true*; *false* $\ll$ 1 = *true*), which should be avoided.

3.1.3 Discussion

We discuss the important results of this section.

Microarchitectural Faults. The previous results involve subtle fault effects that cannot be directly addressed with an ISA-level analysis. Specifically, we targeted mechanisms in the microarchitecture, such as the pipeline or speculative behavior of the PFB, which are not visible at the ISA level. We observed that the vulnerabilities are closely tied to the microarchitectural state. Both the state before the fault injection—the replayed PFB content—and the state after the fault injection—remaining data in flip-flops—must be considered. These microarchitectural elements can lead to the manifestation of vulnerabilities a long time after the injection and depend on software conditions, such as the next branch, to exploit the fault effects.

Finally, it is worth mentioning that vulnerabilities are often the result of a good timing of the fault injection, with a well-chosen instruction sequence, and a specific

microarchitectural state left by the previously executed instructions. Consequently, minor differences in the binary code, such as the number of instructions, code layout, or instruction order, can cause some vulnerabilities to disappear or emerge. Therefore, vulnerabilities stemming from hardware implementation specificities are subtle and require analysis of both the software and hardware, as proposed by µArchiFI.

VerifyPIN_v3 vs. VerifyPIN_v7. The differences between the results from VerifyPIN_v3 and VerifyPIN_v7 can be attributed to two main reasons. First, many previously identified fault injections resulted in inverting or skipping a conditional branch. The *test duplication* countermeasure prevents skipping both tests with a single fault injection, thus eliminating these vulnerabilities. However, two fault injections exploiting the *ALGNR* category remain, as the duplication test countermeasure is ineffective against this fault effect. Second, as described above, many successful fault attacks depend on a specific microarchitectural state and software conditions that make the fault exploitable. A different ordering of instructions between both versions is sufficient to remove vulnerabilities.

Previous Version of µArchiFI. The results presented in this section were published at the FDTC conference [TAC+22] using µArchiFIv0, which only supports the Smt-lib and Yosys-Bmc model-checking backend. As described in Section 2.4.5, faults were directly modeled in Smt-lib rather than injecting them into Rtlil. This made µArchiFIv0 less expressive for fault injection, and the fault model $\mathcal{F}$ used was not exhaustive due to optimizations by Yosys's backend, which rendered some fault locations untargetable in the Smt file. However, this limitation did not impact the purpose of this section, which was to identify security weaknesses due to the processor microarchitecture. We ensured the reproducibility of the results described using µArchiFI.

Conclusion. In this section, we demonstrated various uses of µArchiFI to investigate and highlight the consequences of faults on the hardened VerifyPIN program. We validated our approach by reproducing known attacks exploiting the forwarding mechanism and discovering new effects, such as those involving the prefetch buffer. Consequently, the design of security-critical systems requires a thorough analysis of both hardware and software, as the intricate coupling between microarchitecture and the running program leads to subtle fault consequences.

The next section focuses on evaluating a robust version of the CV32E40P core implementing the MAFIA countermeasure, which ensures code, control flow, and control signal integrity.

3.2 Case Study II: Control Signal Integrity

As discussed in Section 1.3, related works on hardware/software system evaluation have primarily conducted their analyses in a safety and reliability context using

simulation techniques [JAR+94, STB97]. In this section, we demonstrate the capability of µArchiFI to perform these analyses exhaustively using formal methods to prove system security against fault attacks when no vulnerabilities are found. The rest of this section is organized as follows.

First, we describe a case study implementing the MAFIA countermeasure. Second, we present and discuss the evaluation results.

3.2.1 Experimental Set-Up

The following paragraphs describe the use case analyzed with µArchiFI and the formal model it generates for model-checking verification.

3.2.1.1 MAFIA Protection

MAFIA [CCH22, CCH23] is a countermeasure designed to protect against fault injection attacks by ensuring the integrity of code, control-flow, and control signals. The cornerstone of this countermeasure is the *pipeline state*, which is derived from the control signals emitted by the ID stage in the processor. The pipeline state is designed to detect up to 8-bit alterations in (i) the binary encoding of program instructions, or (ii) the control signals in the IF and ID stages. At each cycle, an integrity signature is calculated based on the current pipeline state value and the previous integrity signature. Thus, any modification of the pipeline state should result in an altered integrity signature. This integrity signature is then compared against reference values placed at specific locations in the binary program (e.g., control-flow transfers). This process ensures code and control-flow integrity up to the end of the ID stage. To complete the integrity protection throughout the processor, a redundancy mechanism protects the control signals issued by the ID stage (downstream of the pipeline state calculation). An alarm signal is triggered whenever an integrity violation is detected.

A thorough security assessment of a MAFIA implementation against fault attacks must consider the following security properties:

ϕ_{up} Faults applied upstream from the pipeline state lead to an alteration of the pipeline state.

ϕ_{down} Faults applied downstream from the pipeline state are detected by the redundancy mechanism, i.e., the alarm signal is triggered.

ϕ_{check} Pipeline state alterations or faults in the integrity signature calculation do not produce valid signature values. Faults applied to integrity signature verification and alarm signal propagation do not create exploitable vulnerabilities.

In this section, we examine the MAFIA implementation integrated into the baseline CV32E40P core. The pipeline state integrates 13 carefully chosen signals

from the microarchitecture.

3.2.1.2 VerifyPIN

MAFIA is a combined countermeasure that inserts signatures into the binary program at compile time. For the countermeasure to operate correctly, we assume that the program is correctly built and that the signatures are inserted in the appropriate locations. Since µArchiFI is designed to evaluate reachability properties for hardware/software systems, it cannot prove the robustness of MAFIA independently of the executed program and must evaluate it in the context of a given program, e.g., VerifyPIN. Additionally, we use concrete values for `userPIN` and `cardPIN` instead of symbolic ones, as given below:

$$\texttt{userPIN} = \{1, 2, 3\}\,, \quad \text{and} \quad \texttt{cardPIN} = \{5, 6, 7\}$$

As a result, we focus on evaluating MAFIA in the context of the VerifyPIN function, as introduced in Section 3.1.

MAFIA is verified using two versions of VerifyPIN: VerifyPIN_v3, identified as the most vulnerable version in the previous section, and a weaker version of VerifyPIN without the step counter countermeasure (**SC**). Both versions are compiled with the `0g` optimization to ease the manual inspection of the result and ensure that the compiler does not optimize away the countermeasure. These versions are referred to as `mafia_v3_0g` and `mafia_v3_weak_0g`, respectively.

3.2.1.3 Generated Circuit Model

Similar to Section 3.1, Table 3.5 reports the characteristics of the circuit model. First, we observe that the design size is 56 500 GE, representing an 18% increase compared to the baseline version of the CV32E40P core, which is in line with the original work [CCH22]. Additionally, the size of the memory model is larger than that of the baseline VerifyPIN versions because MAFIA requires the insertion of CFI signatures into the binary. Finally, the memory is initialized with the content of the VerifyPIN program, with unused memory locations zeroed out.

In contrast to Section 3.1, we cannot precompute the system's initial state to start the analysis from the first VerifyPIN instruction. The technique employed to constrain the initial state cannot perfectly restore the pipeline state, which results in incorrect signatures at runtime. Consequently, we begin the analysis from the booting point, requiring the execution of 50 instructions before reaching the first VerifyPIN instruction.

3.2.1.4 Attacker Model

Fault model. Similar to the previous section, the fault model $\mathcal{F}$ targets control signals, arbitrarily defined to be less than 6 bits, with symbolic fault effects as formalized in Equation (3.2). Given this fault model $\mathcal{F}$, we assume that an attacker

Table 3.5: MAFIA circuit model characteristics.

Design Name	Size (GE)	Wires		Gates (#)	Registers		Memory Model	
		(#)	(# bits)		(#)	(# bits)	Size	Symbolic Data
MAFIA	$\sim$ 56 500	5 629	56 986	4 954	212	3 444	32×2^{11}	0

can inject a single transient fault at any time during the VerifyPIN execution, as formalized below:

$$\mathbf{F} \subseteq \mathcal{F} \times [51, n], \quad \text{with } |\mathbf{F}| = 1$$

Faults are not injected during the first 50 instructions as they are not part of the VerifyPIN program. The integer n corresponds to the program execution time, as summarized in Table 3.6. The verification bound is set to 140 for each program.

Table 3.6: Execution time and BMC bound for each VerifyPIN version on MAFIA.

VerifyPIN versions	`mafia_v3_weak_0g`	`mafia_v3_0g`
Length n (clock cycle)	115	120
BMC bound $n + \epsilon$	140	140

Attacker goal. As introduced above, our study focuses on verifying properties ϕ_{up} and ϕ_{down}. The verification of ϕ_{check} is related to the security analysis of the integrity signature wrt. an attacker model that depends on the function signature implemented. This was already carried out in the original work [CCH22] and we assume that ϕ_{check} is correct in the following.

Our security assessment verifies that an attacker cannot inject a fault that modifies the control flow, more precisely that bypass the authentication (φ_{authen}), without modifying the pipeline state or being detected by the redundancy mechanism. Equation (3.4) summarizes this property.

$$\varphi_{mafia} := \varphi_{authen} \wedge (\neg\phi_{up} \vee \neg\phi_{down}) \tag{3.4}$$

Intuitively, Equation (3.4) allows an attacker to reach his goal using two different attack scenarios. On the one hand, a fault can be injected upstream to the pipeline state and allow authentication without modifying the pipeline state, i.e., $\varphi_{authen} \wedge \neg\phi_{up}$ is true. On the other hand, a fault can be injected downstream to the pipeline state and allow authentication without being detected by the redundancy mechanism, i.e., $\varphi_{authen} \wedge \neg\phi_{down}$ is true.

3.2.2 Evaluation Results

Results. The formal verification we carried out did not identify any vulnerabilities in `mafia_v3_0g` or `mafia_v3_weak_0g`. Therefore, we have proven that MAFIA effectively protects these versions of VerifyPIN against the considered fault model, preventing an attacker from bypassing the authentication mechanism.

Discussion. In this section, MAFIA's security analysis was conducted on the VerifyPIN program, making the results software-dependent. However, to our knowledge, no existing methodology can verify combined countermeasures like MAFIA, which requires modeling both the microarchitecture—to represent control signals, perform control-flow integrity calculations, and inject faults—and the software that integrates the reference signatures to be verified at runtime. µArchiFI is, therefore, the first methodology capable of formally modeling a hardware CFI countermeasure and analyzing the effects of faults on software security. Proving the robustness of MAFIA independently of the executed program would be an interesting future research direction, but this would require modeling the insertion of signatures for an arbitrary class of software, which is beyond the scope of this thesis.

Another important topic for discussion is the completeness of the fault evaluation. Similar to Section 3.1, evaluations in this section used the previous version of the tool, µArchiFIv0. The fault model considered is exhaustive in terms of injection timing, as faults were injected throughout the program execution, and exhaustive in terms of effects, as we considered symbolic effects. However, due to restrictions on the tool's expressiveness, the fault model does not include all possible microarchitectural elements. Consequently, the security proof only holds for the chosen fault model $\mathcal{F}$ (cf. Equation (3.2)). To achieve stronger security guarantees, it is necessary to consider a fault model targeting all circuit logic elements. However, increasing the size of the fault model may negatively impact performance, making the analysis impractical. Therefore, performance is the focus of Section 3.3.

3.2.3 Conclusion

This section demonstrated µArchiFI's capability to formally prove the security of a CFI countermeasure against a specified fault model. Achieving such a security proof was previously impossible with either software-only or hardware-only evaluation techniques, as they could not model the operation of a CFI countermeasure. Additionally, existing co-verification approaches based on simulation could not provide a formal proof of security as they are unable to cover the entire state space. µArchiFI thus presents a novel approach to tackling these co-verification challenges.

Sections 3.1 and 3.2 deliberately focused on experimental results, setting aside performance considerations. The next section concentrates on performance analysis.

3.3 Performance Evaluation

Previous sections of this chapter have focused on demonstrating µArchiFI's ability to automatically identify corner-case microarchitectural vulnerabilities and to prove system robustness against a given fault model. However, formal verifica-

Table 3.7: Verification time for each VerifyPIN analysis. The *faults* column indicates the number of fault locations (spatial and temporal) explored.

Hardware	VerifyPIN version	Run Time (h)	Faults (#)	BMC bound
CV32E40P	`v3_0g`	12.9	15 240	74
	`v7_0g`	13.9	17 526	76
	`v3_0s`	14.1	14 478	62
	`v7_0s`	14.5	15 240	66
+ MAFIA	`mafia_v3_weak_0g`	26.7	9 706	140
	`mafia_v3_0g`	20.5	8 685	140

tion techniques come at a cost: they are time-consuming. This section focuses on µArchiFI's performance.

Section 3.3 is not intended to exhaustively benchmark µArchiFI's performance, as too many parameters enter into consideration. From the modeling side, verification time depends on the size of the hardware/software model—including the circuit and program size—and the number of possible fault injections. From the verification side, computation time depends on the complexity of the attacker's goals, the bound of the BMC analysis, and the model checker employed, including all the optimizations they implement. Furthermore, any change in these parameters may have non-intuitive effects on verification performance due to the solvers being used as a black box. For example, reducing the state space with additional constraints can increase verification time since the SMT queries become more complex to solve. Consequently, exhaustively quantifying the performance impact of all these parameters is beyond our reach. Instead, we propose additional case studies representative of various security scenarios to highlight the impact of solvers on performance and the effect of using *sandboxing* and *concretization* strategies. We also demonstrate that µArchiFI scales with multiple fault injections. The rest of this section is organized as follows:

First, we report the performance of evaluations conducted in Sections 3.1 and 3.2. Second, we introduce three use cases to evaluate the impact of the backend solver and verification strategies. Finally, we conclude this section with a discussion on performance evaluation. Results presented in this section are available in open access for reproducibility[2].

3.3.1 Performance of µArchiFIv0

We begin this section by reporting the performance of the analyses conducted in Sections 3.1 and 3.2. Table 3.7 shows the verification times using an Intel Xeon E7-4870 2.40GHz CPU. First, the fault evaluation on the unprotected CV32E40P

[2] https://zenodo.org/records/7958412

took between 12.9 and 14.5 hours to complete for each version of VerifyPIN. In contrast, evaluating VerifyPIN programs with the CFI protection MAFIA took up to 26.7 hours. The *Faults* column indicates the number of faults considered in the fault model, and the *BMC bound* column indicates the length of the execution trace we evaluate. It is important to note that faults were not injected during the first 50 clock cycles of programs running on MAFIA.

To compare our approach against existing simulation frameworks and understand the number of concrete executions required to achieve the same coverage, it is essential to emphasize that symbolic variables were used during our analyses. First, the 48 bits of PIN codes are left unconstrained on VerifyPIN running on the baseline CV32E40P. Second, the effects of the injected faults are also symbolic, covering every possible effect. These results demonstrate that the developed workflow can formally verify the execution of a hundred instructions with symbolic data in the presence of faults within a reasonable verification time.

To further evaluate the verification performance of µArchiFI, the remainder of this section introduces three use cases designed to have a reasonable execution time and discusses the impact of various formal verification backends and optimizations.

3.3.2 Evaluation Scenarios

In contrast to the results presented in Sections 3.1 and 3.2, the evaluations conducted in this section have been performed using the latest version of µArchiFI as described in Section 2.4. Below, we introduce three use cases that illustrate various security scenarios: *robust software*, *robust hardware*, and *cryptographic software*. These use cases are manually dimensioned to ensure reasonable verification times, allowing us to evaluate the impact of the backend solver and the verification strategies introduced in Section 2.4.4.

Table 3.8: Use cases characteristics.

Use Case	Hardware Design				Software Program		Fault Model			Attacker	
	Name	kGE	Gates	Regs	Name	gcc	Location	Effect	Timing	Goal	Order
I	CV32E40P	46.8	2 842	179	VerifyPIN_v7	`0g`	R in Control path	Symbolic	60:75	φ_{I}	1
II	Secure Ibex	61.3	4 422	211	VerifyPIN_v1	`0s`	R in Lockstep	Symbolic	∞	φ_{II}	5
III	Ibex	26.4	1 983	114	KeySchedule	`0s`	C in EX Stage	Reset	∞	φ_{III}	2

3.3.2.1 Use Case I: Robust Software

Use Case I illustrates the possibility for a user to analyze the robustness of a secure program running on a processor. It is an adaptation of Section 3.1 with a different fault model. Key components of the use case are described below and summarized in Table 3.8.

Software. In Use Case I, we target VerifyPIN_v7, the most secure version of the VerifyPIN suite as introduced in Section 3.1.1.2. The program is compiled with the optimization flag `Og` to prevent the compiler from removing the countermeasures, and it runs in constant time, finishing in 69 clock cycles. In this scenario, we consider 3-digit symbolic PIN values, and we assume that the user PIN and the secret PIN differ in each digit (cf. Equation (3.1)).

Hardware. The program is executed on the CV32E40P processor [Opeb], as introduced in Section 3.1.1.1. The processor circuit is represented at the word level and flattened, i.e., we instantiate every module. This flattening is necessary to constrain the maximum number of faults, as explained in Section 2.4.5 regarding tool limitations.

Attacker goal. In this system, the attacker aims to bypass the secure authentication mechanism without triggering the software countermeasures.

$$\varphi_{\text{I}} := (\texttt{g\_authenticated} \wedge \neg \texttt{g\_countermeasure})$$

Fault model. The attacker we consider targets the comparison evaluating the results of the `byteArrayCompare` function, as shown between lines 26 to 30 in Listing 3.1. We target this code sequence as it was revealed to be vulnerable during previous evaluations in Section 3.1. We evaluate the robustness of these instructions against a single fault injected into the sequential logic of the processor control path, i.e., registers whose size is less than 6 bits. The considered fault model targets 102 registers out of the 179 registers in the processor and is formalized below:

$$\mathbf{F} \subseteq \mathcal{F}_{\text{I}} \times [60, 75], \quad \text{with } |\mathbf{F}| = 1, \text{ and } \mathcal{F}_{\text{I}} = \{(g, e) \in G \times E \mid g \in R \wedge \mathit{size\_of}(g) \leq 6\}$$

3.3.2.2 Use Case II: Robust Hardware

Use Case II details how a user can determine whether a fault injected into a secure processor can induce a vulnerable behavior on the software without being detected by the hardware countermeasure. In contrast with Section 3.1, we evaluate another secure processor, the Ibex core.

Software. We consider VerifyPIN_v1, the baseline version of the VerifyPIN collection, without any countermeasure except hardened Booleans. As in Use Case I, we consider symbolic 3-digit PINs. VerifyPIN_v1 is compiled with the optimization flag `Os`.

Hardware. The Ibex [low18] is a parametrizable open-source 32-bit, in-order processor. We analyze the *small* version of the core [lowb] in its *secure* configuration. The secure Ibex implements protections against physical attacks like the dual-core lockstep mechanism that instantiates the core twice and compares outputs. The duplicated core is called the *shadow core* and an alert signal is triggered if an attack has been detected during the operation of the processor. More details on Ibex security are later provided as it is the focus of Chapter 4.

Attacker goal. In this second use case, the attacker still aims to bypass the secure authentication mechanism without triggering the hardware countermeasures.

$$\varphi_{\text{II}} := (\texttt{g\_authenticated} \wedge \neg \texttt{hardware\_alert})$$

Fault model. The attacker we consider can inject at most five faults into the system. In order to investigate if the dual-core lockstep is well designed, we restrict fault locations to the shadow core only as we do not want to inject the same fault in both cores. Faults are injected into registers as formalized below:

$$\mathbf{F} \subseteq \mathcal{F}_{\text{II}} \times \mathbb{N}, \quad \text{with } |\mathbf{F}| = 5, \text{ and } \mathcal{F}_{\text{II}} = \{(g, e) \in G \times E \mid g \in R_{\text{SHADOW}}\}$$

3.3.2.3 Use Case III: Cryptographic Software

Use Case III details how a user can apply the tool to software implementations of cryptographic algorithms.

Software. Tiny AES [kok19] is a small software implementation of the standard AES. The key schedule function of the AES program expands the key into several separate round keys. We focus here on a round of the key schedule function from the 128-bit AES. Input data such as the key and the plaintext have been set to arbitrary values. The program is compiled with the optimization flag `Os`.

Hardware. We run the key schedule function on the baseline version of the *small* Ibex core without any countermeasure.

Attacker goal. The attacker wants to set to zero a byte in the penultimate round key as it is a requirement for some known differential fault attacks [TFY07, AM11]. Fault consequences are evaluated at the end of the key schedule function to limit the analysis to a small sequence of instructions.

$$\varphi_{\text{III}} := (g^{th}\ Round\_key_{\text{byte}} = 0)$$

Fault model. We allow an attacker to inject up to two *reset* faults anywhere in the execute stage combinational logic of Ibex, as follows:

$$\mathbf{F} \subseteq \mathcal{F}_{\text{III}} \times \mathbb{N}, \quad \text{with } |\mathbf{F}| = 2, \text{ and } \mathcal{F}_{\text{III}} = \{(g, e) \in G \times E \mid g \in C_{\text{EX}} \wedge e = reset\}$$

3.3.3 Performance Results

All verifications are executed on an 11th Gen Intel Core i7-1185G7 CPU. Each program presented in this section is compiled with the RISC-V toolchain for the RV32IMC architecture (gcc version 10.2.0). For each use case verification, the BMC bound n is set according to the longest program execution trace plus a 10-percent increment to capture possible modifications in the control flow.

Table 3.9: Use-cases verification time (in seconds) with three model checkers.

Use Case	BMC Bound	BMC without Fault				BMC with Faults			
		PONO	YOSYS-BMC	BTORMC	Reach φ	PONO	YOSYS-BMC	BTORMC	Reach φ
I	75	12.6	11.1	1.5	✗	107	249	273	✓
II	46	20.7	10.6	3.5	✗	250	373	timeout	✗
III	38	0.3	2.4	0.1	✗	313	1 945	3 427	✗

Table 3.10: Verification time improvement with the sandboxing technique wrt. the baseline verification time (in seconds) with faults in Table 3.9.

Use Case	PC Sandboxing	PONO	YOSYS-BMC	BTORMC
I	0x1c4 ≤ PC ≤ 0x234	110 (+2.8%)	242 (-2.8%)	205 (-24.9%)
II	0x84 ≤ PC ≤ 0x114	206 (-17.6%)	297 (-20.4%)	timeout
III	0x40 ≤ PC ≤ 0xc0	107 (-65.8%)	1 454 (-25.2%)	1 659 (-52.0%)

Use Case I: Robust Software Table 3.9 compares the verification performance of use cases with and without faults, using three model checkers: YOSYS-BMC, PONO, and BTORMC. Performing the verification without fault ensures that the attacker goal φ does not hold outside of an attack. The analysis results in Table 3.9 highlight that the attacker can bypass the authentication by injecting a single fault. Counterexamples provided by the model checkers allow the user to find the exact location of the fault that leads to the vulnerability φ_{I}. All solvers have identified the same fault model for this use case, but PONO was the fastest in solving the model-checking problem. The location of the identified fault is the `alu_vec_mode_ex` signal which is located in the `id_stage` module. This fault was already discovered in Section 3.1 as shown in Table 3.4.

Use Case II: Robust Hardware Table 3.9 reports that an attacker cannot bypass the secure authentication with the considered fault model. This use case leverages the fact that the secure Ibex implements hardware countermeasures. The assumption that the *hardware_alert* cannot be triggered makes sense as the attacker wants to bypass the authentication without being detected. Additionally, this assumption helps the solver to simplify the formula during verification. The second line of Table 3.9 shows the verification performance. BTORMC fails to solve the problem, and we stopped the verification with a 2-hour timeout.

Use Case III: Cryptographic Software As reported in Table 3.9, an attacker cannot reach his goal with the considered fault model. Notably, verifying φ_{III} on the AES program without fault is faster than both Use Case I and II because the AES key is fixed, whereas the two 3-digit PINs are symbolic for the VerifyPIN program.

Table 3.11: Verification time improvement with the concretization technique wrt. the baseline verification time (in seconds) with faults from Table 3.9.

Use Case	Concretized step	Baseline Yosys-Bmc	Concretization	
			Parallelized	Accumulated
I	62 (Status comparison)	249	189 (-24.1%)	509 (+104%)
II	31 (PIN comparison)	373	304 (-18.5%)	891 (+139%)
III	23 (No branch instruction)	1 945	1 504 (-22.7%)	2 955 (+51%)

3.3.4 Influence of Verification Strategies

In the following, we evaluate and discuss the impact of software-based verification strategies on performance with regard to the baseline version of µArchiFI.

Sandboxing

For each use case, we determine the range of possible values for the program counter (PC) by manually examining addresses from the binary file. Here, the possible addresses are contiguous, and we add a global constraint on the system to force the PC to stay in this set of values. Table 3.10 illustrates that the sandboxing strategy results in a performance improvement up to 65%, and these additional constraints do not prevent model checkers from retrieving the vulnerability highlighted in Use Case I.

Such improvements are due to two factors. First, some fault effects are not analyzed if they lead to PC values that are out of the memory range considered. Second, the verification may end before the bound n if all possible execution paths exit from the considered range of PC addresses. We also observe that improvements vary between the different solvers even if Pono remains more efficient on the use cases analyzed.

Concretization

Performance is given for the Yosys-Bmc since other evaluated model checkers do not permit to retrieve the SMT formula encoding the unrolled system. We apply the concretization strategy for each use case with an arbitrarily chosen enumeration bound $L = 3$ to split the bounded verification procedure into $L+1$ sub-verifications (cf. Algorithm 2.3).

Table 3.11 reports the concretization steps, the baseline verification time from Table 3.9, and the concretization performance. We show each experiment's wall-clock time and accumulated verification time since we can parallelize the executions.

On Use Case I, we concretize the execution at the first branching instruction targeted by fault injection. It corresponds to the PIN-status comparison to allow authentication (step 62 mentioned at line 1 in Table 3.11). This results in an im-

provement of the verification time by 24%. On Use Case II, we apply concretization during a PIN-digit comparison and enumerate PC values associated to different execution paths. However, few performance improvements are observed, especially regarding the accumulated verification time. We believe this is due to the hardware countermeasure that already prevents executing different paths due to the faults. No branching instruction exists on Use Case III. However, many execution paths are possible due to fault injections. Concretization is applied at step 23, at half of the verification time. This results in a 22.7% improvement of the verification time (cf. line 3 in Table 3.11).

The concretizing technique splits a complex formula *reachable* into simpler formulas. Intuitively, the resulting formulas are easier to solve since symbolic variables are replaced with constant values. In practice, even if the resulting state space has fewer states to explore, solvers may have more difficulty simplifying and solving the concretized formula. As a result, the concretization strategy showed a limited improvement with respect to the baseline version.

In conclusion, concretization often improves the verification time thanks to the parallelization of the executions. However, these verification times remain higher than those obtained using the Pono model checker (Table 3.9).

3.3.5 Discussion

Several questions on µArchiFI performance may stem from the previous analyses. We discuss them in the following.

Case Study Selection. In this section, we arbitrarily choose three case studies representative of various security-critical scenarios. However, even if the choice of these use cases was not discussed in this section, it results from a thorough expertise from the evaluator. We select these cases to have reasonable verification time that can run on a laptop and for which we can benchmark the impact of some verification parameters. The user of µArchiFI must find a sweet spot between the size of the hardware design, the size of the analyzed program, and the complexity of the fault model.

Influence of Solvers. The first lesson to be learned from this section is the importance of the backend solver. We evaluated two state-of-the-art solvers, Pono [MIL+21], and BtorMC [NPWB18], and Yosys-Bmc which is the in-house solver of Yosys. Pono was on average, two to ten times faster than the others, and we experienced some timeouts using BtorMC.

Impact of Verification Strategies. On the one hand, sandboxing has decreased the verification time by 25% on average. It is an interesting under-approximation to find security vulnerabilities but must be used carefully to prove system robustness to faults as it loses completeness. On the other hand, concretization applied to Yosys-Bmc is less efficient than state-of-the-art solvers like Pono. Other configurations

or parameters of these optimizations could have been explored, such as modifying the number of concretizations or the depth at which they are performed. Also, a combination of the two strategies would have been interesting to study. However, these methods showed small improvements with respect to the baseline version. To scale up to larger case studies, we must improve our approach by orders of magnitude, which software-based optimizations cannot achieve.

Performance Limitations of µArchiFI. The bottleneck of µArchiFI is the complexity of the generated model checking problem which is still too difficult to be efficiently solved by state-of-the-art model checkers. This is primarily due to the monolithic approach that models every system component inside a single formal model. Intuitively, the complexity of the model-checking problem depends on the size of the design, the number of combinations of fault injections, and the number of unrolling corresponding to the program length. As a future work, an interesting idea to investigate is to prove intermediate properties and use them to simplify the model checking procedure. This idea is developed in the next chapter.

3.4 Conclusion

Chapter 3 concludes the presentation of µArchiFI, a workflow designed to formally analyze the impact of hardware-level faults on software security. In Section 3.1, we showcased µArchiFI's capability to identify multiple vulnerabilities in security-critical contexts. These results highlight its practical relevance for automatically detecting corner-case vulnerabilities that simulation-based analyses could miss. In Section 3.2, we showed that µArchiFI is valuable for proving the robustness of countermeasures like MAFIA. While state-of-the-art methodologies focus on either the formal verification of cryptographic circuits or fault analysis at the software level, none could verify joint protections such as hardware CFI, which require representation at both levels. µArchiFI fills this gap. In Section 3.3, we reviewed µArchiFI's performance, showing how different model checkers, sandboxing, and concretization strategies influence verification time. µArchiFI can verify up to a hundred instructions on a small microcontroller with a single fault injection. The analysis can be extended to multiple faults if the attack area is limited to a processor sub-module.

Developing µArchiFI has involved overcoming several scientific challenges we identified in Section 1.4. First, we proposed a joint modeling of software, hardware, and faults, and selected the most relevant verification algorithm, as outlined in Chapter 2. Next, we adapted the theoretical model to real-world security evaluations through representative case studies of various security scenarios. The final challenge introduced in Section 1.4 that remains to be addressed is scalability. This issue has been addressed in previous sections through various methods for abstracting and simplifying the verification problem. However, even if µArchiFI benefits from state-

of-the-art model checkers, the generated transition-state system is often too complex to solve. For example, the fault model evaluated in Section 3.2 was incomplete because we limited our focus to control signals. A more exhaustive proof would not have been feasible with the current approach. Achieving a thorough formal security proof for real-world case studies, such as secure elements where security is critical, requires new verification paradigms due to the inherent complexity of these systems.

Overcoming the scalability problem to achieve an exhaustive proof of security is the focus of the next chapter.

Chapter 4

Preliminary Hardware Analysis using Fault-Resistant Partitioning

Contents

In Chapter 2, we introduced µArchiFI, a methodology that integrates software and hardware into a unified formal model for applying model-checking techniques. However, as shown in Chapter 3, modeling all system components together leads to scalability issues, and state-of-the-art model checkers cannot handle such complex models. This limitation is particularly restrictive when proving system robustness, as an exhaustive fault model is required to cover every possible attack scenario. Consequently, only up to a hundred instructions executed on a microcontroller-like processor can be analyzed for a single fault injection. To challenge our approach, we decided to tackle the formal security proof of the OpenTitan secure element running the first stage of its secure boot. However, such a verification problem is beyond the capabilities of the current µArchiFI version, necessitating an enhanced verification methodology. This chapter focuses on addressing this challenge.

This Chapter 4 is a step aside from the previous work done with µArchiFI. Verifying a secure element while considering its software and hardware layers is too significant to be treated as a single verification step. Instead, the approach developed in this chapter involves breaking down the verification problem into several preliminary steps based on the existing countermeasures in the secure element. An overview of this chapter's content is given in Section 4.1 with an emphasis on the new methodology we propose, the challenges we face, and the contributions we make.

The content of this chapter is adapted from the publication at *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)* [THN+24]. This work was also presented at *Groupe de Travail sur les Méthodes Formelles pour la Sécurité (GTMFS)* [Tol24b] and *Journée sur les attaques par injection de fautes (JAIF)* [Tol24a] in 2024. All the code and experimental artifacts presented in this chapter are publicly available[1].

4.1 Overview

This chapter presents a methodology that decomposes hardware/software co-verification into more manageable steps. Our approach leverages that security-critical systems, such as secure elements, implement hardware and software countermeasures to prove intermediate security properties. Specifically, we propose a preliminary evaluation of hardware-level countermeasures to ensure they effectively detect or correct faults, preventing incorrect circuit behavior. Consequently, only faults not captured by hardware protections need to be evaluated in the hardware/software co-verification step, helping to contain the state space explosion.

In the following paragraphs, we present an overview of the two-stage methodology. We also discuss the hardware verification techniques needed to provide sufficient guarantees for simplifying the fault model in the co-verification step. This section concludes with a summary of the chapter's contributions.

[1] https://github.com/CEA-LIST/Fault-Resistant-Partitioning

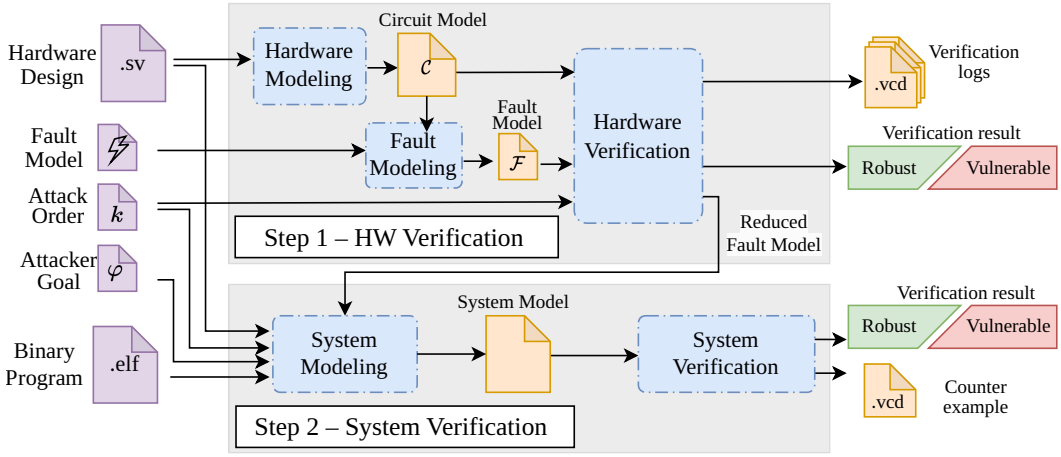


Figure 4.1: Two-step methodology to improve verification of hardware/software systems against faults attacks.

4.1.1 Methodology

The proposed methodology is illustrated in Figure 4.1. Similarly to µArchiFI, the workflow takes as input a hardware design, a binary program, and the fault attack parameters. The novelty of the approach lies in the two verification steps. The first step, labeled *hardware verification* in the top box of Figure 4.1, evaluates the robustness of the hardware circuit's countermeasures against faults at order k. As a result of this first step, the analysis outputs verification results indicating whether the circuit is robust against the considered fault model and provides verification logs for understanding the results. Additionally, a reduced fault model is generated to indicate the faults not protected by the hardware that could be harmful to the system and potentially exploited by an attacker.

The second step, labeled *system verification* and depicted in the bottom box of Figure 4.1, focuses on the co-verification of the entire system, including both hardware and software. This step can be seen as a generalization of the µArchiFI workflow, replacing the input fault model with a reduced fault model generated by the hardware verification.

The advantages of conducting this preliminary hardware verification are manifold. First, it is independent of the executed program, meaning it only needs to be conducted once, and the verification results can be used for multiple software evaluations. If no exploitable faults are identified, the circuit is secure unconditionally of the executed software, and the *system verification* is no longer necessary. Conversely, if the hardware cannot be proved secure, every possible fault must be considered in the co-verification step, similar to the baseline situation with µArchiFI alone. Generally, only some areas of the circuit can be proven secure. Remaining exploitable faults help selecting the best-suited abstraction level during the system modeling step. For example, an ISA-level model suffices when only the values read from memory can be corrupted. When hardware description is necessary, the sys-

tem modeling process can optimize sub-circuits if the reduced fault model does not target them. There is no need to consider a fine-grained detail level for protected parts of the circuit when a behavioral modeling is sufficient.

4.1.2 Hardware Verification

Composing the hardware verification step with the co-verification step requires appropriate hardware guarantees so that the size of the fault model can be reduced. Subsequent paragraphs describe existing definitions and techniques in the literature for evaluating the circuit's robustness to faults.

In 2020, Dhooghe and Nikova introduced the notion of *k-order active security* to build secure systems using small pieces of circuits named gadgets [DN20]. In essence, a *k-active secure* gadget attacked with at most k faults either aborts or produces a correct output. However, their definition focuses only on combinational circuits and does not consider sequential logic. Section 3.1 has demonstrated that faults can remain hidden in microarchitectural registers, such as the prefetch buffer, and modify outputs after an unknown amount of time. Consequently, Dhooghe and Nikova's definition cannot be generalized to sequential circuits like CPUs.

Similarly, the related work on circuit verification described in Section 1.3, such as FIVER [RRSS+21] and SYNFI [NOV+22], relies on bounded equivalence checking. These bounded techniques provide guarantees assuming a fault propagation bound n but cannot prove fault security in the general case and may struggle as the checking complexity increases with n. Unfortunately, state-of-the-art tools scale only for a few clock cycles, which is insufficient to ensure that faults have no long-term effects when evaluating software with hundreds or thousands of instructions.

Since no existing definition or methodology provides strong enough guarantees to abstract faults in the co-verification step, this chapter also contributes in that direction by proposing the new notion of *fault-resistant partitioning* to provide unbounded guarantees on hardware security.

4.1.3 Summary

The contributions of this chapter are twofold:

1) We first extend the *k-order active security* definition to support sequential circuits and propose the new notion of *fault-resistant partitioning* to prove *k-fault security* with unbounded guarantees. We provide an algorithm to build and prove fault-resistant partitioning of circuits and compare its effectiveness against state-of-the-art bounded methods.

2) We leverage *fault-resistant partitioning* as a preliminary analysis of OpenTitan's hardware countermeasures and evaluate the consequences of remaining undetected faults on software security.

The remainder of this chapter is organized as follows:

First, Section 4.2 introduces the additional notations and background necessary for this work. Section 4.3 describes the *fault-resistant partitioning* property at the root of our contributions before detailing its implementation in Section 4.4. Section 4.5 validates our approach against prior work on cryptographic circuits before leveraging our full co-verification methodology to evaluate the fault resistance of OpenTitan's secure processor in Section 4.6. Finally, Section 4.7 evaluates the improvements relative to µArchiFI, and Section 4.8 concludes this work.

4.2 Background

Before formalizing the notion of *k-fault-resistant partitioning* and demonstrating its practical use in evaluating systems like secure elements against fault attacks, we must first introduce additional notations and background essential for this chapter.

First, we introduce the OpenTitan secure element, which motivates the methodology presented in this chapter. Second, the systems we evaluate implement hardware countermeasures that must be analyzed with a bit-level representation, as synthesis tooling might otherwise optimize them away. Therefore, we adapt the notations introduced in Section 2.1 to model bit-level systems with faults. Third, we introduce hardware protections, also known as Concurrent Error Detection (CED) schemes, and specify the security properties they must fulfill. Finally, we describe formal equivalence-checking techniques commonly used in the literature to prove the fault security of application-specific circuits, such as cryptographic ones, and highlight their limitations for general-purpose CPUs.

4.2.1 OpenTitan Secure Element

OpenTitan is an open-source project [JRR+18] which provides a trustworthy hardware secure element [lowe]. The core components of OpenTitan are hardware designs, such as a processor and accelerators; firmware, such as a secure bootloader; and software and security services, such as cryptographic functions or utilities for key management. The secure element implements a wide range of countermeasures to protect the chip's confidentiality, integrity, and authenticity [lowd]. It was notably designed to be resistant to an attacker having physical access to the platform capable of interfering with its operation by performing fault injection attacks.

Internally, OpenTitan uses the 32-bit RISC-V Ibex processor [low18] which interacts with memories and cryptographic accelerators, such as AES or OpenTitan Big Number (OTBN) for asymmetric cryptographic operations like RSA, as shown in Figure 4.2. In this chapter, the hardware analysis focuses on the secure configuration of the Ibex core [lowa], which uses different spatial Concurrent Error

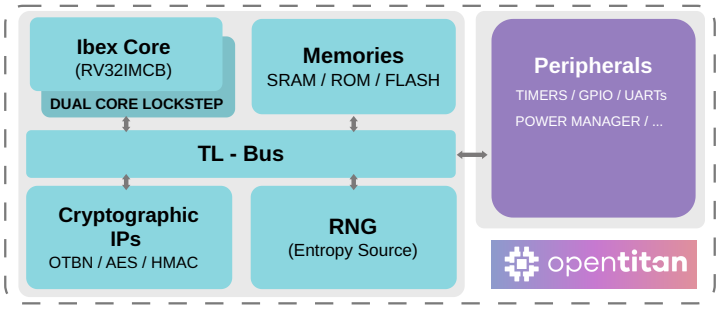


Figure 4.2: OpenTitan block diagram, Earl Grey design [lowc].

Detection (CED) schemes. The *dual-core lockstep* (DCLS) mechanism instantiates the Ibex core twice, compares outputs between the main core and the shadow core, and triggers an alert signal on a mismatch. To increase the protection against faults, the shadow core inputs are delayed for d cycles, where d is fixed at synthesis time. Both core instances share the register file, which is protected against faults with Error Detection Codes (EDC) and a write-enable glitch detection mechanism.

4.2.2 Bit-Level System Modeling

Analyzing the security of hardware countermeasures against fault attacks requires working with a post-synthesis bit-level circuit representation. This approach ensures that countermeasures can effectively detect faults and are not optimized away by synthesis tools. To this end, we adapt the notations introduced in Section 2.1 to work with bit-level circuits.

Bit-Level Circuit Model

A circuit $\mathcal{C} = (G, W)$ is composed of circuit elements including inputs I, outputs O, combinational gates C, and registers R (cf. Definition 2.1). Definition 4.1 specializes these circuits in the bit-precise manner as follows:

Definition 4.1 (Bit-Level Circuit). Let $\mathcal{C} = (G, W)$ be a circuit. We say that $\mathcal{C}$ is a *bit-level* circuit if each gate $g \in G$ outputs a single Boolean value, i.e., $g \colon \mathbb{B}^v \to \mathbb{B}$.

To reason about circuit execution in a bit-precise manner, we use sequences of circuit states, known as *execution traces*. Explicitly manipulating circuit states, rather than transition systems, facilitates access to bit-level circuit elements. Definition 4.2 extends the circuit state notation (cf. Definition 2.3) to include inputs, allowing all gate values to be deduced from a circuit state σ. Definition 4.3 then recalls the notion of execution trace.

Definition 4.2 (Circuit State). Let $\mathcal{C}$ be a bit-level circuit, $I = \{x_1, \ldots, x_{|I|}\} \subseteq G$ be its inputs, and $R = \{r_1, \ldots, r_{|R|}\} \subseteq G$ be its registers. The state of circuit $\mathcal{C}$ at clock cycle i is the value tuple $\sigma_i^{\mathcal{C}} = (\texttt{val}(x_1), \ldots, \texttt{val}(x_{|I|}), \texttt{val}(r_1), \ldots, \texttt{val}(r_{|R|}))$

containing its inputs and registers values at the given clock cycle. The set of possible circuit states is denoted by Σ.

Definition 4.3 (Circuit Execution Trace). Let $\mathcal{C}$ be a bit-level circuit. A *circuit execution trace* $(\sigma_i)_{i=1}^{n} \in \Sigma^n$ is a sequence of n circuit states $(\sigma_1, \ldots, \sigma_n)$ such that σ_1 is a valid initial state, and, for all $i < n$, σ_{i+1} is a valid next state of σ_i.

Based on Definitions 4.2 and 4.3, the value of every gate $g \in G$ in the current clock cycle i can be considered a function of the current circuit state σ_i, denoted as $g(\sigma_i)$. Assuming the gates G are topologically sorted, we define the notation $S(\sigma_i)$, with $S \subseteq G$ an arbitrary subset of gates, as the value tuple of all gates $g \in S$ in the state σ_i. As an example, this notation is used in this chapter to refer to the circuit's output values $O(\sigma_i)$ at state σ_i or to express equalities over output values between different circuit states, e.g., $O(\sigma_i) = O(\sigma_j)$.

Circuit Partitioning

The methodology we introduce in this chapter relies on special ways of partitioning circuits' registers. Definition 4.4 gives a general definition of a *circuit partitioning*.

Definition 4.4 (Circuit Partitioning). Let $\mathcal{C} = (G, W)$ be a circuit. We define a circuit partitioning $\mathcal{P} = \{P_j\}_{j=1}^{m}$ as a complete partitioning of R such that P_j are disjoint sets of registers, i.e., $P_j \subseteq R$, with $\forall j \neq j' : P_j \cap P_{j'} = \emptyset$ and $R = \bigcup_{j=1}^{m} P_j$. Furthermore, for two states σ and $\hat{\sigma}$, we write $\Delta_{\mathcal{P}}(\sigma, \hat{\sigma}) := |\{P \in \mathcal{P} \mid P(\sigma) \neq P(\hat{\sigma})\}|$ for the number of partitions in $\mathcal{P}$ that have different values between states σ and $\hat{\sigma}$.

Example 4.1. Figure 4.3 illustrates a simple circuit partitioning where r_1 and r_2 belongs to partition P_1 and r_3 to P_2. Moreover, assuming two states σ and $\hat{\sigma}$ where:

- $r_1(\sigma) = 0, \quad r_2(\sigma) = 1, \quad r_3(\sigma) = 0$
- $r_1(\hat{\sigma}) = 1, \quad r_2(\hat{\sigma}) = 1, \quad r_3(\hat{\sigma}) = 0$

we have $\Delta_{\mathcal{P}}(\sigma, \hat{\sigma}) = 1$ since $P_1(\sigma) \neq P_1(\hat{\sigma})$ and $P_2(\sigma) = P_2(\hat{\sigma})$.

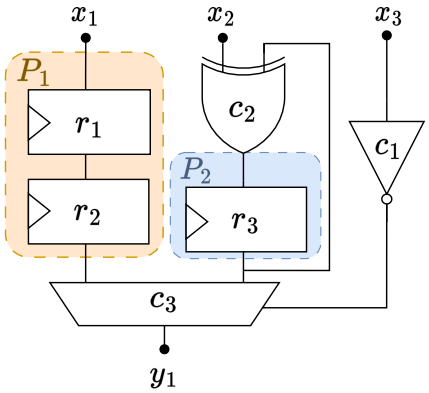


Figure 4.3: Circuit partitioning example with $\mathcal{P} = \{\{r_1, r_2\}, \{r_3\}\}$.

Fault Injection Attacks

The notations for fault injection attacks are consistent with the definitions introduced in Section 2.1. However, Definition 4.5 restricts fault models (cf. Definition 2.5) to bit-level fault effects.

Definition 4.5 (Bit-Level Fault Model). Let $\mathcal{C} = (G, W)$ be a bit-level circuit. A *bit-level transient fault model* for circuit $\mathcal{C}$ is characterized as a set of pairs $\mathcal{F} \subseteq G \times U$, with $U = \{\textit{bit-reset}\colon x \mapsto 0,\ \textit{bit-set}\colon x \mapsto 1,\ \textit{bit-flip}\colon x \mapsto \neg x\}$. Each fault $(g, e) \in \mathcal{F}$ describes a potential fault with the *fault location* $g \in G$ and the *fault effect* $e \in U$.

Faults in bit-level circuits are represented using unary functions U. The remainder of Chapter 4 only considers the *bit-flip* fault effect as it encompasses every other effect, such as *bit-set* and *bit-reset* faults. For instance, a *bit-reset* fault causes a gate to always output 0. If the expected output should have been 0, the fault has no consequences. Otherwise, it is akin to a bit-flip.

In addition, the definition of a transient fault attack $\mathbf{F}$ is identical to Definition 2.6. As a reminder, a *fault attack* $\mathbf{F} \subseteq \mathcal{F} \times [1, n]$ is a set of timed faults injected into the circuit $\mathcal{C}$ with an *attack order* of $|\mathbf{F}|$. For the purpose of notation in the remainder of this chapter, we write $\mathbf{F}_J = \{(g, e, j) \in \mathbf{F} \mid j \in J\}$ to denote the restriction of $\mathbf{F}$ to faults occurring in clock cycles $J \subseteq [1, n]$.

The notion of *fault-resistant partitioning*, later introduced in Section 4.3, requires a deep understanding of fault consequences and how they propagate in bit-level circuits. Example 4.2 details three types of fault consequences.

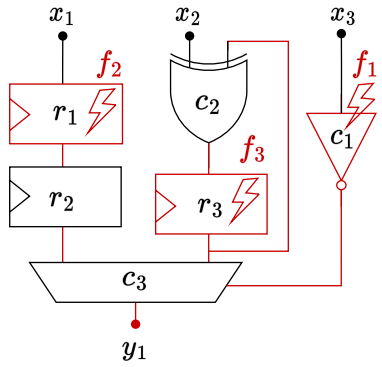


Figure 4.4: Fault propagation on a simple circuit.

Example 4.2. As shown in Figure 4.4, the fault $f_1 = (c_1, \textit{bit-flip}, j)$ has *immediate consequences*, i.e., within the same clock cycle, on the combinational gates c_1 and c_3, and the output y_1. The fault $f_2 = (r_1, \textit{bit-set}, j)$ on the register r_1 propagates through the circuit and has *delayed consequences* on r_2, c_3, and y_1 at clock cycle $j+1$. Fault f_2 can also have *no consequences* if the effect does not induce a different value, i.e., if a 1 is already stored in r_1 or if the mux c_3 never selects the output from r_2. Finally, $f_3 = (r_3, \textit{bit-flip}, j)$ illustrates a specific case of delayed consequences, where the fault can remain *hidden* in the register r_3 for an unknown amount of time without propagating to the output y_1, depending on the value of x_3 and c_1.

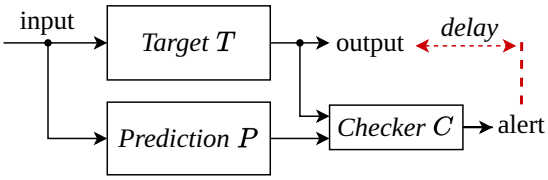


Figure 4.5: Concurrent Error Detection (CED) scheme.

4.2.3 Concurrent Error Detection Schemes

Concurrent Error Detection (CED) schemes attempt to protect a system against fault attacks using spatial redundancy [MM00]. Figure 4.5 depicts such a scheme where the *target* function T produces an output $T(x)$ for a given input x, while the *prediction* function P independently generates a predicted characteristic of the output based on the input x, and the *checker* function compares the outputs and raises an *alert* signal on a mismatch. In its simplest form, the prediction circuit is a duplication of the target, and the checker simply compares them for equality. Alternatively, P can also be implemented with error detection codes [BBK+03, AMR+20]. Some implementations also introduce a *delay* between the operation of the target and the prediction functions [VM02, MP23]. This delay has the advantage of increasing the practical difficulty of faulting both functions but introduces a circuit area overhead due to buffering. Definition 4.6 formalizes CED schemes.

Definition 4.6 ((d, A)-CED)**.** A circuit $\mathcal{C} = (G, W)$ with outputs O implements a (d, A)-CED when its outputs are divided into alert signals $A \subseteq O$ with associated delay of d clock cycles and primary outputs $O' = O \setminus A$. Without loss of generality, we say that $\mathcal{C}$ raises an alert at clock cycle i if $A(\sigma_i) \neq (0, \ldots, 0)$, which we will write $A(\sigma_i) \neq 0$ for brevity.

As noted in Section 4.1, existing definitions of circuit security against fault attacks focus only on combinational circuits. The concept of *fault-secure* CED was introduced by Siewiorek et al. in 1998 [SS98] to ensure system reliability against external perturbations. This notion was adapted for security by Dhooghe and Nikova in 2020 [DN20]. However, these definitions are too restrictive as they do not consider faults that remain in the circuit, modifying the outputs over multiple clock cycles or after an indeterminate amount of time, as illustrated in Example 4.2. Additionally, the CED countermeasures we study implement delayed detection, as formalized in Definition 4.6. Definition 4.7 extends the previous definitions to formalize *k-fault security*, considering potential detection delays.

Definition 4.7 (k-fault secure (d, A)-CED)**.** Let $\mathcal{C} = (G, W)$ be a circuit implementing a (d, A)-CED, $(\sigma_i)_{i=1}^{n+d}$ be an arbitrary execution trace of length $n + d$, $\mathcal{F}$ be a fault model and k be the attack order. We say that the (d, A)-CED is *k-fault secure*

against the fault model $\mathcal{F}$ if and only if, $\forall n \in \mathbb{N}^*$,

$$\begin{aligned} &\forall \left(\sigma_i\right)_{i=1}^{n+d},\ \forall \mathbf{F} \subseteq \mathcal{F} \times [1, n+d],\ |\mathbf{F}| \leq k : \\ &\left(\bigwedge_{i=1}^{n+d} A\left(\sigma_i^{\mathbf{F}[1,i]}\right) = 0\right) \Longrightarrow \left(\bigwedge_{i=1}^{n} O'\left(\sigma_i\right) = O'\left(\sigma_i^{\mathbf{F}[1,i]}\right)\right). \end{aligned} \tag{4.1}$$

Intuitively, Definition 4.7 says that k-fault security against fault model $\mathcal{F}$ guarantees that whenever there are no alerts in the first $n+d$ clock cycles, the primary outputs are correct up to clock cycle n. Since this must hold for all executions of arbitrary length, we can infer that an alert is raised at most d cycles after a corrupted primary output. A delay $d = 0$ implies an immediate detection, whereas $d = 2$ means the alert is raised up to two cycles after the corrupted output.

4.2.4 Hardware Equivalence Checking

Equivalence Checking (EC) formally ensures whether two design specifications are functionally equivalent [LMMS17, §I.4]. EC found a wide field of application in the electronic design automation world to verify the absence of functional discrepancies introduced by the tooling during the different conception-flow steps. Today, formal equivalence checking is a key component of the overall chip validation to guarantee the correspondence between different levels of specification—from higher levels like the instruction set architecture, mid-range levels such as the register-transfer-level (RTL) implementation, or lower levels like the post-synthesis gate-level netlist. Commercial tools from Mentor, Cadence, or Synopsys heavily rely on these techniques, and so does Yosys through the verification tool ABC [BM10] or using its in-house tool EQY [Yos21].

The following paragraphs first introduce the general concept of equivalence checking before describing its applications in the context of fault attacks. Finally, we expose its limitations when applied to general-purpose circuits like CPUs.

Classic Equivalence Checking

In essence, two synchronous designs are said to be functionally equivalent if, at each clock cycle, they produce exactly the same output values for any valid sequence of input values.

Let $\mathcal{M}_1 = (X_1, Y_1, S_1, S_{0,1}, \delta_1, \lambda_1)$ and $\mathcal{M}_2 = (X_2, Y_2, S_2, S_{0,2}, \delta_2, \lambda_2)$ be two Mealy machines. In the sequel, we assume the following simplifications. First, we restrict the comparison between two machines that have exactly the same inputs and outputs, i.e., $X_1 = X_2$ and $Y_1 = Y_2$. We denote the product Mealy machine

$\mathcal{M} = (X, Y, S, S_0, \delta, \lambda) = \mathcal{M}_1 \times \mathcal{M}_2$ as follows:

$$
\begin{aligned}
X &= X_1 = X_2 \\
O &= \{0, 1\} \\
S &= S_1 \times S_2 \\
S_0 &= S_{0,1} \times S_{0,2}
\end{aligned}
\qquad
\begin{aligned}
\delta(x, (s_1, s_2)) &= (\delta_1(x, s_1), \delta_2(x, s_2)) \\
\lambda(x, (s_1, s_2)) &= \begin{cases} 1 & \text{if } \lambda_1(x, s_1) = \lambda_2(x, s_2) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

A schematic view of the product Mealy machine is given in Figure 4.6. We say that the two machines are functionally equivalent if, and only if, the output function λ produces 1 for all reachable states.

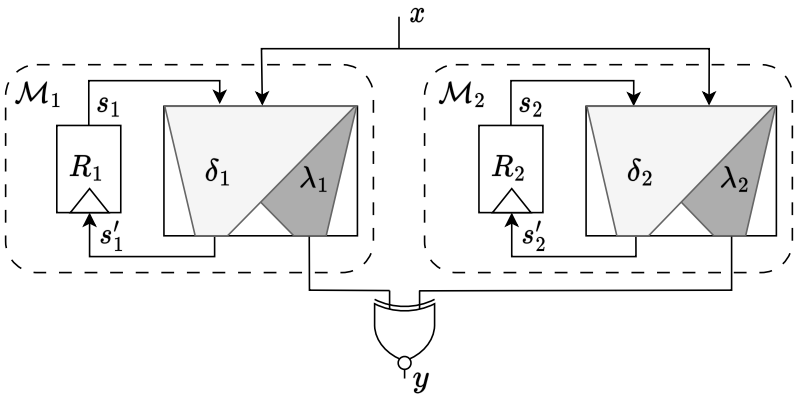


Figure 4.6: Product Mealy machine for comparing two finite-state machines.

Two main approaches are classically employed to compute the equivalence between two machines: combinational or sequential. *Combinational equivalence checking*, on the one hand, exploits a one-to-one register correspondence between $\mathcal{M}_1$ and $\mathcal{M}_2$. If such a correspondence can be found, proving the equivalence between $\mathcal{M}_1$ and $\mathcal{M}_2$ can be reduced to show that δ_1 and δ_2 are equivalent as well as λ_1 and λ_2. On the other hand, *sequential equivalence checking* addresses the general case where no register correspondence can be identified, i.e., check if the outputs are equal for the same inputs in all the reachable states. However, this second approach is more expensive as it requires unrolling the circuit to compute all the reachable states. As a result, classical symbolic reachability analysis (cf. Algorithm 2.1 in Section 2.2) can be applied to the product Mealy machine $\mathcal{M}$ with the reachability property φ defined as follows:

$$\varphi \colon s \mapsto \exists x \colon \lambda_1(x, s) \neq \lambda_2(x, s)$$

Finally, both approaches rely on classical Boolean decision procedures such as BDD and SAT solving, as introduced in Section 2.2.3.

Fault Equivalence Checking

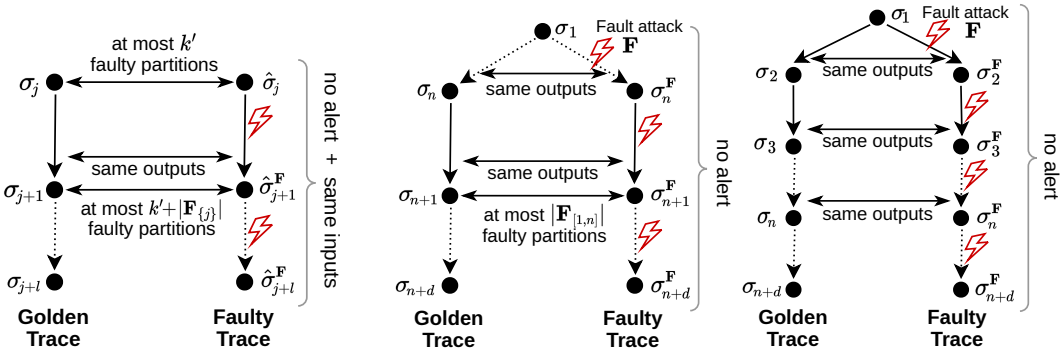
Given a fault model $\mathcal{F}$, proving that a circuit $\mathcal{C}$ is k-fault secure against $\mathcal{F}$ amounts to demonstrate that the fault-free transition system $\mathcal{M}$ and its faulty version $\mathcal{M}^{\mathbf{F}}$

are equivalent, i.e., always produce the same primary outputs assuming there is no alert. At first, one might want to apply *combinational* equivalence checking as the two circuits are structurally equivalent and agree on each of their registers. However, this verification is too fine-grained and fails as soon as a register memorizes a faulty value, i.e., the fault-free next-state function δ and its faulty version $\delta^{\mathbf{F}}$ are not equivalent. Such a situation usually arises in processors where registers, like pipeline stages or the register file, memorize values at each clock cycle. Additionally, countermeasures implementing a detection delay necessarily need to be evaluated over multiple clock cycles. Consequently, *sequential* equivalence checking must be used to observe the fault propagation over multiple clock cycles.

Figure 4.7c illustrates sequential equivalence checking for fault analysis as implemented in state-of-the-art tools like FIVER [RRSS+21] and SYNFI [NOV+22]. In essence, the analysis compares a golden trace $(\sigma_i)_{i=1}^{n+d}$ on the left side of Figure 4.7c with a faulty trace $\left(\sigma_i^{\mathbf{F}}\right)_{i=1}^{n+d}$ on the right. Both traces start from the same initial state σ_1 but diverge due to the fault attack $\mathbf{F}$. k-fault security is ensured by verifying that both traces produce the same outputs for each pair of states σ_i and $\sigma_i^{\mathbf{F}}$, assuming no alert is raised. The other figures, **(a)** and **(b)**, shown in Figure 4.7, are described in the next section.

However, as discussed in Section 4.1 and illustrated in Figure 4.4, the duration of fault propagation is not always known *a priori*, making it challenging to find a bound n because faults can remain hidden in the circuit indefinitely. Consequently, bounded techniques cannot prove k-fault security in the general case and may struggle as the checking complexity increases with n.

Unbounded security guarantees are indeed necessary to correctly use the results of the preliminary hardware analysis in the co-verification step. This issue is solved in the next section.



(a) k-fault-resistant partitioning. **(b)** Inductive property. **(c)** k-fault secure CED.

Figure 4.7: Overview of different properties a circuit implementing CED can fulfill, where **(a)** is the strongest property that implies **(b)**, which in turn implies **(c)**.

4.3 Fault-Resistant Partitioning

Section 4.1 has introduced our enhanced methodology based on a preliminary analysis of the hardware. It has also highlighted the lack of verification techniques offering sufficient guarantees to abstract faults during the co-verification step. Section 4.2 provided the necessary background for modeling bit-level circuit execution with faults, defined the k-fault security property that such circuits must fulfill, and detailed the limitations of equivalence checking with faults.

This section is the core of the methodology introduced in this chapter. It introduces the notion of *k-fault-resistant partitioning* to provide unbounded security guarantees, thus addressing the limitations of equivalence checking. First, we describe the notion of k-fault-resistant partitioning and present its underlying intuition. Next, we provide the formal definition of k-fault-resistant partitioning and prove that it implies k-fault security. Finally, we describe an algorithm that automatically identifies such a partitioning and proves its k-fault resistance.

4.3.1 Intuition

As discussed in Section 4.2.4, directly proving that a circuit implementing a CED fault countermeasure provides k-fault security by means of equivalence checking is not always feasible. As depicted in Figure 4.7c, direct bounded proofs must unroll both the golden and faulty versions of the circuit an *a priori* unknown number of times, until symbolically visiting all reachable states. Considering that transient faults can often linger within the state of the circuit indefinitely, this methodology quickly becomes intractable and a completeness threshold is never reached. In the following, we define such a property called *k-fault-resistant partitioning* and prove it guarantees the k-fault security of a (d, A)-CED.

Intuition. Before providing the formal definition of a *k-fault-resistant partitioning*, this paragraph gives the underlying intuition with $k = 1$. Let $\mathcal{C}$ be a circuit implementing a (d, A)-CED as illustrated on Figure 4.8. As such, the circuit $\mathcal{C}$ has a set of primary outputs $O = \{y_1\}$, and a set of alert signals $A = \{alert\}$. This CED implements an immediate detection, i.e., $d = 0$. Let $\mathcal{P} = \{P_1, P_2\}$ be an arbitrary circuit partitioning with $P_1 = \{r_1, r_2\}$ and $P_2 = \{r_3, r_4\}$. We remind that circuit partitioning applies to circuit registers only, as defined in Definition 4.4.

We say that a partitioning $\mathcal{P}$ is a k-fault-resistant partitioning under two conditions. First, the *fault confinement* property ensures that incorrect register values are either detected or confined in partitions for any set of k faults injected in the circuit. In other words, it means that a single fault injected in one partition cannot propagate to other partitions without being detected. However, the consequence of a fault can freely propagate within a partition without further consequences. As illustrated in Figure 4.8, with $k = 1$, a single fault injected in g_2 (resp. g_6) can propagate to r_1 and r_2 (resp. r_3 and r_4), but cannot modify r_1 and r_3 at the same time. As a

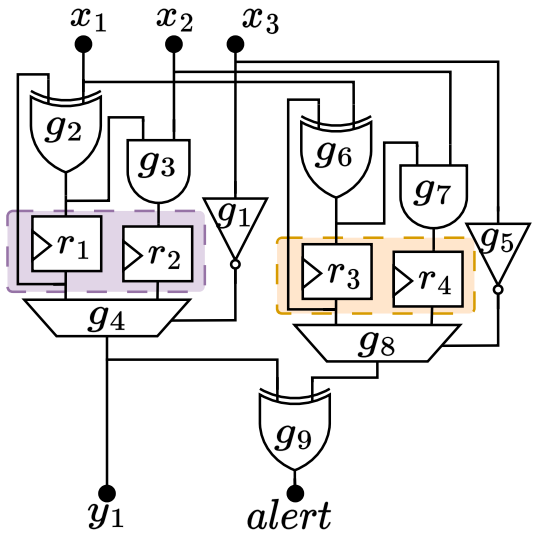


Figure 4.8: Fault-resistant partitioning on a circuit implementing duplication.

result, the partitioning $\mathcal{P}$ fulfills the *fault confinement* property. Note that multiple partitions may exist in a circuit because of circuit structure, i.e., the partitions are not connected with wires, or because of semantical reasons, i.e., the propagation of the fault is stopped by a checker mechanism that raises an alert. Fault confinement implies that the injection of k faults cannot corrupt more than k partitions without being detected. Therefore, a k-fault-resistant partitioning necessarily has at least $k+1$ partitions to ensure fault detection at attack order k.

The second property that $\mathcal{P}$ must ensure is the *output integrity*. Assuming there is no alert, circuit outputs must be correct under any set of k faults targeting either combinational gates or partitions of registers. As illustrated in Figure 4.8, any single fault modifying the output y_1 will automatically trigger the alert signals. In the sequel, Definition 4.8 formally defines the k-fault-resistant partitioning concept.

4.3.2 Formal Definition

Definition 4.8 (k-Fault-Resistant Partitioning). Let $\mathcal{C} = (G, W)$ be a circuit implementing a (d, A)-CED. Let $j \in \mathbb{N}^*$ be an arbitrary offset and let $(\sigma_i)_{i=j}^{j+l}$ and $(\hat{\sigma}_i)_{i=j}^{j+l}$ be two arbitrary execution traces of length $l+1$, where $l = \max(1, d)$. Finally, let $\mathcal{P}$ be a partitioning of the circuit $\mathcal{C}$, $\mathcal{F}$ be a fault model, and let $k \in \mathbb{N}^*$ be an attack order. We say that $\mathcal{P}$ is a *k-fault-resistant partitioning* of $\mathcal{C}$ against the fault model $\mathcal{F}$ if and only if

$$\begin{aligned}
&\forall (\sigma_i)_{i=j}^{j+l},\ (\hat{\sigma}_i)_{i=j}^{j+l},\ \mathbf{F} \subseteq \mathcal{F} \times [j, j+d],\ k' \in \mathbb{N},\ |\mathbf{F}| + k' \leq k : \\
&\left(\bigwedge_{i=j}^{j+d} I(\sigma_i) = I(\hat{\sigma}_i) \right) \wedge \left(\Delta_{\mathcal{P}}(\sigma_j, \hat{\sigma}_j) \leq k' \right) \wedge \left(\bigwedge_{i=j}^{j+d} A\left(\hat{\sigma}_i^{\mathbf{F}_{[j,i]}}\right) = 0 \right) \implies \\
&\left(\Delta_{\mathcal{P}}\left(\sigma_{j+1}, \hat{\sigma}_{j+1}^{\mathbf{F}_{\{j\}}}\right) \leq k' + \left|\mathbf{F}_{\{j\}}\right| \right) \wedge \left(O'(\sigma_j) = O'\left(\hat{\sigma}_j^{\mathbf{F}_{\{j\}}}\right) \right).
\end{aligned} \tag{4.2}$$

Similar to proving the k-fault security with equivalence checking, Definition 4.8 also considers two execution traces $(\sigma_i)_{i=j}^{j+l}$ and $(\hat{\sigma}_i)_{i=j}^{j+l}$ where the former is the reference trace and a fault attack targets the latter. In Equation (4.2), the left-hand side of the implication can be considered as an assumption under which the design must guarantee that the right-hand side holds. First, it is assumed that both execution traces have the same inputs, their initial states σ_j and $\hat{\sigma}_j$ differ in at most k' partitions at clock cycle j, and no alerts are triggered in the faulty trace $\left(\hat{\sigma}_i^{\mathbf{F}}\right)_{i=j}^{j+d}$. Intuitively, this situation represents two execution traces of circuit $\mathcal{C}$, depicted in Figure 4.7a, processing the same inputs but where at most k' partitions have a different state due to faults injected before clock cycle j. In addition, we consider a fault attack $\mathbf{F}$ with an attack order $|\mathbf{F}| \leq k - k'$ modifying execution trace $(\hat{\sigma}_i)_{i=j}^{j+l}$ between clock cycles j and $j+d$ but without triggering any alert signal. The right-hand side of the implication in Equation (4.2) specifies the two characteristics a k-fault-resistant partitioning must fulfill. First, the number of newly corrupted partitions is less than or equal to $\left|\mathbf{F}_{\{j\}}\right|$ which is equal to the number of faults introduced by the fault attack $\mathbf{F}$ at clock cycle j (*k-fault confinement*). Newly corrupted partitions are evaluated after one transition, i.e., at clock cycle $j+1$, since faults have delayed consequences on registers. Second, the circuit's primary outputs must be identical at clock cycle j between the two execution traces since faults have immediate consequences on the outputs (*output integrity*).

Theorem 4.1 states that a circuit with a k-fault-resistant partitioning is necessarily also k-fault secure.

Theorem 4.1 (k-fault-resistant partitioning implies k-fault security)**.** Let $\mathcal{C} = (G, W)$ be a circuit implementing a (d, A)-CED and let $\mathcal{F}$ be a fault model targeting the circuit. If there exists a *k-fault-resistant partitioning* $\mathcal{P}$ against $\mathcal{F}$ then $\mathcal{C}$ is k-fault secure.

Proof. To prove that Definition 4.8 (Figure 4.7a) implies Definition 4.7 (Figure 4.7c), we first show that it implies a stronger inductive property (Figure 4.7b), which in turn, implies Definition 4.7. Figure 4.7 shows the proof intuition, where k' faulty partitions in the initial state $\hat{\sigma}_i$ of (4.7a) correspond to k' faults injected during the previous clock cycles of the same execution trace $\left(\sigma_i^{\mathbf{F}}\right)_{i=1}^{n}$ in (4.7b). The complete proof is given in Appendix A. □

Theorem 4.1 provides a new strategy to prove the k-fault security of a (d, A)-CED circuit, giving unbounded guarantees on the fault attack consequences. Although k-fault-resistant partitioning is only a sufficient condition for k-fault security, it significantly simplifies the endeavor of the proof since the circuit is only unrolled $max(1, d)$ times, compared to the bounded equivalence checking approach. In converse, a k-fault secure circuit may not fulfill the k-fault-resistant partitioning property. Our approach is not sufficient to highlight genuine vulnerabilities, and counterexamples require further analysis as false positives exist. The following section provides an algorithm to build such a k-fault-resistant partitioning.

Algorithm 4.1: Build and prove a k-fault-resistant partitioning of circuit $\mathcal{C}$.

Input: a circuit $\mathcal{C} = (G, W)$ implementing a (d, A)-CED, a fault model $\mathcal{F} \subseteq G \times u$, an attack order k, and an initial partitioning $\mathcal{P}$

Output: On success, returns a k-fault-resistant partitioning $\mathcal{P} = \{P_j\}_{j=1}^{m}$, a set of exploitable faults $\mathcal{F}' \subseteq \mathcal{F}$, and a set of exploitable partitions $\mathcal{P}' \subseteq \mathcal{P}$.

1 Create symbolic executions $(\sigma_i)_{i=1}^{l+1}$ and $(\hat{\sigma}_i)_{i=1}^{l+1}$, with $l = \max(1, d)$;
2 Create symbolic fault attack $\mathbf{F} \subseteq \mathcal{F} \times [1, d+1]$;
3 $\psi_{\text{InsEqAndNoAlert}} \leftarrow \left(\bigwedge_{i=1}^{d+1} I(\sigma_i) = I(\hat{\sigma}_i)\right) \wedge \left(\bigwedge_{i=1}^{d+1} A\left(\hat{\sigma}_i^{\mathbf{F}}\right) = 0\right)$;
4 **Procedure** *BuildPartitioning*: ▷ `Build k-fault-confining partitioning`
5 **for** k' **from** 0 **to** k **do**
6 $\psi_{\text{MoreInfected}} \leftarrow (\Delta_{\mathcal{P}}(\sigma_1, \hat{\sigma}_1) \leq k') \wedge (|\mathbf{F}| \leq k - k') \wedge \left(\Delta_{\mathcal{P}}(\sigma_2, \hat{\sigma}_2) > k' + \left|\mathbf{F}_{\{1\}}\right|\right)$;
7 **while** $(\psi_{InsEqAndNoAlert} \wedge \psi_{MoreInfected})$ *is SAT* **do**
8 $\mathcal{P}_{\text{Init}} \leftarrow \{P \in \mathcal{P} \mid P(\sigma_1) \neq P(\hat{\sigma}_1)\}$;
9 $\mathcal{P}_{\text{Next}} \leftarrow \left\{P \in \mathcal{P} \mid P(\sigma_2) \neq P\left(\hat{\sigma}_2^{\mathbf{F}}\right)\right\}$;
10 $\mathcal{P} \leftarrow \text{merge}(\mathcal{P}, \mathcal{P}_{\text{Init}}, \mathcal{P}_{\text{Next}})$;
11 **if** $|\mathcal{P}| \leq k$ **then return failure**;
12 $\mathcal{F}' \leftarrow \{\}$, $\mathcal{P}' \leftarrow \{\}$;
13 **Procedure** *CheckIntegrity*: ▷ `Find faults that compromise outputs`
14 **for** k' **from** 0 **to** k **do**
15 $\psi_{\text{NoFaultsOnForbidden}} \leftarrow \left(\bigwedge_{P \in \mathcal{P}'} P(\sigma_1) = P(\hat{\sigma}_1)\right) \wedge (\mathbf{F} \cap \mathcal{F}' \times [1, d+1] = \emptyset)$;
16 $\psi_{\text{OutsBad}} \leftarrow (\Delta_{\mathcal{P}}(\sigma_1, \hat{\sigma}_1) \leq k') \wedge (|\mathbf{F}| \leq k - k') \wedge \left(O'(\sigma_1) \neq O'\left(\hat{\sigma}_1^{\mathbf{F}}\right)\right)$;
17 **while** $(\psi_{InsEqAndNoAlert} \wedge \psi_{NoFaultsOnForbidden} \wedge \psi_{OutsBad})$ *is SAT* **do**
18 $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{P \in \mathcal{P} \mid P(\sigma_1) \neq P(\hat{\sigma}_1)\}$;
19 $\mathcal{F}' \leftarrow \mathcal{F}' \cup \{(g, e) \in G \times u \mid \exists j, (g, e, j) \in \mathbf{F}\}$;
20 **return** $(\mathcal{P}, \mathcal{P}', \mathcal{F}')$;

4.3.3 Algorithm to Identify a Fault-Resistant Partitioning

Overview. Algorithm 4.1 describes a process to identify a circuit partitioning $\mathcal{P}$ resistant to k fault injections using SAT solving. It takes as input a circuit model $\mathcal{C}$, a fault model $\mathcal{F}$, an attack order k, and an initial partitioning $\mathcal{P}$. Algorithm 4.1 comprises two main procedures, *BuildPartitioning* and *CheckIntegrity*, where the former builds a partitioning ensuring k-fault confinement or detection, and the latter finds all remaining fault injections compromising the integrity of outputs. Eventually, the algorithm either returns a k-fault-resistant partitioning $\mathcal{P}$ with a set of assumptions under which the circuit is k-fault secure or fails to find such a partitioning and provides counterexamples detailing what happened.

Build Partitioning. The initial partitioning can be chosen freely, but for an initial run of the algorithm, it should be chosen as $\mathcal{P} = \{\{r\} \mid r \in R\}$, where each register $r \in R$ belongs to a separate partition. In subsequent runs of the algorithm, one can set the partitioning $\mathcal{P}$ to one that was previously computed with a different value of k, for instance. At the start, the algorithm first creates two symbolic execution traces $(\sigma_i)_{i=1}^{l+1}$ and $(\hat{\sigma}_i)_{i=1}^{l+1}$ of length $l+1$, with $l = \max(1, d)$, and a symbolic fault

attack $\mathbf{F}$ that describes all possible faults an attacker can induce.

Procedure *BuildPartitioning* iteratively analyzes whether the current partitioning $\mathcal{P}$ guarantees that, whenever k' partitions are compromised and there are $\left|\mathbf{F}_{\{1\}}\right|$ new faults in the first clock cycle, there are at most $k' + \left|\mathbf{F}_{\{1\}}\right|$ compromised partitions in the second clock cycle (line 6). It does this by iterating through all combinations of k' and $|\mathbf{F}| = k - k'$ (line 5) and asking a SAT solver whether there are execution traces $(\sigma_i)_{i=1}^{l+1}$ and $(\hat{\sigma}_i)_{i=1}^{l+1}$ as well as a concrete fault attack $\mathbf{F}$ where the left-hand side of (4.2) is true but the first part of the right-hand side, i.e., $\left(\Delta_{\mathcal{P}}\left(\sigma_2, \hat{\sigma}_2\right) \leq k' + \left|\mathbf{F}_{\{1\}}\right|\right)$, is false (line 7). If the SAT solver finds such an example, i.e., the formula is SAT, the procedure gathers the compromised partitions $\mathcal{P}_{\text{Init}}$ and $\mathcal{P}_{\text{Next}}$ from respectively the first and second clock cycle, and then merges partitions from $\mathcal{P}_{\text{Next}}$ until there are only $k' + \left|\mathbf{F}_{\{1\}}\right|$ left, while avoiding merges between the partitions also present in $\mathcal{P}_{\text{Init}}$ (lines 8 to 10). As $|\mathcal{P}|$ decreases at each iteration, procedure *BuildPartitioning* converges to a fixed point where partitioning $\mathcal{P}$ fulfills the relevant part of (4.2), and the solver must return UNSAT. After all k' are analyzed, the procedure concludes.

After *BuildPartitioning* finishes, the algorithm checks whether the partitioning has less than k partitions (line 11), as such a partitioning cannot guarantee the output integrity for the second procedure. Procedure *BuildPartitioning* may fail for one of the three following reasons:

(i) the circuit $\mathcal{C}$ has some flaws and is not k-fault secure,

(ii) there is no k-resistant partitioning even if $\mathcal{C}$ is k-fault secure (cf. Section 4.3.2),

(iii) the merging heuristics fails to build a k-resistant partitioning even though one exists.

We generate log files for each counterexample given by the solver and the corresponding partition merges the algorithm performs. In general, the logs are invaluable for understanding why a design might be insecure, but this analysis must be performed manually.

For higher-order fault analysis, one should start with $k = 1$ and iteratively feed the found partitioning $\mathcal{P}$ back into the algorithm for the next higher k as a k-fault resistant partitioning must be $(k-1)$-fault resistant.

Check Output Integrity. Procedure *CheckIntegrity* iteratively determines the sets $\mathcal{P}'$ and $\mathcal{F}'$ of partitions and locations where faults can compromise the output integrity. The procedure iteratively verifies if the partitioning $\mathcal{P}$ guarantees outputs' integrity in the presence of k' faulty partitions and $k - k'$ new faults while not targeting the known-to-be exploitable partitions $\mathcal{P}'$ and fault locations $\mathcal{F}'$ identified in previous iterations (lines 15 and 16). Whenever the solver returns SAT, it means that it found a new set of fault locations and initially corrupted partitions that compromises output integrity and must be added to $\mathcal{P}'$ and $\mathcal{F}'$, respectively (lines 17

to 19). If the solver returns UNSAT instead, it means that the partitioning $\mathcal{P}$ is proven k-fault secure assuming there are no faults on the $\mathcal{P}'$ and $\mathcal{F}'$.

Optimizations. The number of Boolean variables in the SAT queries, and consequently the complexity of these queries, directly depends on the size of the circuit and the number of faults to consider. In the following, we propose optimizations based on a structural analysis of the circuit to remove unnecessary faults. As a result, we simplify the SAT query provided to the solver and speed up the proof. Let us start by understanding the three possible consequences of a fault on the k-fault-resistant partitioning property: *i)* the fault can infect more partitions at state σ_2 ($\psi_{\text{MoreInfected}}$, line 6), *ii)* the fault can modify primary outputs at state σ_1 (ψ_{OutsBad}, line 16), or *iii)* the fault can disable the alert signals ($\psi_{\text{InsEqAndNoAlert}}$, line 3).

i) In order to infect more partitions at state σ_2, faults must be injected during the initial state σ_1, as they have delayed consequences on registers. Additionally, gates or partitions where the faults are injected must be combinationally connected to at least two partitions. Faults not fulfilling these conditions do not impact the satisfiability of formula $\psi_{\text{MoreInfected}}$.

ii) In order to modify primary outputs at state σ_1, faults must be injected at the initial state σ_1, as they have immediate consequences on outputs. Additionally, the gates (resp. partitions) where the faults are injected must be combinationally connected to primary outputs. Faults not fulfilling these conditions do not impact the satisfiability of formula ψ_{OutsBad}.

iii) In order to disable the alert signal between states σ_1 and σ_{1+d}, faults can be injected in gates in each one of these clock cycles, as they have immediate or delayed consequences on the alert signal. However, gate locations of the fault have to be structurally connected to the alert signal, and the propagation delay to the alert signal must be less than d. Otherwise, the fault has no impact on the formula $\psi_{\text{InsEqAndNoAlert}}$.

As a result, faults considered during procedure *BuildPartitioning* have to satisfy conditions *i)* and *iii)*. Other faults are optimized away to simplify the SAT query. For example, combinational gates connected to one single partition no longer need to be faulted. Indeed, the consequences obtained by faulting these gates are totally subsumed by considering a symbolic faulty state of the partitions. Similarly, faults during *CheckIntegrity* have to satisfy conditions *ii)* and *iii)*.

4.4 Implementation

Section 4.3 has introduced the theoretical concept of fault-resistant partitioning, which is the cornerstone of the methodology proposed in this chapter. To demonstrate its practical application in evaluating different case studies, this section presents the implementation of the methodology.

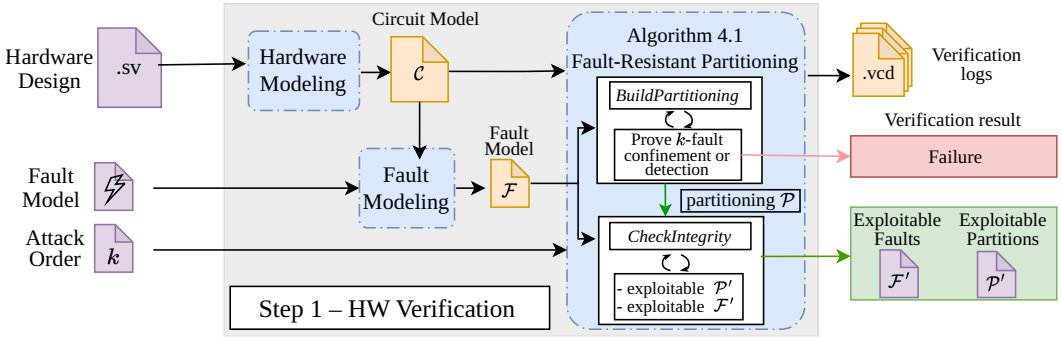


Figure 4.9: Workflow implementing fault-resistant partitioning (Step 1).

First, we detail the *hardware verification flow* that implements Algorithm 4.1. Next, we describe the *system verification flow* implementation we use to evaluate the OpenTitan secure element.

4.4.1 Hardware Verification Flow

As illustrated in Figure 4.9, the design is first converted into a bit-level netlist using the synthesis tool Yosys [Wolb] to produce a circuit model $\mathcal{C}$ according to Definition 2.1. Additionally, input, output and alert signals must be provided by the user to define the CED circuit to be analyzed. The fault model $\mathcal{F}$, which specifies the exact locations of faults, is then derived from the circuit model $\mathcal{C}$. In this work, we only focus on bit-flip fault effects as it encompasses every other bit-level effect.

We then rely on the C++ API of the CaDiCaL SAT solver [BFFH20] for the formal analysis described in Algorithm 4.1 and shown in the *Fault-Resistant Partitioning* box in Figure 4.9. Circuit elements are encoded with Boolean variables and execution traces $(\sigma_i)_{i=1}^{l+1}$ and $(\hat{\sigma}_i)_{i=1}^{l+1}$ are modeled unrolling the circuit $l = \max(1, d)$ times. Fault injections are applied to execution traces using new Boolean variables to control the effect of faults. For each procedure, *BuildPartitioning* and *CheckIntegrity*, assumptions ψ made by Algorithm 4.1 are provided to the SAT solver to check their satisfiability. CaDiCaL is used in incremental mode to update assumptions during subsequent iterations of the procedures. Log files and VCD waveforms are generated to keep track of successive iterations, understand how the algorithm builds the circuit partitioning, and analyze why the proof may fail. The implementation is about 4000 lines of C++ code and is publically available[2].

4.4.2 System Co-verification using Verilator

The whole chapter is motivated by the security proof of the OpenTitan secure element running the first stage of its secure boot. However, such a verification requires

[2] https://github.com/CEA-LIST/Fault-Resistant-Partitioning

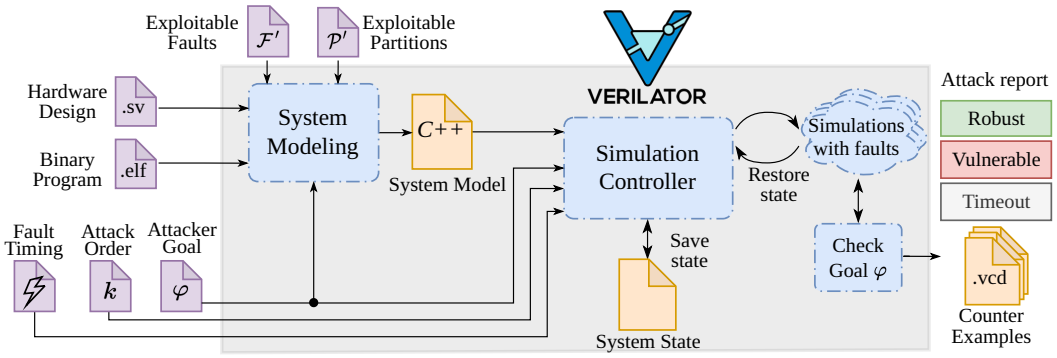


Figure 4.10: Hardware/software co-verification flow using Verilator (Step 2).

emulating the entire OpenTitan chip, as introduced in Section 4.2.1, because hardware accelerators like OTBN are involved in computing cryptographic signatures. Modeling these components within the µArchiFI infrastructure is currently impracticable. To address this limitation, we utilized the Verilator simulation environment provided by the OpenTitan project to perform fault injections. This Verilator-based co-verification framework serves as a proof of concept to validate our methodology.

In the following, we first explain how the exploitable faults and partitions identified during the fault-resistant partitioning can be converted into a unique reduced fault model. Next, we describe the implementation of the co-verification workflow.

Reduced Fault Model. The set of exploitable faults $\mathcal{F}'$ and partitions $\mathcal{P}'$ needs to be translated into a unique fault model $\mathcal{G}$, as defined in Definition 2.5, before being provided to the co-verification step. This operation is done as follows:

$$\mathcal{G} = \left\{ (g, e) \in G \times E \;\middle|\; (g, e) \in \mathcal{F}' \vee g \in \bigcup_{P \in \mathcal{P}'} P \right\}$$

That is, $\mathcal{G}$ introduces a new fault model where the faults are either in the set of exploitable faults $\mathcal{F}'$ or the gates are a register that belongs to the set of exploitable partitions $\mathcal{P}'$. As a result, integrating the results of the k-fault-resistant partitioning analysis into the co-verification flow can significantly reduce the size of the initial fault model $\mathcal{F}$, assuming the hardware countermeasures are proven robust. In other words, we are only interested in analyzing faults that are not detected by the hardware and can potentially cause incorrect system behavior.

Co-verification workflow Figure 4.10 illustrates our simulation-based co-verification framework. First, the *system modeling* step relies on the open-source tool Verilator [Sny] to convert the hardware design and the binary program into a cycle-accurate C++ model. The system modeling also takes as input the sets of exploitable faults $\mathcal{F}'$ and partitions $\mathcal{P}'$ computed in Step 1 to determine the remaining fault locations in gates and registers. Verilator optimizes the generated model for simulation performance reasons, and the effectiveness of optimizations depends on the number of fault locations. Then, the *simulation controller* exhaustively simu-

lates the circuit with a maximum of k faults. The fault timing specifies the cycles where the faults must be injected during the simulation. The predicate φ is evaluated on the system state at each clock cycle to determine if the attacker can reach its goal. Simulations can be parallelized on multiple threads. Finally, the framework provides an attack report for each fault attack evaluated. The report classifies the attack between i) *robust*, i.e., the fault attack does not fulfill φ and the simulation terminates as expected, ii) *vulnerable*, i.e., φ has been reached, and iii) *timeout*, i.e., neither the attacker goal nor the normal program exit point has been reached, and the simulation stops after a timeout. The timeout is computed according to the program length. Verilator generates logs such as the ISA states or VCD waveforms to understand where the faults were injected and how they propagate in the system to create the vulnerability.

To speed up the analysis, we adapted a simulator feature to save the system state in a file. The state is restored for each new verification, which avoids simulating irrelevant parts of the program for the fault analysis. In addition, we used the Verification Procedural Interface (VPI), supported by Verilator, to observe the circuit state and compute φ or to inject faults on circuit elements retrieved according to their hierarchical names.

This Verilator-based co-verification framework offers the advantage of easily including the behavior of peripherals that can be simulated simultaneously with the CPU. These peripherals include interconnects, memories, or hardware accelerators for which building a formal model would have been neither relevant nor scalable. However, it also has the same limitations as simulation-based analysis tools: it is not exhaustive on program inputs and it does not provide security guarantees in the general case. In addition, a timeout is needed to stop the simulation when the control flow has been modified by the attack but without reaching the attacker goal.

4.5 Validation on Impeccable Circuits

This section validates our methodology against prior work on formal verification of CED schemes. We evaluate the robustness of Skinny-64 and AES-128 implementations from Impeccable Circuits [AMR+20] protected with code-based CEDs against faults attacks. Although providing unbounded guarantees on cryptographic circuits is not as crucial as on a CPU, i.e., their operation usually takes a few clock cycles, these case studies allow us to compare against related work, e.g., FIVER [RRSS+21], as no similar work exists on CPUs. This section also discusses the impact of optimizations on performance.

Table 4.1: Evaluation of Skinny-64 and AES-128 circuits using k-fault-resistant partitioning.

Circuit Characteristics			**Faults**		**Algorithm 4.1 Performance**		**Results**	
Name	**Size (GE)**	**Regs (#)**	**Loc. (#)**	**Order k**	***Build Partitioning* (s)**	***Check Integrity* (s)**	**Partitions (#)**	**Exploitable Faults (#)**
Skinny-64 red-1	3 270	235	1 707	1	1.18	0.043	235	
Skinny-64 red-3	4 163	305	2 959	1	1.32	0.052	305	
				2	9.06	0.324	305	128 inputs
Skinny-64 red-4	6 316	341	3 417	1	2.48	0.127	341	+ 64 outputs
				2	10.23	0.404	341	
				3	38.50	0.693	341	+ 335 in checker
AES-128 red-1	20 532	427	16 262	1	597	1.20	427	
AES-128 red-4	29 092	527	23 390	1	1 073	2.14	527	257 inputs
				2	13 983	2.28	527	+ 129 outputs
AES-128 red-5	32 284	561	26 166	1	1 471	2.32	561	
				2	17 376	2.57	561	
				3	201 272	2.79	561	+ 22 in checker

4.5.1 Evaluation Results

As shown in Table 4.1, our algorithm successfully proves the 2-fault security of AES-128 (resp. Skinny-64) implementation in less than 4 h (resp. 10 s) using an Intel Core i7-1185G7 laptop. Our approach also reports that circuit inputs and outputs can be faulted as EDC does not protect them.

Our methodology also assessed the 3-fault security of these circuits, an attacker model unreachable by any existing state-of-the-art tools. However, our fault analysis first fails to build a circuit partitioning. The manual investigation of the logs produced during procedure *BuildPartitioning* shows that three faults defeat the EDC protection by simultaneously targeting: 1) the original function, 2) the redundant function, and 3) the checker disabling the alert at a specific clock cycle. During the following clock cycles, the injected faults propagate and lead to collisions that are undetected by the checker mechanism. Explaining the exploitable faults in more detail is beyond the scope of this thesis.

However, assuming that the exploitable faults identified in the checker cannot be corrupted, i.e., 335 bits in Skinny-64, and only 22 bits in AES-128 where fewer collisions exist, we prove the 3-fault security of AES (resp. Skinny) in 55 h (resp. 39 s). For each algorithm, we reproduced the attack for a full encryption to ensure that the reported counterexamples are not false positives. Attack scenarios are given in Appendix B.

4.5.2 Comparison against Related Work

This section compares our approach against two similar tools: FIVER and FIRMER.

FIVER [RRSS+21] is a formal tool that compares the outputs of a golden model with those of a faulty model to reveal the consequences of faults. Similarly, FIRMER [TGC+23] evaluates the resistance of protected cryptographic circuits by translating the hardware design description into SAT formulas. Both tools perform bounded equivalence checking, unrolling the circuit over several clock cycles for analysis. This technique is not suitable for processor verification, as fault consequences may manifest after an unknown amount of time.

Performance-wise, the authors of FIVER reported 130 hours to prove the 2-fault security of the same AES circuit using an Intel Xeon server, while our approach requires only 4 hours on an Intel Core i7-1185G7 laptop. The authors of FIRMER reported 31 minutes to prove the 2-fault security of AES-128 red-1 using 8 threads of an Intel Xeon Gold 6342 2.80GHz. Our approach thus outperforms FIVER by orders of magnitude when proving the 2-fault security of AES but is slower than FIRMER, which executes on eight threads. However, our proof is inductive, meaning it is valid for multi-round encryption, whereas FIVER's and FIRMER's results hold for a single round. Additionally, we analyzed the 3-fault security of AES, which was previously impossible with state-of-the-art tools.

Furthermore, SYNFI [NOV+22] is a pre-silicon fault analysis framework that allows hardware designers to evaluate the robustness of a circuit and its countermeasures against faults. However, a meaningful performance comparison is not possible as SYNFI only analyzed parts of the AES block (e.g., the FSM).

4.6 Evaluation of OpenTitan

In this section, we apply our co-verification methodology to analyze the resilience of a development version of the OpenTitan platform (cf. Section 4.2.1) running three different programs.

Hardware Countermeasures. Our hardware analysis focuses on the secure configuration of the Ibex core [lowa], which uses different spatial Concurrent Error Detection (CED) schemes (Figure 4.11). The *dual-core lockstep* (DCLS) mechanism instantiates the Ibex core twice, compares the outputs between the main core and the shadow core, and triggers an alert signal on a mismatch. To increase the protection against faults, the shadow core inputs are delayed for d cycles, where d is fixed at synthesis time. Our evaluation uses the default value $d = 2$ but also discusses the results for $d = 3$. Both core instances share the register file, which is protected against faults with Error Detection Codes (EDC) and a write-enable glitch detection mechanism.

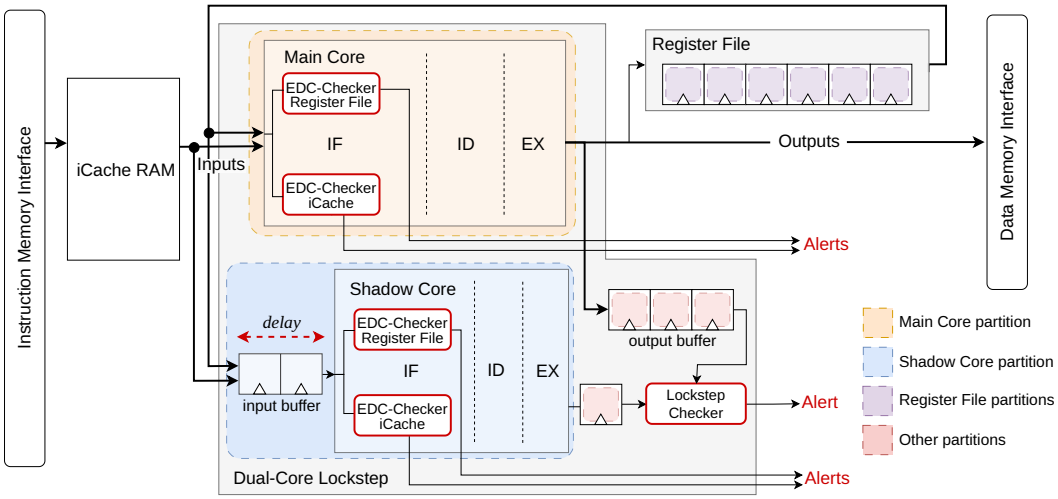


Figure 4.11: Secure Ibex countermeasures and partitioning obtained with Algorithm 4.1.

Threat Model. We consider an attacker having physical access to the Secure Ibex processor capable of interfering with its operation by performing fault injection attacks (*attacker goal*). We consider a single transient bit-flip everywhere in the microarchitecture (*fault model*), which is in line with the protection level provided by the countermeasures. The rest of this section is organized as follows:

First, we evaluate the robustness of the hardware countermeasures implemented in the Secure Ibex processor. Second, we leverage the hardware verification results to analyze whether the identified vulnerabilities can be exploited in different programs. Third, we provide a hardware fix for the vulnerability discovered and re-evaluate the security.

4.6.1 Hardware Verification: the Secure Ibex

In the following, we apply our hardware verification methodology individually to the register file and the DCLS before analyzing the entire Ibex core. Table 4.2 summarizes the area in gate equivalent (GE) for each circuit, provides the number of possible fault locations, and reports verification results and performance using an Intel Xeon Gold 6154 CPU. Our analysis does not consider the sleep mode of the Secure Ibex processor that disables the clock signal, as our circuit model only considers synchronous circuits (Definition 2.1). First, we focus on $k = 1$ since the countermeasures of Secure Ibex aim to mitigate a single fault before discussing the evaluation results with $k = 2$.

4.6.1.1 Register File Analysis

The register file consists of thirty-two 32-bit registers, each protected by a 7-bit EDC. Register file countermeasures ensure that written data is stored at the correct address

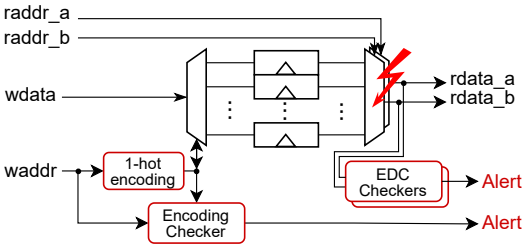


Figure 4.12: Register file protections and the identified vulnerability.

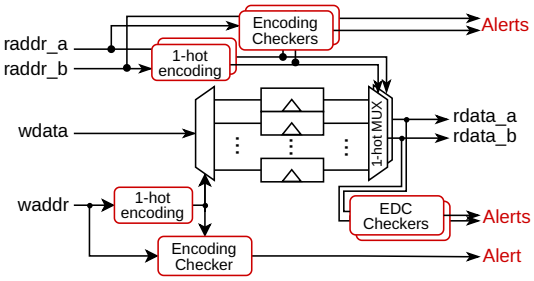


Figure 4.13: Register file with fixed vulnerability.

(*encoding checker* box in Figure 4.12) and that read data have not been modified (*EDC checkers* box in Figure 4.12). Procedure *BuildPartitioning* has proven that a fault injected in the circuit cannot propagate to multiple registers without being detected by the protections. Table 4.2 reports that each register is an independent partition.

However, procedure *CheckIntegrity* has enumerated 172 fault locations in the combinational logic that lead to the corruption of primary outputs. As shown in Figure 4.12, the internal multiplexer tree that selects the register to read according to the inputs signals `raddr_a_i` or `raddr_b_i` is not protected. Hence, a single fault in the mux logic can change which register file value is written back to the core. This is not detected by DCLS as the register file is only read once by the main core, and the value is then stored in the input buffer of the shadow core, i.e., both cores retrieve the same faulty register file value (Figure 4.11). We discuss the mitigation we designed in Section 4.6.3.

In addition, we also evaluate the register file against a weaker fault model targeting only the sequential logic. The EDC protection claims to be robust against 3 faults injected in the data and our analysis proves it as reported in Table 4.2.

Table 4.2: Evaluation of Secure Ibex and its modules using k-fault-resistant partitioning.

Circuit Characteristics			Faults		Algorithm 4.1 Performance			Results		
Name	Size (GE)	Regs (#)	Locations (#)	Order k	*BuildPartitioning* Iter. (#)	*BuildPartitioning* Time	*CheckIntegrity* Time	Partitions (#)	Exploitable Faults $\mathcal{P}'$ (#)	Exploitable Faults $\mathcal{F}'$ (#)
Register File	12 075	1 326	8 392	1	172	38 s	53 s	1 326	0	172
			1 326[a]	3	1	349 s	344 s	1 326	0	0
Register File with fix	11 913	1 326	8 668	1	1	17 s	73 s	1 326	0	0
			1 326[a]	3	1	135 s	383 s	1 326	0	0
DCLS	117 998	5 918	116 561	1	508	20 h 12	5 h 10	1 108	0	0
				2	11	11 s	—	445	—	—
Secure Ibex (no iCache)	130 194	7 248	125 080	1	1	10 h 45	30 h 50	2 438	0	0 (+172)
				2	48	53 s	—	421	—	—

[a] Restricted fault model targeting the sequential logic only

4.6.1.2 Dual-Core Lockstep (DCLS) Analysis.

At first, procedure *BuildPartitioning* described in Algorithm 4.1 failed to build a correct partitioning of the DCLS and grouped every register inside the same partition. Counterexamples provided by the analysis showed that the checker mechanism can be disabled when initializing a specific register to 0. This register drives the `enable_cmp_q` signal and is intended to disable the alert during the first d clock cycles after a system reset as the shadow core and the main core produce different outputs because of the delay. Formal verification leverages this register to turn off the protection failing to prove system 1-fault security. In the following, and without loss of generality, we assume this register is initialized to 1 as it should be during the normal processor operation. Faults can still be injected into this register. Nonetheless, this highlights that the whole DCLS security relies on a 1-bit register that can be written to 0 to disable the protection. We reported this finding to the OpenTitan project and provided a security enhancement that got integrated[3] into the project.

Assuming `enable_cmp_q` = 1, our analysis builds 1 108 partitions. The *main core* and the *shadow core* are two of these, while the others are registers that faults cannot corrupt without raising an alert. Figure 4.11 denotes them as *other partitions*. Building $\mathcal{P}$ takes 508 iterations in 11 h, and then the proof of fault confinement in $\mathcal{P}$ takes 9 h 20 (Table 4.2). Finally, procedure *CheckIntegrity* proves that the DCLS can detect any single bit-flip in one of the two cores and in its internal comparison logic in 5 h 10.

To observe the influence of the DCLS detection delay on the evaluation, we also carried out experiments with $d = 3$. As a result, the design size increases by 3.1%, the number of registers by 13.4%, and the verification time by 24.6%, since the circuit has to be unrolled once more. The analysis concludes with the same results as with $d = 2$.

4.6.1.3 Full Ibex Analysis

Full Ibex comprises the DCLS and the register file. The remaining gates are involved in the sleep unit module, which we disabled. First, we assume that the 172 faults already identified in the register file cannot be reproduced here. Then, we reuse the partitions found when verifying the DCLS and the register file modules to initialize Algorithm 4.1. As a result, procedure *BuildPartitioning* only needs one iteration to prove the fault confinement (Table 4.2), and our methodology proves the 1-fault security of the full Ibex processor against a single fault injection.

4.6.1.4 Discussion on Ibex Analysis with $k = 2$

OpenTitan is designed to be 1-fault secure. Two faults are logically not detected when targeting both cores or disabling the 1-bit alert. However, we report in Table 4.2 how our methodology behaves, with $k = 2$, on an unprotected design. For

[3] https://github.com/lowRISC/ibex/pull/2129

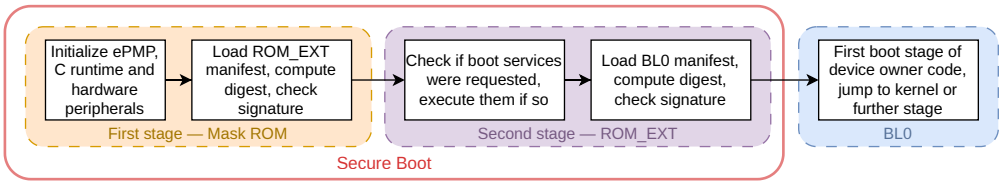


Figure 4.14: OpenTitan secure boot flow.

both the DCLS and the Secure Ibex, procedure *BuildPartitioning* merges most of the previously identified partitions, with $k = 1$, within a few seconds. Multiple partitions remain due to structurally impossible merges or because of non-faultable registers that drive the alert signal directly. Notably, the main core and the shadow core are merged, and the partitioning that results no longer guarantees the output integrity against one fault. We omit the *CheckIntegrity* results as enumerating all the exploitable faults is irrelevant and does not necessarily report genuine vulnerabilities.

4.6.2 System Verification

In this section, we analyze if the exploitable faults previously identified in the register file can be exploited in an attack on the running software. Since the OpenTitan development environment is based on Verilator, all co-verifications have been conducted in the Verilator-based framework described in Section 4.4.2. This framework is very convenient to simulate the whole chip with its hardware accelerators necessary to operate the secure element. Our analysis focuses on the secure boot provided by the OpenTitan project. The other evaluated programs are typical fault injection benchmarks [DRPR19, PHB+19, TAC+22], i.e., VerifyPIN and tiny AES that are not provided by the OpenTitan project. Table 4.3 reports evaluation results and performance using an Intel Xeon Gold 6154 CPU.

4.6.2.1 Secure Boot

The secure boot process guarantees the integrity and authenticity of the code running on the device after a system reset, as illustrated on Figure 4.14. The first stage configures the peripherals, sets up the software environment, and also verifies the integrity of the second boot stage, `ROM_EXT`, stored in Flash memory before booting on it. The second stage of the secure boot provides boot services and verifies the next stage's integrity, i.e., the boot loader (`BL0`) code for the kernel. We focus on verifying the first boot stage, a typical target for fault injection attacks, since it is stored in read-only memory and cannot be modified. We analyze the `rom_verify` function in the `Mask_ROM` code, which is responsible for verifying the authenticity and integrity of the next boot stage. It first computes the digest of the `ROM_EXT` image and checks its RSA signature against the signature stored in the boot manifest.

Attacker Goal. Assuming a malicious `ROM_EXT` code, the attacker wants to bypass the signature check and call the `rom_boot` function:

$$\varphi_{boot\_flash} := (\text{PC} = @\texttt{rom\_boot})$$

Our analysis evaluates faults injected in the `rom_verify` function, assuming that the RSA hardware accelerator (OTBN module) has already computed the signature. Our framework shows that controlling, with a fault, the register file value that is written back is insufficient to bypass the first stage of the secure boot. Even if not detected by the hardware, these faults are captured by the software countermeasures. Hence, the secure boot's signature verification is robust to single bit-flip attacks.

4.6.2.2 Differential Fault Analysis on tiny AES

Differential fault analysis [BS97] enables adversaries to retrieve the cryptographic key by injecting faults during the AES encryption. These attacks can be performed on hardware or software implementations of AES. As our work focuses on the evaluation of hardened CPUs, we do not analyze the AES driver provided in the OpenTitan cryptography library as it utilizes the AES hardware accelerator. Instead, we port the tiny AES [kok19] program, which is not officially provided by the project, to OpenTitan. As previously, we used the framework described in Section 4.4.2 to inject faults into the register file during the AES execution. We illustrate how an attacker can exploit these faults at the software level by reproducing the requirements of two attacks known from the literature [KQ08, TMA11]. An arbitrary plaintext and symmetric key were used for the analysis.

Attacker Goal. The first attack targets the *key schedule* function to corrupt one byte in the first column of the 9^{th} round key (φ_{key_sched}) [TFY07, KQ08]. The second attack targets the *AES algorithm* to corrupt a single byte in the 8^{th} round state matrix (φ_{aes}) [TMA11].

For each experiment, the fault is injected during the round preceding the round of interest. We observe the 9^{th} round key and the 8^{th} round state matrix stored in the data memory[4] and compare them against the precomputed reference values to determine if the fault induced a single-byte corruption. Table 4.3 summarizes our evaluation results for each attack. Our analysis reported 532 successful fault injections over the 5 760 possibilities to satisfy φ_{key_sched}. Similarly, 4 084 successful fault injections were identified over the 38 912 configurations tested to reach φ_{aes}. Inspecting the analysis reports shows that successful fault injections are mainly applied to memory load and store operations.

4.6.2.3 Analysis of VerifyPIN

For the last software verification, we focus on the VerifyPIN test suite [DPP+16] that we port to the chip as it is not part of the OpenTitan project. In this simple

[4] Actually, we observe values on the data memory interface as OpenTitan implements memory scrambling.

Table 4.3: Co-verification results on software use cases exploiting the undetected fault in the register file.

Program Characteristics			Fault Characteritics			Analysis Results			Performance	
Name	**Function/ Version**	**Instr. (#)**	**Attacker Goal φ**	**Timing (clock cycles)**	**Locations (#)**	**Success**	**Fail**	**Timeout**	**Threads (#)**	**Verification Time (s)**
Secure Boot	Mask ROM *signature check*	2 526	φ_{boot_flash}	0 - 1 907	122 048	0	95 238	26 810	8	9 235
Tiny AES	Key Schedule *8th-9th round*	221	φ_{key_sched}	0 - 90	5 760	532	4 666	562	2	458
	AES *7th-8th round*	1 144	φ_{aes}	0 - 610	38 912	4 084	29 477	5 228	8	1 742
VerifyPIN	v0	114	φ_{authen}	0 - 49	3 136	2	2 890	160	1	145
			φ_{ptc}			84				
	v1	121	φ_{authen}	0 - 51	3 264	1	2 990	185	1	154
			φ_{ptc}			89				
	v2	162	φ_{authen}	0 - 91	5 824	1	5 200	537	1	311
			φ_{ptc}			87				
	v3	166	φ_{authen}	0 - 95	6 080	1	5 456	537	1	309
			φ_{ptc}			87				
	v4	189	φ_{authen}	0 - 117	7 488	1	6 714	679	1	468
			φ_{ptc}			95				
	v5	169	φ_{authen}	0 - 97	6 208	0	5 503	628	1	311
			φ_{ptc}			77				
	v6	160	φ_{authen}	0 - 88	5 632	0	5 019	528	1	264
			φ_{ptc}			85				
	v7	187	φ_{authen}	0 - 116	7 424	1	6 682	681	1	399
			φ_{ptc}			61				

authentication mechanism, we have described in detail in Section 3.1.1.2, a user has a maximum number of `g_ptc` attempts to enter the correct 4-digit userPIN matching the secret `cardPIN`. When the authentication succeeds, a global variable `g_authenticated` is set to *true*. The program is available in eight versions with an increasing number of protections against fault attacks and we evaluate all of them in the following.

Attacker Goal. The attacker aims to *i)* bypass the secure authentication:

$$\varphi_{authen} := (\texttt{g\_authenticated} = true)$$

or *ii)* manipulate the maximum number of authentication tries:

$$\varphi_{ptc} := (\texttt{g\_ptc} \geq 3)$$

For each analysis, the attacker's goal is evaluated at the end of VerifyPIN function once the program counter reaches the exit point. Faults can be injected during the entire execution of the program. As a result, Table 4.3 illustrates that φ_{authen} and φ_{ptc} are reachable by an attacker in most of the eight versions of VerifyPIN. For example, injecting a fault when decrementing `g_ptc` fulfills φ_{ptc}. In addition, we observed that φ_{authen} can be reached by setting the cardPIN pointer equal to the `userPIN` pointer and comparing the cardPIN code to itself.

4.6.3 Fixing Register File Vulnerability

As demonstrated in Section 4.6.1, a single fault into the output mux tree of the register file could modify which value is written back to the Secure Ibex. Figure 4.13

depicts our hardware modifications to protect the register file from faults.

First, the read addresses *raddr_ a* and *raddr_ b* are converted to one-hot encoded signals, and their integrity is ensured by checker modules. Then, the one-hot encoded read addresses are each fed into a mux directly operating on these signals. Internally, the one-hot mux selects each output bit individually by performing AND- and OR-reductions on the one-hot encoded addresses and the register file values. As a result, a single bit-flip is immediately detected either by the one-hot encoding checkers or the EDC protections.

As reported in Table 4.2, our verification flow proves the 1-fault security of the fixed register file. Since the fixed Ibex is robust to one single fault injection, no exploitable faults need to be verified in the system verification step, which reduces the overall security verification time. We reported this finding to the OpenTitan project and provided a security enhancement that got integrated[5] into the project.

4.7 Discussion on Methodology Improvements

This section compares the baseline version of µArchiFI described in Chapters 2 and 3 with the co-verification workflow presented in this chapter.

Complexity-wise, the state space to analyze depends on the design size in terms of gates $|G|$, the program length n, and the size of the fault model $|\mathcal{F}|$ raised to the power of the fault order k. The approach in this chapter divides the verification into two steps to reduce this complexity. First, by analyzing the hardware design independently of the program, we no longer depend on the program length n and only need to unroll the circuit d times, where the countermeasure delay d is much smaller than n, i.e., $d \ll n$. Second, the co-verification introduces the software program in the model while the design complexity $|G|$ and the remaining exploitable faults $|\mathcal{F}'|$ can be tremendously reduced, provided countermeasures are proven robust.

Security-wise, the two-step methodology provides much stronger guarantees. First, the hardware is checked only once. If no vulnerabilities are found, all programs are secure. Otherwise, the hardware verification results are used for any program co-verification. Second, in the specific case of Secure Ibex, this chapter formally proves that only faults in the register file are possible and analyzes their consequences on various programs. In contrast, the results obtained on Secure Ibex in Section 3.3 only hold for VerifyPIN, and the entire software/hardware verification must be rerun for each program.

Performance-wise, Use Case II presented in Section 3.3 evaluated the Secure Ibex processor with a restricted fault model $\mathcal{F}$ targeting the shadow core's registers only, which included 2 500 fault locations and verified a 46-instruction program in

[5] https://github.com/lowRISC/ibex/pull/2117

5 minutes. However, Chapter 3 concludes that the current version of µArchiFI cannot process more than a hundred instructions nor evaluate a larger design with more faults. In contrast, our two-step method considered all possible bit-flips, i.e., 125 000 faults, and reduced them to a smaller set of exploitable faults $\mathcal{F}'$ with 172 undetected faults in the register file. This proved that targeting only the shadow core is futile. The co-verification step then verified the robustness of 2 526 secure boot instructions in 2 hours and 30 minutes.

4.8 Conclusion

Summary. This chapter has introduced the novel notion of *k-fault-resistant partitioning* to decompose hardware/software co-verification into more manageable steps. Specifically, we propose a preliminary security evaluation of hardware-level countermeasures to analyze only the faults not captured by hardware protections in the hardware/software co-verification step, thereby containing the state space explosion. We validate our approach by replicating known results on Skinny and AES cryptographic circuits protected with a code-based CED from the Impeccable Circuits [AMR+20], as no similar work exists on CPUs. Further, we show that our methodology outperforms related work, i.e., proves the 2-fault security of AES in less than 4 h compared to 130 h for FIVER [RRSS+21]. In addition, we demonstrate the scalability of k-fault-resistant partitioning by analyzing the 3-fault security of AES, which was not conducted by related work.

Then, to demonstrate the capabilities of the complete fault co-verification methodology, we analyze the k-fault security of a development version of the fault-hardened Ibex processor [lowa] used in OpenTitan [JRR+18]. We first verify two hardware countermeasures, namely its Dual-Core LockStep (DCLS) and the Error Detection Code (EDC) of its register file. Our analysis reveals that DCLS correctly detects any single bit-flip in one of the two cores or in its internal comparison logic, i.e., it is labeled 1-fault secure. However, some single bit-flips injected in the Ibex's register file are not captured by the EDC protection, thus leading to potential software exploitations. The hardware/software co-verification step showcases that an adversary can exploit this vulnerability to manipulate the control flow of the VerifyPIN authentication program [DPP+16] or to perform a differential fault analysis on an AES software implementation [kok19]. Nevertheless, we verify the robustness of the OpenTitan secure element running the first step of a secure boot process, as its software countermeasures prevent the register file vulnerability from being exploited. Performance-wise, k-fault-resistant partitioning allows us to analyze a secure processor with a 130 kGE circuit. The hardware/software co-verification step can then address previously intractable software verification of thousands of instructions.

We disclosed the fault vulnerability of the register file in the Ibex core used in a development version of OpenTitan to the project, which acknowledged our findings.

The fix we provided and formally proved was integrated[6] into the OpenTitan project.

[6] https://github.com/lowRISC/ibex/pull/2117

Chapter 5

Conclusion

Contents

5.1 Conclusion

The literature has proposed numerous formal analysis techniques to analyze fault attacks, either at the circuit level—to accurately represent faults in logic gates—or at the software level—to evaluate the consequences of faults on software programs. However, recent works have reported experimental observations that are inexplicable with classical ISA fault models, such as the instruction skip, or have highlighted subtle fault effects due to processor microarchitecture, such as pipelining. These findings emphasize the need for a system co-verification that considers faults in processor microarchitecture and that analyzes their consequences on software security. In this context, the goal of this Ph.D. thesis was to design a cross-layer fault verification methodology that is exhaustive, efficient, automatic, and capable of proving system security with formal guarantees or detecting subtle fault effects that are difficult to spot manually.

First, Chapter 2 introduced µArchiFI, the first workflow to evaluate the security of hardware/software systems against fault attacks using formal techniques. To develop this tool, we first modeled the hardware/software into a unified representation and formalized the effects of faults on such a system. We then integrated our approach into Yosys to leverage its existing synthesis framework and extend it to represent fault attacks. Finally, µArchiFI generates a formal system specification embedding faults that can be analyzed with model-checking techniques.

Second, Chapter 3 showcases the use of µArchiFI on several versions of the CV32E40P RISC-V processor. We identified multiple microarchitectural mechanisms, such as forwarding, prefetching, and multiplication unit, that can be lever-

aged in attacks to defeat software security. These observations validate our approach by reproducing known results from the literature and highlight the necessity of considering both software and hardware in a joint analysis to fully understand the potential of fault injections. Additionally, we demonstrated that µArchiFI is valuable for proving the robustness of the CFI-like countermeasure MAFIA, which was previously impossible with hardware- or software-only methodologies. To address more complex systems embedding countermeasures and to cope with the state space explosion problem inherent in formal methods, we investigated software-based optimizations like sandboxing or concretization of the program counter value. These methods showed small performance improvement but were insufficient for verifying more ambitious systems like secure elements.

Finally, Chapter 4 presented the novel notion of *fault-resistant partitioning* to decompose hardware/software co-verification into more manageable steps. Specifically, we proposed a preliminary security evaluation of hardware-level countermeasures to analyze only the faults not captured by hardware protections in the hardware/-software co-verification step, thereby containing state space explosion. Experimental results demonstrated that our notion of fault-resistant partitioning, when used for hardware verification alone, outperforms state-of-the-art methodologies in proving the robustness of a secure AES implementation. Additionally, when used in a co-verification process, fault-resistant partitioning allows us to address previously intractable problems, such as OpenTitan running a secure boot process.

Essentially, the work done in this Ph.D. thesis is a significant step toward formal hardware/software security evaluations and a better understanding of microarchitectural fault effects on running software.

5.2 Perspectives

The following paragraphs present several directions to extend the work presented in this thesis.

µArchiFI Improvements. After proposing the initial version of µArchiFIv0 in Chapter 2, the remainder of this manuscript has focused on enhancing the scalability of the approach. First, we integrated the methodology into Yosys to improve the fault modeling process and leverage state-of-the-art backend solvers. Next, we investigated software-based optimization strategies based on restrictions on the program counter (PC) values. Additionally, we decomposed the co-verification process by conducting a preliminary analysis of hardware countermeasures, thereby reducing the number of faults to be analyzed in the subsequent step. Further improvements based on compositional approaches could be explored to reduce problem complexity. For example, a compositional verification strategy could decompose the attacker goal—which is often a global property encompassing the entire system state—into local properties.

Control Flow Integrity (CFI) Support. To propose more effective countermeasures with limited overhead, recent works have introduced combined hardware/software protections, such as hardware-based CFI [DCGÜ+17, CCH22, NSL+23]. Assessing the security of these protections is crucial and cannot be achieved using software- or hardware-only verification techniques. µArchiFI is currently the only tool capable of formally evaluating these countermeasures, as demonstrated in Section 3.2. However, as these countermeasures become more prevalent, improving verification scalability is essential. Future research direction could focus on adapting the k-fault-resistant partitioning approach to these protections.

Better Software Support. This thesis focused on hardware/software co-verification. To accurately model faults in the microarchitecture, we adopted a circuit-level system representation in µArchiFI. Additionally, the work in Chapter 4 further emphasized this near-circuit modeling using a bit-level representation to assess hardware countermeasures. As a result, the software is essentially seen as a set of constraints restricting hardware behavior. Future work could better involve software in the verification process. For instance, it would be interesting to exploit instruction semantics to determine whether a fault is more likely to affect a given instruction based on its class, such as arithmetic or jump instructions. Moreover, this manuscript highlighted the importance of faults remaining hidden in the microarchitecture. When evaluating the fault effects on instructions, the microarchitecture state must also be considered before and after execution. By achieving a detailed characterization of the effects of faults on the microarchitecture, these fault models could be elevated to the ISA level and enrich state-of-the-art fault analyses at the software level.

New Horizons. Performance limitations are the current bottleneck of hardware/software co-verification and efficient methodologies are needed to scale to more ambitious use cases. Assuming these limitations are overcome, the range of verification use cases we can explore is exciting. We will address more complex processor architectures and evaluate the consequences of faults on microarchitectural mechanisms such as out-of-order execution and branch prediction. Additionally, recent research in cryptography has focused on other implications of fault attacks, such as statistically ineffective fault attacks (SIFA) [DEK+18, SRM20] and combined side-channel/fault-injection attacks [DN20, RFSG22, FGM+23]. Although properties on combined attacks are not currently supported with µArchiFI as the attacker goal cannot be directly modeled as a reachability property, studying these attacks promises exciting findings by adding microarchitectural considerations.

Publications

Papers

- Benjamin Binder, Samira Ait Bensaid, Simon Tollec, Farhat Thabet, Mihail Asavoae, and Mathieu Jan. Formal Processor Modeling for Analyzing Safety and Security Properties. In *Embedded Real Time Systems (ERTS)*, 2022.
- Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. Exploration of Fault Effects on Formal RISC-V Microarchitecture Models. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 73–83. IEEE, 2022. doi:10.1109/FDTC57191.2022.00017.
- Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. µArchiFI: Formal Modeling and Verification Strategies for Microarchitectural Fault Injections. In *2023 Formal Methods in Computer Aided Design (FMCAD)*, 2023. doi:10.34727/2023/isbn.978-3-85448-060-0_18.
- Simon Tollec, Vedad Hadžić, Pascal Nasahl, Mihail Asavoae, Roderick Bloem, Damien Couroussé, Karine Heydemann, Mathieu Jan, and Stefan Mangard. Fault-Resistant Partitioning of Secure CPUs for System Co-Verification against Faults. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, pages 179–204, 2024. doi:10.46586/tches.v2024.i4.179-204.

Communications

- *The Spring 2022 RISC-V Week*, Poster, May 2022, Paris, France.
- *The Eleventh Summer School on Formal Techniques* (SSFT), Oral presentation, June 2022, Palo Alto, USA.
- *Journée sur les attaques par injection de fautes* (JAIF), Oral presentations, November 2022, Valence, and October 2024, Rennes, France.
- Winter School on *« Technologies de Conception des Systèmes Embarqués Hétérogènes »* (FETCH), Poster, February 2023, Lavey-les-Bains, Suisse.
- *Journées du groupe de travail sur les méthodes formelles pour la sécurité* (GTMFS), Oral presentation, March 2023 and April 2024, France.

Artifacts

- µArchiFI: https://github.com/CEA-LIST/uArchiFI
  https://zenodo.org/records/7958412
- Partitioning: https://github.com/CEA-LIST/Fault-Resistant-Partitioning

Appendix A

Proof of Theorem 4.1

Proof. To prove that a circuit $\mathcal{C}$ with partitioning $\mathcal{P}$ fulfilling Definition 4.8 also fulfills Definition 4.7, we first prove that it satisfies a stronger inductive property for all $n \in \mathbb{N}^*$:

$$\begin{aligned} &\forall\,(\sigma_i)_{i=1}^{n+d}\,,\ \forall \mathbf{F} \subseteq \mathcal{F} \times [1, n+d],\ |\mathbf{F}| \leq k : \left(\bigwedge_{i=1}^{n+d} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0\right) \implies \\ &\left(\Delta_{\mathcal{P}}\left(\sigma_{n+1}, \sigma_{n+1}^{\mathbf{F}_{[1,n]}}\right) \leq \left|\mathbf{F}_{[1,n]}\right|\right) \wedge \left(\bigwedge_{i=1}^{n} O'\left(\sigma_i\right) = O'\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)\right). \end{aligned} \tag{A.1}$$

Trivially, (A.1) implies it must also satisfy (4.1) for all $n \in \mathbb{N}^*$. As mentioned, the proof proceeds inductively over n, generalizing from an arbitrary execution $(\sigma_i)_{i=1}^{n+d}$.

*(**Basis.**)* For the base case, we must demonstrate (A.1) for $n = 1$, i.e.,

$$\begin{aligned} &\forall\,(\sigma_i)_{i=1}^{d+1}\,,\ \forall \mathbf{F} \subseteq \mathcal{F} \times [1, d+1],\ |\mathbf{F}| \leq k : \left(\bigwedge_{i=1}^{d+1} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0\right) \implies \\ &\left(\Delta_{\mathcal{P}}\left(\sigma_2, \sigma_2^{\mathbf{F}_{[1,1]}}\right) \leq \left|\mathbf{F}_{[1,1]}\right|\right) \wedge \left(O'\left(\sigma_1\right) = O'\left(\sigma_1^{\mathbf{F}_{[1,1]}}\right)\right). \end{aligned} \tag{A.2}$$

This follows directly from (4.2). Let $\mathbf{F} \subseteq \mathcal{F} \times [1, d+1]$ be an attack with $|\mathbf{F}| \leq k$, $(\sigma_i)_{i=1}^{d+1}$ be an execution, and lastly, $(\hat{\sigma}_i)_{i=1}^{d+1}$ be a second execution with $\hat{\sigma}_i = \sigma_i^{\mathbf{F}_{[1,i-1]}}$. Applying (4.2) with $j = 1$ and $k' = 0$ yields

$$\begin{aligned} &\left(\bigwedge_{i=1}^{d+1} I\left(\sigma_i\right) = I\left(\sigma_i^{\mathbf{F}_{[1,i-1]}}\right)\right) \wedge \left(\Delta_{\mathcal{P}}\left(\sigma_1, \sigma_1^{\mathbf{F}_{[1,0]}}\right) \leq 0\right) \wedge \left(\bigwedge_{i=1}^{d+1} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right) = 0\right) \\ &\implies \left(\Delta_{\mathcal{P}}\left(\sigma_2, \sigma_2^{\mathbf{F}_{[1,1]}}\right) \leq \left|\mathbf{F}_{[1,1]}\right|\right) \wedge \left(O'\left(\sigma_1\right) = O'\left(\sigma_1^{\mathbf{F}_{[1,1]}}\right)\right). \end{aligned}$$

As faults in prior states cannot corrupt the input in the current state, we can conclude that $\left(\bigwedge_{i=1}^{d+1} I\left(\sigma_i\right) = I\left(\sigma_i^{\mathbf{F}_{[1,i-1]}}\right)\right) = \top$. Furthermore, since $\sigma_1^{\mathbf{F}_{[1,0]}} = \sigma_1^{\mathbf{F}_{\emptyset}} = \sigma_1$, we get $\Delta_{\mathcal{P}}\left(\sigma_1, \sigma_1^{\mathbf{F}_{[1,0]}}\right) = 0$, simplifying the left-hand side of the implication to just the last term. Generalizing the result, i.e., introducing quantification over free variables $(\sigma_i)_{i=1}^{d+1}$ and $\mathbf{F} \subseteq \mathcal{F} \times [1, d+1]$ with $|\mathbf{F}| \leq k$, produces (A.2) and concludes the induction basis.

*(**Step.**)* For the induction step, we have to show that necessarily

$$\forall\,(\sigma_i)_{i=1}^{n+d+1},\ \forall \mathbf{F}\subseteq\mathcal{F}\times[1,n+d+1],\ |\mathbf{F}|\leq k:\ \left(\bigwedge_{i=1}^{n+d+1} A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)=0\right)\Longrightarrow$$
$$\left(\Delta_{\mathcal{P}}\left(\sigma_{n+2},\sigma_{n+2}^{\mathbf{F}_{[1,n+1]}}\right)\leq\left|\mathbf{F}_{[1,n+1]}\right|\right)\wedge\left(\bigwedge_{i=1}^{n+1}O'\left(\sigma_i\right)=O'\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)\right) \tag{A.3}$$

under the assumption that (A.1) holds. First, let $(\sigma_i)_{i=1}^{n+d+1}$ be an arbitrary execution and $\mathbf{F}\subseteq\mathcal{F}\times[1,n+d+1]$ be an arbitrary attack with $|\mathbf{F}|\leq k$. Consider the expression $\left(\bigwedge_{i=1}^{n+d+1}A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)=0\right)$ and assume it is true ($\top$). Consequently, the weaker expression from 1 up to $n+d$ is also true, i.e., $\left(\bigwedge_{i=1}^{n+d}A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)=0\right)=\top$. This, together with an application of (A.1) to the execution $(\sigma_i)_{i=1}^{n+d}$, means that $\left(\Delta_{\mathcal{P}}\left(\sigma_{n+1},\sigma_{n+1}^{\mathbf{F}_{[1,n]}}\right)\leq\left|\mathbf{F}_{[1,n]}\right|\right)=\top$ and $\left(\bigwedge_{i=1}^{n}O'\left(\sigma_i\right)=O'\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)\right)=\top$. Next, instantiate (4.2) for the executions $(\sigma_i)_{i=n+1}^{n+l+1}$ and $(\hat{\sigma}_i)_{i=n+1}^{n+l+1}$, with $\hat{\sigma}_i=\sigma_i^{\mathbf{F}_{[1,i-1]}}$, the number $k'=\left|\mathbf{F}_{[1,n]}\right|$ and fault attack $\mathbf{F}_{[n+1,n+d+1]}$, which works because $\left|\mathbf{F}_{[n+1,n+d+1]}\right|+k'=\left|\mathbf{F}_{[n+1,n+d+1]}\right|+\left|\mathbf{F}_{[1,n]}\right|=|\mathbf{F}|\leq k$, to get

$$\left(\bigwedge_{i=n+1}^{n+d+1}I\left(\sigma_i\right)=I\left(\sigma_i^{\mathbf{F}_{[1,i-1]}}\right)\right)\wedge\left(\Delta_{\mathcal{P}}\left(\sigma_{n+1},\sigma_{n+1}^{\mathbf{F}_{[1,n]}}\right)\leq\left|\mathbf{F}_{[1,n]}\right|\right)\wedge\left(\bigwedge_{i=n+1}^{n+d+1}A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)=0\right)$$
$$\Longrightarrow\ \left(\Delta_{\mathcal{P}}\left(\sigma_{n+2},\sigma_{n+2}^{\mathbf{F}_{[1,n+1]}}\right)\leq\left|\mathbf{F}_{[1,n]}\right|+\left|\mathbf{F}_{\{n+1\}}\right|\right)\wedge\left(O'\left(\sigma_{n+1}\right)=O'\left(\sigma_{n+1}^{\mathbf{F}_{[1,n+1]}}\right)\right).$$

Similarly to the basis step, past faults cannot lead to different inputs in the current state, and therefore $\left(\bigwedge_{i=n+1}^{n+d+1}I\left(\sigma_i\right)=I\left(\sigma_i^{\mathbf{F}_{[1,i-1]}}\right)\right)=\top$. Moreover, the weaker term from $n{+}1$ to $n{+}d{+}1$ of our assumption must also be true, i.e., $\left(\bigwedge_{i=n+1}^{n+d+1}A\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)=0\right)=\top$. The left-hand side of the implication is $\top$, yieling $\left(\Delta_{\mathcal{P}}\left(\sigma_{n+2},\sigma_{n+2}^{\mathbf{F}_{[1,n+1]}}\right)\leq\left|\mathbf{F}_{[1,n+1]}\right|\right)=\top$ and $\left(O'\left(\sigma_{n+1}\right)=O'\left(\sigma_{n+1}^{\mathbf{F}_{[1,n+1]}}\right)\right)=\top$. Joining the previous facts about the output into $\left(\bigwedge_{i=1}^{n+1}O'\left(\sigma_i\right)=O'\left(\sigma_i^{\mathbf{F}_{[1,i]}}\right)\right)$, we have proven the implication in (A.3). After generalization, we get (A.3) itself. □

Appendix B

Vulnerabilities in Impeccable Circuits Implementations

The Skinny-64 and AES-128 implementations from Impeccable Circuit [AMR+20] protected against 3 faults revealed to be vulnerable. For each cipher, we detail reproducible attack scenarios providing the Plaintext, the Key, and the faults to be injected according to the circuit nomenclature to produce incorrect ciphertexts without triggering an alert.

Skinny-64 red-4

- Plaintext: `0x06034f957724d19d`;
- Key: `0xf5269826fc681238`;
- Expected ciphertext: `0xbb39dfb2429b8ac7`;
- 1^{st} fault: (bit-flip, round 0, `Red_StateReg. s_current_state[44]`);
- 2^{nd} fault: (bit-flip, round 0, `SubCellOutput[46]`);
- 3^{rd} fault: (bit-flip, round 0, `Check1.in1[244]`);
- Faulty ciphertext: `0x0897810d2aa02f8e`.

AES-128 red-5

- Plaintext: `0xd2228cc9f8b8f239b0162a9ad3632127`;
- Key: `0x7f287089fbbebdb8f364377b97f5c9ef`;
- Expected ciphertext: `0x6666b1677c13464929f286aca090eb74`;
- 1^{st} fault: (bit-flip, at round 0, `InputMUX.Q[31]`) ;
- 2^{nd} fault: (bit-flip, at round 0, `RedFinalRoundControlLogicInst.Red_FinalRoundBit[2]`);
- 3^{rd} fault: (bit-flip, at round 0, `Check1.result[2]`);
- Faulty ciphertext: `0xfd711dada3bfa30b6406f71be54e20a1`.

List of Figures

List of Tables

Listings

Bibliography

[ABC+17] Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-luc Rainard, and Rémi Tucoulou. Nanofocused X-Ray Beam to Reprogram Secure Circuits. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, volume 10529, pages 175–188. Springer, 2017. doi:10.1007/978-3-319-66787-4_9. (Cited on page 8).

[ACD+22] Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle, and Paolo Maistri. Variable-Length Instruction Set: Feature or Bug? In *Euromicro Conference on Digital System Design (DSD)*, pages 464–471. IEEE, 2022. doi:10.1109/DSD57027.2022.00068. (Cited on pages 15 and 56).

[AD10] Michel Agoyan and Jean-Max Dutertre. When Clocks Fail: On Critical Paths and Clock Faults. In *Smart Card Research and Advanced Application*, volume 6035, pages 182–193. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-12510-2_13. (Cited on page 6).

[AM11] Sk. Subidh Ali and Debdeep Mukhopadhyay. A Differential Fault Analysis on AES Key Schedule Using Single Fault. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 35–42. IEEE, 2011. doi:10.1109/FDTC.2011.10. (Cited on pages 12 and 76).

[AMR+20] Anita Aghaie, Amir Moradi, Shahram Rasoolzadeh, Aein Rezaei Shahmirzadi, Falk Schellenberg, and Tobias Schneider. Impeccable Circuits. *IEEE Transactions on Computers*, 69:361–376, 2020. doi:10.1109/TC.2019.2948617. (Cited on pages 4, 17, 18, 90, 102, 112, and 120).

[AVFM07] Frederic Amiel, Karine Villegas, Benoit Feix, and Louis Marcel. Passive and Active Combined Attacks: Combining Fault Attacks and Side Channel Analysis. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 92–102. IEEE, 2007. doi:10.1109/FDTC.2007.12. (Cited on page 12).

[AWMN20] Victor Arribas, Felix Wegener, Amir Moradi, and Svetla Nikova. Cryptographic Fault Diagnosis using VerFI. In *International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 229–240. IEEE, 2020. doi:10.1109/HOST45689.2020.9300264. (Cited on pages 20 and 21).

[BAM+23] Sophie Bouat, Stéphanie Anceau, Laurent Maingault, Jessy Clédière, Luc Salvo, and Rémi Tucoulou. X ray nanoprobe for fault attacks and circuit edits on 28-nm integrated circuits. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6. IEEE, 2023. doi:10.1109/DFT59622.2023.10313553. (Cited on page 8).

[BBK+03] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri. Error analysis and detection procedures for a hardware implementation of the advanced encryption standard. *IEEE Transactions on Computers*, 52:492–505, 2003. doi:10.1109/TC.2003.1190590. (Cited on pages 17 and 90).

[BBK+10] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented AES: Effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, pages 1–10. ACM, 2010. doi:10.1145/1873548.1873555. (Cited on page 18).

[BBKN12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proceedings of the IEEE*, 100:3056–3076, 2012. doi:10.1109/JPROC.2012.2188769. (Cited on page 6).

[BBPP09] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. Low Voltage Fault Attacks on the RSA Cryptosystem. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 23–31. IEEE, 2009. doi:10.1109/FDTC.2009.30. (Cited on page 7).

[BBT+22] Benjamin Binder, Samira Ait Bensaid, Simon Tollec, Farhat Thabet, Mihail Asavoae, and Mathieu Jan. Formal Processor Modeling for Analyzing Safety and Security Properties. In *Embedded Real Time Systems (ERTS)*, 2022. No cited.

[BCC+19] Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. *nuXmv User Manual*. Fondazione Bruno Kessler, 2019. (Cited on page 40).

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, 2018. URL: http://arxiv.org/abs/1610.00502, arXiv:1610.00502. (Cited on pages 22 and 51).

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^20 States and beyond. *Information and Computation*, 98:142–170, 1992. doi:10.1016/0890-5401(92)90017-A. (Cited on pages 35 and 36).

[BCN+06] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94:370–382, 2006. doi:10.1109/JPROC.2005.862424. (Cited on pages 6, 13, and 17).

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification (CAV)*, volume 818, pages 68–80. Springer, 1994. doi:10.1007/3-540-58179-0_44. (Cited on page 48).

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology (EUROCRYPT)*, pages 37–51, 1997. doi:10.1007/3-540-69053-0_4. (Cited on page 6).

[BFFH20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, pages 51–53, 2020. (Cited on page 100).

[BFP19] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, pages 199–224, 2019. doi:10.46586/tches.v2019.i2.199-224. (Cited on page 7).

[BGE+17] Jan Burchard, Mañl Gay, Ange-Salomé Messeng Ekossono, Jan Horáček, Bernd Becker, Tobias Schubert, Martin Kreuzer, and Ilia Polian. AutoFault: Towards Automatic Construction of Algebraic Fault Attacks. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 65–72. IEEE, 2017. doi:10.1109/FDTC.2017.13. (Cited on page 20).

[BGV11] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 105–114. IEEE, 2011. doi:10.1109/FDTC.2011.9. (Cited on pages 6, 13, and 14).

[BHE+19] Jean-Baptiste Bréjon, Karine Heydemann, Emmanuelle Encrenaz, Quentin Meunier, and Son-Tuan Vu. Fault attack vulnerability assessment of binary code. In *Workshop on Cryptography and Security in Computing Systems (CS2)*, pages 13–18. ACM, 2019. doi:10.1145/3304080.3304083. (Cited on pages 20 and 22).

[BHW11] Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. 2011. doi:10.35011/FMVTR.2011-2. (Cited on page 40).

[BIL11] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. *Smart Card Research and Advanced Applications*, 7079:283–296, 2011. doi:10.1007/978-3-642-27257-8_18. (Cited on page 12).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, April 2008. (Cited on page 41).

[BLE+16] Noemie Beringuier-Boher, Marc Lacruche, David El-Baze, Jean-Max Dutertre, Jean-Baptiste Rigaud, and Philippe Maurine. Body Biasing Injection Attacks in Practice. In *Workshop on Cryptography and Security in Computing Systems (CS2)*, pages 49–54. ACM, 2016. doi:10.1145/2858930.2858940. (Cited on page 8).

[BLLL18] Sebanjila K. Bukasa, Ronan Lashermes, Jean-Louis Lanet, and Axel Leqay. Let's shock our IoT's heart: ARMv7-M under (fault) attacks. In *International Conference on Availability, Reliability and Security (ARES)*, pages 1–6. ACM, 2018. doi:10.1145/3230833.3230842. (Cited on page 7).

[BLS02] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Computer Aided Verification (CAV)*, volume 2404, pages 78–92. Springer, 2002. doi:10.1007/3-540-45657-0_7. (Cited on page 48).

[BM10] Robert Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification (CAV)*, volume 6174, pages 24–40. Springer, 2010. doi:10.1007/978-3-642-14295-6_5. (Cited on page 91).

[BN08] Alberto Bosio and Giorgio Di Natale. LIFTING: A Flexible Open-Source Fault Simulator. In *2008 17th Asian Test Symposium*, pages 35–40, 2008. doi:10.1109/ATS.2008.17. (Cited on pages 20 and 22).

[Bra11] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 2011. doi:10.1007/978-3-642-18275-4_7. (Cited on page 37).

[Bry86] Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986. doi:10.1109/TC.1986.1676819. (Cited on page 37).

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992. doi:10.1145/136035.136043. (Cited on page 37).

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology (CRYPTO)*, volume 1294, pages 513–525. Springer Berlin Heidelberg, 1997. doi:10.1007/BFb0052259. (Cited on pages 12 and 109).

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard. *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, page 104, 2010. (Cited on page 38).

[BTG10] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. *Smart Card Research and Advanced Application*, 6035:148–163, 2010. doi:10.1007/978-3-642-12510-2_11. (Cited on page 12).

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 2001. doi:10.1023/A:1011276507260. (Cited on page 36).

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *Computer Aided Verification (CAV)*, pages 334–342. Springer, 2014. doi:10.1007/978-3-319-08867-9_22. (Cited on page 41).

[CCH22] Thomas Chamelot, Damien Courousse, and Karine Heydemann. SCI-FI: Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 556–559. IEEE, 2022. doi:10.23919/DATE54114.2022.9774685. (Cited on pages 19, 69, 70, 71, and 116).

[CCH23] Thomas Chamelot, Damien Couroussé, and Karine Heydemann. MAFIA: Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2023. doi:10.1109/TCAD.2023.3276507. (Cited on pages 19 and 69).

[CDHK15] Eduard Cerny, Surrendra Dudani, John Havlicek, and Dmitry Korchemny. *SVA: The Power of Assertions in SystemVerilog*. 2015. doi:10.1007/978-3-319-07139-8. (Cited on page 49).

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131, pages 52–71, 1981. doi:10.1007/BFb0025774. (Cited on page 34).

[CGV+22] Brice Colombier, Paul Grandamme, Julien Vernay, Émilie Chanavat, Lilian Bossuet, Lucie De Laulanié, and Bruno Chassagne. Multi-Spot Laser Fault Injection Setup: New Possibilities for Fault Injection Attacks. In *Smart Card Research and Advanced Applications (CARDIS)*, volume 13173, pages 151–166, 2022. doi:10.1007/978-3-030-97348-3_9. (Cited on page 8).

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer International Publishing, 2018. doi:10.1007/978-3-319-10575-8. (Cited on page 41).

[CKV10] Edmund M. Clarke, Robert P. Kurshan, and Helmut Veith. The Localization Reduction and Counterexample-Guided Abstraction Refinement. In *Time for Verification*, volume 6200, pages 61–71. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-13754-9_4. (Cited on page 39).

[CO23] Zitai Chen and David Oswald. PMFault: Faulting and Bricking Server CPUs through Management Interfaces: Or: A Modern Example of Halt and Catch Fire. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, pages 1–23, 2023. doi:10.46586/tches.v2023.i2.1-23. (Cited on page 8).

[Con88] Wikipedia Contributors. Morris worm — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Morris_worm, 1988. Accessed: June 12, 2024. (Cited on page 1).

[Con14] Wikipedia Contributors. Heartbleed — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Heartbleed, 2014. Accessed: June 12, 2024. (Cited on page 1).

[CPHR21] Ludovic Claudepierre, Pierre-Yves Péneau, Damien Hardy, and Erven Rohou. TRAITOR: A Low-Cost Evaluation Platform for Multifault Injection. In *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems*, pages 51–56. ACM, 2021. doi:10.1145/3457340.3458303. (Cited on page 6).

[Cri22] Common Criteria. Common criteria for information technology security evaluation – cc v3.1. release 5. https://www.commoncriteriaportal.org/cc/index.cfm, 2022. Accessed: July 12, 2024. (Cited on page 2).

[CT05] Hamid Choukri and Michael Tunstall. Round Reduction Using Faults. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2005. (Cited on page 13).

[DBC+18] Jean-Max Dutertre, Vincent Beroulle, Philippe Candelier, Stephan De Castro, Louis-Barthelemy Faber, Marie-Lise Flottes, Philippe Gendrier, David Hely, Regis Leveugle, Paolo Maistri, Giorgio Di Natale, Athanasios Papadimitriou, and Bruno Rouzeyre. Laser Fault Injection at the CMOS 28 nm Technology Node: An Analysis of the Fault Model. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–6. IEEE, 2018. doi:10.1109/FDTC.2018.00009. (Cited on page 7).

[DBP23] Soline Ducousso, Sébastien Bardin, and Marie-Laure Potet. Adversarial Reachability for Program-level Security Analysis. In *European Symposium on Programming (ESOP)*, pages 59–89, 2023. doi:10.1007/978-3-031-30044-8_3. (Cited on pages 20 and 22).

[DCGÜ+17] Ruan De Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. SOFIA: Software and control flow integrity architecture. *Computers & Security*, 68:16–35, 2017. doi:10.1016/j.cose.2017.03.013. (Cited on pages 18 and 116).

[DDCS+14] Jean-Max Dutertre, Stephan De Castro, Alexandre Sarafianos, Noemie Boher, Bruno Rouzeyre, Mathieu Lisart, Joel Damiens, Philippe Candelier, Marie-Lise Flottes, and Giorgio Di Natale. Laser attacks on integrated circuits: From CMOS to FD-SOI. In *International Conference on Design & Technology of*

Integrated Systems in Nanoscale Era (DTIS), pages 1–6. IEEE, 2014. doi:10.1109/DTIS.2014.6850664. (Cited on page 7).

[DDRT12] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 7–15. IEEE, 2012. doi:10.1109/FDTC.2012.15. (Cited on page 7).

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, pages 547–572, 2018. doi:10.46586/tches.v2018.i3.547-572. (Cited on page 116).

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commununications of the ACM*, 1962. doi:10.1145/368273.368557. (Cited on page 38).

[DLM19] Mathieu Dumont, Mathieu Lisart, and Philippe Maurine. Electromagnetic Fault Injection : How Faults Occur. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 9–16. IEEE, 2019. doi:10.1109/FDTC.2019.00010. (Cited on page 7).

[DLM21] M. Dumont, M. Lisart, and P. Maurine. Modeling and Simulating Electromagnetic Fault Injection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40:680–693, 2021. doi:10.1109/TCAD.2020.3003287. (Cited on page 7).

[DMM+13] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. Electromagnetic Glitch on the AES Round Counter. *Constructive Side-Channel Analysis and Secure Design (COSADE)*, 7864:17–31, 2013. doi:10.1007/978-3-642-40026-1_2. (Cited on page 7).

[DN20] Siemen Dhooghe and Svetla Nikova. My Gadget Just Cares for Me - How NINA Can Prove Security Against Combined Attacks. In *Topics in Cryptology – CT-RSA 2020*, volume 12006, pages 35–55. Springer International Publishing, 2020. doi:10.1007/978-3-030-40186-3_3. (Cited on pages 85, 90, and 116).

[DPP+16] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. FISSC: A Fault Injection and Simulation Secure Collection. In *Computer Safety, Reliability, and Security (SAFECOMP)*, 2016. doi:10.1007/978-3-319-45477-1_1. (Cited on pages 59, 60, 109, 112, and iii).

[DRPR19] Jean-Max Dutertre, Timothé Riom, Olivier Potin, and Jean-Baptiste Rigaud. Experimental Analysis of the Laser-Induced Instruction Skip Fault Model. In *Nordic Conference on Secure IT Systems*, volume 11875, pages 221–237. Springer International Publishing, 2019. doi:10.1007/978-3-030-35055-0_14. (Cited on pages 8 and 108).

[Dut14] Bruno Dutertre. Yices 2.2. In *Computer Aided Verification (CAV)*, volume 8559, pages 737–744. Springer, 2014. doi:10.1007/978-3-319-08867-9_49. (Cited on page 38).

[EMB11] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134. IEEE, 2011. (Cited on page 37).

[ESH+11] Sho Endo, Takeshi Sugawara, Naofumi Homma, Takafumi Aoki, and Akashi Satoh. An on-chip glitchy-clock generator for testing fault injection attacks. *Journal of Cryptographic Engineering*, 1:265–270, 2011. doi:10.1007/s13389-011-0022-y. (Cited on page 6).

[FGA+23] Clément Fanjas, Clément Gaine, Driss Aboulkassimi, Simon Pontié, and Olivier Potin. Combined Fault Injection and Real-Time Side-Channel Analysis for Android Secure-Boot Bypassing. In *Smart Card Research and Advanced Applications*, volume 13820, pages 25–44. Springer International Publishing, 2023. doi:10.1007/978-3-031-25319-5_2. (Cited on page 12).

[FGM+23] Jakob Feldtkeller, Tim Güneysu, Thorben Moos, Jan Richter-Brockmann, Sayandeep Saha, Pascal Sasdrich, and François-Xavier Standaert. Combined Private Circuits - Combined Security Refurbished. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1004. ACM, 2023. doi:10.1145/3576915.3623129. (Cited on page 116).

[GHHR23] Antoine Gicquel, Damien Hardy, Karine Heydemann, and Erven Rohou. SAMVA: Static Analysis for Multi-fault Attack Paths Determination. *Constructive Side-Channel Analysis and Secure Design (COSADE)*, pages 3–22, 2023. doi:10.1007/978-3-031-29497-6_1. (Cited on pages 20 and 22).

[Gir05] Christophe Giraud. DFA on AES. In *Advanced Encryption Standard – AES*, volume 3373, pages 27–41. Springer Berlin Heidelberg, 2005. doi:10.1007/11506447_4. (Cited on page 12).

[GJL20] Thomas Given-Wilson, Nisrine Jafri, and Axel Legay. Combined software and hardware fault injection vulnerability detection. *Innovations in Systems and Software Engineering*, 16:101–120, 2020. doi:10.1007/s11334-020-00364-5. (Cited on page 13).

[GJLL17] Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, and Axel Legay. An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT. In *IEEE Trustcom/BigDataSE/ICESS*, pages 293–300. IEEE Computer Society, 2017. doi:10.1109/TRUSTCOM/BIGDATASE/ICESS.2017.250. (Cited on page 20).

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. *Detection of Intrusions and Malware, and Vulnerability Assessment*, 9721:300–321, 2016. doi:10.1007/978-3-319-40667-1_15. (Cited on page 12).

[GS20] Aman Goel and Karem Sakallah. AVR: Abstractly Verifying Reachability. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 413–422. Springer, 2020. doi:10.1007/978-3-030-45190-5_23. (Cited on page 41).

[GS21] Jacob Grycel and Patrick Schaumont. SimpliFI: Hardware Simulation of Embedded Software Fault Attacks. *Cryptography*, 5:15, 2021. doi:10.3390/cryptography5020015. (Cited on pages 20 and 22).

[GST12] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. Infective computation and dummy rounds: Fault protection for block ciphers without check-before-output. *Progress in cryptology – LATINCRYPT 2012*, pages 305–321, 2012. doi:10.1007/978-3-642-33481-8_17. (Cited on page 17).

[GYTS14] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa Taha, and Patrick Schaumont. Differential Fault Intensity Analysis. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 49–58. IEEE, 2014. doi:10.1109/FDTC.2014.15. (Cited on page 12).

[HGA+21] Florian Hauschild, Kathrin Garb, Lukas Auer, Bodo Selmke, and Johannes Obermaier. ARCHIE: A QEMU-Based Framework for Architecture-Independent Evaluation of Faults. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 20–30. IEEE, 2021. doi:10.1109/FDTC53659.2021.00013. (Cited on pages 20, 21, and 22).

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2012. (Cited on page 10).

[HSP21] Max Hoffmann, Falk Schellenberg, and Christof Paar. ARMORY: Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries. *IEEE Transactions on Information Forensics and Security*, 16:1058–1073, 2021. doi:10.1109/TIFS.2020.3027143. (Cited on pages 20 and 21).

[HWM20] Hardware model checking competition (hwmcc). https://fmv.jku.at/hwmcc20, 2020. Accessed: April 29, 2024. (Cited on pages 40 and 41).

[JAR+94] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing*, pages 66–75, 1994. doi:10.1109/FTCS.1994.315656. (Cited on pages 20, 22, and 69).

[JRR+18] Scott Johnson, Dominic Rizzo, Parthasarathy Ranganathan, Jon McCune, and Richard Ho. Titan: enabling a transparent silicon root of trust for cloud. In *Hot Chips: A Symposium on High Performance Chips*, volume 194, 2018. (Cited on pages 4, 86, and 112).

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42:361–372, 2014. doi:10.1145/2678373.2665726. (Cited on page 8).

[KHEB14] Thomas Korak, Michael Hutter, Baris Ege, and Lejla Batina. Clock Glitch Attacks in the Presence of Heating. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 104–114. IEEE, 2014. doi:10.1109/FDTC.2014.20. (Cited on page 6).

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–394, 1976. doi:10.1145/360248.360252. (Cited on pages 22 and 51).

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109, pages 104–113. Springer Berlin Heidelberg, 1996. doi:10.1007/3-540-68697-5_9. (Cited on page 1).

[kok19] kokke. Tiny AES, release 1.0. https://github.com/kokke/tiny-AES-c, 2019. Accessed: February 22, 2024. (Cited on pages 4, 76, 109, and 112).

[KP17] Daniel Kroening and Mitra Purandare. Ebmc. https://github.com/diffblue/hw-cbmc, 2017. Accessed: April 29, 2024. (Cited on page 41).

[KQ07] Chong Hee Kim and Jean-Jacques Quisquater. Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures. *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, 4462:215–228, 2007. doi:10.1007/978-3-540-72354-7_18. (Cited on page 13).

[KQ08] Chong Hee Kim and Jean-Jacques Quisquater. New Differential Fault Analysis on AES Key Schedule: Two Faults Are Enough. In *Smart Card Research and Advanced Applications*, volume 5189, pages 48–60. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-85893-5_4. (Cited on page 109).

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2016. doi:10.1007/978-3-662-50497-0. (Cited on page 37).

[KT14] Daniel Kroening and Michael Tautschnig. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413, pages 389–391. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54862-8_26. (Cited on page 41).

[Kur95] Robert P Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton university press, 1995. (Cited on page 39).

[Lau20] Johan Laurent. *Modélisation de fautes utilisant la description RTL de microarchitectures pour l'analyse de vulnérabilité conjointe matérielle-logicielle*. PhD thesis, Université Grenoble Alpes, 2020. (Cited on page 2).

[LBC+15] Marc Lacruche, Nicolas Borrel, Clement Champeix, Cyril Roscian, Alexandre Sarafianos, Jean-Baptiste Rigaud, Jean-Max Dutertre, and Edith Kussener. Laser fault injection into SRAM cells: Picosecond versus nanosecond pulses. In *2015 IEEE 21st International On-Line Testing Symposium (IOLTS)*, pages 13–18. IEEE, 2015. doi:10.1109/IOLTS.2015.7229820. (Cited on page 8).

[LBD+18] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. On the Importance of Analysing Microarchitecture for Accurate Software Fault Models. In *Euromicro Conference on Digital System Design (DSD)*, pages 561–564, 2018. doi:10.1109/DSD.2018.00097. (Cited on pages 14, 15, and 56).

[LBD+19] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor. *Microprocessors and Microsystems*, 71:102862, 2019. doi:10.1016/j.micpro.2019.102862. (Cited on pages 14 and 15).

[LBDP19] Johan Laurent, Vincent Beroulle, Christophe Deleuze, and Florian Pebay-Peyroula. Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 252–255, 2019. doi:10.23919/DATE.2019.8715158. (Cited on pages 15 and 63).

[LM06] H. Li and S. Moore. Security evaluation at design time against optical fault injection attacks. *IEE Proceedings - Information Security*, 153:3, 2006. doi:10.1049/ip-ifs:20055021. (Cited on page 17).

[LMMS17] Luciano Lavagno, Igor L Markov, Grant Martin, and Lou Scheffer. *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology*. CRC Press, 2017. (Cited on page 91).

[lowa] lowRISC. Ibex RISC-V Core github repository. https://github.com/lowRISC/ibex. Accessed: February 22, 2024. (Cited on pages 4, 86, 104, and 112).

[lowb] lowRISC. Ibex RISC-V Core github repository. https://github.com/lowRISC/ibex#configuration. Accessed: February 22, 2024. (Cited on page 75).

[lowc] lowRISC. OpenTitan Documentation. https://opentitan.org/documentation/index.html. Accessed: December 22, 2023. (Cited on pages 17, 87, and ii).

[lowd] lowRISC. OpenTitan: Lightweight Threat Model. https://opentitan.org/book/doc/security/threat_model/index.html. Accessed: December 22, 2023. (Cited on page 86).

[lowe] lowRISC. OpenTitan: Open source silicon root of trust. https://github.com/lowRISC/opentitan. Accessed: December 22, 2023. (Cited on page 86).

[low18] lowRISC. Ibex: An embedded 32 bit RISC-V CPU core. https://ibex-core.readthedocs.io/en/latest/, 2018. Accessed: February 22, 2024. (Cited on pages 17, 75, and 86).

[McM03] K. L. McMillan. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification (CAV)*, volume 2725, pages 1–13. Springer, 2003. doi:10.1007/978-3-540-45069-6_1. (Cited on page 37).

[MDH+13] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 77–88. IEEE, 2013. doi:10.1109/FDTC.2013.9. (Cited on page 13).

[MDP+20] Alexandre Menu, Jean-Max Dutertre, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Luc Danger. Experimental Analysis of the Electromagnetic Instruction Skip Fault Model. In *International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–7. IEEE, 2020. doi:10.1109/DTIS48698.2020.9081261. (Cited on page 13).

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, September 1955. doi:10.1002/j.1538-7305.1955.tb03788.x. (Cited on pages 26 and 28).

[MIL+21] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark Barrett. Pono: A Flexible and Extensible SMT-Based Model Checker. In *Computer Aided Verification (CAV)*, pages 461–474. Springer, 2021. doi:10.1007/978-3-030-81688-9_22. (Cited on pages 41, 45, 49, and 79).

[MM00] S. Mitra and E.J. McCluskey. Which concurrent error detection scheme to choose ? In *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*, pages 985–994. Int. Test Conference, 2000. doi:10.1109/TEST.2000.894311. (Cited on pages 17 and 90).

[MMB+18] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G. Daly, Dillon Huff, and Pat Hanrahan. CoSA: Integrated Verification for Agile Hardware Design. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–5. IEEE, 2018. doi:10.23919/FMCAD.2018.8603014. (Cited on page 41).

[MNH+16] Noriyuki Miura, Zakaria Najm, Wei He, Shivam Bhasin, Xuan Thuy Ngo, Makoto Nagata, and Jean-Luc Danger. PLL to the rescue: A novel EM fault countermeasure. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. ACM, 2016. doi:10.1145/2897937.2898065. (Cited on page 17).

[MOG+20] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020. doi:10.1109/SP40000.2020.00057. (Cited on page 8).

[Mor14] Nicolas Moro. *Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués*. PhD thesis, Pierre and Marie Curie University, Paris, France, 2014. (Cited on page 18).

[MP23] Krzysztof Marcinek and Witold A. Pleskacz. Variable Delayed Dual-Core Lockstep (VDCLS) Processor for Safety and Security Applications. *Electronics*, 12:464, 2023. doi:10.3390/electronics12020464. (Cited on page 90).

[MS99] J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999. doi:10.1109/12.769433. (Cited on page 38).

[MTOL12] Philippe Maurine, Karim Tobich, Thomas Ordas, and Pierre Yvan Liardet. Yet Another Fault Injection Technique: By Forward Body Biasing Injection. *YACC'2012: Yet Another Conference on Cryptography*, 2012. (Cited on page 8).

[MW78] Timothy C. May and Murray H. Woods. A New Physical Mechanism for Soft Errors in Dynamic Memories. In *16th International Reliability Physics Symposium*, pages 33–40. IEEE, 1978. doi:10.1109/IRPS.1978.362815. (Cited on page 6).

[NM23] Pascal Nasahl and Stefan Mangard. SCRAMBLE-CFI: Mitigating Fault-Induced Control-Flow Attacks on OpenTitan. In *Proceedings of the Great Lakes Symposium on VLSI 2023*, pages 45–50. ACM, 2023. doi:10.1145/3583781.3590221. (Cited on page 19).

[NOV+22] Pascal Nasahl, Miguel Osorio, Pirmin Vogel, Michael Schaffner, Timothy Trippel, Dominic Rizzo, and Stefan Mangard. SYNFI: Pre-Silicon Fault Analysis of an Open-Source Secure Element. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, 2022:56–87, 2022. doi:10.46586/TCHES.V2022.I4.56-87. (Cited on pages 20, 21, 85, 93, and 104).

[NP23] Aina Niemetz and Mathias Preiner. Bitwuzla. In *Computer Aided Verification (CAV)*, volume 13965, pages 3–17. Springer, 2023. doi:10.1007/978-3-031-37703-7_1. (Cited on page 38).

[NPWB18] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , BtorMC and Boolector 3.0. In *Computer Aided Verification (CAV)*, pages 587–595. Springer, 2018. doi:10.1007/978-3-319-96145-3_32. (Cited on pages 40, 41, 45, 49, and 79).

[NSL+23] Pascal Nasahl, Salmin Sultana, Hans Liljestrand, Karanvir Grewal, Michael LeMay, David M. Durham, David Schrammel, and Stefan Mangard. EC-CFI: Control-Flow Integrity via Code Encryption Counteracting Fault Attacks. In *International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 24–35. IEEE, 2023. doi:10.1109/HOST55118.2023.10132915. (Cited on pages 19 and 116).

[OGM15] S. Ordas, L. Guillaume-Sage, and Philippe Maurine. EM Injection: Fault Model and Locality. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 3–13. IEEE, 2015. doi:10.1109/FDTC.2015.9. (Cited on page 7).

[OGM17] S. Ordas, L. Guillaume-Sage, and P. Maurine. Electromagnetic fault injection: The curse of flip-flops. *Journal of Cryptographic Engineering*, 7:183–197, 2017. doi:10.1007/s13389-016-0128-3. (Cited on page 7).

[OGT+15] S. Ordas, L. Guillaume-Sage, K. Tobich, J.-M. Dutertre, and P. Maurine. Evidence of a Larger EM-Induced Fault Model. In *Smart Card Research and Advanced Applications (CARDIS)*, volume 8968, pages 245–259. Springer International Publishing, 2015. doi:10.1007/978-3-319-16763-3_15. (Cited on page 7).

[Opea] OpenHW group. CV32E40P GitHub repository. https://github.com/openhwgroup/cv32e40p. Accessed: April 22, 2023. (Cited on page 57).

[Opeb] OpenHW group. CV32E40P User Manual. https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/index.html. Accessed: February 22, 2024. (Cited on pages 57 and 75).

[OSM02a] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51:111–122, 2002. doi:10.1109/24.994926. (Cited on page 18).

[OSM02b] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51:63–75, 2002. doi:10.1109/24.994913. (Cited on page 18).

[PCNM15] Sikhar Patranabis, Abhishek Chakraborty, Phuong Ha Nguyen, and Debdeep Mukhopadhyay. A Biased Fault Attack on the Time Redundancy Countermeasure for AES. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, volume 9064, pages 189–203. Springer International Publishing, 2015. doi:10.1007/978-3-319-21476-4_13. (Cited on page 15).

[PHB+19] Julien Proy, Karine Heydemann, Alexandre Berzati, Fabien Majéric, and Albert Cohen. A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures: A Secure Software Perspective. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, pages 1–10. ACM, 2019. doi:10.1145/3339252.3339253. (Cited on pages 14 and 108).

[PMPD14] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*, pages 213–222, 2014. doi:10.1109/ICST.2014.34. (Cited on pages 20 and 22).

[PNKI08] Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 472–481, 2008. doi:10.1109/DSN.2008.4630118. (Cited on pages 20 and 21).

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, 2779:77–88, 2003. doi:10.1007/978-3-540-45238-6_7. (Cited on page 12).

[QS02] Jean-Jacques Quisquater and David Samyde. Eddy current for magnetic analysis with active sensor. *Proceedings of eSMART*, 2002. (Cited on page 7).

[RCV+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254. IEEE, 2005. doi:10.1109/CGO.2005.34. (Cited on page 18).

[RFSG22] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. VERICA - Verification of Combined Attacks: Automated formal verification of security against simultaneous information leakage and tampering. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, pages 255–284, 2022. doi:10.46586/tches.v2022.i4.255-284. (Cited on page 116).

[Ric53] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953. arXiv:1990888, doi:10.2307/1990888. (Cited on page 34).

[Ris20] Riscure. Fisim. https://github.com/Riscure/FiSim, 2020. Accessed: February 22, 2024. (Cited on pages 20, 21, and 22).

[RLK11] Thomas Roche, Victor Lomné, and Karim Khalfallah. Combined Fault and Side-Channel Attack on Protected Implementations of AES. *Smart Card Research and Advanced Applications*, 7079:65–83, 2011. doi:10.1007/978-3-642-27257-8_5. (Cited on page 12).

[RNR+15] Lionel Rivière, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures. In *International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2015. arXiv:1510.01537, doi:10.1109/HST.2015.7140238. (Cited on page 13).

[RRSS+21] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. FIVER – Robust Verification of Countermeasures against Fault Injections. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, pages 447–473, 2021. doi:10.46586/tches.v2021.i4.447-473. (Cited on pages 4, 20, 21, 85, 93, 102, 104, and 112).

[RSDT13] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 89–98. IEEE, 2013. doi:10.1109/FDTC.2013.17. (Cited on page 7).

[RSG22] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Guneysu. Revisiting Fault Adversary Models – Hardware Faults in Theory and Practice. *IEEE Transactions on Computers*, pages 1–1, 2022. doi:10.1109/TC.2022.3164259. (Cited on pages 6 and 20).

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2002. doi:10.1007/3-540-36400-5_2. (Cited on pages 6 and 7).

[SBHS16] Bodo Selmke, Stefan Brummer, Johann Heyszl, and Georg Sigl. Precise Laser Fault Injections into 90 nm and 45 nm SRAM-cells. In *Smart Card Research and Advanced Applications (CARDIS)*, volume 9514, pages 193–205. Springer International Publishing, 2016. doi:10.1007/978-3-319-31271-2_12. (Cited on page 7).

[SGD08] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. Practical Setup Time Violation Attacks on AES. In *2008 Seventh European Dependable Computing Conference*, pages 91–96. IEEE, 2008. doi:10.1109/EDCC-7.2008.11. (Cited on page 7).

[SH07] Jörn-Marc Schmidt and Michael Hutter. Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results. *Austrochip*, 2007. (Cited on page 7).

[SH08] Jörn-Marc Schmidt and Christoph Herbst. A Practical Fault Attack on Square and Multiply. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 53–58. IEEE, 2008. doi:10.1109/FDTC.2008.10. (Cited on page 13).

[SJ00] Sung-Ming Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49:967–970, Sept./2000. doi:10.1109/12.869328. (Cited on page 12).

[Sko06] Sergei Skorobogatov. Optically Enhanced Position-Locked Power Analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, 4249:61–75, 2006. doi:10.1007/11894063_6. (Cited on page 12).

[Sko09] Sergei Skorobogatov. Local heating attacks on Flash memory devices. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 1–6. IEEE, 2009. doi:10.1109/HST.2009.5225028. (Cited on page 8).

[SMT23] 18th international satisfiability modulo theories competition – smt-comp'23. https://smt-comp.github.io/2023/, 2023. Accessed: April 30, 2024. (Cited on page 38).

[Sno] Zachary Snow. sv2v. https://github.com/zachjs/sv2v. Accessed: February 22, 2024. (Cited on page 46).

[Sny] Wilson Snyder. Verilator. https://veripool.org/verilator/. Accessed: February 22, 2024. (Cited on page 101).

[SRM20] Aein Rezaei Shahmirzadi, Shahram Rasoolzadeh, and Amir Moradi. Impeccable Circuits II. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020. doi:10.1109/DAC18072.2020.9218615. (Cited on page 116).

[SS98] Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems - Design and Evaluation (3. Ed.)*. A K Peters, 1998. (Cited on page 90).

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 127–144. Springer, 2000. doi:10.1007/3-540-40922-X_8. (Cited on page 37).

[STB97] V. Sieh, O. Tschache, and F. Balbach. VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 32–36. IEEE Comput. Soc, 1997. doi:10.1109/FTCS.1997.614074. (Cited on pages 20, 22, and 69).

[TAC+22] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. Exploration of Fault Effects on Formal RISC-V Microarchitecture Models. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 73–83. IEEE, 2022. doi:10.1109/FDTC57191.2022.00017. (Cited on pages 26, 56, 68, and 108).

[TAC+23] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. µArchiFI: Formal Modeling and Verification Strategies for Microarchitectural Fault Injections. In *2023 Formal Methods in Computer Aided Design (FMCAD)*, 2023. doi:10.34727/2023/isbn.978-3-85448-060-0_18. (Cited on pages 26 and 56).

[TBC20] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clédière. Fault Injection Characterization on Modern CPUs: From the ISA to the Micro-Architecture. In *Information Security Theory and Practice*, volume 12024, pages 123–138. Springer International Publishing, 2020. doi:10.1007/978-3-030-41702-4_8. (Cited on page 13).

[TBE+21] Thomas Trouchkine, Sébanjila Kevin Bukasa, Mathieu Escouteloup, Ronan Lashermes, and Guillaume Bouffard. Electromagnetic fault injection against a complex CPU, toward new micro-architectural fault models. *Journal of Cryptographic Engineering*, 11:353–367, 2021. doi:10.1007/s13389-021-00259-6. (Cited on page 15).

[TFY07] Junko Takahashi, Toshinori Fukunaga, and Kimihiro Yamakoshi. DFA Mechanism on the AES Key Schedule. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 62–74. IEEE, 2007. doi:10.1109/FDTC.2007.13. (Cited on pages 12, 76, and 109).

[TGC+23] Huiyu Tan, Pengfei Gao, Taolue Chen, Fu Song, and Zhilin Wu. SAT-based Formal Fault-Resistance Verification of Cryptographic Circuits, 2023. URL: http://arxiv.org/abs/2307.00561, arXiv:2307.00561. (Cited on pages 4, 20, 21, and 104).

[THN+24] Simon Tollec, Vedad Hadžić, Pascal Nasahl, Mihail Asavoae, Roderick Bloem, Damien Couroussé, Karine Heydemann, Mathieu Jan, and Stefan Mangard. Fault-Resistant Partitioning of Secure CPUs for System Co-Verification against Faults. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, pages 179–204, 2024. doi:10.46586/tches.v2024.i4.179-204. (Cited on page 83).

[TK10] Elena Trichina and Roman Korkikyan. Multi Fault Laser Attacks on Protected CRT-RSA. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 75–86. IEEE, 2010. doi:10.1109/FDTC.2010.14. (Cited on page 8).

[TM17] Niek Timmers and Cristofaro Mune. Escalating Privileges in Linux Using Voltage Fault Injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8. IEEE, 2017. doi:10.1109/FDTC.2017.16. (Cited on pages 7 and 12).

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 224–233. 2011. doi:10.1007/978-3-642-21040-2_15. (Cited on pages 12 and 109).

[Tol22] Simon Tollec. Exploration of fault effects on formal risc-v microarchitecture models. In *Journée sur les attaques par injection de fautes – JAIF'22*, 2022. https://jaif.io/2022/. (Cited on page 56).

[Tol23] Simon Tollec. Analysis of fault effects on formal risc-v microarchitecture models. In *Annual Meeting of the working group "Formal Methods for Security" – GTMFS'23*, 2023. https://gtmfs2023.sciencesconf.org/. (Cited on page 26).

[Tol24a] Simon Tollec. Fault-resistant partitioning of secure cpus for system co-verification against faults. In *Journée sur les attaques par injection de fautes – JAIF'24*, 2024. https://jaif.io/2024/. (Cited on page 83).

[Tol24b] Simon Tollec. Proving hardware security of cpus to analyze software resistance against faults attacks. In *Annual Meeting of the working group "Formal Methods for Security" – GTMFS'24*, 2024. https://gtmfs2024.sciencesconf.org/. (Cited on page 83).

[Tri] Tristan Gingold and contributors. GHDL 3.0. https://github.com/ghdl/ghdl. Accessed: February 22, 2024. (Cited on pages 42 and 46).

[TS16] Niek Timmers and Albert Spruyt. Bypassing Secure Boot using Fault Injection, 2016. (Cited on pages 7 and 12).

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1057–1074. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang. (Cited on pages 8 and 12).

[TSW16] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM Using Fault Injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016. doi:10.1109/FDTC.2016.18. (Cited on page 12).

[Tur37] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42:230–265, 1937. doi:10.1112/plms/s2-42.1.230. (Cited on page 34).

[VM02] T. Verdel and Y. Makris. Duplication-based concurrent error detection in asynchronous circuits: Shortcomings and remedies. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 345–353. IEEE Comput. Soc, 2002. doi:10.1109/DFTVS.2002.1173531. (Cited on page 90).

[VTM+17] Aurelien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adele Morisset, and Sebastien Ermeneux. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 41–48. IEEE, 2017. doi:10.1109/FDTC.2017.18. (Cited on pages 8 and 12).

[VWWM11] Jasper G.J. Van Woudenberg, Marc F. Witteman, and Federico Menarini. Practical Optical Fault Injection on Secure Microcontrollers. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 91–99. IEEE, 2011. doi:10.1109/FDTC.2011.12. (Cited on page 15).

[WMP20] Vincent Werner, Laurent Maingault, and Marie-Laure Potet. An End-to-End Approach for Multi-Fault Attack Vulnerability Assessment. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 10–17. IEEE, 2020. doi:10.1109/FDTC51366.2020.00009. (Cited on page 8).

[Wola] Claire Wolf. Symbiyosys (sby) documentation. https://symbiyosys.readthedocs.io/en/latest/index.html. Accessed: June 22, 2024. (Cited on pages 45 and 53).

[Wolb] Claire Wolf. Yosys open synthesis suite. https://yosyshq.net/yosys. Accessed: February 22, 2024. (Cited on pages 41, 42, and 100).

[WUSM18] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. Sponge-Based Control-Flow Protection for IoT Devices. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 214–226, 2018. doi:10.1109/EuroSP.2018.00023. (Cited on page 18).

[YGS15] Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. Improving Fault Attacks on Embedded Software Using RISC Pipeline Characterization. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 97–108. IEEE, 2015. doi:10.1109/FDTC.2015.16. (Cited on pages 14 and 56).

[YGS+16] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58. IEEE, 2016. doi:10.1109/FDTC.2016.21. (Cited on pages 6, 14, 15, and 18).

[Yos21] Equivalence checking with yosys (eqy). https://yosyshq.readthedocs.io/projects/eqy, 2021. Accessed: May 15, 2024. (Cited on page 91).

[YSW18] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault Attacks on Secure Embedded Software: Threats, Design and Evaluation. *Journal of Hardware and Systems Security*, pages 111–130, 2018. URL: http://arxiv.org/abs/2003.10513, arXiv:2003.10513. (Cited on pages 6, 8, 9, 15, and i).

[Yuc18] Bilgiday Yuce. *Fault Attacks on Embedded Software: New Directions in Modeling, Design, and Mitigation*. PhD thesis, Virginia Polytechnic Institute, 2018. (Cited on page 2).

[ZDCT13] Loic Zussa, Jean-Max Dutertre, Jessy Clediere, and Assia Tria. Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, pages 110–115. IEEE, 2013. doi:10.1109/IOLTS.2013.6604060. (Cited on page 7).

[ZDT+14] Loic Zussa, Amine Dehbaoui, Karim Tobich, Jean-Max Dutertre, Philippe Maurine, Ludovic Guillaume-Sage, Jessy Clediere, and Assia Tria. Efficiency of a glitch detector against electromagnetic fault injection. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE Conference Publications, 2014. doi:10.7873/DATE.2014.216. (Cited on pages 7 and 17).

[ZL79] J. F. Ziegler and W. A. Lanford. Effect of Cosmic Rays on Computer Memories. *Science*, 206:776–788, 1979. doi:10.1126/science.206.4420.776. (Cited on page 6).

[ZPRD23] Anthony Zgheib, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Max Dutertre. CIFER: Code Integrity and control Flow verification for programs Executed on a RISC-V core. In *International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 100–110. IEEE, 2023. doi:10.1109/HOST55118.2023.10133542. (Cited on page 19).